Making Apps with Moqui

Holistic Enterprise Applications Made Easy

by David E. Jones

Sponsored by:

HotWax Media, Inc.

http://www.hotwaxmedia.com

HotWax Media designs, implements, and supports custom ERP applications and systems of innovation that help businesses run faster, leaner, and better.

Leapfrogging the inflexible legacy ERP suites of years past, HotWax Media leverages flexible open source software to create business systems for today's innovators and tomorrow's industry leaders.

If you are running an old fashioned ERP mega suite, the time has come to think different. If you have a new vision for system innovation in your industry, open source ERP is the way to make it happen, and Moqui is a great option to consider.

HotWax Media is proud to sponsor this book, and we actively cheer on Moqui's long-term success!

Moqui Ecosystem

http://www.moqui.org

Sponsor this book to see your logo and a description of your offerings here!

Contact author at dej@dejc.com for details.

Version 1.0 - First Edition

Based on **Moqui Framework version 1.4.1** and **Mantle Business Artifacts version 0.5.2**. These open source projects are public domain licensed and are available for download through http://www.moqui.org.

A special thanks to the sponsors who helped make this book what it is, keep the price low on the printed book, and make the PDF download version available for free.

Designed for a full color 8x10" bound book. With this format the PDF version can also be printed on A4 or US Letter paper.

For the CreateSpace print edition:

ISBN-13: 978-0692267059

ISBN-10: 0692267050

Table of Contents

Also sponsored by:

David E. Jones Consulting

http://www.dejc.com

I help organizations build custom ERP, CRM, and eCommerce systems based on open source software.

I am the founder of various open source projects including Apache OFBiz, Moqui Framework, and Mantle Business Artifacts. Since starting OFBiz in 2001 I have worked on over 100 custom systems and commercial products based on these open source projects.

Jimmy Shen

http://jimmyshen.info

I have 12 years experienced in enterprise application development and operation as well as infrastructure operation. Since 2012, I have been moving to open source solutions to build enterprise-class applications and infrastructure with scalability, high availability and openness, such as Moqui, AngularJS, OpenStack, Docker, etc.

Sharan Foga

ERP Project Manager and Functional Consultant

http://cz.linkedin.com/in/sfoga/

Author of "Getting Started with Apache OFBiz Accounting" and "Getting Started with Apache OFBiz Manufacturing & MRP". I enjoy working with people to show them how Apache OFBiz works and how it can be configured to fit their existing business processes. I also focus on producing practical and good quality End User documentation and other specific training related materials.

Moqui Ecosystem

http://www.moqui.org

Sponsor this book to see your logo and a short description of your offerings here!

See the full page ad for Ant Websystems (http://www.antwebsystems.com/) on page 18.

See the full page ad for HotWax Media (http://www.hotwaxmedia.com/) on page 32.

Foreword

I am not a professional framework developer. I am, just like you, a professional application developer. My career is oriented around building and customizing applications for a wide variety of organizations to manage processes and automate information management.

Like any craftsman an application developer needs a good set of tools, and my quest for the best tools possible started in 1999 when I got into this business. At the time Enterprise Java was maturing and going through a period of standardization to help consolidate and organize the many different tools and technologies that were available in the marketplace.

There was only one problem: for building large-scale systems like an ERP application these tools and technologies were painful to develop with, required massive hardware to run satisfactorily, and were plagued by inadequate standards that practically guaranteed lock-in to application servers that featured enterprise-grade price tags. These applications were also difficult and expensive to customize and maintain after initial implementation. It was, in a word, horrible.

Various open source alternatives were starting to emerge to compete with the commercial players that drove much of the standardization, and this helped with the licensing cost but did little for the inefficiencies in both development and production performance.

There was much room for improvement. In 2001 I started an open source project called The Open For Business Project (OFBiz) with the wide ranging goal of acting as a foundation for all manner of information automation applications. This was meant to enable consolidated systems and include eCommerce, ERP, CRM, MRP, and so on. Based on my experience with enterprise Java tools and exposure to some novel ideas and patterns people were starting to develop, I designed a very different sort of tool set. This tool set was not plagued by object mapping to organize data and encapsulate logic, and embraced the service-oriented design patterns for internal use that have become the standard for interoperation between applications.

Along with technical development tools, a good application developer also needs a flexible and comprehensive data model to give structure and consistency to applications developed. Fortunately in early March 2001, just two months before I started The Open For Business Project, Len Silverston published The Data Model Resource Book, Revised Edition, Volume 1

and Volume 2. This was a huge expansion and rewrite of an earlier book with a similar name by Silverston, Inmon, and Graziano in 1997.

The data model ideas and patterns presented in these two volumes became the foundation for the data model in OFBiz. They have gracefully acted as a foundation for that system during the growth of the project from a simple eCommerce application to a full-featured ERP and CRM system that is used by thousands of organizations and is the basis for over a dozen commercial and open source extensions.

Over years of working on a wide variety of projects based on OFBiz the framework was expanded along with the higher level business artifacts in the project. The ideas for improvements to the framework flowed in steadily, and some extensions and competitors to it outside of OFBiz emerged as well. Many of the ideas were incorporated, but as the project grew and as the community of users and contributors exploded it became increasingly difficult to change fundamental aspects of the system.

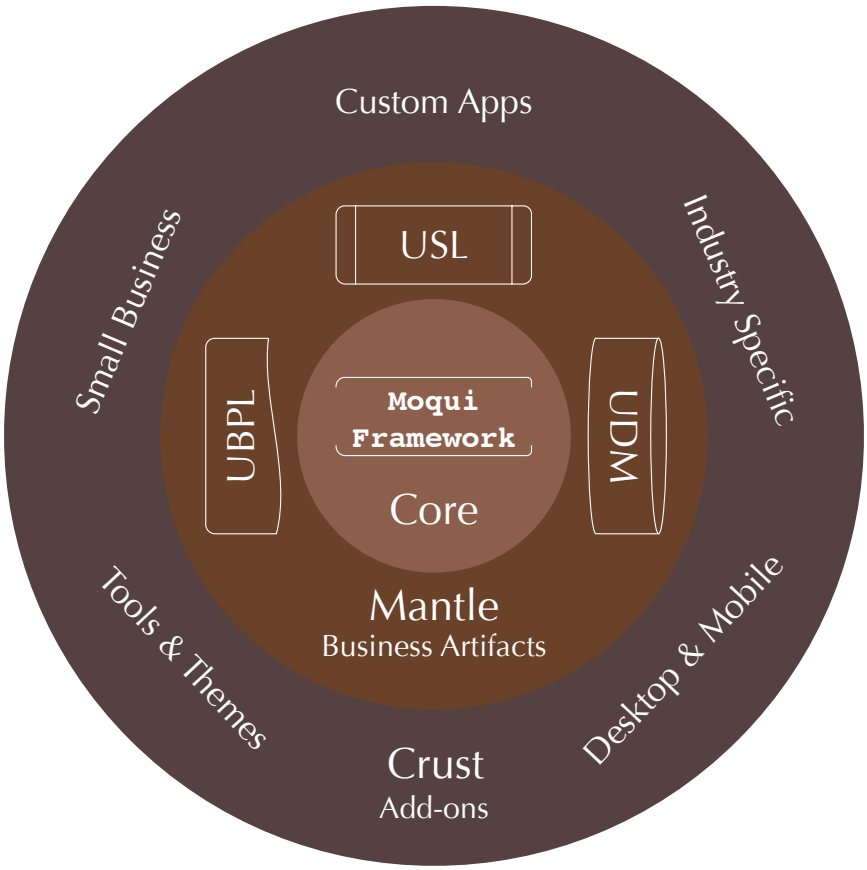
For years I kept a list of dozens of great ideas that constituted major changes to improve and expand the framework. As the list got longer I knew a different approach would be necessary to enter the next phase of my aforementioned quest for the best toolset possible. The result was the birth of the Moqui Framework as an independent project, and the Mantle Business Artifacts to provide a generic foundation for an ecosystem of open source projects, internal applications, and commercial products that go way beyond what one community could do with a single generic open source project.

This book will help you get started with the Moqui Framework and Mantle Business Artifacts, and provide a reference during months and years of building excellent applications.

1. Introduction to Moqui

What is the Moqui Ecosystem?

The Moqui Ecosystem is a set of software packages centered on a common framework and universal business artifacts. The central packages (in the Core and Mantle) are organized as separate open source projects to keep their purpose, management, and dependencies focused and clean. Both are managed with a moderated community model, much like the Linux Kernel.



The goal of the ecosystem is to provide a number of interoperating and yet competing enterprise applications (in the Crust), all based on a common framework for flexibility and easy customization, and a common set of business artifacts (data model and services) so they are implicitly integrated.

The ecosystem includes:

- **Moqui Framework**: Synergistic tools for efficient and flexible application building
- **Mantle Business Artifacts**: Universal business artifacts to make your applications easier to build and implicitly integrated with other apps built on Moqui and Mantle
  - Universal Business Process Library (UBPL)
  - Universal Data Model (UDM)
  - Universal Service Library (USL)
- **Moqui Crust**: themes, tool integrations, and applications for different industries, company sizes, business areas, etc

The focus of this book is Moqui Framework, and the last chapter is a summary of Mantle Business Artifacts.

What is Moqui Framework?

Moqui Framework is an all-in-one, enterprise-ready application framework based on Groovy and Java. The framework includes tools for screens, services, entities, and advanced functionality based on them such as declarative artifact-aware security and multi-tenancy.

The Framework is well suited for a wide variety of applications from simple web sites (like moqui.org) and small form-based applications to complex ERP systems. Applications built with Moqui are easy to deploy on a wide variety of highly scalable infrastructure software such as Java Servlet containers (or app servers) and both traditional relational and more modern NoSQL databases.

Moqui Framework is based on a decade of experience with The Open For Business Project (now Apache OFBiz, see http://ofbiz.apache.org) and designed and written by the person who founded that project. Many of the ideas and approaches, including the pure relational data layer (no object-relational mapping) and the service-oriented logic layer, stem from this legacy and are present in Moqui in a more refined and organized form.

With a cleaner design, more straightforward implementation, and better use of other excellent open source libraries that did not exist when OFBiz was started in 2001, the Moqui Framework code is about 20% of the size of the OFBiz Framework while offering significantly more functionality and more advanced tools.

The result is a framework that helps you build applications that automatically handles many concerns that would otherwise require a significant percentage of overall effort for every application you build.

Moqui Concepts

Application Artifacts

The Moqui Framework toolset is structured around artifacts that you can create to represent common parts of applications. In Moqui the term artifact refers to anything you create as a developer and includes various XML files as well as scripts and other code. The framework supports artifacts for things like:

- **entities** for the relational data model used throughout applications (used directly, no redundant object-relational mapping)
- **screens** and **forms** for web-based and other user interfaces (base artifacts in XML files with general or user-specific extensions in the database)
- screen **transitions** to configure flow from screen to screen and process input as needed along the way
- **services** for logic run internally or exposed for remote execution
- **ECA** (event-condition-action) rules triggered on system events like entity and service operations and received email messages

Here is a table of common parts of an application and the artifact or part of an artifact that handles each:

screen	XML Screen (rendered as various types of text, or can be used to generate other UIs; OOTB support for html, xml, xsl-fo, csv, and plain text)
form	XML Form (defined within a screen; various OOTB widgets and easy to add custom ones or customize existing ones)
prepare data for display	screen actions (defined within a screen, can call external logic)
flow from one screen to another	screen transition with conditional and default responses (defined within the originating screen, response points to destination screen or external resource)
process input	transition actions (either a single service defined to match the form and share validations/etc, or actions embedded in the screen definition or call external logic)
menu	automatic based on sub-screen hierarchy and configured menu title and order for each screen, or define explicitly

internal service	XML service definition and various options for embedded or external service implementations
XML-RPC and JSON-RPC services	internal service with `allow-remote=true` and called through generic interfaces using the natural List and Map structure mappings
RESTful web services	internal service called through simple transition definition supporting path, form body, and JSON body requests and JSON or XML responses
remote service calls	define an internal service as a proxy with automatic XML-RPC, JSON-RPC, and other mappings, or use simple tools for RESTful and other service types
send email	screen designed to be rendered directly as html and plain text and configured along with subject, etc in an `EmailTemplate` record
receive email	define an Email ECA rule to call an internal service that processes the email
use scripts, templates, and JCR content	access and execute/render through the Resource Facade

The Execution Context

The `ExecutionContext` is the central application-facing interface in the Moqui API. An instance is created specifically for executing edge artifacts such as a screen or service. The `ExecutionContext`, or "`ec`" for short, has various facade interfaces that expose functionality for the various tools in the framework.

The `ec` also keeps a context map that represents the variable space that each artifact runs in. This context map is a stack of maps and as each artifact is executed a fresh map is pushed onto the stack, then popped off it once the artifact is done executing. When reading from the map stack it starts at the top and goes down until it finds a matching map entry. When writing to the map stack it always writes to the map at the top of the stack (unless to explicitly reference the root map, i.e., at the bottom of the stack).

With this approach each artifact can run without concern of interfering with other artifacts, but still able to easily access data from parent artifacts (the chain of artifacts that called or included down to the current artifact). Because the `ec` is created for the execution of each

edge artifact it has detailed information about every aspect of what is happening, including the user, messages from artifacts, and much more.

The Artifact Stack

As each artifact is executed and includes or calls other artifacts the artifact is pushed onto a stack that keeps track of the active artifacts, and is added to an artifact history list tracking each artifact used.

As artifacts are pushed onto the stack authorization for each artifact is checked, and security information related to the artifact is tracked. With this approach authz settings can be simplified so that artifacts that include or call or artifacts can allow those artifacts to inherit authorization. With inherited authorization configurations are only needed for key screens and services that are accessed directly.

Peeking Under the Covers

When working with Moqui Framework you'll often be using higher-level artifacts such as XML files. These are designed to support most common needs and have the flexibility to drop down to lower level tools such as templates and scripts at any point. At some point though you'll probably either get curious about what the framework is doing, or you'll run into a problem that will be much easier to solve if you know exactly what is going on under the covers.

While service and entity definitions are handled through code other artifacts like XML Actions and the XML Screens and Forms are just transformed into other text using macros in FreeMarker template files. XML Actions are converted into a plain old Groovy script and then compiled into a class which is cached and executed. The visual (widget) parts of XML Screens and Forms are also just transformed into the specified output type (html, xml, xsl-fo, csv, text, etc) using a template for each type.

With this approach you can easily see the text that is generated along with the templates that produced the text, and through simple configuration you can even point to your own templates to modify or extent the OOTB functionality.

Development Process

Moqui Framework is designed to facilitate implementation with natural concept mappings from design elements such as screen outlines and wireframes, screen flow diagrams, data statements, and automated process descriptions. Each of these sorts of design artifacts can be turned into a specific implementation artifact using the Moqui tools.

These design artifacts are usually best when based on requirements that define and structure specific activities that the system should support to interact with other actors including

people and systems. These requirements should be distinct and separate from the designs to help drive design decisions and make sure that all important aspects of the system are considered and covered in the designs.

With this approach implementation artifacts can reference the designs they are based on, and in turn designs can reference the requirements they are based on. With implementation artifacts that naturally map to design artifacts both tasking and testing are straightforward.

When implementing artifacts based on such designs the order that artifacts are created is not so important. Different people can even work simultaneously on things like defining entities and building screens.

For web-based applications, especially public-facing ones that require custom artwork and design, the static artifacts such as images and CSS can be in separate files stored along with screen XML files using the same directory structure that is used for subscreens using a directory with the same name as the screen. Resources shared among many screens live naturally under screens higher up in the subscreen hierarchy.

The actual HTML generated from XML Screens and Forms can be customized by overriding or adding to the FreeMarker macros that are used to generate output for each XML element. Custom HTML can also be included as needed. This allows for easy visual customization of the generic HTML using CSS and JavaScript, or when needed totally custom HTML, CSS, and JavaScript to get any effect desired.

Web designers who work with HTML and CSS can look at the actual HTML generated and style using separate CSS and other static files. When more custom HTML is needed the web designers can produce the HTML that a developer can put in a template and parameterize as needed for dynamic elements.

Another option that sometimes works well is to have more advanced web designers build the entire client side as custom HTML, CSS, and JavaScript that interacts with the server through a service interface using some form of JSON over HTTP. This approach also works well with client applications for mobile or desktop devices that will interact with the application server using web services. The web services can use the automatic JSON-RPC or XML-RPC or other custom automatic mappings, or can use custom wrapper services that call internal services to support any sort of web service architecture.

However your team is structured and however work is to be divided on a given project, with artifacts designed to handle defined parts of applications it is easier to split up work and allow people to work in parallel based on defined interfaces.

Development Tools

For requirements and designs you need a group content collaboration tool that will be used by users and domain experts, analysts, designers, and developers. The collaboration tool should support:

- hierarchical documents
- links between documents and parts of documents (usually to headers within the target document)
- attachments to documents for images and other supporting documents
- full revision history for each document
- threaded comments on each document
- email notification for document updates
- online access with a central repository for easy collaboration

There are various options for this sort of tool, though many do not support all the above and collaboration suffers because of it. One good commercial option is Atlassian Confluence. Atlassian offers a very affordable hosted solution for small groups along with various options for larger organizations. There are various open source options, including the wiki built into HiveMind PM which is based on Moqui Framework and Mantle Business Artifacts.

Note that this content collaboration tool is generally separate from your code repository, though putting requirement and design content in your code repository can work if everyone involved is able to use it effectively. Because Moqui itself can render wiki pages and pass through binary attachments you might even consider keeping this in a Moqui component. The main problem with this is that until there is a good wiki application built on Moqui to allow changing the content, this is very difficult for less technical people involved.

For the actual code repository there are various good options and this often depends on personal and organizational preferences. Moqui itself is hosted on GitHub and hosted private repositories on GitHub are very affordable (especially for a small number of repositories). If you do use GitHub it is easy to fork the moqui/moqui repository to maintain your own runtime directory in your private repository while keeping up to date with the changes in the main project code base.

Even if you don't use GitHub a local or hosted git repository is a great way to manage source code for a development project. If you prefer other tools such as Subversion or Mercurial then there is no reason not to use them.

For actual coding purposes you'll need an editor or IDE that supports the following types of files:

- XML (with autocompletion, validation, annotation display, etc)
- Groovy (for script files and scripts embedded in XML files)
- HTML, CSS, and JavaScript
- FreeMarker (FTL)
- Java (optional)

My preferred IDE these days is IntelliJ IDEA from JetBrains. The free Community Edition has excellent XML and Groovy support. For HTML, CSS, JavaScript, and FreeMarker to go beyond a simple text editor you'll have to pay for the Ultimate Edition. I implemented most of Moqui, including the complex FreeMarker macro templates, using the Community

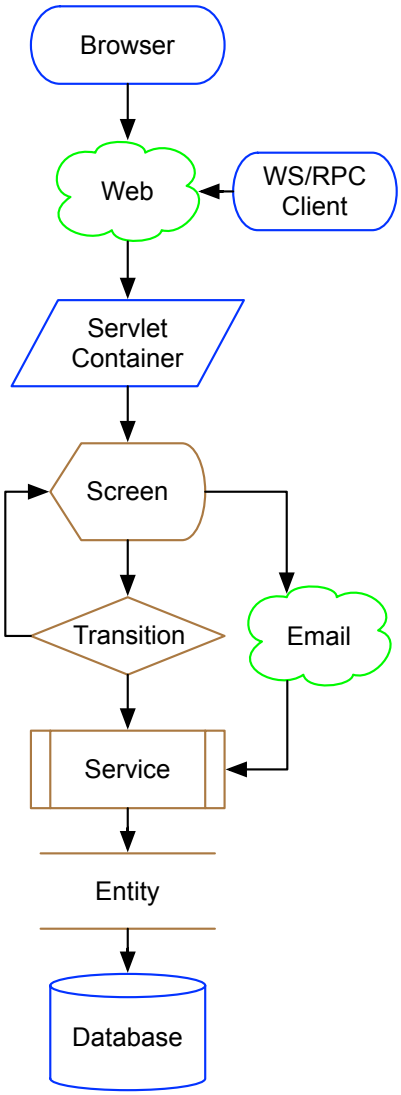
Edition. After breaking down and buying a personal license for the Ultimate Edition I am happy with it, but the Community Edition is impressively capable.

Other popular Java IDEs like Eclipse and NetBeans are also great options and have built-in or plugin functionality to support all of these types of files. I personally prefer having autocomplete and other advanced IDE functionality around, but if you prefer a more simple text editor then of course use what makes you happy and productive.

The Moqui Framework itself is built using Gradle. While I prefer the command line version of Gradle (and Git), most IDEs (including IntelliJ IDEA) include decent user interfaces for these tools that help simplify common tasks.

A Top to Bottom Tour

Web Browser Request



A request from a Web Browser will find its way to the framework by way of the Servlet Container (the default is the embedded Winstone Servlet Container, also works well with Apache Tomcat or any Java Servlet implementation). The Servlet Container finds the requested path on the server in the standard way using the `web.xml` file and will find the MoquiServlet mounted there. The MoquiServlet is quite simple and just sets up an ExecutionContext, then renders the requested Screen.

The screen is rendered based on the configured "root" screen for the webapp, and the subscreens path to get down to the desired target screen. Beyond the path to the target screen there may be a transition name for a transition of that screen.

A transition is part of a screen definition and is used to go one from screen to another (or back to the same). Transitions are used to process input (not to prepare data for presentation), which is separated from the screen actions which are used to prepare data for presentation (not to process input).

If there is a transition name in the URL path the service or actions of the transition will be run, a response to the transition selected (based on conditions and whether there was an error), and then the response will be followed, usually to another screen.

When a service is called (often from a transition or screen action) the Service Facade validates and cleans up the input parameters to the service call using the defined input parameters on the service definition, and then calls the defined inline or external script, Java method, auto or implicit entity operation, or remote service.

Entity operations, which interact with the database, should only be called from services for write operations and can be called from actions anywhere for read operations (transition or screen actions, service scripts/methods, etc).

Web Service Call

Web Service requests generally follow the same path as a form submission request from a web browser that is handled by a Screen Transition. The incoming data will be handled by the transition actions, and typically the response will be handled by an action that sends back the encoded response (in XML, JSON, etc) and the default-response for the transition will be of type "none" so that no screen is rendered and no redirecting to a screen is done.

Incoming and Outgoing Email

Incoming email is handled through Email ECA rules which are called by the **`pollEmailServer`** service (configured using the `EmailServer` entity). These rules have information about the email received parsed and available to them in structured Maps. If the condition of a rule passes, then the actions of the rule will be run. Rules can be written to do anything you would like, typically saving the message somewhere, adding it to a queue for review based on content, generating an automated response, and so on.

Outgoing email is most easily done with a call to the **`sendEmailTemplate`** service. This service uses the passed in `emailTemplateId` to lookup an `EmailTemplate` record that has settings for the email to render, including the subject, the from address, the XML Screen to render and use for the email body, screens or templates to render and attach, and various other options. This is meant to be used for all sorts of emails, especially notification messages and system-managed communication like customer service replies and such.

2. Running Moqui

Download Moqui and Required Software

The only required software for the default configuration of Moqui Framework is the Java SE JDK version 7 or later. The Oracle Java SE downloads are generally the best option:

http://www.oracle.com/technetwork/java/javase/downloads

To build the framework from source you'll need Gradle (http://www.gradle.org) version 1.6 or later. Note that Gradle often has non-backward compatible changes so much more recent versions may not work.

You can download Moqui Framework releases from GitHub at:

`https://github.com/moqui/moqui/releases`

The most recent version is first one on the page. You may choose either the binary or source distribution archive. The binary release of the framework is named "`moqui-<version>.zip`" and there are links to download source archives.

The Moqui Framework source is available on GitHub for download and online browsing here:

`https://github.com/moqui/moqui`

Similarly the Mantle Business Artifacts are available on GitHub here:

`https://github.com/moqui/mantle`

There is also a releases page for Mantle on GitHub.

The Runtime Directory and Moqui Conf XML File

The Moqui Framework has three main parts to deploy:

- Executable WAR File (see below)
- Runtime Directory
- Moqui Configuration XML File

However you use the executable WAR file, you must have a runtime directory and you may override default settings (in the `MoquiDefaultConf.xml` file) with a Moqui Conf XML file, such as the `MoquiProductionConf.xml` file in the `runtime/conf` directory.

The runtime directory is the main place to put components you want to load, the root files (root screen) for the web application, and general configuration files. It is also where the framework will put log files, Derby db files (if you are using Derby), etc. You will eventually want to create your own runtime directory and keep it in your own source repository. You can use the default project runtime directory as a starting point for your own project's runtime resources.

When running specify these two properties:

`moqui.runtime`	Runtime directory (defaults to "./runtime" if exists or just "." if there is no runtime sub-directory)
`moqui.conf`	Moqui Conf XML file (URL or path relative to moqui.runtime)

There are two ways to specify these two properties:

- `MoquiInit.properties` file on the classpath
- System properties specified on the command line (with java -D arguments)

The Executable WAR File

Yep, that's right: an executable WAR file. The main things you can do with this (with example commands to demonstrate, modify as needed):

Load Data	`$ java -jar moqui-<version>.war -load`
Run embedded web server	`$ java -jar moqui-<version>.war`
Deploy as WAR (in Tomcat, etc)	`$ cp moqui-<version>.war ../tomcat/webapps`
Display settings and help	`$ java -jar moqui-<version>.war -help`

When running the data loader (with the -load argument), the following options are available as additional parameters:

`-types=<type>[,<type>]`	Data types to load, matches the `entity-facade-xml.type` attribute (can be anything, common are: `seed`, `seed-initial`, `demo`, ...)

-location=<location>	Location of a single data file to load
-timeout=<seconds>	Transaction timeout for each file, defaults to 600 seconds (10 minutes)
-dummy-fks	Use dummy foreign-keys to avoid referential integrity errors
-use-try-insert	Try insert and update on error instead of checking for record first
-tenantId=<tenantId>	ID for the Tenant to load the data into

Note that If no `-types` or `-location` argument is used all known data files of all types will be loaded.

The examples above show running with the `moqui.runtime` and `moqui.conf` values coming from the `MoquiInit.properties` file on the classpath. To specify these parameters on the command line, use something like:

```
$ java -Dmoqui.conf=conf/MoquiStagingConf.xml -jar moqui-<version>.war
```

Note that the `moqui.conf` path is relative to the `moqui.runtime` directory, or in other words the file lives under the runtime directory.

When running the embedded web server (without the `-load` or `-help` parameters) the Winstone Servlet Container is used. For a full list of arguments available in Winstone, see:

http://winstone.sourceforge.net/#commandLine

For your convenience here are some of the more common Winstone arguments to use:

--httpPort	set the http listening port. -1 to disable, Default is 8080
--httpListenAddress	set the http listening address. Default is all interfaces
--httpsPort	set the https listening port. -1 to disable, Default is disabled
--ajp13Port	set the ajp13 listening port. -1 to disable, Default is 8009
--controlPort	set the shutdown/control port. -1 to disable, Default disabled

Embedding the Runtime Directory in the WAR File

Moqui can run with an external runtime directory (independent of the WAR file), or with the runtime directory embedded in the WAR file. The embedded approach is especially helpful

when deploying to WAR hosting providers like Amazon ElasticBeanstalk. To create a WAR file with an embedded runtime directory:

1. Add components and other resources as needed to the runtime directory
2. Change ${moqui.home}/MoquiInit.properties with desired settings
3. Change Moqui conf file (runtime/conf/Moqui*Conf.xml) as needed
4. Create a derived WAR file based on the moqui.war file and with your runtime directory contents and MoquiInit.properties file with one of:
   a. `$ gradle addRuntime`
   b. `$ ant add-runtime`
5. Copy the created WAR file (moqui-plus-runtime.war) to deployment target
6. Run server (or restart/refresh to deploy live WAR)

The resulting WAR file will have the runtime directory under its root directory (a sibling to the standard WEB-INF directory) and all JAR files under the `WEB-INF/lib` directory.

Building Moqui Framework

Moqui Framework uses the build automation tool Gradle (http://www.gradle.org) for building from source. There are various custom tasks to automate frequent things, but most work is done with the built-in tasks from Gradle. There is also an Ant build file for a few common tasks, but not for building from source.

Build JAR, WAR	`$ gradle build`	
Load All Data	`$ gradle load`	`$ ant load`
Run Server in dev mode	`$ gradle run`	`$ ant run`
Clean up JARs, WAR	`$ gradle clean`	
Clean up ALL built and runtime files (logs, DBs, etc)	`$ gradle cleanAll`	

Note that in Gradle the load and run tasks depend on the build task. With this dependency the easiest to get a new development system running with a populated database is:

```
$ gradle load run
```

This will build the WAR file, run the data loader, then run the server. To stop it just press `<ctrl-c>` (or your preferred alternative).

Database Configuration

Database (or datasource) setup is done in the Moqui Conf XML file with `moqui-conf.entity-facade.datasource` elements. There is one element for each entity group and the `datasource.group-name` attribute matches against `entity.group-name` attribute. By default in Moqui there are 4 entity groups: `transactional`, `analytical`, `nosql`, and `tenantcommon`. If you only configure a `datasource` for the `transactional` group it will also be used for the other groups. One exception to this: if you want to use multiple tenants in your deployment you must also define a `datasource` for `tenantcommon`.

Here is the default configuration for the Apache Derby database:

```
<datasource group-name="transactional" database-conf-name="derby"
    schema-name="MOQUI">
  <inline-jdbc pool-minsize="5" pool-maxsize="50">
    <xa-properties databaseName="${moqui.runtime}/db/derby/MoquiDEFAULT"
        createDatabase="create"/>
  </inline-jdbc>
</datasource>
```

The `database-conf-name` attribute points to a database configuration and matches against a `database-list.database.name` attribute to identify which. Database configurations specify things like SQL types to use, SQL syntax options, and JDBC driver details.

This example uses a `xa-properties` element to use the XA (transaction aware) interfaces in the JDBC driver. The attributes on the element are specific to each JDBC driver. Some examples for reference are included in the `MoquiDefaultConf.xml` file, but for a full list of options look at the documentation for the JDBC driver.

Here is an example of a non-XA configuration for MySQL:

```
<datasource group-name="transactional" database-conf-name="mysql"
    schema-name="">
  <inline-jdbc jdbc-uri="jdbc:mysql://127.0.0.1:3306/MoquiDEFAULT?
autoReconnect=true&amp;useUnicode=true&amp;characterEncoding=UTF-8"
      jdbc-username="moqui" jdbc-password="moqui"
      pool-minsize="2" pool-maxsize="50"/>
</datasource>
```

For non-XA configurations the various `jdbc-*` attributes are on the `inline-jdbc` element as opposed to a subelement. This example shows the main ones needed: the JDBC URI, username, and password. To use something like this put the `datasource` element under the `entity-facade` element in the runtime Moqui Conf XML file (like the `MoquiProductionConf.xml` file).

This book sponsored by **Ant Websystems** (http://www.antwebsystems.com/)

3. Framework Tools and Configuration

What follows is a summary of the various tools in the Moqui Framework and corresponding configuration elements in the Moqui Conf XML file. The default settings are in the `MoquiDefaultConf.xml` file, which is included in the executable WAR file in a binary distribution of Moqui Framework. This is a great file to look at to see some of the settings that are available and what they are set to by default. If you downloaded a binary distribution of Moqui Framework you can view this file online at (note that this is from the master branch on GitHub and may differ slightly from the one you downloaded):

https://github.com/moqui/moqui/blob/master/framework/src/main/resources/MoquiDefaultConf.xml

Any setting in this file can be overridden in the Moqui Conf XML file that is specified at runtime along with the runtime directory (and generally in the `conf` directory under the `runtime` directory). The two files are merged before any settings are used, with the runtime file overriding the default one. Because of this, one easy way to change settings is simply copy from the default conf file and paste into the runtime one, and then make changes as desired.

Execution Context and Web Facade

The Execution Context is the central object in the Moqui Framework API. This object maintains state within the context of a single server interaction such as a web screen request or remote service call. Through the `ExecutionContext` object you have access to a number of "facades" that are used to access the functionality of different parts of the framework. There is detail below about each of these facades.

The main state tracked by the Execution Context is the variable space, or "context", used for screens, actions, services, scripts, and even entity and other operations. This context is a hash or map with name/value entries. It is implemented with the `ContextStack` class and supports protected variable spaces with **`push()`** and **`pop()`** methods that turn it into a stack

of maps. As different artifacts are executed they automatically `push()` the context before writing to it, and then `pop()` the context to restore its state before finishing. Writing to the context always puts the values into the top of the stack, but when reading the named value is searched for at each level on the stack starting at the top so that fields in deeper levels are visible.

In some cases, such as calling a service, we want a fresh context to better isolate the artifact from whatever called it. For this we use the `pushContext()` method to get a fresh context, then the `popContext()` method after the artifact is run to restore the original context.

The context is the literal variable space for the executing artifact wherever possible. In screens when XML actions are executed the results go in the local context. Even Groovy scripts embedded in service and screen actions share a variable space and so variables declared exist in the context for subsequent artifacts.

Some common expressions you'll see in Moqui-based code (using Groovy syntax) include:

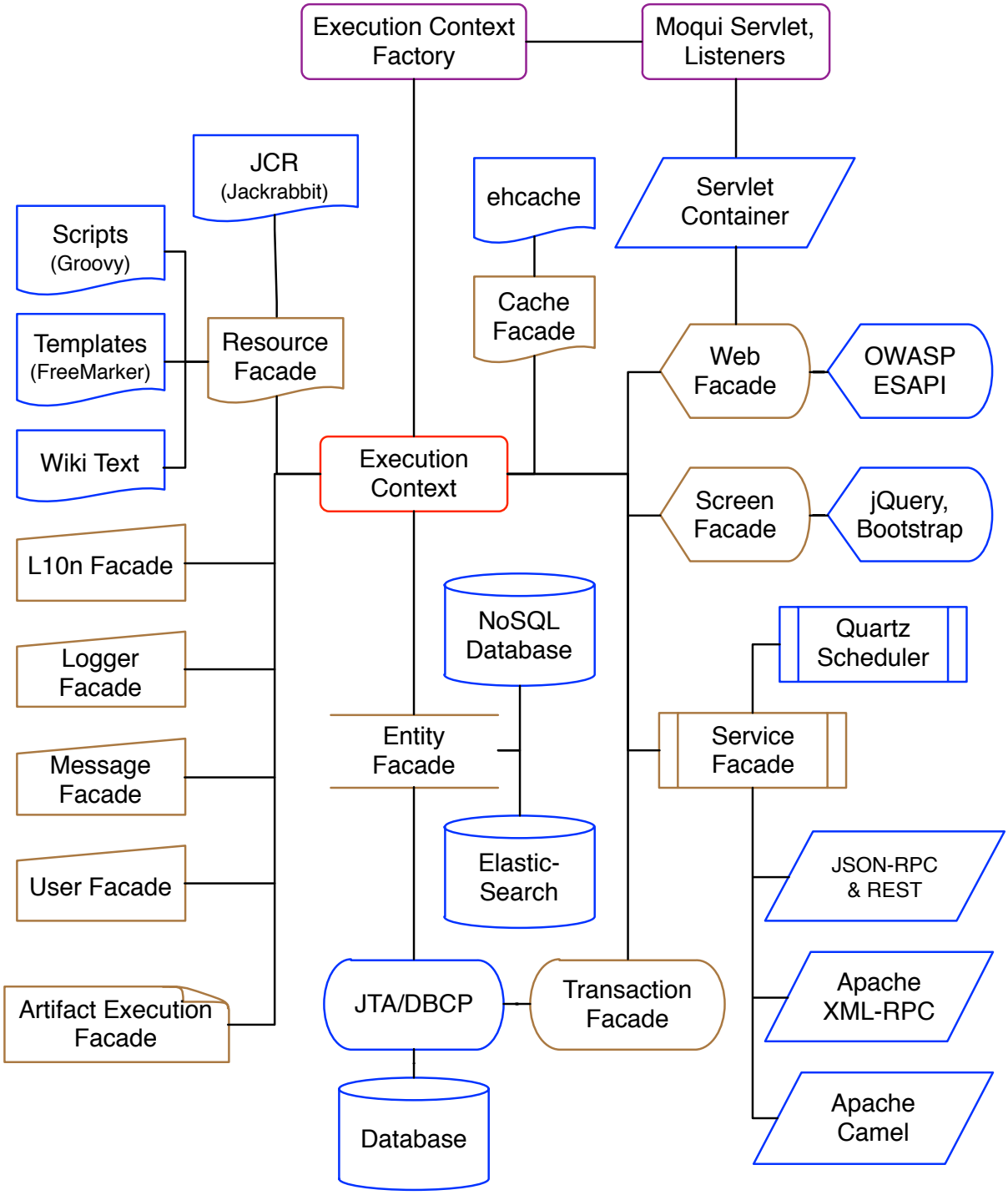
- refer to the current variable context: `ec.context`
- refer to the "exampleId" field from the context: `ec.context.exampleId`
- set the exampleId to "foo": `ec.context.exampleId = "foo"`
- for inline scripts you can also just do: `exampleId = "foo"`

For an `ExecutionContext` instance created as part of a web request (`HttpServletRequest`) there will be a special facade called the Web Facade. This facade is used to access information about the servlet environment for the context including request, response, session, and application (`ServletContext`). It is also used to access the state (attributes) of these various parts of the servlet environment including request parameters, request attributes, session attributes, and application attributes.

Web Parameters Map

The request parameters "map" (`ec.web.requestParameters`) is a special map that contains parameters from the URL parameter string, inline URL parameters (using the "`/~name=value/`" format), and multi-part form submission parameters (when applicable). There is also a special parameters map (`ec.web.parameters`) that combines all the other maps in the following order (with later overriding earlier): request parameters, application attributes, session attributes, and request attributes. That parameters map is a stack of maps just like the context so if you write to it the values will go in the top of the stack which is the request attributes.

For security reasons the request parameters map is canonicalized and filtered using the OWASP ESAPI library. This and the Service Facade validation help to protect agains XSS and injection attacks.


Execution Context Factory
Moqui Servlet, Listeners
JCR (Jackrabbit)
ehcache
Servlet Container
Scripts (Groovy)
Cache Facade
Templates (FreeMarker)
Resource Facade
Web Facade
OWASP ESAPI
Wiki Text
Execution Context
Screen Facade
jQuery, Bootstrap
L10n Facade
NoSQL Database
Quartz Scheduler
Logger Facade
Entity Facade
Service Facade
Message Facade
Elastic-Search
JSON-RPC & REST
User Facade
Apache XML-RPC
Artifact Execution Facade
JTA/DBCP
Transaction Facade
Database
Apache Camel

Factory, Servlet & Listeners

Execution Context instances are created by the Execution Context Factory. This can be done directly by your code when needed, but is usually done by a container that Moqui Framework is running in.

The most common way to run Moqui Framework is as a webapp through either a WAR file deployed in a servlet container or app server, or by running the executable WAR file and using the embedded Winstone Servlet Container. In either case the Moqui root webapp is loaded and the `WEB-INF/web.xml` file tells the servlet container to load the `MoquiServlet`, the `MoquiSessionListener`, and the `MoquiContextListener`. These are default classes included in the framework, and you can certainly create your own if you want to change the lifecycle of the `ExecutionContextFactory` and `ExecutionContext`.

With these default classes the `ExecutionContextFactory` is created by the `MoquiContextListener` on the **`contextInitialized()`** event, and is destroyed by the same class on the **`contextDestroyed()`** event. The `ExecutionContext` is created using the factory by the `MoquiServlet` for each request in the **`doGet()`** and **`doPost()`** methods, and is destroyed by the `MoquiServlet` at the end of each request by the same method.

Resource and Cache Facades

The Resource Facade is used to access and execute resource such as scripts, templates, and content. The Cache Facade is used to do general operations on caches, and to get a reference to a cache as an implementation of the `Cache` interface. Along with supporting basic get/put/remove/etc operations you can get statistics for each cache, and modify cache properties such as timeouts, size limit, and eviction algorithm. The default Cache Facade implementation is just a wrapper around ehcache, and beyond the cache-facade configuration in the Moqui Conf XML file you can configure additional options using the `ehcache.xml` file.

The Resource Facade uses the Cache Facade to cache plain text by its source location (for **`getLocationText()`** method), compiled Groovy and XML Actions scripts by their locations (for the runScriptInCurrentContext method), and compiled FreeMarker (FTL) templates also by location (for the **`renderTemplateInCurrentContext()`** method).

There is also a cache used for the small Groovy expressions that are scattered throughout XML Screen and Form definitions, and that cache is keyed by the actual text of the expression instead of by a location that it came from (for the **`evaluateCondition()`**, **`evaluateContextField()`**, and **`evaluateStringExpand()`** methods).

For more generic access to resources the **`getLocationReference()`** method returns an implementation of the `ResourceReference` interface. This can be used to read resource contents (for files and directories), and get information about them such as content/MIME type, last modified time, and whether it exists. These resource references are used by the rest

of the framework to access resources in a generic and extensible way. Implementations of the `ResourceReference` interface can be implemented as needed and default implementations exist for the following protocols/schemes: http, https, file, ftp, jar, classpath, component, and content (JCR, i.e., Apache Jackrabbit).

Screen Facade

The API of the Screen Facade is deceptively simple, mostly just acting as a factory for the ScreenRender interface implementation. Through the ScreenRender interface you can render screens in a variety of contexts, the most common being in a service with no dependence on a servlet container, or in response to a `HttpServletRequest` using the `ScreenRender.render(request, response)` convenience method.

Generally when rendering and a screen you will specify the root screen location, and optionally a subscreen path to specify which subscreens should be rendered (if the root screen has subscreens, and instead of the default-item for each screen with subscreens). For web requests this sub-screen path is simply the request "pathInfo" (the remainder of the URL path after the location where the webapp/servlet are mounted).

Screen Definition

The real magic of the Screen Facade is in the screen definition XML files. Each screen definition can specify web-settings, parameters, transitions with responses, subscreens, pre-render actions, render-time actions, and widgets. Widgets include subscreens menu/active/panel, sections, container, container-panel, render-mode-specific content (i.e. html, xml, csv, text, xsl-fo, etc), and forms.

There are two types of forms: form-single and form-list. They both have a variety of layout options and support a wide variety of field types. While Screen Forms are primarily defined in Screen XML files, they can also be extended for groups of users with the `DbForm` and related entities.

One important note about forms based on a service (using the `auto-fields-service` element) is that various client-side validations will be added automatically based on the validations defined for the service the form field corresponds to.

Screen/Form Render Templates

The output of the ScreenRender is created by running a template with macros for the various XML elements in screen and form definitions. If a template is specified through the `ScreenRender.macroTemplate()` method then it will be used, otherwise a template will be determined with the **`renderMode`** and the configuration in the `screen-facade.screen-text-output` element of the Moqui Conf XML file. You can create your own templates that

override the default macros, or simply ignore them altogether, and configure them in the Moqui Conf XML file to get any output you want. There is an example of one such template in the `runtime/template/screen-macro/ScreenHtmlMacros.ftl` file, with the override configuration in the `runtime/conf/development/MoquiDevConf.xml` file.

The default HTML screen and form template uses jQuery Core and UI for dynamic client-side interactions. Other JS libraries could be used by modifying the screen HTML macros as described above, and by changing the theme data (defaults in `runtime/component/webroot/data/WebrootThemeData.xml` file) to point to the desired JavaScript and CSS files.

Service Facade

The Service Facade is used to call services through a number of service call interfaces for synchronous, asynchronous, scheduled and special (TX commit/rollback) service calls. Each interface has different methods to build up information about the call you want to do, and they have methods for the name and parameters of the service.

When a service is called the caller doesn't need to know how it is implemented or where it is located. The service definition abstracts that out to the service definition so that those details are part of the implementation of the service, and not the calling of the service.

Service Naming

Service names are composed of 3 parts: path, verb, and noun. When referring to a service these are combined as: "**`${path}.${verb}#${noun}`**", where the hash/pound sign is optional but can be used to make sure the verb and noun match exactly. The path should be a Java package-style path such as `org.moqui.impl.UserServices` for the file at `classpath://service/org/moqui/impl/UserServices.xml`. While it is somewhat inconvenient to specify a path this makes it easier to organize services, find definitions based on a call to the service, and improve performance and caching since the framework can lazy-load service definitions as they are needed.

That service definition file will be found based on that path with location patterns: `"classpath://service/$1"` and `"component://.*/service/$1"` where $1 is the path with '.' changed to '/' and ".xml" appended to the end.

The verb (required) and noun (optional) parts of a service name are separate to better to describe what a service does and what it is acting on. When the service operates on a specific entity the noun should be the name of that entity.

The Service Facade supports CrUD operations based solely on entity definitions. To use these entity-implicit services use a service name with no path, a noun of create, update, or delete, a hash/pound sign, and the name of the entity. For example to update a `UserAccount` use the service name **`update#UserAccount`**. When defining `entity-auto` services the noun must also be the name of the entity, and the Service Facade will use the in- and out-parameters

along with the entity definition to determine what to do (most helpful for create operations with primary/secondary sequenced IDs, etc).

The full service name combined from the examples in the paragraphs above would look like this:

```
org.moqui.impl.UserServices.update#UserAccount
```

Parameter Cleaning, Conversion and Validation

When calling a service you can pass in any parameters you want, and the service caller will clean up the parameters based on the service definition (remove unknown parameters, convert types, etc) and validate parameters based on validation rules in the service definition before putting those parameters in the context for the service to run. When a service runs the parameters will be in the `ec.context` map along with other inherited context values, and will be in a map in the context called parameters to access the parameters segregated from the rest of the context.

One important validation is configured with the `parameter.allow-html` attribute in the service definition. By default no HTML is allowed, and you can use that attribute to allow any HTML or just safe HTML for the service parameter. Safe HTML is determined using the OWASP ESAPI and Antisamy libraries, and configuration for what is considered safe is done in the `antisamy-esapi.xml` file.

Quartz Scheduler

The Service Facade uses Quartz Scheduler for asynchronous and scheduled service calls. Some options are available when calling the services and configuration in the Moqui Conf XML file, but to configure Quartz itself use the `quartz.properties` file (there is a default in the `framework/src/main/resources/` directory that may be overridden on the classpath).

Web Services

For web services the Service Facade uses Apache XML-RPC for incoming and outgoing XML-RPC service calls, and custom code using Moqui JSON and web request tools for incoming and outgoing JSON-RPC 2.0 calls. The outgoing calls are handled by the `RemoteXmlRpcServiceRunner` and `RemoteJsonRpcServiceRunner` classes, which are configured in the `service-facade.service-type` element in the Moqui Conf XML file. To add support for other outgoing service calls through the Service Facade implement the `ServiceRunner` interface (as those two classes do) and add a `service-facade.service-type` element for it.

Incoming web services are handled using default transitions defined in the `runtime/component/webroot/screen/webroot/rpc.xml` screen. The remote URL for these, if

`webroot.xml` is mounted on the root ("/") of the server, would be something like: "`http://hostname/rpc/xml`" or "`http://hostname/rpc/json`". To handle other types of incoming services similar screen transitions can be added to the `rpc.xml` screen, or to any other screen.

For REST style services a screen transition can be declared with a HTTP request method (get, put, etc) as well as a name to match against the incoming URL. For more flexible support of parameters in the URL beyond the transition's place in the URL path values following the transition can be configured to be treated the same as named parameters. To make things easier for JSON payloads they are also automatically mapped to parameters and can be treated just like parameters from any other source, allowing for easily reusable server-side code. To handle these REST service transitions an internal service can be called with very little configuration, providing for an efficient mapping between exposed REST services and internal services.

Entity Facade

The Entity Facade is used for common database interactions including create/update/delete and find operations, and for more specialized operations such as loading and creating entity XML data files. While these operations are versatile and cover most of the database interactions needed in typical applications, sometimes you need lower-level access, and you can get a JDBC Connection object from the Entity Facade that is based on the `entity-facade` datasource configuration in the Moqui Conf XML file.

Entities correspond to tables in a database and are defined primarily in XML files. These definitions include list the fields on the entity, relationships betweens entities, special indexes, and so on. Entities can be extended using database record with the `UserField` and related entities.

Each individual record is represented by an instance of the `EntityValue` interface. This interface extends the Map interface for convenience, and has additional methods for getting special sets of values such as the primary key values. It also has methods for database interactions for that specific record including create, update, delete, and refresh, and for getting setting primary/secondary sequenced IDs, and for finding related records based on relationships in the entity definition. To create a new `EntityValue` object use the `EntityFacade.makeValue()` method, though most often you'll get `EntityValue` instances through a find operation.

To find entity records use the `EntityFind` interface. To get an instance of this interface use the `EntityFacade.makeFind()` method. This find interface allows you to set various conditions for the find (both where and having, more convenience methods for where), specify fields to select and order by, set offset and limit values, and flags including use cache, for update, and distinct. Once options are set you can call methods to do the actual find including: `one()`, `list()`, `iterator()`, `count()`, `updateAll()`, and `deleteAll()`.

Multi-Tenant

When getting an `EntityFacade` instance from the `ExecutionContext` the instance retrieved will be for the active tenantId on the `ExecutionContext` (which is set before authentication either specified by the user, or set by the servlet or a listener before the request is processed). If there is no `tenantId` the `EntityFacade` will be for the `"DEFAULT"` tenant and use the settings from the Moqui Conf XML file. Otherwise it will use the active `tenantId` to look up settings on the `Tenant*` entities that will override the defaults in the Moqui Conf XML file for the datasource.

Connection Pool and Database

The Entity Facade uses Atomikos TransactionsEssentials or Bitronix BTM (default) for XA-aware database connection pooling. To configure Atomikos use the `jta.properties` file. To configure Bitronix use the `bitronix-default-config.properties` file. With configuration in the `entity-facade` element of the Moqui Conf XML file you can change this to use any `DataSource` or `XADataSource` in JNDI instead.

The default database included with Moqui Framework is Apache Derby. This is easy to change with configuration in the `entity-facade` element of the Moqui Conf XML file. To add a database not yet supported in the `MoquiDefaultConf.xml` file, add a new `database-list.database` element. Currently databases supported by default include Apache Derby, DB2, HSQL, MySQL, Postgres, Oracle, and MS SQL Server.

Database Meta-Data

The first time (in each run of Moqui) the Entity Facade does a database operation on an entity it will check to see if the table for that entity exists (unless configured not to). You can also configure it to check the tables for all entities on startup. If a table does not exist it will create the table, indexes, and foreign keys (for related tables that already exist) based on the entity definition. If a table for the entity does exist it will check the columns and add any that are missing, and can do the same for indexes and foreign keys.

Transaction Facade

Transactions are used mostly for services and screens. Service definitions have transaction settings, based on those the service callers will pause/resume and begin/commit/rollback transactions as needed. For screens a transaction is always begun for transitions (if one is not already in place), and for rendering actual screens a transaction is only begun if the screen is setup to do so (mostly for performance reasons).

You can also use the `TransactionFacade` for manual transaction demarcation. The JavaDoc comments have some code examples with recommended patterns for begin/commit/rollback and for pause/begin/ commit/rollback/resume to use try/catch/finally clauses to make sure the transaction is managed properly.

When debugging transaction problems, such as tracking down where a rollback-only was set, the `TransactionFacade` can also be use as it keeps a stack trace when `setRollbackOnly()` is called. It will automatically log this on later errors, and you can manually get those values at other times too.

Transaction Manager (JTA)

By default the Transaction Facade uses the Bitronix TM library (also used for a connection pool by the Entity Facade). To configure Bitronix use the `bitronix-default-config.properties` file. Moqui also supports Atomikos OOTB. To configure Atomikos use the `jta.properties` file.

Any JTA transaction manager, such as one from an application server, can be used instead through JNDI by configuring the locations of the `UserTransaction` and `TransactionManager` implementations in the `entity-facade` element of the Moqui Conf XML file.

Artifact Execution Facade

The Artifact Execution Facade is called by other facades to keep track of which artifacts are "run" in the life of the `ExecutionContext`. It keeps both a history of all artifacts, and a stack of the current artifacts being run. For example if a screen calls a subscreen and that calls a service which does a find on an entity the stack will have (bottom to top) the first screen, then the second screen, then the service and then the entity.

Artifact Authorization

While useful for debugging and satisfying curiosity, the main purpose for keeping track of the stack of artifacts is for authorization and permissions. There are implicit permissions for screens, transitions, services and entities in Moqui Framework. Others may be added later, but these are the most important and the one supported for version 1.0 (see the `"ArtifactType" Enumeration` records in the `SecurityTypeData.xml` file for details).

The `ArtifactAuthz*` and `ArtifactGroup*` entities are used to configure authorization for users (or groups of users) to access specific artifacts. To simplify configuration authorization can be "inheritable" meaning that not only is the specific artifact authorized but also everything that it uses.

There are various examples of setting up different authorization patterns in the `ExampleSecurityData.xml` file. One common authorization pattern is to allow access to a screen and all of its subscreens where the screen is a higher-level screen such as the `ExampleApp.xml` screen that is the root screen for the example app. Another common pattern is that only a certain screen within an application is authorized but the rest of it is not. If a subscreen is authorized, even if its parent screen is not, the user will be able to use that subscreen.

Artifact Hit Tracking

There is also functionality to track performance data for artifact "hits". This is done by the Execution Context Factory instead of the Artifact Execution Facade because the Artifact Execution Facade is created for each Execution Context, and the artifact hit performance data needs to be tracked across a large number of artifact hits both concurrent and over a period of time. The data for artifact hits is persisted in the `ArtifactHit` and `ArtifactHitBin` entities. The `ArtifactHit` records are associated with the `Visit` record (one visit for each web session) so you can see a history of hits within a visit for auditing, user experience review, and various other purposes.

User, L10n, Message, and Logger Facades

The User Facade is used to manage information about the current user and visit, and for login, authentication, and logout. User information includes locale, time zone, and currency. There is also the option to set an effective date/time for the user that the system will treat as the current date/time (through `ec.user.nowTimestamp`) instead of using the current system date/time.

The L10n (Localization) Facade uses the locale from the User Facade and localizes the message it receives using cached data from the `LocalizedMessage` entity. The `EntityFacade` also does localization of entity fields using the `LocalizedEntityField` entity. The L10n Facade also has methods for formatting currency amounts, and for parsing and formatting for `Number`, `Timestamp`, `Date`, `Time`, and `Calendar` objects using the `Locale` and `TimeZone` from the User Facade as needed.

The Message Facade is used to track messages and error messages for the user. The error message list (ec.message.errors) is also used to determine if there was an error in a service call or other action.

The Logger Facade is used to log information to the system log. This is meant for use in scripts and other generic logging. For more accurate and trackable logging code should use the SLF4J Logger class (`org.slf4j.Logger`) directly. The JavaDoc comments in the `LoggerFacade` interface include example code for doing this.

Extensions and Add-ons

The Compelling Component

A Moqui Framework component is a set of artifacts that make up an application built on Moqui, or reusable artifacts meant to be used by other components such as the `mantle-udm` and `mantle-usl` components, a theme component, or a component that integrates some other tool or library with Moqui Framework to extend the potential range of applications based on Moqui.

Component Directory Structure

The structure of a component is driven by convention as opposed to configuration. This means that you must use these particular directory names, and that all Moqui components you look at will be structured in the same way.

- **`data`** - Entity XML data files with root element `entity-facade-xml`, loaded by **`type`** attribute matching types specified on command line (executable WAR with `-load`), or all types if no type specified
- **`entity`** - All Entity Definition and Entity ECA XML files in this directory will be loaded; Entity ECA files must be in this directory and have the dual extension "`.eecas.xml`"
- **`lib`** - JAR files in this directory will be added to the classpath when the webapp is deployed
- **`screen`** - Screens are referenced explicitly (usually by "`component://*`" URL), so this is a convention
- **`script`** - Scripts are referenced explicitly (usually by "`component://*`" URL), so this is a convention; Groovy, XML Action, and any other scripts should go under this directory
- **`service`** - Services are loaded by path to the Service Definition XML file they are defined in, and those paths are found either under these component service directories or under "`classpath://service/`"; Service ECA files must be in this directory and have the dual extension "`.secas.xml`"; Email ECA files must be in this directory and have the extension "`.emecas.xml`"

Installing a Component

Load the Component

There are two ways to tell Moqui about a component:

- put the component directory in the runtime/component directory
- add a component-list.component element in the Moqui Conf XML file

Mounting Screen(s)

Each webapp in Moqui (including the default webroot webapp) must have a root screen specified in the `moqui-conf.webapp-list.webapp.root-screen-location` attribute. The default root screen is called webroot which is located at `runtime/component/webroot/screen/webroot.xml`.

For screens from your component to be available in a screen path under the `webroot` screen you need to make each top-level screen in your component (i.e. each screen in the component's screen directory) a subscreen of another screen that is an ancestor of the `webroot` screen. There are two ways to do this (this does not include putting it in the `webroot` directory as an implicit subscreen since that is not an option for screens defined elsewhere):

- add a `screen.subscreens.subscreen-item` element to the parent screen (what the subscreen will be under); for example see the apps screen (`runtime/component/webroot/screen/WebRoot/apps.xml`) where the example and tools root screens are "mounted"
- add a record in the `SubscreensItem` entity, specifying the parent screen in the **`screenLocation`** field, the subscreen in the **`subscreenLocation`** field, the "mount point" in the **`subscreenName`** field (equivalent to the `subscreens-item.name` attribute), and either `ALL_USERS` in the **`userGroupId`** field for it to apply to all users, or an actual `userGroupId` for it to apply to just that user group

If you want your screen to use its own decoration and be independent from other screens, put it under the `webroot` screen directly. To have your screen part of the default apps menu structure and be decorated with the default apps decoration, put it under the `apps` screen.

Moqui Conf XML File Settings

You may want have things in your component add to or modify various things that come by default with Moqui Framework, including:

- **Resource Reference**: see the `moqui-conf.resource-facade.resource-reference` element
- **Template Renderer**: see the `moqui-conf.resource-facade.template-renderer` element
- **Screen Text Output Template**: see the `moqui-conf.screen-facade.screen-text-output` element
- **Service Type Runner**: see the `moqui-conf.service-facade.service-type` element
- Explicit **Entity Data and Definition** files: see the `moqui-conf.entity-facade.load-entity` and `moqui-conf.entity-facade.load-data` elements

There are examples of all of these in the `MoquiDefaultConf.xml` file since the framework uses the Moqui Conf XML file for its own default configuration.

4. Create Your First Component

Summary

This chapter is a step-by-step guide to creating and running your own Moqui component with a user interface, logic, and database interaction.

- **Part 1**: To get started you'll be creating your own component and a simple "Hello world!" screen.
- **Part 2**: Continuing from there you'll define your own entity (database table) and add forms to your screen to find and create records for that entity.
- **Part 3**: To finish off the fun you will create some custom logic instead of using the default CrUD logic performed by the framework based on the entity definition.

The running approach used in this document is a simple one using the embedded servlet container and database.

The tutorial code from this chapter is available on moqui.org at:

http://www.moqui.org/tutorial.zip

Part 1

Download Moqui Framework

If you haven't already downloaded Moqui Framework, do that now. You should have a `moqui-<version>` directory with at least the `moqui-<version>.war` file and the default `runtime` directory that comes with Moqui. Start out in that moqui root directory.

If you have a clean download, do a data load and try running it real quick:

```
$ gradle load
$ gradle run
```

In your browser go to `http://localhost:8080/`, log in as John Doe with the button in the lower-left corner of the screen, and look around a bit.

Now quit (<ctrl>-c in the command line) and you're ready for the next step.

Create a Component

Moqui follows the "convention over code" principle for components, so all you really have to do to create a Moqui component is create a directory:

```
$ cd runtime/component
$ mkdir tutorial
```

Now go into the directory and create some of the standard directories that you'll use later in this tutorial:

```
$ cd tutorial
$ mkdir data
$ mkdir entity
$ mkdir screen
$ mkdir script
$ mkdir service
```

With your component in place just start up Moqui (with "`$ gradle run`" or similar).

Add a Screen

Using your preferred IDE or text editor add a screen XML file in:

`runtime/component/tutorial/screen/tutorial.xml`

For now let this be a super simple screen with just a "Hello world!" label in it. The contents should look something like:

```
<screen require-authentication="false">
    <widgets><label type="h1" text="Hello world!"/></widgets>
</screen>
```

Note that the **`require-authentication`** attribute is set to `false`. By default this is `true` and the screen will require authentication and authorization. We'll discuss the artifact-aware configurable authorization later in the Security chapter.

Mount as a Subscreen

To make your screen available it needs to be added as a subscreen to a screen that is already under the root screen somewhere. In Moqui screens the URL path to the screen and the menu structure are both driven by the subscreen hierarchy, so this will setup the URL for the screen and add a menu tab for it.

For the purposes of this tutorial we'll use the existing root screen and header/footer/etc that are in the included runtime directory. This runtime directory has a `webroot` component with the root screen at:

`runtime/component/webroot/screen/webroot.xml`

On a side note, the root screen is specified in the Moqui Conf XML file using the `webapp-list.webapp.root-screen` element, and you can use multiple elements to have different root screens for different host names.

To make the subscreen hierarchy more flexible this root screen only has a basic HTML head and body, with no header and footer content, so let's put our screen under the "apps" screen which adds a header menu and will give our screen some context. Modify the apps screen by changing:

`runtime/component/webroot/screen/webroot/apps.xml`

Add a `subscreens-item` element under the `subscreens` element in the **apps.xml** file like:

```
<subscreens-item name="tutorial" menu-title="Tutorial"
    location="component://tutorial/screen/tutorial.xml"/>
```

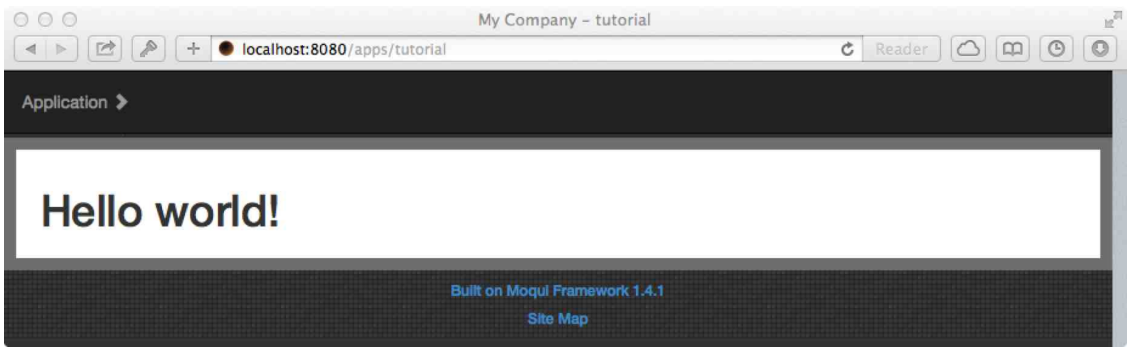
The `name` attribute specifies the value for the path in the URL to the screen, so your screen is now available in your browser at:

`http://localhost:8080/apps/tutorial`

If you don't want to modify an existing screen file and still want to mount your screen as a subscreen of another you can do so with a record in the database that looks like this (in the `entity-facade-xml` format with elements representing entities and attributes representing fields):

```
<SubscreensItem subscreenName="tutorial" userGroupId="ALL_USERS"
    screenLocation="component://webroot/screen/webroot/apps.xml"
    subscreenLocation="component://tutorial/screen/tutorial.xml"
    menuTitle="Tutorial" menuIndex="1" menuInclude="Y"/>
```

Once it's all wired up this is what your screen should look like:



Try Included Content

Instead of using the label element we can get the HTML from a file that is "under" the screen.

First create a simple HTML file located at:

`runtime/component/tutorial/screen/tutorial/hello.html`

The HTML file can contain any HTML, and since this will be included in a screen whose parent screens take care of header/footer/etc we can keep it very simple:

```
<h1>Hello world! (from the hello.html file)</h1>
```

Now just explicitly include the HTML file in the `tutorial.xml` screen definition using the `render-mode.text` element:

```
<screen>
    <widgets>
        <label type="h1" text="Hello world!"/>
        <render-mode>
            <text type="html"
                location="component://tutorial/screen/tutorial/hello.html"/>
        </render-mode>
    </widgets>
</screen>
```

So what is this `render-mode` thingy? Moqui XML Screens are meant to platform agnostic and may be rendered in various environments. Because of this we don't want anything in the screen that is specific to a certain mode of rendering the screen without making it clear that it is. Under the `render-mode` element you can have various sub-elements for different render modes, even for different text modes such as HTML, XML, XSL-FO, CSV, and so on so that a single screen definition can be rendered in different modes and produce output as needed for each mode.

The screen is available at the same URL, but now includes the content from the HTML file instead of having it inline as a label in the screen definition.

Try Sub-Content

Another way to show the contents of the `hello.html` file is to treat it as screen sub-content.

To do this the `hello.html` file must by in a subdirectory with the same name as the screen, i.e. in a `tutorial` directory as a sibling of the `tutorial.xml` file.

Now all we have to do is:

- tell the tutorial.xml screen to include child content by setting the `screen.include-child-content` attribute to `true`
- tell the screen where to include subscreens and child content by adding a `widgets.subscreens-active` element
- specify the default subscreens item as the `hello.html` sub-content with the `subscreens.default-item` attribute

With those done your screen XML file should look like:

```
<screen require-authentication="false" include-child-content="true">
    <subscreens default-item="hello.html"/>
    <widgets>
        <label type="h1" text="Hello world!"/>
        <subscreens-active/>
    </widgets>
</screen>
```

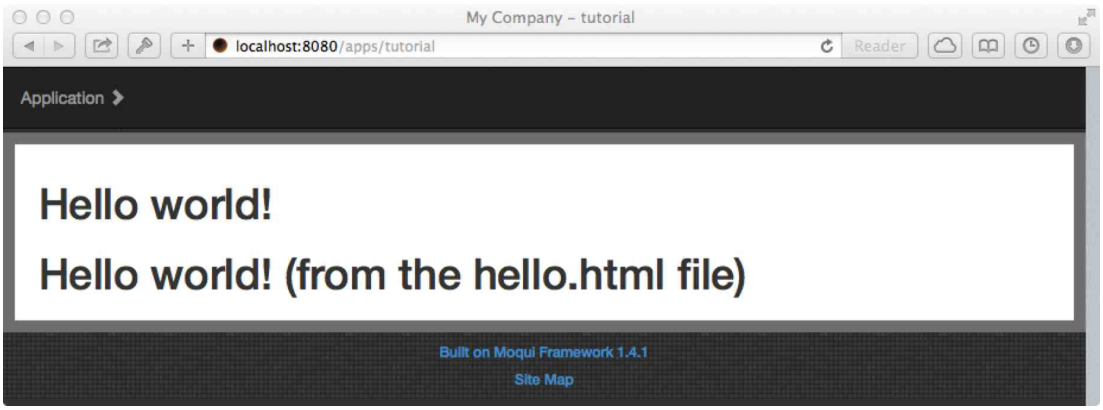
To see the content go to a URL that tells Moqui that you want the `hello.html` file that is under the `tutorial` screen:

`http://localhost:8080/apps/tutorial/hello.html`

With the default subscreens item specified you can also just go to the tutorial screen's URL:

`http://localhost:8080/apps/tutorial`

With this in place this is how your screen should look, with both hello world lines:



Part 2

My First Entity

An entity is a basic tabular data structure, and usually just a table in a database. An entity value is equivalent to a row or record in the database. Moqui does not do object-relational mapping, so all we have to do is define an entity, and then start writing code using the Entity Facade (or other higher level tools) to use it.

To create a simple entity called `Tutorial` with fields **`tutorialId`** and **`description`** create an entity XML file at:

`runtime/component/tutorial/entity/TutorialEntities.xml`

That contains:

```
<entities>
    <entity entity-name="Tutorial" package-name="tutorial">
        <field name="tutorialId" type="id" is-pk="true"/>
        <field name="description" type="text-long"/>
    </entity>
</entities>
```

If you're running Moqui in dev mode the entity definition cache clears automatically so you don't have to restart, and for production mode or if you don't want to wait (since Moqui does start very fast) you can just stop and start the JVM.

How do you create the table? Unless you turn the feature off (in the Moqui Conf XML file) the Entity Facade will create the table the first time the entity is used if it doesn't already exist.

Add Some Data

The Entity Facade has functionality to load data from, and write data to, XML files where elements map to entity names and attributes map to field names.

We'll create a UI to enter data later on, and you can use the Auto Screen or Entity Data UI in the Tools application to work with records in your new entity. Data files are useful for seed data that code depends on, data for testing, and data to demonstrate how a data model should be used. So, let's try it.

Create an entity facade XML file at:

`runtime/component/tutorial/data/TutorialData.xml`

That contains:

```
<entity-facade-xml type="seed">
    <tutorial.Tutorial tutorialId="TestOne"
        description="Test one description."/>
    <tutorial.Tutorial tutorialId="TestTwo"
        description="Test two description."/>
</entity-facade-xml>
```

To load this just run "`$ gradle load`" or one of the other load variations described in the Running Moqui chapter.

Automatic Find Form

Add the XML screen definition below as a subscreen for the `tutorial` screen by putting it in the file:

`runtime/component/tutorial/screen/tutorial/FindTutorial.xml`

```
<screen require-authentication="anonymous-all">
    <transition name="findTutorial">
        <default-response url="."/></transition>
    <actions>
        <entity-find entity-name="tutorial.Tutorial" list="tutorialList">
            <search-form-inputs/></entity-find>
    </actions>
    <widgets>
        <form-list name="ListTutorials" list="tutorialList"
                transition="findTutorial">
            <auto-fields-entity entity-name="tutorial.Tutorial"
                    field-type="find-display"/>
        </form-list>
    </widgets>
</screen>
```

This screen has a few key parts:

- **`transition`** Think of links between screens as an ordered graph where each screen is a node and the transitions defined in each screen are how you go from that screen to another (or back to the same), and as part of that transition possibly run actions or a service.
  - A single `transition` can have multiple responses with conditions and for errors resulting in transition to various screens as needed by your UI design.
  - This particular `transition` refers to the current screen.
- **`actions.entity-find`** There is just one action run when this screen is rendered: an `entity-find`.
  - Normally with an `entity-find` element (or in the Java API an `EntityFind` object) you would specify conditions, fields to order by, and other details about the find to run.
  - In this case we are doing a find on an entity using standard parameters from a XML Form, so we can use the `search-form-inputs` sub-element to handle these automatically.
  - To get an idea of what the parameters should be like just view the HTML source in your browser that is generated by the XML Form.
- **`widgets.form-list`** This is the actual form definition, specifically for a "list" form for multiple records/rows (as opposed to a "single" form).
  - The `name` here can be anything as long as it is unique within the XML Screen.
  - Note that the `list` refers to the result of the `entity-find` in the `actions` block, and the **`transition`** attribute refers to the `transition` element defined at the top of the screen.
  - Since the goal was to have a form automatically defined based on an entity we use the `auto-fields-entity` element with the name of our Tutorial entity, and `"find-display"` option for the **`field-type`** attribute which creates find fields in the header and display fields for each record in the table body.

To view this screen use this URL:

`http://localhost:8080/apps/tutorial/FindTutorial`

An Explicit Field

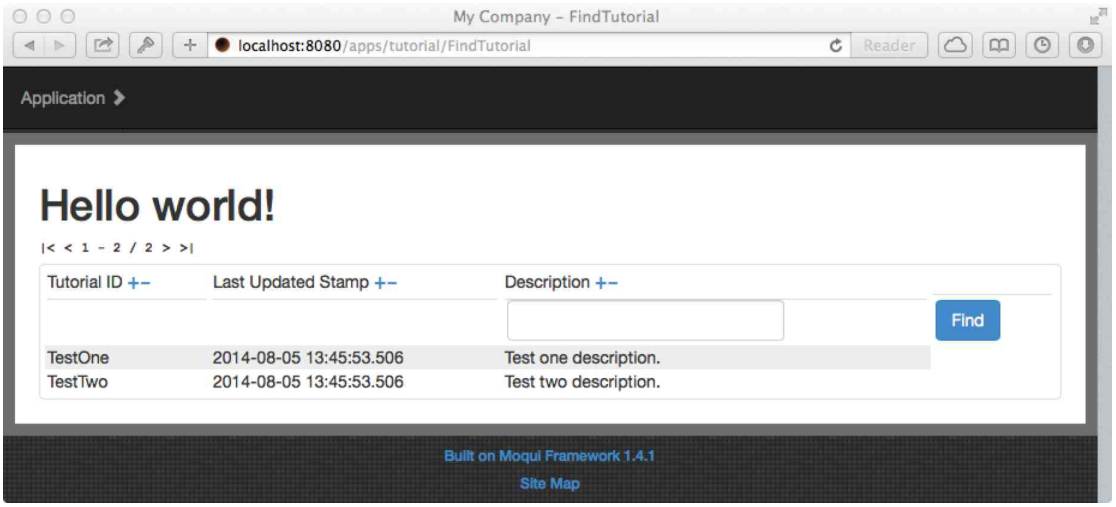
Instead of the default for the description field, what if you wanted to specify how it should look and what type of field it should be?

To do this just add a `field` element inside the `form-list` element, and just after the `auto-fields-entity` element, like this:

```
<form-list name="ListTutorials" list="tutorialList"
        transition="findTutorial">
    <auto-fields-entity entity-name="tutorial.Tutorial"
        field-type="display"/>
    <field name="description">
        <header-field show-order-by="true">
            <text-find hide-options="true"/></header-field>
        <default-field><display/></default-field>
    </field>
    <field name="find">
        <header-field><submit/></header-field>
    </field>
</form-list>
```

Because the field **`name`** attribute is the same as a field already created by the `auto-fields-entity` element it will override that field. If the **`name`** was different an additional field would be created. The result of this is mostly the same as what was automatically generated using the `auto-fields-entity` element, and this is how you would do it explicitly.

With your screen and form defined like this the FindTutorial screen should look something like this:



Add a Create Form

Let's add a button that will pop up a Create Tutorial form, and a transition to process the input.

First add the transition to the `FindTutorial.xml` screen you created before, right next to the `findTutorial` transition:

```
<transition name="createTutorial">
    <service-call name="create#tutorial.Tutorial"/>
    <default-response url="."/>
</transition>
```

This transition just calls the **`create#tutorial.Tutorial`** service, and then goes back to the current screen.

Where did the **`create#tutorial.Tutorial`** service come from? We haven't defined anything like that yet. The Moqui Service Facade supports a special kind of service for entity CrUD operations that don't need to be defined, let alone implemented. This service name consists of two parts, a verb and a noun, separated by a hash (#).

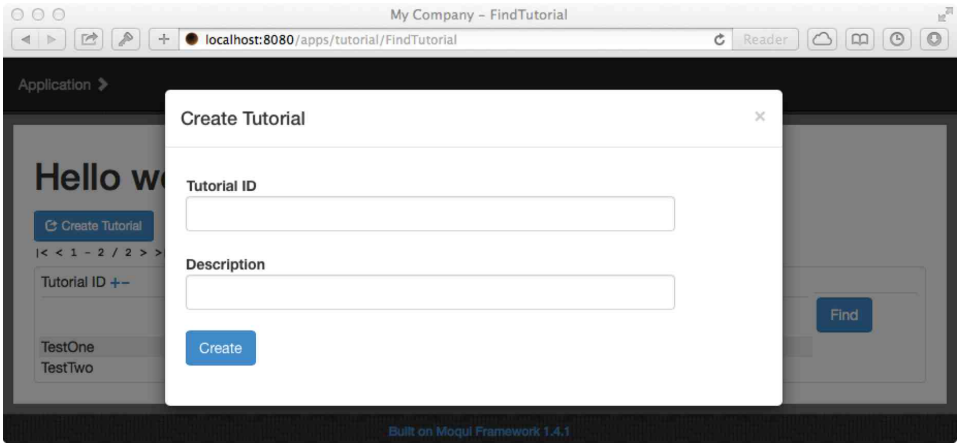
As long as the verb is `create`, `update`, `store`, or `delete` and the noun is a valid entity name the Service Facade will treat it as an implicit entity-auto service and do the desired operation. It does so based on the entity definition and the parameters passed to the service call. For example, with the `create` verb and an entity with a single primary key field if you pass in a value for that field it will use it, otherwise it will automatically sequence a value using the entity name as the sequence key.

Next let's add the create form, in a hidden container that will expand when a button is clicked. Put this inside the `widget` element, just above the `form-list` element in the original `FindTutorial` screen you created before so that it appears above the list form in the screen:

```
<container-dialog id="CreateTutorialDialog" button-text="Create Tutorial">
    <form-single name="CreateTutorial" transition="createTutorial">
        <auto-fields-entity entity-name="tutorial.Tutorial"
            field-type="edit"/>
        <field name="submitButton">
            <default-field title="Create"><submit/></default-field>
        </field>
    </form-single>
</container-dialog>
```

The form definition refers to the `transition` you just added to the screen, and uses the `auto-fields-entity` element with `edit` for the **`field-type`** attribute to generate edit fields. The last little detail is to declare a button to submit the form, and it is ready to go. Try it out and see the records appear in the list form that was part of the original screen.

Here is a screen shot of the create form, and you can see the button added to the find screen in the background:



Part 3

Custom Create Service

The `createTutorial` transition from our screen above used the implicit entity-auto service **`create#tutorial.Tutorial`**. Let's see what it would look like to define and implement a service manually.

First lets define a service and use the automatic entity CrUD implementation. Put the services XML text below in a file in this location:

`runtime/component/tutorial/service/tutorial/TutorialServices.xml`

```
<services>
    <service verb="create" noun="Tutorial" type="entity-auto">
        <in-parameters><auto-parameters include="all"/></in-parameters>
        <out-parameters>
            <auto-parameters include="pk" required="true"/>
        </out-parameters>
    </service>
</services>
```

This will allow all fields of the `Tutorial` entity to be passed in, and will always return the PK field (**`tutorialId`**). Note that with the `auto-parameters` element we are defining the service based on the entity, and if we added fields to the entity they would be automatically represented in the service.

Now change that service definition to add an inline implementation as well. Notice that the `service.type` attribute has changed, and the `actions` element has been added.

```
<service verb="create" noun="Tutorial" type="inline">
    <in-parameters><auto-parameters include="all"/></in-parameters>
    <out-parameters>
        <auto-parameters include="pk" required="true"/>
    </out-parameters>
    <actions>
        <entity-make-value entity-name="tutorial.Tutorial"
            value-field="tutorial"/>
        <entity-set value-field="tutorial" include="all"/>
        <if condition="!tutorial.tutorialId">
            <entity-sequenced-id-primary value-field="tutorial"/>
        </if>
        <entity-create value-field="tutorial"/>
    </actions>
</service>
```

Now to call the service instead of the implicit entity-auto one just change the `transition` to refer to this service:

```
<transition name="createTutorial">
    <service-call name="tutorial.TutorialServices.create#Tutorial"/>
    <default-response url="."/>
</transition>
```

Note that the service name for a defined service like this is like a fully qualified Java class name. It has a "package", in this case `tutorial` which is the directory (possibly multiple directories separated by dots) under the `component/service` directory. Then there is a dot and the equivalent of the class name, in this case "`TutorialServices`" which is the name of the XML file the service is in, but without the `.xml` extension. After that is another dot, and then the service name with the verb and noun optionally separated by a hash (`#`).

Groovy Service

What if you want to implement the service in Groovy (or some other supported scripting language) instead of the inline XML Actions? In that case the service definition would look like this:

```
<service verb="create" noun="Tutorial" type="script"
     location="component://tutorial/script/tutorial/createTutorial.groovy">
    <in-parameters><auto-parameters include="all"/></in-parameters>
    <out-parameters>
        <auto-parameters include="pk" required="true"/>
    </out-parameters>
</service>
```

Notice that the `service.type` attribute has changed to `"script"`, and there is now a `service.location` attribute which specifies the location of the script.

Here is what the script would look like in that location:

```
def tutorial = ec.entity.makeValue("tutorial.Tutorial")
tutorial.setAll(context)
if (!tutorial.tutorialId) tutorial.setSequencedIdPrimary()
tutorial.create()
```

When in Groovy, or other languages, you'll be using the Moqui Java API which is based on the `ExecutionContext` class which is available in the script with the variable name "`ec`". For more details on the API see the API JavaDocs (`http://www.moqui.org/javadoc/index.html`) and specifically the doc for the `ExecutionContext` (`http://www.moqui.org/javadoc/org/moqui/context/ExecutionContext.html`) class which has links to the other major API interface pages.

5. Data and Resources

Resources, Content, Templates, and Scripts

Resource Locations

A Resource Facade location string is structured like a URL with a protocol, host, optional port, and filename. It supports the standard Java URL protocols (http, https, ftp, jar, and file). It also supports some additional useful protocols:

- **`classpath://`** for resources on the Java classpath
- **`content://`** for resources in a content repository (JCR, via Jackrabbit client); the first path element after the protocol prefix is the name of the content repository as specified in the `repository.name` attribute in the Moqui Conf XML file
- **`component://`** for locations relative to a component base location, no matter where the component is located (file system, content repository, etc)
- **`dbresource://`** for a virtual filesystem persisted with the Entity Facade in a database using the `moqui.resource.DbResource` and `DbResourceFile` entities

Additional protocols can be added by implementing the `org.moqui.context.ResourceReference` interface and adding a `resource-facade.resource-reference` element to the Moqui Conf XML file. The supported protocols listed above are configured this way in the MoquiDefaultConf.xml file.

Using Resources

The simplest way to use a resource, and supported by all location protocols, is to read the text or binary content. To get the text from a resource location use the `ec.resource.getLocationText(String location, boolean cache)` method. To get an InputStream for binary or large text resources use the `ec.resource.getLocationStream(String location)` method.

For a wider variety of operations beyond just reading resource data use the `ec.resource.getLocationReference(String location)` method to get an instance of the `org.moqui.context.ResourceReference` interface. This interface has methods to get text

or binary stream data from the resource like the Resource Facade methods. It also has methods for directory resources to get child resources, find child files and/or directories recursively by name, write text or binary stream data, and move the resource to another location.

Rendering Templates and Running Scripts

There is a single method for rendering a template in a resource at a location: `ec.resource.renderTemplateInCurrentContext(String location, Writer writer)`. This method returns nothing and simply writes the template output to the writer. By default FTL (Freemarker Template Language) and GString (Groovy String) are supported.

Additional template renderers can be supported by implementing the `org.moqui.context.TemplateRenderer` interface and adding a `resource-facade.template-renderer` element to the Moqui Conf XML file.

To run a script through the Resource Facade use the `Object ec.resource.runScriptInCurrentContext(String location, String method)` method. Specify the `location` and optionally the `method` within the script at the location and this method will run the script and return the `Object` that the script returns or evaluates to. There is a variation on this method in the Resource Facade that also accepts a `Map additionalContext` parameter for convenience (it just pushes the Map onto the context stack, runs the script, then pops from the context stack). By default Moqui supports Groovy, XML Actions, JavaScript, and any scripting engine available through the `javax.script.ScriptEngineManager`.

To add a script runner you have two options. You can use the `javax.script` approach for any scripting language that implements the `javax.script.ScriptEngine` interface and is discoverable through the `javax.script.ScriptEngineManager`. Moqui uses this to discover the script engine using the extension on the script's filename and execute the script. If the script engine implements the `javax.script.Compilable` interface then Moqui will compile the script and cache it in compiled form for the faster repeat execution of a script at a given location.

The other option is to implement the `org.moqui.context.ScriptRunner` interface and add a `resource-facade.script-runner` element to the Moqui Conf XML file. Moqui uses Groovy the XML Actions through this interface as it provides additional flexibility not available through the `javax.script` interfaces.

Because Groovy is the default expression language in Moqui there are a few Resource Facade methods to easily evaluate expressions for different purposes:

- `boolean evaluateCondition(String expression, String debugLocation)` is used to evaluate a Groovy condition `expression` and return the boolean result

- `Object evaluateContextField(String expression, String debugLocation)` is used to evaluate the `expression` to return a field within the context, and more generally to evaluate any Groovy expression and return the result
- `String evaluateStringExpand(String inputString, String debugLocation)` is used to expand the `inputString`, treating it as a GString (Groovy String) and returns the expanded value

These methods accept a `debugLocation` parameter that is used in error messages. For faster evaluation these expressions are all cached, using the expression itself as the key for maximal reuse.

Data Model Definition

Entity Definition XML

Let's start with a simple entity definition that shows the most common elements. This is an actual entity that is part of Moqui Framework:

```
<entity entity-name="DataSource" package-name="moqui.basic" cache="true">
    <field name="dataSourceId" type="id" is-pk="true"/>
    <field name="dataSourceTypeEnumId" type="id"/>
    <field name="description" type="text-medium"/>
    <relationship type="one" title="DataSourceType"
            related-entity-name="Enumeration">
        <key-map field-name="dataSourceTypeEnumId"/>
    </relationship>
    <seed-data>
        <moqui.basic.EnumerationType description="Data Source Type"
            enumTypeId="DataSourceType"/>
        <moqui.basic.Enumeration description="Purchased Data"
            enumId="DST_PURCHASED_DATA" enumTypeId="DataSourceType"/>
    </seed-data>
</entity>
```

Just like a Java class an entity has a package name and the full name of the entity is the package name plus the entity name, in the format:

`${package-name}.${entity-name}`

Based on that pattern the full name of this entity is:

`moqui.basic.DataSource`

This example also has the `entity.cache` attribute set to `true`, meaning that it will be cached unless the code doing the find says otherwise.

The first field (`dataSourceId`) has the `is-pk` attribute set to true, meaning it is one of the primary key fields on this entity. In this case it is the only primary key field, but any number of fields can have this attribute set to true to make them part of the primary key.

The third field (**description**) is a simple field to hold data. It is not part of the primary key, and it is not a foreign key to another entity.

The field.type attribute is used to specify the data type for the field. The default options are defined in the MoquiDefaultConf.xml file with the database-list.dictionary-type element. These elements specify the default type settings for each dictionary type and there can be an override to this setting for each database using the database.database-type element.

You can use these elements to add your own types in the data type dictionary. Those custom types won't appear in autocomplete for the field.type attribute in your XML editor unless you change the XSD file to add them there as well, but they will still function just fine.

The second field (**dataSourceTypeEnumId**) is a foreign key to the Enumeration entity, as denoted by the relationship element in this entity definition. The two records in under the seed-data element define the EnumerationType to group the Enumeration options, and one of the Enumeration options for the **dataSourceTypeEnumId** field. The records under the seed-data element are loaded with the command-line -load option (or the corresponding API call) along with the seed type.

There is an important pattern here that allows the framework to know which **enumTypeId** to use to filter Enumeration options for a field in automatically generated form fields and such. Notice that the value in the relationship.**title** attribute matches the enumTypeId. In other words, for enumerations anyway, there is a convention that the relationship.**title** value is the type ID to use to filter the list.

This is a pattern used a lot in Moqui and in the Mantle Business Artifacts because the Enumeration entity is used to manage types available for many different entities.

In this example there is a key-map element under the relationship element, but that is only necessary if the field name(s) on this entity does not match the corresponding field name(s) on the related entity. In other words, because the foreign key field is called **dataSourceTypeEnumId** instead of simply **enumId** we need to tell the framework which field to use. It knows which field is the primary key of the related entity (Enumeration in this case), but unless the field names match it does not know which fields on this entity correspond to those fields.

In most cases you can use something more simple without key-map elements like:

```
<relationship type="one" related-entity-name="Enumeration"/>
```

The seed-data element allows you to define basic data that is necessary for the use of the entity and that is an aspect of defining the data model. These records get loaded into the database along with the entity-facade-xml files where the **type** attribute is set to seed.

With this introduction to the most common elements of an entity definition, lets now look at some of the other elements and attributes available in an entity definition.

- other entity attributes

- **group-name**: Each datasource available through the Entity Facade is used by putting an entity in the group for that datasource. The value here should match a value on the moqui-conf.entity-facade.datasource.**group-name** attribute in the Moqui Conf XML file. If no value is specified will default to the value of the moqui-conf.entity-facade.**default-group-name** attribute. By default configuration the valid values include transactional (default), analytical, tenantcommon, and nosql.
- **sequence-bank-size**: The size of the sequence bank to keep in memory. Each time the in-memory bank runs out the **seqNum** in the SequenceValueItem record will be incremented by this amount.
- **sequence-primary-stagger**: The maximum amount to stagger the sequenced ID. If 1 the sequence will be incremented by 1, otherwise the current sequence ID will be incremented by a random value between 1 and staggerMax.
- **sequence-secondary-padded-length**: If specified front-pads the secondary sequenced value with zeroes until it is this length. Defaults to 2.
- **optimistic-lock**: Set to true to have the Entity Facade compare the **lastUpdatedStamp** field in memory to the one in the database before doing an update on the record. If the timestamps don't match an error will be generated. Defaults to "false" (no timestamp locking).
- **no-update-stamp**: By default the Entity Facade adds a single field (**lastUpdatedStamp**) to each entity for use in optimistic locking and data synchronization. If you do not want it to create that stamp field for this entity then set this to "false".
- **cache**: can be set to these values (defaults to false):
  - true: use cache for finds (code may override this)
  - false: no cache for finds (code may override this)
  - never: no cache for finds (code may NOT override this)
- **authorize-skip**: can be set to these values (defaults to false):
  - true: skip all authz checks for this entity
  - false: do not skip authz checks
  - create: skip authz checks for create operations
  - view: skip authz checks for finds or read-only operations
  - view-create: skip authz checks for find and create ops

- other field attributes
  - **encrypt**: Set to true to encrypt this field in the database. Defaults to false (not encrypted).
  - **enable-audit-log**: Set to true to log all changes to the field along with when it was changed and the user who changes. The data is stored using the EntityAuditLog entity. Defaults to false (no audit logging).
  - **enable-localization**: If set to true gets on this field will be looked up with the LocalizedEntityField entity and if there is a matching record the localized value will be returned instead of the original record's value. Defaults to false for performance reasons, only set to true for fields that will have translations.

While some database optimizations must be done in the database itself because so many such features vary between databases, you can declare indexes along with the entity definition using the `index` element. As an element under the `entity` element it would look something like this:

```
<index name="EX_NAME_IDX1" unique="true">
    <index-field name="exampleName"/>
</index>
```

Entity Extension - XML

An entity can be extended without modifying the XML file where the original is defined. This is especially useful when you want to extend an entity that is part of a different component such as the Mantle Universal Data Model (mantle-udm) or even part of the Moqui Framework and you want to keep your extensions separate.

This is done with the `extend-entity` element which can mixed in with the `entity` elements in an entity definition XML file. This element has most of the same attributes and sub-elements as the `entity` element used to define the original entity. Simply make sure the **`entity-name`** and **`package-name`** match the same attributes on the original `entity` element and anything else you specify will add to or override the original entity.

Here is an example if a XML snippet to extend the `moqui.example.Example` entity:

```
<extend-entity entity-name="Example" package-name="moqui.example">
    <field name="auditedField" type="text-medium" enable-audit-log="true"/>
    <field name="encryptedField" type="text-medium" encrypt="true"/>
</extend-entity>
```

Entity Extension - DB

You can also extend an entity with a database record using the `UserField` entity. This is a bit different from extending an entity with the `extend-entity` XML element because it is a virtual extension and the data goes in a separate data structure using the `UserFieldValue` entity.

The main reason for this difference is that User Fields are generally added for a group of users or a single user, and are not visible outside the group they are associated with. You can use the `ALL_USERS` User Group to have a User Field applies to all users.

Even though it operates this way under the covers, from the perspective of the `EntityValue` object it is treated the same way as any other field on the entity.

Here is an example element from the `ExampleTypeData.xml` file showing how you would add a **`testUserField`** field accessible by all users to the `moqui.example.Example` entity:

```
<moqui.entity.UserField entityName="moqui.example.Example"
    fieldName="testUserField" userGroupId="ALL_USERS" fieldType="text-long"
```

```
    enableAuditLog="Y" enableLocalization="N" encrypt="N"/>
```

Data Model Patterns

There are various useful data model patterns that Moqui Framework has conventions and functionality to help support. These data model patterns are also used extensively in the Moqui and Mantle data models.

Master Entities

A Master Entity is one whose records exist independent of other entities, and generally has a single field primary key. Examples of this include the `moqui.example.Example`, `moqui.security.UserAccount`, `mantle.party.Party`, `mantle.product.Product`, and `mantle.order.OrderHeader` entities.

To set a primary sequenced ID, which is the sequenced value for the primary key of a master entity, use the `EntityValue.setSequencedIdPrimary()` method. You can also manually set the primary key field to any value, as long as it is unique.

Detail Entities

A Detail Entity adds detail to a Master Entity for fields that have a one-to-many relationship with the Master. The primary key is usually two fields and one of the fields is the single primary key field of the master entity. The second field is a special sort of sequenced ID that instead of having an absolute sequence value its value is in the context of the master entity's primary key.

An example of a detail entity is `ExampleItem`, which is a detail to the master entity `Example`. `ExampleItem` has two primary keys: **`exampleId`** (the primary key field of the master entity) and **`exampleItemSeqId`** which is a sub-sequence to distinguish the detail records within the context of a master record.

To populate the secondary sequenced ID first set the master's primary key (**`exampleId`** for `ExampleItem`), then use the `EntityValue.setSequencedIdSecondary()` method to automatically populate it (for `ExampleItem` the **`exampleItemSeqId`**).

A single master entity can have multiple detail entities associated with it to structure distinct data as needed.

Join Entities

A Join Entity is used to associate Master Entities, usually two. A Join Entity is a physical representation of a many-to-many relationship between entities in a logical model.

A join entity is useful for tracking associated records among the master entities, and for any data that is associated with both master entities as opposed to just one of them. For example if you want to specify a sequence number for one master entity record in the context of a record of the other master entity, the sequence number field should go on the join entity and not on either of the master entities.

The join entity may have a single generated primary key, or a natural composite primary key consisting of the single primary key field of each of the master entities and optionally a **`fromDate`** field with a corresponding **`thruDate`** field that is not part of the join entity's primary key.

One example of this is the `ExampleFeatureAppl` entity which joins the `Example` and `ExampleFeature` master entities. The `ExampleFeatureAppl` entity has three primary key fields: exampleId (the PK of the `Example` entity), **`exampleFeatureId`** (the PK of the `ExampleFeature` entity), and a **`fromDate`**. It also has a **`thruDate`** field to accompany the **`fromDate`** PK field.

To better describe the relationship between an `Example` and an `ExampleFeature`, the `ExampleFeatureAppl` entity also has a **`sequenceNum`** field for ordering features within and example, and a **`exampleFeatureApplEnumId`** field to describe how the feature applies to the example (Required, Desired, or Not Allowed).

To see the actual entity definition and seed data for the `ExampleFeatureAppl` entity see the `ExampleEntities.xml` file (in the example component that comes with Moqui Framework).

Dependent Entities

A few parts of the API and Tools app support the concept of "dependent" entities. Dependent entities can be found for any entity, but the concept is most useful for dependents of Master Entities. The general idea is that things like the items of an order (`mantle.order.OrderItem`) are dependent on the header (`mantle.order.OrderHeader`). It is useful to do operations such as data export including the master entity and all of its dependents.

Conceptually this is pretty simple, but the implementation is more complex because the information we have to work with for this is the entity relationships. The general idea is that each type one relationship points from a dependent entity to its master, and by this definition many dependent entities have more than one master entity and an entity can be both a dependent and a master entity so what an entity is depends on how you are treating it. When defining entities there is an automatic reverse type relationship for each type one relationship, and while it is generally a type many reverse relationship if the two entities have the same PK field(s) then it is a type one automatic reverse relationship.

For example, `OrderItem` has a type one relationship to `OrderHeader` so there is an automatic reverse relationship of type many from `OrderHeader` to `OrderItem`. This establishes `OrderItem` as a dependent of `OrderHeader`.

When getting dependents for an entity the method (which is part of the internal Entity Facade implementation: `EntityDefinition.getDependentsTree()`) runs recursively to get the dependents of dependents as well. The general idea is that for entities like `OrderHeader` you can get all records that define the order.

Enumerations

An Enumeration is simply a pre-configured set of possible values. Enumerations are used to describe single records or relationships between records. An entity may have multiple fields enumerated values.

The entity in Moqui where all enumerations are stored is named `Enumeration`, and values in it are split by type with a record in the `EnumerationType` entity.

When a field is to have a constrained set of possible enumerated values it should have the suffix "EnumId", like the **`exampleTypeEnumId`** field on the `Example` entity. For each field there should also be a relationship element to describe the relationship from the current entity to the `Enumeration` entity. The **`title`** attribute on the `relationship` element should have the same value as the **`enumTypeId`** that is used for the `Enumeration` records that are possible values for that field. Generally the **`title`** attribute should be the same as the enum field's name up to the "EnumId" suffix. For example the relationship title for the **`exampleTypeEnumId`** field is `ExampleType`.

Status, Flow, Transition and History

Another useful data concept is tracking the status of a record. Various business concepts have a lifecycle of some sort that is easily tracked with a set of possible status values. The possible status values are tracked using the `StatusItem` entity and exist in sets distinguished by a **`statusTypeId`** pointing to a record in the `StatusType` entity.

A set of status values are kind of like nodes in a graph and the transitions between those nodes represent possible changes from one status to another. The possible transitions from one status to another are configured using records in the `StatusFlowTransition` entity.

There can be multiple status flows for a set of status items with a given **`statusTypeId`**, each represented by a `StatusFlow` record. The `StatusItem` records are associated with a `StatusFlow` using `StatusFlowItem` records. For example the `WorkEffort` entity has a **`statusFlowId`** field to specify which status flow should be used for a project or task.

If an entity has only a single status associated with it the field to track the status can simply be named **`statusId`**. If an entity needs to have multiple status values then the field name should have a distinguishing prefix and end with "StatusId".

There should be a relationship defined for each status field to tie the current entity to the `StatusItem` entity. Similar to the pattern with the `Enumeration` entity, the **`title`** attribute on the `relationship` element should match the **`statusTypeId`** on each `StatusItem` record.

The audit log feature of the Entity Facade is the easiest way to keep a history of status changes including who made the change, when it was made, and the old and new status values. To turn this on just use set the **`enable-audit-log`** attribute to true on the `entity.field` element. With this the field definition would look something like:

```
<field name="statusId" type="id" enable-audit-log="true"/>
```

Units of Measure

A unit of measure is a standardized or custom unit for measures such as length, weight, temperature, data size, and even currency. These are the types of UOM. A `moqui.basic.Uom` record, identified by **`uomId`**, has type (**`uomTypeEnumId`**), **`description`**, and **`abbreviation`** fields. The OOTB data for units of measure is in the `UnitData.xml` file.

Most UOM types have a conversion between different units of the same type. These conversions are modeled in the `UomConversion` entity. For example there are 1000 meters in a kilometer, and that is recorded this way:

```
<moqui.basic.UomConversion uomConversionId="LEN_km_m" uomId="LEN_km"
    toUomId="LEN_m" conversionFactor="1000"/>
```

The **`conversionFactor`** is multiplied by the value with the **`uomId`** unit to get a value in the **`toUomId`** unit. You can also divide to go in the other direction. For example 1km = 1000m so a 1 value with the `LEN_km` unit is multiplied by the **`conversionFactor`** of `1000` to get a value of 1000 for the `LEN_m` unit.

There is also a **`conversionOffset`** field for cases such as Celsius and Fahrenheit temperatures where a value must be added (or subtracted) to go from one unit to the other. The **`conversionFactor`** is multiplied first, then the **`conversionOffset`** is added to the result. When converting in the reverse direction the **`conversionOffset`** is subtracted first, then the result is divided by the **`conversionFactor`**.

Some UOM types, such as currency, have conversion factors that change over time. To handle this the `UomConversion` entity has optional effective date (**`fromDate`**, **`thruDate`**) fields.

Geographic Boundaries and Points

A geographic boundary can be a political division, business region, or any other geographic area. Each `moqui.basic.Geo` record, identified by a **`geoId`**, has a type (**`geoTypeEnumId`**) such as city, country, or sales region. Each `Geo` has a name (**`geoName`**) and may have 2 letter (**`geoCodeAlpha2`**), 3 letter (**`geoCodeAlpha3`**), and numeric (**`geoCodeNumeric`**) codes

following the ISO 3166 pattern for country code (see the `GeoCountryData.xml` file for the country data that comes with Moqui).

The `Geo` entity also has a **`wellKnownText`** field for machine-readable detail about the geometry of the geographic boundary. It is meant to contain text following the ISO/IEC 13249-3:2011 specification which is supported by various databases and tools (including Java libraries). For a good introduction to WKT see:

http://en.wikipedia.org/wiki/Well-known_text

Use the `GeoAssoc` entity to associate `Geo` records. This has different types (**`geoAssocTypeEnumId`**) and can be used for regions of larger geographic boundaries (`GAT_REGIONS`; like cities within states, states within countries), for `Geo` records that are more general groups to associate them with the `Geo` records in the group (`GAT_GROUP_MEMBER`; like the lower 48 states in the USA), or other types you might define. The **`geoId`** field should point to the group or larger area, and the **`toGeoId`** to the group member or region within the area. See the `GeoUsaData.xml` file for examples of both.

A `GeoPoint` is a specific geographic point, i.e. a point on the Earth's surface. It has **`latitude`**, **`longitude`**, and **`elevation`** fields and a **`elevationUomId`** field to specify the unit for the **`elevation`** (such as feet, which is `LEN_ft`). There is also a **`dataSourceId`** to specify where the data came from and an **`information`** field for general text about the point.

The Entity Facade

Basic CrUD Operations

The basic CrUD operations for an entity record are available through the `EntityValue` interface. There are two main ways to get an EntityValue object:

- Make a Value (use `ec.entity.makeValue(entityName)`)
- Find a Value (more details on this below)

Once you have an `EntityValue` object you can call the **`create()`**, **`update()`**, or **`delete()`** methods to perform the desired operation. There is also a **`createOrUpdate()`** method that will create a record if it doesn't exist, or update it if it does.

Note that all of these methods, like many methods on the `EntityValue` interface, return a self-reference for convenience so that you can chain operations. For example:

```
ec.entity.makeValue("Example").setAll(fields)
         .setSequencedIdPrimary().create()
```

While this example is interesting, only in rare cases should you create a record directly using the Entity Facade API (accessed as `ec.entity`). You should generally do CrUD operations through services, and there are automatic CrUD services for all entities available through the Service Facade. These services have no definition, they exist implicitly and are driven only the entity definition.

We'll discuss the Service Facade more below in the context of the logic layer, but here is an example of what that operation would look like using an implicit automatic entity service:

```
ec.service.sync().name("create#Example").parameters(fields).call()
```

Most of the Moqui Framework API methods return a self-reference for convenient chaining of method calls like this. The main difference between the two is that one goes through the Service Facade and the other doesn't. There are some advantages of going through the Service Facade (such as transaction management, flow control, security options, and so much more), but many things are the same between the two calls including automatic cleanup and type conversion of the fields passed in before performing the underlying operation.

Also note that with the implicit automatic entity service you don't have to explicitly set the sequenced primary ID as it automatically determines that there is a single primary and if it is not present in the parameters passed into the service then it will generate one.

However you do the operation, only the entity fields that are modified or passed in are updated. The `EntityValue` object will keep track of which fields have been modified and only create or update those when the operation is done in the database. You can ask an `EntityValue` object if it is modified using the **`isModified()`** method, and you can restore it to its state in the database (populating all fields, not just the modified ones) using the **`refresh()`** method.

If you want to find all the differences between the field values currently in the `EntityValue` and the corresponding column values in the database, use the **`checkAgainstDatabase`**`(List messages)` method. This method is used when asserting (as opposed to loading) an `entity-facade-xml` file and can also be used manually if you want to write Java or Groovy code check the state of data.

Finding Entity Records

Finding entity records is done using the `EntityFind` interface. Rather than using a number of different methods with different optional parameters through the `EntityFind` interface you can call methods for the aspects of the find that you care about, and ignore the rest. You can get a find object from the `EntityFacade` with something like:

```
ec.getEntity().makeFind("moqui.example.Example")
```

Most of the methods on the `EntityFind` interface return a reference to the object so that you can chain method calls instead of putting them in separate statements. For example a find by the primary on the `Example` entity would look like this:

```
EntityValue example = ec.entity.makeFind("moqui.example.Example")
      .condition("exampleId", exampleId).useCache(true).one()
```

The `EntityFind` interface has methods on it for:

- conditions (both where and having)

- **condition**(String fieldName, Object value): Simple condition, named field equals value.
- **condition**(String fieldName, EntityCondition.ComparisonOperator operator, Object value): Compare the named field to the value using the operator which can be EQUALS, NOT_EQUAL, LESS_THAN, GREATER_THAN, LESS_THAN_EQUAL_TO, GREATER_THAN_EQUAL_TO, IN, NOT_IN, BETWEEN, LIKE, or NOT_LIKE.
- **conditionToField**(String fieldName, EntityCondition.ComparisonOperator operator, String toFieldName): Compare a field to another field using the operator.
- **condition**(Map<String, ?> fields): Constrain by each entry in the Map whose key matches a field name on the entity. If a field has been set with the same name and any of the Map keys, this will replace that field's value. Fields set in this way will be combined with other conditions (if applicable) just before doing the query. This will do conversions if needed from Strings to field types as needed, and will only get keys that match entity fields. In other words, it does the same thing as: EntityValue.**setFields**(fields, true, null, null).
- **condition**(EntityCondition condition): Add a condition created through the EntityConditionFactory.
- **conditionDate**(String fromFieldName, String thruFieldName, Timestamp compareStamp): Add conditions for the standard effective date query pattern including from field is null or earlier than or equal to compareStamp and thru field is null or later than or equal to compareStamp.
- **havingCondition**(EntityCondition condition): Add a condition created through the EntityConditionFactory to the having conditions. Having is the standard SQL concept and used for conditions applied after the grouping and functions.
- **searchFormInputs**(String inputFieldsMapName, String defaultOrderBy, boolean alwaysPaginate): Adds conditions for the fields found in the inputFieldsMapName Map. The fields and special fields with suffixes supported are the same as the *-find fields in the XML Forms. This means that you can use this to process the data from the various inputs generated by XML Forms. The suffixes include things like *_op for operators and *_ic for ignore case. If inputFieldsMapName is empty will look at the ec.web.parameters map if the web facade is available, otherwise the current context (ec.context). If there is not an orderByField parameter (one of the standard parameters for search XML Forms) defaultOrderBy is used instead. If alwaysPaginate is true pagination offset/limit will be set even if there is no pageIndex parameter.

- fields to select with **selectField**(String fieldToSelect) and/or **selectFields**(Collection<String> fieldsToSelect)
- fields to order the results by
  - **orderBy**(String orderByFieldName): A field of the find entity to order the query by. Optionally add a " ASC" to the end or "+" to the beginning for ascending, or " DESC" to the end of "-" to the beginning for descending. If any other order by fields have

already been specified this will be added to the end of the list. The `String` may be a comma-separated list of field names. Only fields that actually exist on the entity will be added to the order by list.
    - **`orderBy`**`(List<String> orderByFieldNames)`: Each `List` entry is passed to the **`orderBy`**`(String orderByFieldName)` method.
- whether or not to cache the results with **`useCache`**`(Boolean useCache)`, defaults to the value on the entity definition
- the offset and limit to pass to the datasource to limit results
    - **`offset`**`(Integer offset)`: The offset, i.e. the starting row to return. Default (null) means start from the first actual row. Only applicable for **`list`**`()` and **`iterator`**`()` finds.
    - **`offset`**`(int pageIndex, int pageSize)`: Specify the offset in terms of page index and size. Actual offset is `pageIndex * pageSize`.
    - **`limit`**`(Integer limit)`: The limit, i.e. max number of rows to return. Default (null) means all rows. Only applicable for **`list`**`()` and **`iterator`**`()` finds.
- database options including distinct with the **`distinct`**`(boolean distinct)` method and for update with the **`forUpdate`**`(boolean forUpdate)` method
- JDBC options
    - **`resultSetType`**`(int resultSetType)`: Specifies how the `ResultSet` will be traversed. Available values are `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE (default)` or `ResultSet.TYPE_SCROLL_SENSITIVE`. See the `java.sql.ResultSet` JavaDoc for more information. If you want it to be fast, use the common option `ResultSet.TYPE_FORWARD_ONLY`. For partial results where you want to jump to an index make sure to use `ResultSet.TYPE_SCROLL_INSENSITIVE`, which is the default.
    - **`resultSetConcurrency`**`(int resultSetConcurrency)`: Specifies whether or not the `ResultSet` can be updated. Available values are `ResultSet.CONCUR_READ_ONLY` (default) or `ResultSet.CONCUR_UPDATABLE`. Should pretty much always be `ResultSet.CONCUR_READ_ONLY` with the Entity Facade since updates are generally done as separate operations.
    - **`fetchSize`**`(Integer fetchSize)`: The JDBC fetch size for this query. Default (null) will fall back to datasource settings. This is not the fetch as in the OFFSET/FETCH SQL clause (use the offset/limit methods for that), and is rather the JDBC fetch to determine how many rows to get back on each round-trip to the database. Only applicable for **`list`**`()` and **`iterator`**`()` finds.
    - **`maxRows`**`(Integer maxRows)`: The JDBC max rows for this query. Default (null) will fall back to datasource settings. This is the maximum number of rows the `ResultSet` will keep in memory at any given time before releasing them and if requested they are retrieved from the database again. Only applicable for **`list`**`()` and **`iterator`**`()` finds.

There are various options for conditions, some on the `EntityFind` interface itself and a more extensive set available through the `EntityConditionFactory` interface. To get an instance of this interface use the `ec.entity.`**`getConditionFactory`**`()` method, something like:

```
EntityConditionFactory ecf = ec.entity.getConditionFactory();
ef.condition(ecf.makeCondition(...));
```

For find forms that follow the standard Moqui pattern (used in XML Form find fields and can be used in templates or JSON or XML parameter bodies too), just use the `EntityFind.searchFormInputs()` method.

Once all of these options have been specified you can do any of these actual operations to get results or make changes:

- get a single `EntityValue` (`one()` method)
- get an `EntityValueList` with multiple value objects (`list()` method)
- get an `EntityListIterator` to handle a larger set of results in smaller batches (with the `iterator()` method)
- get a count of matching results (`count()` method)
- update all matching records with specified fields (`updateAll()` method)
- delete all matching records (`delete()` method)

Flexible Finding with View Entities

You probably noticed that the `EntityFind` interface operates on a single entity. To do a query across multiple entities joined together and represented by a single entity name you can create a static view entity using a XML definition that lives along side normal entity definitions.

A view entity can also be defined in database records (in the `DbViewEntity` and related entities) or with dynamic view entities built with code using the `EntityDynamicView` interface (get an instance using the `EntityFind.makeEntityDynamicView()` method).

Static View Entity

A view entity consists of one or more member entities joined together with key mappings and a set of fields aliased from the member entities with optional functions associated with them. The view entity can also have conditions associated with it to encapsulate some sort of constraint on the data to be included in the view.

Here is an example of a view-entity XML snippet from the `ExampleViewEntities.xml` file in the example component:

```
<view-entity entity-name="ExampleFeatureApplAndEnum"
        package-name="moqui.example">
    <member-entity entity-alias="EXFTAP" entity-name="ExampleFeatureAppl"/>
    <member-entity entity-alias="ENUM"
            entity-name="moqui.basic.Enumeration"
            join-from-alias="EXFTAP">
       <key-map field-name="exampleFeatureApplEnumId"/>
    </member-entity>
```

```
    <alias-all entity-alias="EXFTAP"/>
    <alias-all entity-alias="ENUM"/>
</view-entity>
```

Just like an entity a view entity has a name and exists in a package using the **entity-name** and **package-name** attributes on the view-entity element.

Each member entity is represented by a member-entity element and is uniquely identified by an alias in the **entity-alias** attribute. Part of the reason for this is that the same entity can be a member in a view entity multiple times with a different alias for each one.

Note that the second member-entity element also has a **join-from-alias** attribute to specify that it is joined to the first member entity. Only the first member entity does not have a **join-from-alias** attribute. If you want the current member entity to be optional in the join (a left outer join in SQL) then just set the **join-optional** attribute to true.

To describe how the two entities relate to each other use one or more key-map elements under the member-entity element. The key-map element has two attributes: **field-name** and **related-field-name**. Note that the **related-field-name** attribute is optional when matching the primary key field on the current member entity.

Fields can be aliased in sets using the alias-all element, as in the example above, or individually using the alias element. If you want to have a function on the field then alias them individually with the alias element. Note for SQL databases that if any aliased field has a function then all other fields that don't have a function but that are selected in the query will be added to the group by clause to avoid invalid SQL.

View Entity Auto Minimize on Find

When doing a query with the Entity Facade EntityFind you can specify fields to select and only those fields will be selected. For view entities this does a little more to give you a big boost in performance without much work.

A common problem with static view entities is that you want to join in a bunch of member entities to provide a lot of options for search screens and similar flexible queries and when you do this the temporary table for the query in the database can get HUGE. When the common use is to only select certain fields and only have conditions and sorting on a limited set of fields you may end up joining in a number of tables that are not actually used. In effect you are asking the database to do a LOT more work that it really needs to for the data you need.

One approach to solving this is to build a EntityDynamicView on the fly and only join in the entities you need for the specific query options used. This works, but is cumbersome.

The easy approach is to just take advantage of the feature in EntityFind that automatically minimizes the fields and entities joined in for each particular query. On a view entity just specify the fields to select, the conditions, and the order by fields. The Entity Facade will

automatically go through the view entity definition and only alias the fields that are used for one of these (select, conditions, order by), and only join in the entities with fields that are actually used (or that are need to connect a member entity with other member entities to complete the join).

A good example of this is the `FindPartyView` view entity defined in the `PartyViewEntities.xml` file in Mantle Business Artifacts. This view entity has a respectable 13 member entities. Without the automatic minimize that would be 13 tables joined in to every query on it. With millions of customer records or other similarly large party data each query could take a few minutes. When only querying on a few fields and only joining in a small number of member entities and a minimal number of fields, the query gets down to sub-second times.

The actual find is done by the `mantle.party.PartyServices.find#Party` service. The implementation of this service is a simple 45 line Groovy script (`findParty.groovy`), and most of that script is just adding conditions to the find based on parameter being specified or not. Doing the same thing with the `EntityDynamicView` approach requires hundreds of lines of much more complex scripting, more complex to both write and maintain.

Database Defined View Entity

In addition to defining view entities in XML you can also define them in database records using `DbViewEntity` and related entities. This is especially useful for building screens where the user defines a view on the fly (like the `EditDbView.xml` screen in the tools component, get to it in the menu with `Tool => Data View`), and then searches, views, and exports the data using a screen based on the user-defined view (like the `ViewDbView.xml` screen).

There aren't quite as many options when defining a DB view entity, but the main features are there and the same patterns apply. There is a view entity with a name (**`dbViewEntityName`**), package (**`packageName`**), and whether to **`cache`** results. It also has member entities (`DbViewEntityMember`), key maps to specify how the members join together (`DbViewEntityKeyMap`), and field aliases (`DbViewEntityAlias`). Here is an example, from the `example` component:

```
<moqui.entity.view.DbViewEntity dbViewEntityName="StatusItemAndTypeDb"
    packageName="moqui.example" cache="Y"/>
<moqui.entity.view.DbViewEntityMember
    dbViewEntityName="StatusItemAndTypeDb" entityAlias="SI"
    entityName="moqui.basic.StatusItem"/>
<moqui.entity.view.DbViewEntityMember
    dbViewEntityName="StatusItemAndTypeDb" entityAlias="ST"
    entityName="moqui.basic.StatusType" joinFromAlias="SI"/>
<moqui.entity.view.DbViewEntityKeyMap
    dbViewEntityName="StatusItemAndTypeDb" joinFromAlias="SI"
    entityAlias="ST" fieldName="statusTypeId"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
    entityAlias="SI" fieldAlias="statusId"/>
```

```
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
    entityAlias="SI" fieldAlias="description"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
    entityAlias="SI" fieldAlias="sequenceNum"/>
<moqui.entity.view.DbViewEntityAlias dbViewEntityName="StatusItemAndTypeDb"
    entityAlias="ST" fieldAlias="typeDescription" fieldName="description"/>
```

As you can see the entity and field names correlate with the XML element and attribute names. To use these entities just refer to them by name just like any other entity.

Dynamic View Entity

Even with the automatic view entity minimize that the Entity Facade does during a find there are still cases where you'll need or want to build a view programmatically on the fly instead of having a statically defined view entity.

To do this get an instance of the `EntityDynamicView` interface using the `EntityFind.makeEntityDynamicView()` method. This interface has methods on it that do the same things as the XML elements in a static view entity. Add member entities using the `addMemberEntity(String entityAlias, String entityName, String joinFromAlias, Boolean joinOptional, Map<String, String> entityKeyMaps)` method.

One convenient option that doesn't exist for static (XML defined) view entities is to join in a member entity based on a relationship definition. To do this use the `addRelationshipMember(String entityAlias, String joinFromAlias, String relationshipName, Boolean joinOptional)` method.

To alias fields use the `addAlias(String entityAlias, String name, String field, String function)` method, the shortcut variation of it `addAlias(String entityAlias, String name)`, or the `addAliasAll(String entityAlias, String prefix)` method.

You can optionally specify a name for the dynamic view with the `setEntityName()` method, but usually this mostly useful for debugging and the default name (`DynamicView`) is usually just fine.

Once this is done just specify conditions and doing the find operation as normal on the `EntityFind` object that you used to create the `EntityDynamicView` object.

Entity ECA Rules

Entity ECA (EECA) rules can be used to trigger actions to run when data is modified or searched. It is useful for maintaining entity fields (database columns) that are based on other entity fields or for updating data in a separate system based on data in this system. EECA rules should not generally be used for triggering business processes because the rules are applied too widely. Service ECA rules are a better tool for triggering processes.

For example here is an EECA rule from the `Work.eecas.xml` file in Mantle Business Artifacts that calls a service to update the total time worked on a task (WorkEffort) when a TimeEntry is created, updated, or deleted:

```
<eeca entity="mantle.work.time.TimeEntry" on-create="true" on-update="true"
        on-delete="true" get-entire-entity="true">
    <actions><service-call in-map="context"
            name="mantle.work.TaskServices.update#TaskFromTime"/></actions>
</eeca>
```

An ECA (event-condition-action) rule is a specialized type of rule to conditionally run actions based on events. For Entity ECA rules the events are the various find and modify operations you can do with a record. Set any of these attributes (of the `eeca` element) to true to trigger the EECA rule on the operation: **`on-create`**, **`on-update`**, **`on-delete`**, **`on-find-one`**, **`on-find-list`**, **`on-find-iterator`**, **`on-find-count`**.

By default the EECA rule will run after the entity operation. To have it run before set the **`run-before`** attribute to `true`. There is also a **`run-on-error`** attribute which defaults to `false` and if set to `true` the EECA rule will be triggered even if there is an error in the entity operation.

When the actions run the context will be whatever context the service was run in, plus the entity field values passed into the operation for convenience in using the values. There are also special context fields added:

- `entityValue`: A `Map` with the field values passed into the entity operation. This may not include all field values that are populated in the database for the record. To fill in the field values that are not passed in from the database record set the `eeca`**`.get-entire-entity`** attribute to `true`.
- `originalValue`: If the `eeca`**`.get-original-value`** attribute is set to `true` and the EECA rule runs before the entity operation (**`run-before`**`=true`) this will be an EntityValue object representing the original (current) value in the database.
- `eecaOperation`: A String representing the operation that triggered the EECA rule, basically the **`on-*`** attribute name without the `"on-"`.

The `condition` element is the same condition as used in XML Actions and may contain `expression` and `compare` elements, combined as needed with `or`, `and`, and `not` elements.

The `actions` element is the same as `actions` elements in service definitions, screens, forms, etc. It contains a XML Actions script. See the **Overview of XML Actions** section for more information.

Entity Data Import and Export

Loading Entity XML and CSV

Entity records can be imported from XML and CSV files using the `EntityDataLoader`. This can be done through the Entity Facade API using the `ec.entity.makeDataLoader()` method to get an object that implements the interface and using its methods to specify which data to load and then load it (using the `load()` method), get an `EntityList` of the records (using the `list()` method), or validate the data against the database (using the `check()` method).

There are a few options for specifying which data to load. You can specify one or more locations using the `location(String location)` and `locationList(List<String> locationList)` methods. You can use text directly with the `xmlText(String xmlText)` and `csvText(String csvText)` methods. You can also load from component data directories and the `entity-facade.load-data` elements in the Moqui Conf XML file by specifying the types of data to load (only the files with a matching type will be loaded) using the `dataTypes(Set<String> dataTypes)` method.

To set the transaction timeout to something different from the default, usually larger to handle processing large files, use the `transactionTimeout(int tt)` method. If you expect mostly inserts you can use pass `true` to the `useTryInsert(boolean useTryInsert)` method to improve performance by doing an insert without a query to see if the record exists and then if the insert fails with an error try an update.

To help with foreign keys when records are out of order, but you know all will eventually be loaded, pass `true` to the `dummyFks(boolean dummyFks)` method and it will create empty records for foreign keys with no existing record. When the real record for the FK is loaded it will simply update the empty dummy record. To disable Entity ECA rules as the data is loaded pass `true` to the `disableEntityEca(boolean disableEeca)` method.

For CSV files you can specify which characters to use when parsing the file(s) with `csvDelimiter(char delimiter)` (defaults to ',), `csvCommentStart(char commentStart)` (defaults to '#'), and `csvQuoteChar(char quoteChar)` (defaults to '"').

Note that all of these methods on the `EntityDataLoader` return a self reference so you can chain calls, i.e. it is a DSL style API. For example:

```
ec.entity.makeDataLoader().dataTypes(['seed', 'demo']).load()
```

In addition to directly using the API you can load data using the `Tool => Entity => Import` screen in the `tools` component that comes in the default Moqui runtime. You can also load data using the command line with the executable WAR file using the `-load` argument. Here are the command line arguments available for the data loader:

```
-load -------- Run data loader
    -types=<type>[,<type>] -- Data types to load (can be anything, common
```

```
        are: seed, seed-initial, demo, ...)
    -location=<location> ---- Location of data file to load
    -timeout=<seconds> ------ Transaction timeout for each file, defaults
        to 600 seconds (10 minutes)
    -dummy-fks -------------- Use dummy foreign-keys to avoid referential
        integrity errors
    -use-try-insert --------- Try insert and update on error instead of
        checking for record first
    -tenantId=<tenantId> ---- ID for the Tenant to load the data into
```

For example

```
$ java -jar moqui-${version}.war -load -types=seed,demo
```

The entity data XML file must have the `entity-facade-xml` root element which has a **`type`** attribute to specify the type of data in the file, which is compared with the specified types (if loading by specifying types) and only loaded if the type is in the set or if all types are loaded. Under that root element each element name is an entity or service name. For entities each attribute is a field name and for services each attribute is a input parameter.

Here is an example of a entity data XML file:

```
<entity-facade-xml type="seed">
    <moqui.basic.LocalizedMessage original="Example" locale="es"
            localized="Ejemplo"/>
    <moqui.basic.LocalizedMessage original="Example" locale="zh"
            localized="样例"/>
</entity-facade-xml>
```

Here is an example CSV file that calls a service (the same pattern applies for loading entity data):

```
# first line is ${entityName or serviceName},${dataType}
org.moqui.example.ExampleServices.create#Example, demo
# second line is list of field names
exampleTypeEnumId, statusId, exampleName, exampleSize, exampleDate
# each additional line has values for those fields
EXT_MADE_UP, EXST_IN_DESIGN, Test Example Name 3, 13, 2014-03-03 15:00:00
```

Writing Entity XML

The easiest way export entity data to an XML file is to use the `EntityDataWriter`, which you can get with `ec.entity.makeDataWriter()`. Through this interface you can specify the names of entities to export from and various other options, then it does the query and exports to a file (with the `int file(String filename)` method), a directory with one file per entity (with the `int directory(String path)` method), or to a `Writer` object (with the `int writer(Writer writer)` method). All of these methods return an `int` with the number of records that were written.

The methods for specifying options return a self reference to enable chaining calls. These are the methods for the query and export options:

- `entityName(String entityName)`: Specify the name of an entity to query and export. Data is queried and exporting from entities in the order they are added by calling this or `entityNames()` multiple times.
- `entityNames(List<String> entityNames)`: A `List` of entity names to query and export. Data is queried and exporting from entities in the order they are specified in this list and other calls to this or `entityName()`.
- `dependentRecords(boolean dependents)`: If true export dependent records of each record. This dramatically slows down the export so only use it on smaller data sets. See the **Dependent Entities** section for details about what would be included.
- `filterMap(Map<String, Object> filterMap)`: A `Map` of field name, value pairs to filter the results by. Each name/value is only used on entities that have a field matching the name.
- `orderBy(List<String> orderByList)`: Field names to order (sort) the results by. Each name only used on entities with a field matching the name. May be called multiple times. Each entry may be a comma-separated list of field names.
- `fromDate(Timestamp fromDate)`,`thruDate(Timestamp thruDate)`: The from and thru dates to filter the records by, compared with the `lastUpdatedStamp` field which the Entity Facade automatically adds to each entity (unless turned off in the entity definition).

Here is an example of an export of all `OrderHeader` records within a time range plus their dependents:

```
ec.entity.makeDataWriter().entityName("mantle.order.OrderHeader")
    .dependentRecords(true).orderBy(["orderId"]).fromDate(lastExportDate)
    .thruDate(ec.user.nowTimestamp).file("/tmp/TestOrderExport.xml")
```

Another way to export entity records is to do a query and get an `EntityList` or `EntityListIterator` object and call the `int writeXmlText(Writer writer, String prefix, boolean dependents)` method on it. This methods writes XML to the `writer`, optionally adding the `prefix` to the beginning of each element and including `dependents`.

Similar to the entity data import UI you can export data using the `Tool => Entity => Export` screen in the `tools` component that comes in the default Moqui runtime.

Views and Forms for Easy View and Export

A number of tools come together to make it very easy to view and export database data that comes from a number of different tables. We have explored the options for static (XML), dynamic, and database defined entities. In the **User Interface** chapter there is detail about XML Forms, and in particular list forms.

When a `form-list` has **`dynamic`**=`true` and a `${}` string expansion in the `auto-fields-entity`.**`entity-name`** attribute then it will be expanded on the fly as the screen is rendered, meaning a single form can be used to generate tabular HTML or CSV output for any entity given an entity name as a screen parameter.

To make things more interesting results viewed can be filtered generically using a dynamic `form-single` with an `auto-fields-entity` element to generate a search form based on the entity, and an `entity-find` with `search-form-inputs` to do the query based on the entity name parameter and the search parameters from the search form.

Below is an example of these features along with a transition (`DbView.csv`) to export a CSV file. Don't worry too much about all the details for screens, transitions, forms, and rendering options, they are covered in detail in the **User Interface** chapter. This screen definition is an excerpt from the `ViewDbView.xml` screen in the `tools` component that comes by default with Moqui Framework:

```
<screen>
  <parameter name="dbViewEntityName"/>

  <transition name="filter"><default-response url="."/></transition>
  <transition name="DbView.csv">
    <default-response url="."><parameter name="renderMode" value="csv"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>

  <actions>
    <entity-find entity-name="${dbViewEntityName}" list="dbViewList">
      <search-form-inputs/></entity-find>
  </actions>
  <widgets>
    <link url="DbView.csv" text="Get as CSV"/>
    <label text="Data View for: ${dbViewEntityName}" type="h2"/>

    <form-single name="FilterDbView" transition="filter" dynamic="true">
      <auto-fields-entity entity-name="${dbViewEntityName}"
          field-type="find"/>
      <field name="dbViewEntityName"><default-field>
        <hidden/></default-field></field>
      <field name="submitButton"><default-field title="Find">
        <submit/></default-field></field>
    </form-single>

    <form-list name="ViewList" list="dbViewList" dynamic="true">
      <auto-fields-entity entity-name="${dbViewEntityName}"
          field-type="display"/>
    </form-list>
  </widgets>
</screen>
```

While this screen is designed to be used by a user it can also be rendered outside a web or other UI context to generate CSV output to send to a file or other location. If you were to just write a screen for that it would be far simpler, basically just the `parameter` element, the single `entity-find` action, and the simple `form-list` definition. The transitions and the search form would not be needed.

The code to do this through the screen renderer would look something like:

```
ec.context.putAll([pageNoLimit:"true", lastStandalone:"true",
    dbViewEntityName:"moqui.example.ExampleStatusDetail"])
String csvOutput = ec.screen.makeRender()
  .rootScreen("component://tools/screen/Tools/DataView/ViewDbView.xml")
  .renderMode("csv").render()
```

Data Document

A Data Document is assembled from database records into a JSON document or a Java nested Map/List representation of the document.

Below is an example Data Document instance and the `DataDocument*` records that define it. This example a selection from the HiveMind PM project, which is based on Moqui and Mantle. The document is for a project, which is a type of `WorkEffort`.

```
{
    "_index": "hivemind",
    "_type": "HmProject",
    "_id": "HM",
    "_timestamp": "2013-12-27T00:46:07",
   "WorkEffort": {
        "workEffortId": "HM",
        "name": "HiveMind PM Build Out",
        "workEffortTypeEnumId": "WetProject"
    },
    "StatusItem": { "status": "In Progress" },
    "WorkEffortType": { "type": "Project" },
    "Party": [
        {
            "Person": { "firstName": "John", "lastName": "Doe" },
            "RoleType": { "role": "Person - Manager" },
            "partyId": "EX_JOHN_DOE"
        },
        {
            "Person": { "firstName": "Joe", "lastName": "Developer" },
            "RoleType": { "role": "Person - Worker" },
            "partyId": "ORG_BIZI_JD"
        }
    ]
```

}

These are the database records defining the Data Document, in the format of records in an Entity Facade XML file:

```
<moqui.entity.document.DataDocument dataDocumentId="HmProject"
    indexName="hivemind" documentName="Project"
    primaryEntityName="mantle.work.effort.WorkEffort"
    documentTitle="${name}"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortName" fieldNameAlias="name"/>
<!-- this is aliased so we can have a condition on it -->
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="workEffortTypeEnumId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="WorkEffort#moqui.basic.StatusItem:description"
    fieldNameAlias="status"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"
    fieldPath="mantle.work.effort.WorkEffortParty:partyId"/>
<moqui.entity.document.DataDocumentField dataDocumentId="HmProject"

fieldPath="mantle.work.effort.WorkEffortParty:mantle.party.RoleType:description"
    fieldNameAlias="role"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.work.effort.WorkEffort"
    documentAlias="WorkEffort"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="WorkEffort#moqui.basic.StatusItem"
    documentAlias="StatusItem"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.work.effort.WorkEffortParty"
    documentAlias="Party"/>
<moqui.entity.document.DataDocumentRelAlias dataDocumentId="HmProject"
    relationshipName="mantle.party.RoleType" documentAlias="RoleType"/>
<moqui.entity.document.DataDocumentCondition dataDocumentId="HmProject"
    fieldNameAlias="workEffortTypeEnumId" fieldValue="WetProject"/>
<moqui.entity.document.DataDocumentLink dataDocumentId="HmProject"
    label="Edit Project"
    linkUrl="/apps/hm/Project/EditProject?workEffortId=${workEffortId}"/>
```

The top level object (the JSON term, Map in Java) of the Data Document instance has 3 fields that identify the document:

- **`_index`**: The index the document should live in, from the `DataDocument.indexName` field in the document definition
- **`_type`**: The type of document within the index, and the ID that Moqui Framework uses for the DataDocument definition, from the `DataDocument.dataDocumentId` field

- **`_id`**: The ID for a particular Data Document instance, based on the primary key of the primary entity as specified in the `DataDocument.primaryEntityName` field

The top level also contains a **`_timestamp`** field with the date and time the document was generated.

These 4 fields are named the way they are for easy indexing with ElasticSearch, which is the tool used by the Data Search feature which is based on the Data Document feature. These fields, and Data Documents in general, are useful for notifications, integrations, and various things other than just search.

A Data Document definition is made up of these records:

- `DataDocument`: The main record, identified by a **`dataDocumentId`** and contains the index name, document name (for display purposes)
  - **`primaryEntityName`**: the primary (master) entity for the document that all other entities for document fields relate to and that plain field names belong to
  - **`documentTitle`**: For display purposes, especially in search results and such. Note that the **`documentTitle`** value is expanded using a flattened `Map` from the Data Document, so names of expanded fields must match document field names (or aliases).
- `DataDocumentField`: Each record specifies a field for the document.
  - **`fieldPath`**: The field name, optionally preceded by a colon-separated list of relationship names from the primary entity to the entity the field is on.
  - **`fieldNameAlias`**: Optionally specify a name for the field to use in the document if different from the name of the field on the entity it belongs to. The field name in the document must be unique for the entire document, not just within the entity the field belongs to. This is true whether the entity field name or an alias is used. The reasons for this are: this is the alias used in the query to get the data for the document from the database and to facilitate parametric searching.
- `DataDocumentRelAlias`: Use these records to produce a cleaner document by specifying an alias for relationships in **`fieldPath`** fields, and for the **`primaryEntityName`**.
- `DataDocumentCondition`: These records constrain the query that gets data for the document from the database. In the example above this is used to constrain the query to only get `WorkEffort` records with the `WetProject` type so it only includes projects.
- `DataDocumentLink`: In search results and other user and system interfaces it is useful to have a link to where more information about the document, especially the primary entity in it, is available. Use these records to specify such links. Note that the **`linkUrl`** value is expanded using a flattened Map from the Data Document, so names of expanded fields must match document field names (or aliases).

In the top level object of the example document there is a `WorkEffort` object for the primary entity in the document. There will always be an object like this in the document and its name will be the name of the primary entity. It will be the literal value of the `DataDocument.primaryEntityName` field unless it is aliased in a `DataDocumentRelAlias`

record, which is why in this document that named of the object is `"WorkEffort"` and not `"mantle.work.effort.WorkEffort"`.

All `DataDocumentField` records with a **`fieldPath`** with plain field names (no colon-separated relationship prefix) map to fields on the primary entity and will be included in the primary entity's object in the document.

All document fields with a colon-separated relationship name prefix will result in other entries in the top level document object (Map) with the entry key as the relationship name or the alias for the relationship name if one is configured. The value for that entry will be an object/Map if it is a type one relationship, or an array of objects (in Java a List of Maps) if it is a type many relationship.

The same pattern applies when there is more than one colon-separated relationship name in a **`fieldPath`**. The object/Map entries will be nested as needed to follow the path to the specified field. An example of this from the HmProject document example above is the `"mantle.work.effort.WorkEffortParty:mantle.party.RoleType:description"` **`fieldPath`** value. Note that the two relationship names are aliased to exclude the package names, and the field is aliased to be `role` instead of `description`. The result is this part of the JSON document:

```
{ "Party": [ { "RoleType": { "role": "Person - Manager" } } ] }
```

The JSON syntax for an object (Map) is curly braces ({ }) and for an array (List) is square braces ([ ]). So what we have above is the top-level object with a `Party` entry whose value is an array with an object in it that has a `RoleType` entry whose value is an object with a single entry with the key **`role`** and the value is from the `RoleType`**`.description`** entity field. The reason the **`description`** field is aliased as **`role`** is the one described above in the description for the `DataDocumentField`**`.fieldNameAlias`** field: each field in a Data Document must have a unique name across the entire document.

There are a few ways to generate a Data Document from data in a database. The most generally useful approach is the Data Feed described below, but you can also get it through an API call that looks like this:

```
List<Map> docMapList = ec.entity.getDataDocuments(dataDocumentId,
        condition, fromUpdateStamp, thruUpdatedStamp)
```

In the List returned each Map represents a Data Document. The condition, fromUpdatedStamp and thruUpdatedStamp parameters can all be null, but if specified are used as additional constraints when querying the database. The condition should use the field alias names for the fields in the document. To see if any part of the document has changed in a certain time range the *UpdatedStamp parameters are used to look for any record in any of the entities with the automatically added **`lastUpdatedStamp`** field in the from/thru range.

The Map for a Data Document is structured the same way as the example JSON document above. The ElasticSearch API supports this Map form of a document, but in some cases you

will want it as a JSON String. To create a JSON String from the Map in Groovy use a simple statement like this:

```
String docString = groovy.json.JsonOutput.toJson(docMap)
```

If you want a more friendly human-readable version of the JSON String do this:

```
String prettyDocString = groovy.json.JsonOutput.prettyPrint(docString)
```

To go the other way (get a Map representation from a JSON String) use a statement like this:

```
Map docMap = (Map) new groovy.json.JsonSlurper().parseText(docString)
```

Data Feed

A Data Feed is a configurable way to push Data Documents to a service or group multiple documents for retrieval through an API call.

The example below is a push feed (**dataFeedTypeEnumId**="DTFDTP_RT_PUSH") to send documents to the HiveMind.SearchServices.indexAndNotify#HiveMindDocuments service when any data in any of the documents is changed in the database through the Moqui Entity Facade. The framework automatically keeps track of push Data Feeds and the entities that are part of the Data Documents associated with them to look for changes as create, update, and delete operations are done. This is an efficient way to get updated Data Documents in real time.

Here is an example of entity-facade-xml for the records to configure a push Data Feed:

```
<moqui.entity.feed.DataFeed dataFeedId="HiveMindSearch"
    dataFeedTypeEnumId="DTFDTP_RT_PUSH" feedName="HiveMind Search"
feedReceiveServiceName="HiveMind.SearchServices.indexAndNotify#HiveMindDocuments"/>
<moqui.entity.feed.DataFeedDocument dataFeedId="HiveMindSearch"
    dataDocumentId="HmProject"/>
<moqui.entity.feed.DataFeedDocument dataFeedId="HiveMindSearch"
    dataDocumentId="HmTask"/>
```

Each DataFeedDocument record associates a DataDocument record to the DataFeed record to be included in the feed.

On a side note, when you have data you want to index that is loaded through a XML data file as part of the load process and it may be loaded before the Data Feed is loaded an activated, you can include an element for a ServiceTrigger record and the Service Facade will call the service during the load process to index for the feed. Here is an example of that:

```
<moqui.entity.ServiceTrigger serviceTriggerId="HM_SEARCH_INIT"
    statusId="SrtrNotRun" mapString="[dataFeedId:'HiveMindSearch']"
    serviceName="org.moqui.impl.EntityServices.index#DataFeedDocuments"/>
```

The DataFeed example above is for a push Data Feed. To setup a feed for manual pull just set **dataFeedTypeEnumId**="DTFDTP_MAN_PULL" on the DataFeed record. Any type of Data

Feed can be retrieved manually, but with this type the feed will not be automatically run. To get the documents for any feed through the API use a statement like this:

```
List<Map> docList = ec.entity.getDataFeedDocuments(dataFeedId,
        fromUpdateStamp, thruUpdatedStamp)
```

Data Search

The Data Search feature in Moqui Framework is based on ElasticSearch (http://www.elasticsearch.org). This is a distributed text search tool based on Apache Lucene. ElasticSearch uses JSON documents as the artifact to search, and each named field in a JSON document is a facet for searching. The Data Document feature produces documents with 4 special fields that ElasticSearch uses, as described in the Data Document section (**_index**, **_type**, **_id**, and **_timestamp**).

There are two main touch points for Data Search: indexing and searching. The service for indexing in the framework is org.moqui.impl.EntityServices.index#DataDocuments. This service implements the `org.moqui.EntityServices.receive#DataFeed` interface and accepts all parameters from the interface but only uses the **documentList** parameter, which is the list of Data Documents to index with ElasticSearch.

It also has one other parameter, **getOriginalDocuments**, which when set to true the service will populate and return **originalDocumentList**, a list of the previously indexed version of any matching existing documents from ElasticSearch. The service always returns a **documentVersionList** parameter with a list of the version number for each document in the original list after the index is done to show how many times each document has been updated in the index.

The example in the previous section used an application-specific service to receive the push Data Feed, so here is an example of a push Data Feed configuration that uses the indexing service that is part of the framework:

```
<moqui.entity.feed.DataFeed dataFeedId="PopCommerceSearch"
    dataFeedTypeEnumId="DTFDTP_RT_PUSH" feedName="PopCommerce Search"
    feedReceiveServiceName="org.moqui.impl.EntityServices.index#DataDocuments"/>
<moqui.entity.feed.DataFeedDocument dataFeedId="PopCommerceSearch"
    dataDocumentId="PopcProduct"/>
```

You can also use the ElasticSearch API directly to index documents, either Data Documents produced by the Entity Facade or any JSON document you want to search. For more complete information see the ElasticSearch documentation. Here is an example of indexing a JSON document in nested Map form with the **_index**, **_type**, and **_id** entries set:

```
IndexResponse response = ec.elasticSearchClient
    .prepareIndex(document._index, document._type, document._id)
    .setSource(document).execute().actionGet()
```

To search Data Documents use the `org.moqui.impl.EntityServices.search#DataDocuments` service, like this:

```
<service-call name="org.moqui.impl.EntityServices.search#DataDocuments"
    out-map="context" in-map="context + [indexName:'popc']"/>
```

Note that in this example the **`queryString`**, **`pageIndex`**, and **`pageSize`** parameters come from the search form and get into the `context` from request parameters. The parameters for this service are:

- **`queryString`**: the search query string that will be passed to the Lucene classic query parser, for documentation see: http://lucene.apache.org/core/4_8_1/queryparser/org/apache/lucene/queryparser/classic/package-summary.html
- **`documentType`**: the ElasticSearch document type, matches the **`_type`** field in the document and the `DataDocument.dataDocumentId`; examples of this from previous sections include `PopcProduct` and `HmProject`
- **`pageIndex`**, **`pageSize`**: these are the standard pagination parameters for Moqui XML list forms so this service can be easily used with them; only **`pageSize`** results will be returned and starting at the **`pageIndex`** * **`pageSize`** index in the results
- **`flattenDocument`**: default `false`, if set to `true` each document in the form of a nested Map result (object form, JSON document being the text form) will be flattened into a single flat Map with name/value pairs taken from all of the nested Maps and Lists of Maps; later values in the document will override earlier values if the same Map entry key is found more than once (see the `StupidUtilities.flattenNestedMap()` method)

The service returns a **`documentList`** parameter, which is a List of Maps, each Map representing a Data Document. It also returns the various **`documentList*`** parameters that are part of the pagination pattern for Moqui XML list forms (*Count, *PageIndex, *PageSize, *PageMaxIndex, *PageRangeLow, and *PageRangeHigh). These are used when rendering a list form, and can be used for other purposes where useful as well.

In addition to this service you can also retrieve results directly through the ElasticSearch API. Note that there are two main steps, the search to get back the 3 identifying fields of each document, and then a multi-get to get all of the documents. In this example we get each document as a Map (the **`getSourceAsMap()`** method), and the ElasticSearch API also supports getting each as a JSON document (the **`getSourceAsString()`** method).

```
SearchHits hits =
ec.elasticSearchClient.prepareSearch().setIndices(indexName)
   .setTypes(documentType).setQuery(QueryBuilders.queryString(queryString))
   .setFrom(fromOffset).setSize(sizeLimit).execute().actionGet().getHits()
if (hits.getTotalHits() > 0) {
    MultiGetRequestBuilder mgrb = ec.elasticSearchClient.prepareMultiGet()
    for (SearchHit hit in hits)
        mgrb.add(hit.getIndex(), hit.getType(), hit.getId())
    Iterator mgirIt = mgrb.execute().actionGet().iterator()
    while(mgirIt.hasNext()) {
```

```
        MultiGetItemResponse mgir = mgirIt.next()
        Map document = mgir.getResponse().getSourceAsMap()
        documentList.add(document)
    }
}
```

In addition to indexing and searching another aspect of ElasticSearch to know about is the deployment options. By default Moqui Framework has an embedded node of ElasticSearch running in the same JVM for fast, convenient access. A remote ElasticSearch server can also be used.

The easiest distributed deployment mode is to have each Moqui application server be a node in the ElasticSearch cluster, and if you have separate ES nodes with actual search data persisted on them then set the app server ES nodes to not persist any data. With that approach results may be aggregated on the app servers, but actual searches against index data will be done on the other servers in the cluster.

404 - Page Not Found

(not really, this page is intentionally blank for layout reasons; to make it less blank sponsor this book and see your ad here!)

6. Logic and Services

Service Definition

With Moqui Framework the main unit of logic is the service. This is a service-oriented architecture with services used as internal, granular units of logic as well as external, coarse aggregations of logic. Moqui services are:

- transactional
- secure (both authentication and authorization, plus tarpit for velocity limits)
- validated (data types and various constraints for input parameters)
- implemented with any of a wide variety of languages and tools including scripting languages, Java methods, an even an Apache Camel endpoint
- run from a local or remote caller
- run synchronously, asynchronously, or on a schedule
- a source of triggers at various phases of execution to run other services using service event-condition-action (SECA) rules
- optionally restricted to a single running instance with a database semaphore

Services are defined in a services XML file using the service element. A service name is composed of a path, a verb and a noun in this structure: `"${path.verb#noun}"`. Note that the noun is optional in a service definition, and in a service name the hash (#) between the verb and noun is also optional. Here is an example, the `mantle.party.PartyServices.create#Person` service (from Mantle Business Artifacts):

```
<service verb="create" noun="Person">
    <in-parameters>
       <parameter name="partyId"/>
       <auto-parameters entity-name="mantle.party.Person" include="nonpk"/>
       <parameter name="firstName" required="true"/>
       <parameter name="lastName" required="true"/>
       <parameter name="roleTypeId"/>
    </in-parameters>
    <out-parameters><parameter name="partyId"/></out-parameters>
    <actions>
       <service-call name="create#mantle.party.Party" out-map="context"
               in-map="[partyId:partyId, partyTypeEnumId:'PtyPerson']"/>
```

```
        <service-call name="create#mantle.party.Person" in-map="context"/>
        <if condition="roleTypeId">
            <service-call name="create#mantle.party.PartyRole"
                    in-map="[partyId:partyId, roleTypeId:roleTypeId]"/>
        </if>
    </actions>
</service>
```

The only attribute that is required for a service is **verb**, though use of a **noun** is generally recommended. The **type** attribute is commonly used, but defaults to `"inline"` just like the service above which has an `actions` element containing the service implementation. For other types of services, i.e. other ways of implementing a service, the **location** and optional **method** attributes are used to specify what to run.

The example above has `in-parameters` including individual `parameter` elements and an `auto-parameters` element to pull in all non-PK fields on the `mantle.party.Person` entity. It also has one `out-parameter`, a `partyId` that in this case is either generated if no `partyId` is passed as an input parameter or the passed in value is simply passed through.

The `actions` element has the implementation of the service, containing a XML Actions script. In this case it calls a couple of services, and then conditionally calls a third if a `roleTypeId` is passed in. Note that there is no explicit setting of the `partyId` output parameter (in the `result` Map) as the Service Facade automatically picks up the context value for each declared output parameter after the service implementation is run to populate the output/results Map.

These are the attributes available on the `service` element:

- **verb**: This can be any verb, and will often be one of: create, update, store, delete, or find. The full name of the service will be: "${path}.${verb}#${noun}". The verb is required and the noun is optional so if there is no noun the service name will be just the verb.
- **noun**: For entity-auto services this should be a valid entity name. In many other cases an entity name is the best way to describe what is being acted on, but this can really be anything.
- **type**: The service type specifies how the service is implemented. The default available options include: `inline`, `entity-auto`, `script`, `java`, `interface`, `remote-xml-rpc`, `remote-json-rpc`, and `camel`. Additional types can be added by implementing the `org.moqui.impl.service.ServiceRunner` interface and adding a `service-facade.service-type` element in the Moqui Conf XML file. The default value is `inline` meaning the service implementation is under the `service.actions` element.
- **location**: The location of the service. For scripts this is the Resource Facade location of the file. For Java class methods this is the full class name. For remote services this is the URL of the remote service. Instead of an actual location can also refer to a pre-defined location from the `service-facade.service-location` element in the Moqui Conf XML file. This is especially useful for remote service URLs.
- **method**: The method within the location, if applicable to the service type.

- **`authenticate`**: If not set to false (is true by default) a user much be logged in to run this service. If the service is running in an ExecutionContext with a user logged in that will qualify. If not then either a "authUserAccount" parameter or the "authUsername" and "authPassword" parameters must be specified and must contain valid values for a user of the system. An "authTenantId" parameter may also be specified to authenticate the user in a specific tenant instance. If specified will be used to run the service with that as the context tenant. Can also be set to anonymous-all or anonymous-view and not only will authentication not be required, but this service will run as if authorized (using the _NA_ UserAccount) for all actions or for view-only.
- **`allow-remote`**: Defaults to false meaning this service cannot be called through remote interfaces such as JSON-RPC and XML-RPC. If set to true it can be. Before settings to true make sure the service is adequately secured (for authentication and authorization).
- **`validate`**: Defaults to true. Set to false to not validate input parameters, and not automatically remove unspecified parameters.
- **`transaction`**:
  - `ignore`: Don't do anything with transactions (if one is in place use it, if no transaction in place don't begin one).
  - `use-or-begin`: Use active transaction or if no active transaction begin one. This is the default.
  - `force-new`: Always begin a new transaction, pausing/resuming the active transaction if there is one.
  - `cache`: Like `use-or-begin` but with a write-through per-transaction cache in place (works even if active TX is in place). See notes and warnings in the JavaDoc comments of the TransactionCache class for details.
  - `force-cache`: Like `force-new` with a transaction cache in place like the `cache` option.
- **`transaction-timeout`**: The timeout for the transaction, in seconds. This value is only used if this service begins a transaction (`force-new`, `force-cache`, or `use-or-begin` or `cache` and there is no other transaction already in place).
- **`semaphore`**: Intended for use in long-running services (usually scheduled). This uses a record in the database to lock the service so that only one instance of it can run against a given database at any given time. Options include `none` (default), `fail`, and `wait`.
- **`semaphore-timeout`**: When waiting how long before timing out, in seconds. Defaults to 120s.
- **`semaphore-sleep`**: When waiting how long to sleep between checking the semaphore, in seconds. Defaults to 5s.
- **`semaphore-ignore`**: Ignore existing semaphores after this time, in seconds. Defaults to 3600s (1 hour).

The input and output of a service are each a Map with name/value entries. Input parameters are specified with the `in-parameters` element, and output parameters with the `out-parameters` element. Under these elements use the `parameter` element to define a single

parameter, and the `auto-parameters` element to automatically define parameters based on primary key (pk), non-primary key (nonpk) or all fields of an entity.

An individual `parameter` element has attributes to define it including:

- **`name`**: The name of the parameter, matches against the key of an entry in the parameters Map passed into or returned from the service.
- **`type`**: The type of the attribute, a full Java class name or one of the common Java API classes (including String, Timestamp, Time, Date, Integer, Long, Float, Double, BigDecimal, BigInteger, Boolean, Object, Blob, Clob, Collection, List, Map, Set, Node).
- **`required`**: Defaults to `false`, set to `true` for the parameter to be required. Can also set to `disabled` to behave the same as if the parameter did not exist, useful when overriding a previously defined parameter.
- **`allow-html`**: Applies only to String fields. Only checked for incoming parameters (meant for validating input from users, other systems, etc). Defaults to `none` meaning no HTML is allowed (will result in an error message). If some HTML is desired then use `safe` which will follow the rules in the antisamy-esapi.xml file. This should be safe for both internal and public users. In rare cases when users are trusted or it is not a sensitive field the `any` option may be used to not check the HTML content at all.
- **`format`**: Used only when the parameter is passed in as a String but the type is something other than String to convert to that type. For date/time uses standard Java SimpleDateFormat strings.
- **`default`**: The field or expression specified will be used for the parameter if no value is passed in (only used if required=false). Like default-value but is a field name or expression instead of a text value. If both this and default-value are specified this will be evaluated first and only if empty will default-value be used.
- **`default-value`**: The text value specified will be used for the parameter if no value is passed in (only used if required=false). If both this and default are specified default will be evaluated first and this will only be used if default evaluates to an empty value.
- **`entity-name`**: Optional name of an entity with a field that this parameter is associated with.
- **`field-name`**: Optional field name within the named entity that this parameter is associated with. Most useful for form fields defined automatically from the service parameter. This is automatically populated when parameters are defined automatically with the `auto-parameters` element.

For parameter object types that contain other objects (such as List, Map, and Node) the parameter element can be nested to specify what to expect (and if applicable, validate) within the parameter object.

In addition to the **`required`** attribute, validations can be specified for each parameter with these sub-elements:

- `matches`: Validate the current parameter against the regular expression specified in the **`regexp`** attribute.
- `number-range`: Validate the number within the **`min`** and **`max`** range.

- `number-integer`: Validate that the parameter is an integer.
- `number-decimal`: Validate that the parameter is a decimal number.
- `text-length`: Validate that the length of the text is within the **`min`** and **`max`** range.
- `text-email`: Validate that the text is a valid email address.
- `text-url`: Validate that the text is a valid URL.
- `text-letters`: Validate that the text contains only letters.
- `text-digits`: Validate that the text contains only digits.
- `time-range`: Validate that the date/time is within the **`before`** and **`after`** range, using the specified **`format`**.
- `credit-card`: Validate that the text is a valid credit card number using Luhn MOD-10 and if specified for the given card **`types`**.

Validation elements can be combined using the `val-or` and `val-and` elements, or negated using the `val-not` element.

When a XML Form field is based on a service parameter with validations certain validations are automatically validated in the browser with JavaScript, including **`required`**, `matches`, `number-integer`, `number-decimal`, `text-email`, `text-url`, and `text-digits`.

Now that your service is defined, essentially configuring the behavior of the Service Facade when the service is called, it is time to implement it.

Service Implementation

Some service types have local implementations while others have no implementation (`interface`) or the service definition is a proxy for something else and the location refers to an external implementation (`remote-xml-rpc`, `remote-json-rpc`, and `camel`). The remote and Apache Camel types are described in detail in the System Interfaces chapter.

Service Scripts

A script is generally the best way to implement a service, unless an automatic implementation for entity CrUD operations will do. Scripts are reloaded automatically when their cache entry is clear, and in development mode these caches expire in a short time by default to get updates automatically.

Scripts can run very efficiently, especially Groovy scripts which compile to Java classes at runtime and are cached in their compiled form so they can be run quickly. XML Actions scripts are transformed into a Groovy script (see the `XmlActions.groovy.ftl` file for details) and then compiled and cached, so have a performance profile just like a plain Groovy script.

Any script that the Resource Facade can run can be used as a service implementation. See the **Rendering Templates and Running Scripts** section for details. In summary the scripts supported by default are Groovy, XML Actions, and JavaScript. Any scripting language can

be supported through the `javax.script` or Moqui-specific interfaces. Here is an example of a service implemented with a Groovy script, defined in the `org.moqui.impl.EmailServices.xml` file:

```
<service verb="send" noun="EmailTemplate" type="script"
        location="classpath://org/moqui/impl/sendEmailTemplate.groovy">
    <implements service="org.moqui.EmailServices.send#EmailTemplate"/>
</service>
```

In this case the **`location`** is a classpath location, but any location supported by the Resource Facade can be used. See the **Resource Locations** section for details on how to refer to files within components, in the local file system, or even at general URLs.

At the beginning of a script all of the input parameters passed into the service, or set through defaults in the service definition, will be in the context as fields available for use in the script. As with other artifacts in Moqui there is also an `ec` field with the current `ExecutionContext` object.

Note that the script has a context isolated from whatever called it using the `ContextStack.pushContext()` and **`popContext()`** methods meaning not only do fields created in the context not persist after the service is run, but the service does not have access to the context of whatever called it even though it may be running locally and within the same `ExecutionContext` as whatever called it.

For convenience there is a `result` field in the context that is of type `Map<String, Object>`. You can put output parameters in this `Map` to return them, but doing so is not necessary. After the script is run the script service runner looks for all output parameters defined on the service in the context and adds them to the results. The script can also return (evaluate to) a `Map` object to return results.

Inline Actions

The service definition example near the beginning of this chapter shows a service with the default service type, `inline`. In this case the implementation is in the `service.actions` element, which contains a XML Actions script. It is treated just like an external script referred to by the service location but for simplicity and to reduce the number of files to work with it can be inline in the service definition.

Java Methods

A service implementation can also be a Java method, either a class (static) method or an object method. If the method is not static then the service runner creates a new instance of the object using the default (no arguments) constructor.

The method must take a single `ExecutionContext` argument and return a `Map<String, Object>`, so the signature of the method would be something like:

```
Map<String, Object> myService(ExecutionContext ec)
```

Entity Auto Services

With `entity-auto` type services you don't have to implement the service, the implementation is automatic based on the **`verb`** and **`noun`** attribute values. The verb can be `create`, `update`, `delete`, or `store` (which is a create if the record does not exist, update if it does). The noun is an entity name, either a full name with the package or just the simple entity name with no package.

Entity Auto services can be implicitly (automatically) defined by just calling a service named like `${verb}#${noun}` with no path (package or filename). For example:

```
ec.service.sync().name("create", "moqui.example.Example")
        .parameters([exampleName:'Test Example']).call()
```

When you define a service and use the `entity-auto` implementation you can specify which input parameters to use (must match fields on the entity), whether they are required, default values, etc. When you use an implicitly defined entity auto service it determines the behavior based on what is passed into the service call. In the example above there is no **`exampleId`** parameter passed in, and that is the primary key field of the `moqui.example.Example` entity, so it automatically generates a sequenced ID for the field, and returns it as an output parameter.

For `create` operations in addition to automatically generating missing primary sequenced IDs it will also generate a secondary sequenced ID if the entity has a 2-part primary key and one is specified while the other is missing. There is also special behavior if there is a **`fromDate`** primary key field that is not passed in, it will use the now Timestamp to populate it.

The pattern for is `update` to pass in all primary key fields (this is required) and any non-PK field desired. There is special behavior for `update` as well. If the entity has a **`statusId`** field and a `statusId` parameter is passed in that is different then it automatically returns the original (DB) value in the `oldStatusId` output parameter. Whenever the entity has a statusId field it also returns a `statusChanged` `boolean` parameter which is true if the parameter is different from the original (DB) value, false otherwise. Entity auto services also enforce valid status transitions by checking for the existing of a matching `moqui.basic.StatusFlowTransition` record. If no valid transition is found it will return an error.

Add Your Own Service Runner

To add your own service runner, with its own service type, implement the `org.moqui.impl.service.ServiceRunner` interface and add a `service-facade.service-type` element in the Moqui Conf XML file.

The `ServiceRunner` interface has 3 methods to implement:

```
ServiceRunner init(ServiceFacadeImpl sfi);
Map<String, Object> runService(ServiceDefinition sd,
        Map<String, Object> parameters) throws ServiceException;
void destroy();
```

Here is an example of a `service-facade.service-type` element from the `MoquiDefaultConf.xml` file:

```
<service-type name="script"
    runner-class="org.moqui.impl.service.runner.ScriptServiceRunner"/>
```

The `service-type.name` attribute matches against the `service.type` attribute, and the `runner-class` attribute is simply the class that implements the `ServiceRunner` interface.

Calling Services and Scheduling Jobs

There are DSL-style interfaces available through the `ServiceFacade` (`ec.getService()`, or in Groovy `ec.service`) that have options applicable to the various ways of calling a service. All of these service call interfaces have `name()` methods to specify the service name, and `parameter()` and `parameters()` methods to specify the input parameters for the service. These and other methods on the various interfaces return an instance of themselves so that calls can be chained. Most have some variation of a `call()` method to actually call the service.

For example:

```
Map ahp = [visitId:ec.user.visitId, artifactType:artifactType, …]
ec.service.async().name("create", "moqui.server.ArtifactHit")
        .parameters(ahp).call()
Map result = ec.service.sync()
        .name("org.moqui.impl.UserServices.create#UserAccount")
        .parameters(params).call()
```

The first service call is to an implicitly defined entity CrUD service to create a `ArtifactHit` record asynchronously. Note that for `async()` the `call()` method returns nothing and in this case the service call results are ignored. The second is a synchronous call to a defined service with a `params` input parameter `Map`, and because it is a `sync()` call the `call()` method returns a `Map` with the results of the service call.

Beyond these basic methods each interface for different ways of calling a service has methods for applicable options, including:

- **`sync()`**: Call the service synchronously and return the results.
  - **`requireNewTransaction(boolean requireNewTransaction)`**: If true suspend/resume the current transaction (if a transaction is active) and begin a new transaction for the scope of this service call.
  - **`multi(boolean mlt)`**: If true expect multiple sets of parameters passed in a single map, each set with a suffix of an underscore and the row of the number, i.e. something like "userId_8" for the userId parameter in the 8th row.
  - **`disableAuthz()`**: Disable authorization for the current thread during this service call.
- **`async()`**: Call the service asynchronously and ignore the results, get back a `ServiceResultWaiter` object to wait for the results, or pass in an implementation of the `ServiceResultReceiver` interface to receive the results when the service is complete.
  - **`maxRetry(int maxRetry)`**: Set the maximum number of times to retry running the service when there is an error.
  - **`resultReceiver(ServiceResultReceiver resultReceiver)`**: Specify the object that implements the `ServiceResultReceiver` interface to use for the service call. Use the **`call()`** method after this to actually call the service.
  - **`callWaiter()`**: Calls the service (like **`call()`**) and returns a `ServiceResultWaiter` instance used to wait for and receive the service results.
- **`schedule()`**: Setup call(s) to the service on a schedule.
  - **`jobName(String jobName)`**: Name of the job. If specified repeated schedules with the same jobName will use the same underlying job.
  - **`startTime(long startTime)`**: Time to first run this service (in milliseconds from epoch).
  - **`count(int count)`**: Number of times to repeat.
  - **`endTime(long endTime)`**: Time that this service schedule should expire (in milliseconds from epoch).
  - **`interval(int interval, TimeUnit intervalUnit)`**: A time interval specifying how often to run this service. The `intervalUnit` parameter is a value from the enumeration `ServiceCall.IntervalUnit { SECONDS, MINUTES, HOURS, DAYS, WEEKS, MONTHS, YEARS }`
  - **`cron(String cronString)`**: A string in the same format used by cron to define a recurrence.
  - **`maxRetry(int maxRetry)`**: Maximum number of times to retry running this service.
- **`special()`**: Register the current service to be called when the current transaction is either committed (use **`registerOnCommit()`**) or rolled back (use **`registerOnRollback()`**). This interface does not have a **`call()`** method.

The asynchronous and scheduled service calls are run using Quartz Scheduler. To use Quartz directly get an instance of the `org.quartz.Scheduler` object using the **`ec.getServices().getScheduler()`** method. For details on what you can do with Quartz, see the documentation at http://quartz-scheduler.org/documentation.

The Quartz job store is in memory by default and can be put in a database using the Quartz JDBC job store or the Moqui `EntityJobStore` which uses the Entity Facade for persistence for easier configuration and deployment. When using the RAM job store or to make sure that a certain job is scheduled use the `XMLSchedulingDataProcessorPlugin` from Quartz by configuring it in the `quartz.properties` file. Part of the configuration is the filename of the XML file that has the job settings, `quartz_data.xml` by default in Moqui.

Here is an example of a schedule, which is in place by default in Moqui:

```
<schedule>
    <job>
        <name>clean_ArtifactData_single</name>
        <group>org.moqui.impl.ServerServices.clean#ArtifactData</group>
        <job-class>org.moqui.impl.service.ServiceQuartzJob</job-class>
        <job-data-map><entry><key>daysToKeep</key><value>90</value>
            </entry></job-data-map>
    </job>
    <trigger>
        <cron>
            <name>clean_ArtifactData_daily</name>
            <group>ServerServices</group>
            <job-name>clean_ArtifactData_single</job-name>
            <job-group>org.moqui.impl.ServerServices.clean#ArtifactData
                </job-group>
            <!-- trigger every night at 2:00 am -->
            <cron-expression>0 0 2 * * ?</cron-expression>
            <!-- for testing, run every 2 minutes:
                <cron-expression>0 0/2 * * * ?</cron-expression> -->
        </cron>
    </trigger>
</schedule>
```

The most important elements are `job.job-class` which should be set to `org.moqui.impl.service.ServiceQuartzJob` for Moqui Service Facade jobs, and `job.group` which is the service name. Note that `trigger.job-name` must match `job.name`, and `trigger.job-group` must match `job.group`.

The Tools app in default runtime that comes with Moqui Framework has some screens for viewing, pausing, resuming, and canceling Quartz jobs. The screens include a summary of scheduler details, a history of jobs run, and admin for current jobs and triggers. These screens are under the Tools => Service => Scheduler screen.

Service ECA Rules

An ECA (event-condition-action) rule is a specialized type of rule to conditionally run actions based on events. For Service ECA (SECA) rules the events are the various phases of executing a service, and these are triggered for all service calls.

Service ECAs are meant for triggering business processes and for extending the functionality of existing services that you don't want to, or can't, modify. Service ECAs should NOT generally be used for maintenance of data derived from other entities, Entity ECA rules are a much better tool for that.

Here is an example of an SECA rule from the `AccountingInvoice.secas.xml` file in Mantle Business Artifacts that calls a service to create invoices for orders when a shipment is packed:

```
<seca service="update#mantle.shipment.Shipment" when="post-service">
    <condition><expression>
        statusChanged &amp;&amp; statusId == 'ShipPacked'
    </expression></condition>
    <actions><service-call
        name="mantle.account.InvoiceServices.create#SalesShipmentInvoices"
        in-map="context + [statusId:'InvoiceFinalized']"/></actions>
</seca>
```

The required attributes on the `seca` element are **`service`** with the service name, and **`when`** which is the phase within the service call. These two attributes together make up the event that triggers the SECA rule. There is also a **`run-on-error`** attribute which defaults to `false` and if set to `true` the SECA rule will be triggered even if there is an error in the service call.

The options for the **`when`** attribute include:

- `pre-auth`: Runs before authentication and authorization checks, but after the `authUsername`, `authPassword` and `authTenantId` parameters are used and specified user logged in; useful for any custom behavior related to authc or authz
- `pre-validate`: Runs before input parameters are validated; useful for adding or modifying parameters before validation and data type conversion
- `pre-service`: Runs before the service itself is run; best place for general things to be done before running the service
- `post-service`: Runs just after the service is run; best place for general things to be done after the service is run and independent of the transaction
- `post-commit`: Runs just after the commit would be done, whether it is actually done or not (depending on service settings and existing TX in place, etc); to run something on the actual commit use the `tx-commit` option
- `tx-commit`: Runs when the transaction the service is running in is successfully committed. Gets its data after the run of the service so will have the output/results of the service run as well as the input parameters.
- `tx-rollback`: Runs when the transaction the service is running in is rolled back. Gets its data after the run of the service so will have the output/results of the service run as well as the input parameters.

When the actions run the context will be whatever context the service was run in, plus the input parameters of the service for convenience in using them. If **`when`** is before the service itself is run there will be a context field called `parameters` with the input parameters `Map` in it that you can modify as needed in the ECA actions. If **`when`** is after the service itself the

`parameters` field will contain the input parameters and a `results` field will contain the output parameters (results) that also may be modified.

The `condition` element is the same condition as used in XML Actions and may contain `expression` and `compare` elements, combined as needed with `or`, `and`, and `not` elements.

The `actions` element is the same as `actions` elements in service definitions, screens, forms, etc. It contains a XML Actions script. See the **Overview of XML Actions** section for more information.

Overview of XML Actions

The `xml-actions-${version}.xsd` file has thorough annotations for detailed documentation, this section is just an overview of what is available to help you get started. You can view the annotations through most good XML editors (including the better Java IDEs or IDE plugins), in the XSD file itself, or in the PDF on `moqui.org` that is generated from the XSD file.

Here is a summary of the most important XML Actions elements to be aware of:

`set`	Set a **`field`**, either **`from`** another field or from a **`value`**, optionally specifying the **`type`**, a **`default-value`**, and whether to **`set-if-empty`**.
`if`	Conditionally run the elements directly under the `if` element, or in the `if.then` element. The condition can be in the **`if.condition`** attribute or in `compare` and `expression` elements under the `if.condition` element (combined with `and` or `or` element, negated by the `not` element). For alternate actions use the `else-if` and `else` subelements.
`while`	Repeat the subelements as long as the condition is true. Just like the `if` element the condition can be in the **`if.condition`** attribute or in the `if.condition` element.
`iterate`	Iterate over elements in the given **`list`**, creating a field in the context using the name in the **`entry`** attribute. If the field named in the **`list`** attribute is a `Map`, iterates over the map entries and the key for each entry is put in the context using the name in the **`key`** attribute. Also creates context fields `${entry}_index` and `${entry}_has_next`.
`script`	Run any kind of script the Resource Facade can run at the specified **`location`** or the Groovy script in the text under this element (inline script).

service-call	Call the service specified in the **name** attribute, using the inputs in the **in-map** attribute (which is a Groovy expression, so can use the square-brace [] syntax for an inline Map) or field-map subelements and putting the outputs in the **out-map**. Can optionally be **async** and **include-user-login**. If the service results in an error the simple method will return immediately unless **ignore-error** equals true.
entity-find-one	Find a single record for **entity-name** and put it in an EntityValue object in **value-field** using attributes including **auto-field-map**, **cache**, and **for-update**, and subelements including field-map and select-field.
entity-find	Find records for **entity-name** and put an EntityList object in **list** using attributes including **cache**, **for-update**, **distinct**, **offset**, and **limit**, and subelements including search-form-inputs, date-filter, econdition, econditions, econdition-object, having-econditions, select-field, order-by, limit-range, limit-view, and use-iterator.
entity-find-count	Find the count of the number of records that match the given conditions. Conditions and other application options follow the same structure as the entity-find operation.
entity-make-value	Create a **value-field** entity value object for the given **entity-name** and optionally set fields based on a **map**.
entity-create	Create (**or-update**) a record for the **value-field** entity value.
entity-update	Update the record for the **value-field** entity value.
entity-delete	Delete the record corresponding to the **value-field** entity value.
entity-set	Set fields to **include** (pk, nonpk, or all) on EntityValue object in **value-field** from **map** (defaults to context) with an optional **prefix** and **set-if-empty**.
entity-sequenced-id-primary	For **value-field** of an entity with a single primary key field, populate that primary key field with a sequenced value (the sequence name is the full entity name).
entity-sequenced-id-secondary	For **value-field** of an entity with a two field primary key and one field already populated, populate the other with a secondary sequenced key with the value of the highest existing secondary field for records matching the populated field, plus 1.

`entity-data`	For the given **`mode`**, `load` or `asset` the Entity Facade XML at the specified **`location`**.
`filter-map-list`	Filter the **`list`** and put the results in **`to-list`** if specified or back in **`list`** if not. Use one or more `field-map` or `date-filter` subelements to specify how to filter the list.
`order-map-list`	Order (sort) a **`list`** of `Map` objects by the fields specified in `order-by` subelements.
`message`	Add the text under the `message` element to the Message Facade to the errors list if **`error`**=`true` or the message list otherwise.
`check-errors`	Checks the Message Facade error message list (`ec.message.errors`) and if not empty returns with an error, otherwise does nothing.
`return`	Returns immediately. Can optionally specify a **`message`** to add to the Message Facade errors list if **`error`**=`true` or the message list otherwise.
`log`	Log the **`message`** at the specified **`level`**.

7. User Interface

The main artifact for building user interfaces in Moqui Framework is the XML Screen.

XML Screens are designed to be used with multiple render modes using the same screen definition. This includes various types of text output for user and system interfaces, and code-driven user interfaces in client applications.

To accommodate this design goal most screen elements are render mode agnostic. For elements that are specific to a particular render mode there is a `render-mode` element with subelements designed for specific render modes. To support multiple render mode specific elements in the same screen just put a subelement under the `render-mode` element for each desired type.

In a web-based application a XML Screen is the main way to produce output for incoming requests. The structure of screens makes it easy to support any sort of URL to a screen.

XML Screen

Screens in Moqui are organized in two ways:

- each screen exists in a hierarchy of subscreens
- a screen may be a node in a graph tied to other nodes by transitions

The hierarchy model is used to reference the screen, and in a URL specify which screen to render by its path in the hierarchy. Screens also contain links to other screens (literally a hyperlink or a form submission) that is more like the structure of going from one node to another in a graph through a transition.

Subscreens

The subscreen hierarchy is primarily used to dynamically include another screen, a subscreen or child screen. The subscreens of a screen can also be used to populate a menu.

When a screen is rendered it is done with a root screen and a list of screen names.

The root screen is configured per webapp in the Moqui Conf XML file with the `moqui-conf.webapp-list.webapp.root-screen` element. Multiple root screens can be configured per webapp based on a hostname pattern, providing a convenient means of virtual hosting within a single webapp. Note that there is no root screen specified in the MoquiDefaultConf.xml file, so it needs to be specified in conf file specified at runtime.

You should have at least one catchall `root-screen` element meaning that the **`host`** is set to the regular expression `".*"`. See the sample runtime conf files, such as the MoquiDevConf.xml file, for an example.

If the list of subscreen names does not reach a leaf screen (with no subscreens) then the default subscreen, specified with the `screen.subscreens.default-item` attribute will be used. Because of this any screen that has subscreens should have a default subscreen.

There are three ways to add subscreens to a screen:

1. for screens within a single application, by directory structure: create a directory in the directory where the parent screen is named the same as the parent screen's filename and put XML Screen files in that directory (**`name`**=filename up to .xml, **`title`**=`screen.default-title`, **`location`**=parent screen minus filename plus directory and filename for subscreen)
2. for including screens that are part of another application, or shared and not in any application, use the `subscreens-item` element below the `screen.subscreens` element
3. for adding screens, removing screens, or changing order and title of screens of a separate application add a record in the `moqui.screen.SubscreensItem` entity

For #1 a directory structure would look something like this (from the Example application):

- `ExampleApp.xml`
- `ExampleApp`
  - `Feature.xml`
  - `Feature`
    - `FindExampleFeature.xml`
    - `EditExampleFeature.xml`
  - `Example.xml`
  - `Example`
    - `FindExample.xml`
    - `EditExample.xml`

The pattern to notice is that if there is are subscreens there should be a directory with the same name as the XML Screen file, just without the .xml extension. The `Feature.xml` file is an example of a screen with subscreens, whereas the `FindExampleFeature.xml` has no subscreens (it is a leaf in the hierarchy of screens).

For approach #2 the `subscreens-item` element would look something like this element from the `apps.xml` file used to mount the Example app's root screen:

```
<subscreens-item name="example" menu-title="Example" menu-index="8"
    location="component://example/screen/ExampleApp.xml"/>
```

For #3 the record in the database in the `SubscreensItem` entity would look something like this (an adaptation of the XML element above):

```
<moqui.screen.SubscreensItem subscreenName="example"
    userGroupId="ALL_USERS"
    menuTitle="Example" menuIndex="8" menuInclude="Y"
    screenLocation="component://webroot/screen/webroot/apps.xml"
    subscreenLocation="component://example/screen/ExampleApp.xml"/>
```

Within the widgets (visual elements) part your screen you specify where to render the active subscreen using the `subscreens-active` element. You can also specify where the menu for all subscreens should be rendered using the `subscreens-menu` element. For a single element to do both with a default layout use the `subscreens-panel` element.

While the full path to a screen will always be explicit, when following the default subscreen item under each screen there can be multiple defaults where all but one have a condition. In the webroot.xml screen there is an example of defaulting to an alternate subscreen for the iPad:

```
<subscreens default-item="apps">
    <conditional-default item="ipad"
     condition="(ec.web.request.getHeader('User-Agent')?:'').matches('.*iPad.*')"/>
</subscreens>
```

With this in place an explicit screen path will go to either the "apps" subscreen or the "ipad" subscreen, but if neither is explicit it will default to the `ipad.xml` subscreen if the User-Agent matches, otherwise it will default to the normal `apps.xml` subscreen. Both of these have the example and tools screen hierarchies under them but have slightly different HTML and CSS to accommodate different platforms.

Once a screen such as the FindExample screen is rendered through one of these two its links will retain that base screen path in URLs generated from relative screen paths so the user will stay in the path the original default pointed to.

Standalone Screen

Normally screens will be rendered following the render path, starting with the root screen. Each screen along the way may add to the output. A screen further down the path that is rendered without any previous screens in the path adding to the output is a "standalone" screen.

This is useful when you want a screen to control all of its output and not use headers, menus, footers, etc from the screen it is under in the subscreens hierarchy.

There are two ways to make a screen standalone:

- set the `screen.standalone` attribute to true to make the screen always standalone
- to render any screen standalone pass in the `lastStandalone=true` parameter, or set it in a screen pre-action (action under the `screen.pre-actions` element)

The first option is most useful for screens that are the root of an application separate from the rest and that need different decoration and such. The second option is most useful for screens that are sometimes used in the context of an application, and other times used to produce undecorated output like a CSV file or for loading dynamically in a dialog window or screen section.

Transition

A transition is defined as a part of a screen and is how you get from one screen to another, processing input if applicable along the way. A transition can of course come right back to the same screen and when processing input often does.

The logic in transitions (transition actions) should be used only for processing input, and not for preparing data for display. That is the job of screen actions which, conversely, should not be used to process input (more on that below).

When a XML Screen is running in a web application the transition comes after the screen in the URL. In any context the transition is the last entry in the list of subscreen path elements. For example the first path goes to the `EditExample` screen, and the second to the **`updateExample`** transition within that screen:

```
/apps/example/Example/EditExample
/apps/example/Example/EditExample/updateExample
```

When a transition is the target of a HTTP request any actions associated with the transition will be run, and then a redirect will be sent to ask the HTTP client (usually a web browser) to go to the URL of the screen the transition points to. If the transition has no logic and points right to another screen or external URL when a link is generated to that transition it will automatically go to that other screen or external URL and skip calling the transition altogether. Note that these points only apply to a XML Screen running in a web-based application.

A simple transition that goes from one screen to another, in this case from `FindExample` to `EditExample`, looks like this:

```
<transition name="editExample">
    <default-response url="../EditExample"/>
</transition>
```

The path in the **`url`** attribute is based on the location of the two screens as siblings under the same parent screen. In this attribute a simple dot (`"."`) refers to the current screen and two dots (`".."`) refers to the parent screen, following the same pattern as Unix file paths.

For screens that have input processing the best pattern to use is to have the transition call a single service. With this approach the service is defined to agree with the form that is submitted to the corresponding transition. This makes the designs of both more clear and offers other benefits such as some of the validations on the service definition are used to

generate matching client-side validations. This sort of transition would look like this (the **updateExample** transition on the EditExample screen):

```
<transition name="updateExample">
    <service-call name="org.moqui.example.ExampleServices.updateExample"/>
    <default-response url="."/>
</transition>
```

In this case the default-response.**url** attribute is simple a dot which refers to the current screen and means that after this transition is processed it will go to the current screen.

A screen transition can also have actions instead of a single service call by using the actions element instead of the service-call element. Just as with all actions elements in all XML files in Moqui, the subelements are standard Moqui XML Actions that are transformed into a Groovy script. This is what a screen transition with actions might look like (simplified example, also from the EditExample screen):

```
<transition name="getExampleTypeEnumList">
    <actions>
        <entity-find entity-name="..." list="...">
            <econdition field-name="..." from="..."/>
            <order-by field-name="..."/>
        </entity-find>
        <script>
        ec.web.sendJsonResponse([exampleTypeEnumList:exampleTypeEnumList])
        </script>
    </actions>
    <default-response type="none"/>
</transition>
```

This example also shows how you would do a simple entity find operation and return the results to the HTTP client as a JSON response. Note the call to the ec.web.**sendJsonResponse**() method and the none value for the default-response.**type** attribute telling it to not process any additional response.

As implied by the element default-response you can also conditionally choose a response using the conditional-response element. This element is optional and you can specify any number of them, though you should always have at least one default-response element to be used when none the conditions are met. There is also an optional error-response which you may use to specify the response in the case of an error in the transition actions.

A transition with a conditional-response would look something like this simplified example from the DataExport screen:

```
<transition name="EntityExport.xml">
    <actions><script><![CDATA[if (...) noResponse = true]]>
        </script></actions>
    <conditional-response type="none">
        <condition><expression>noResponse</expression></condition>
    </conditional-response>
    <default-response url="."/>
```

```
</transition>
```

This is allowing the script to specify that no response should be sent (when it sends back the data export), otherwise it transitions back to the current screen. Note that the text under the `condition.expression` element is simply a Groovy expression that will be evaluated as a boolean.

All `*-response` elements can have parameter subelements that will be used when redirecting to the url or other activating of the target screen. Each screen has a list of expected parameters so this is only necessary when you need to override where the parameter value comes from (default defined in the parameter tag under the screen) or to pass additional parameters.

Here are the shared attributes of the `default-response`, `conditional-response`, and `error-response` elements:

`type`	Defaults to `url`, can be: • `none`: No response, do nothing aside from the transition actions. • `screen-last`: Go to the screen from the last request unless there is a saved one from some previous request (using the **`save-current-screen`** attribute, done automatically for login). If no last screen is found the value in the url will be used, and if nothing there will go to the default screen (just to root with whatever defaults are setup for each subscreen). • `screen-last-noparam`: Like `screen-last` but don't pass through any parameters. • `url`: Redirect to the URL specified in the **`url`** attribute, of **`url-type`**
`url`	The URL to follow in response, based on **`url-type`**. The default **`url-type`** is `screen-path` which means the value here is a path from the current screen to the desired screen, transition, or sub-screen content. Use `"."` to represent the current screen, and `".."` to represent the parent screen on the runtime screen path. The `".."` can be used multiple times, such as `"../.."` to get to the parent screen of the parent screen (the grand-parent screen). If the screen-path type url starts with a `"/"` it will be relative to the root screen instead of relative to the current screen. If **`url-type`** is `plain` then this can be any valid URL (relative on current domain or absolute).
`url-type`	Can be either `screen-path` (default) or `plain`. Normally responses will go to another screen, hence the default, but if you want to go to a relative or absolute URL use the `plain` type.

parameter-map	Just like the parameter subelement can be used to specify parameters to pass with the redirect.
save-current-screen	Save the current screen's path and parameters for future use, generally with the screen-last **type** of response.
save-parameters	Save the current parameters (and request attributes) before doing a redirect so that the screen rendered after the redirect renders in a context similar to the original request to the transition.

Parameters and Web Settings

One of the first things in a screen definition is the parameters that are passed to the screen. This is used when building a URL to link to the screen or preparing a context for the screen rendering. You do this using the parameter element, which generally looks something like this:

```
<parameter name="exampleId"/>
```

The **name** attribute is the only required one, and there are others if you want a default static value (with the **value** attribute) or to get the value by default from a field in the context other than one matching the parameter name (with the **from** attribute).

While parameters apply to all render modes there are certain settings that apply only when the screen is rendered in a web-based application. These options are on the screen.web-settings element, including:

- **allow-web-request**: Defaults to true. Set to false to not allow access to an HTTP client.
- **require-encryption**: Defaults to true. Set to false for screens that are less secure and don't requite encryption (i.e. HTTPS).
- **mime-type**: Defaults to text/html. This can vary based on how the screen is rendered (the render mode) but when always producing a certain type of output set the corresponding mime type here.
- **character-encoding**: Defaults to UTF-8 for text output. If you are rendering text with a different encoding, set it here.

Screen Actions, Pre-Actions, and Always Actions

Before rendering the visual elements (widgets) of a screen data preparation is done using XML Actions under the screen.actions element. These are the same XML Actions used for services and other tools and are described in the Logic and Services chapter. There are elements for running services and scripts (inline Groovy or any type of script supported through the Resource Facade), doing basic entity and data moving operations, and so on.

Screen actions should be used only for preparing data for output. Use transition actions to process input.

When screens are rendered it is done in the order they are found in the screen path and the actions for each screen are run as each screen in the list is rendered. To run actions before the first screen in the path is rendered use the `pre-actions` element. This is used mainly for preparing data needed by screens that will include the current screen (i.e., before the current screen in the screen path). When using this keep in mind that a screen can be included by different screens in different circumstances.

If you want actions to run before the screen renders and before any transition is run, then use the `always-actions` element. The main difference between `always-actions` and `pre-actions` is that the `pre-actions` only run before a screen or subscreen is rendered, while `always-actions` will run before any transition in the current screen and any transition in any subscreen. The `always-actions` also run whether the screen will be rendered, while the `pre-actions` only run if the screen will be rendered (i.e., is below a standalone screen in the path).

XML Screen Widgets

The elements under the `screen.widgets` element are the visual elements that are rendered, or when producing text that actually produce the output text. The most common widgets are XML Forms (using the `form-single` and `form-list` elements) and included templates. See the section below for details about XML Forms.

While XML Forms are not specific to any render mode templates by their nature are particular to a specific render mode. This means that to support multiple types of output you'll need multiple templates. The `webroot.xml` screen (the default root screen) has an example of including multiple templates for different render modes:

```
<render-mode>
    <text type="html"
        location="component://webroot/screen/includes/Header.html.ftl"/>
    <text type="xsl-fo" no-boundary-comment="true"
        location="component://webroot/screen/includes/Header.xsl-fo.ftl"/>
</render-mode>
```

The same screen also has an example of supporting multiple render modes with inline text:

```
<render-mode>
    <text type="html"><![CDATA[</body></html>]]></text>
    <text type="xsl-fo">
        <![CDATA[</fo:flow></fo:page-sequence></fo:root>]]></text>
</render-mode>
```

These are the widget elements for displaying basic things:

- **`link`**: a hyperlink to a transition, another screen, or any URL

- **image**: display an image
- **label**: display some text

To structure screens use these widget elements:

- section: a named part of a screen with condition, actions, widgets, and fail-widgets (run when condition evaluates to false)
- section-iterate: like section but is run for each entry in a collection
- container: an area of a screen
- container-panel: an area of a screen structured into a header, footer and left, center and right panels in-between
- container-dialog: a screen area that is initially hidden and that pops up when a button is pressed
- dynamic-dialog: a button and placeholder for a popup that loads its content from the server through a transition of the current screen
- include-screen: literally include another screen

Section, Condition and Fail-Widgets

A section is a special widget that contains other widgets. It can be used anywhere other screen widget elements are used. A section has widgets, condition, and fail-widgets subelements. The screen element also supports these subelements, making it a sort of top-level section of a screen.

The condition element is used to specify a condition. If it evaluates to true the widgets under the widgets element will be rendered, and if false the widgets under the fail-widgets element will be.

Macro Templates and Custom Elements

Moqui XML Screen and XML Form files are transformed to the desired output using a set of macros in a Freemarker (FTL) template file. There is one macro for each XML element to produce its output when the screen is rendered.

There are two ways to specify the macro template used to render a screen:

- for all screens: moqui-conf.screen-facade.screen-text-output.**macro-template-location** attribute in the Moqui Conf XML file; there is one screen-text-output element for each render mode (i.e. html, xml, csv, xsl-fo, etc) identified by the screen-text-output.**type** attribute
- for a single screen: screen.macro-template.**location** attribute; you can also specify a macro-template element for each render-mode, identified by the macro-template.**type** attribute

The location of the macro template can be any location supported by the Resource Facade. The most common types of locations you'll use for this include component, content, and runtime directory locations.

The default macro templates included with Moqui are specified in the `MoquiDefaultConf.xml` file along with all other default settings. You can override them with your own in the Moqui Conf XML file specified at runtime.

When you use a custom macro template file you don't need to include a macro for every element you want to render differently. You can start the file with an include of a default macro file or any other macro file you want to use, and then just override the macros for desired elements. An include of another macro file within your file will look something like:

```
<#include "classpath://template/DefaultScreenMacros.html.ftl"/>
```

The location here can also be any location supported by the Resource Facade.

You can use this approach to add your own custom elements. In other words, the macros in your custom macro template file don't have to be an override of one of the stock elements in Moqui, they can be anything you want.

Use this approach to add your own widget elements and form field types that you want to be consistent across screens in your applications. For example you can add macros for special containers with dynamic HTML like the dialogs in the default macros, or a special form field like a slider or a custom form field widget you create with JavaScript.

When you add a macro for a custom element you can just start using it in your XML Screen files even though they are not validated by the XSD file. If you want them to be validated:

1. create your own custom XSD file
2. include one or more of the default Moqui XSD files
3. add your element definitions to your custom XSD
4. refer to your custom XSD file in the `screen.xsi:noNamespaceSchemaLocation` attribute of your XML Screen file

CSV, XML, PDF and Other Screen Output

Because a single XML Screen file can support output in multiple render modes the render mode to use is selected using a parameter to the screen: the **`renderMode`** parameter. For web-based applications this can be a URL parameter. For any application this can be set in a screen action, usually a pre-action (i.e., under the `screen.pre-actions` element).

The value of this parameter can be any string matching a `screen-text-output.type` attribute in the Moqui Conf XML file. This includes the OOTB types as well as any you add in your runtime conf file.

All screens in the render path are rendered regardless of the render mode, so for output types where you only want the content of the last screen in the path to be included (like CSV), use the **lastStandalone**=true parameter along with the **renderMode** parameter.

XML Form

There are two types of XML Form: single and list. A single form represents a single set of fields with a label and widget for each. A list form is presented as a table with a column for each field, the label in the table header, a widget for the field in each row, and a row for each entry in the list the form output is based on.

While there are other ways to get data, most commonly a single form gets field values from a Map and a list form from a List of Maps.

A XML Form is like a XML Screen in that they are both rendered using a FTL macro for each element, and both support multiple render modes. Just like with XML Screen widgets you can add your own widgets by adding macros for them. The XML Form macros go in the same FTL file as the XML Screen macros, so use the same approach to add custom macros.

Form Field

The main element in a form is the `field`, identified by its **name** attribute. When a form extends another form fields are overridden by using the same field name. For HTML output this is also the name of the HTML form field. The name is also used as the map key or parameter name (if no map key value found, or when there is an error submitting the form) to get the field value from. To get the field value from somewhere else in the context, and still use the **name** for the parameter when applicable, use the **entry-name** attribute which can be any Groovy expression that evaluates to the value desired.

For automatic client-side validation in generated HTML based on a service parameter you can use the **validate-service** and **validate-parameter** attributes on the field element. When the form field is automatically defined based on a service using the `auto-fields-service` element these two attributes will be populated automatically. The XML Form renderer will also look at the **transition** the form submits to and if it has a single `service-call` element (as opposed to processing input using an `actions` element) it will look for a service input parameter with a name matching the field name and use its validations.

The field type or "widget" (visual/interactive element) of a field goes under a subelement of the `field` element. The default widget to use goes under the `default-field` subelement and all fields should have one (and only one). If you want different widgets to be used in specific conditions use the `conditional-field` element with a Groovy expression that evaluates to a boolean in the **condition** attribute. This works for both single and list forms, and for list forms is evaluated for each row.

There is also a `field.header-field` subelement for a widget that goes in the header row of list forms. When used these header field widgets are part of a separate form that is meant to be used for search options. Sort/order links naturally go along with search options in the list form header and these can be turned on by setting the `header-field.show-order-by` attribute to `true` or `case-insensitive`.

A field's title comes from the `default-field.title` attribute unless there is a `header-field` element, then it comes from the `title` attribute on that element. The `default-field` element also has a `tooltip` attribute which shows as a popup tooltip when focused on or hovering over the field (specific behavior depends on the HTML generated or other specific form rendering).

It is often nice when date values are red when a from date has not been reached or after a thru date. This is controlled using the `default-field.red-when` attribute, which by default is `by-name` meaning if the field `name` is `fromDate` then the field is red when the date is in the future and if the field `name` is `thruDate` then the field is red when the date is in the past. The `red-when` attribute can also be `before-now`, `after-now`, and `never`.

Field Widgets

There are a number of OOTB widgets for form fields, and additional widgets can be added using the extension mechanism described for screens in the Macro Templates and Custom Elements section.

Any of the widgets usable in screens can be used in XML Form fields (see the **XML Screen Widgets** section). There are also various widgets that are specific to form fields. Here is a summary of the OOTB field widgets in Moqui:

`auto-widget-service`	Define the field widget automatically based on the `parameter-name` input parameter of the `service-name` service. Use the `field-type` attribute to specify the general type of field widget to use, the specific field widget is selected based on the parameter object type. This can be `edit` (default), `find`, `display`, `find-display` (adds both find and display widgets), or `hidden`.
`auto-widget-entity`	Define the field widget automatically based on the `field-name` field of the `entity-name` entity. Use the `field-type` attribute to specify the general type of field widget to use, the specific field widget is selected based on the field type. This can be `edit`, `find`, `display`, `find-display` (default; adds both find and display widgets), or `hidden`.

widget-template-include	Form field widget templates are defined in a XML file with the widget-templates root element. Each widget-template element can contain any of the field widget elements with ${} parameters as needed. To use a widget template just specify its **location** and set subelements as needed define fields for just the scope of rendering the template.
check	Show check boxes for a list of options from the entity-options, list-options, and/or option subelements (see the drop-down description for details). Optionally specify a box to check by default using the **no-current-selected-key** attribute, or check all boxes by setting **all-checked** to true.
date-find	Displays two date/time input widgets just like date-time with the same **type** and **format** attributes. Use the **default-value-from** attribute for the default value of the from (left) input box, and the **default-value-thru** attribute for the thru (right) one.
date-time	A date/time input widget specific to the **type**, either timestamp, date-time, date, or time. The format of the date/time string is specified in the **format** attribute using a Java SimpleDateFormat string. The text input box part of the widget is **size** characters wide on a single line allowing at most **maxlength** entered characters, though these are optional and automatically set based on the **type**. Use the **default-value** attribute to specify a value to use if there is no context or parameter value for the field.
display	A plain text display of the expanded string from the **text** attribute (or the field value if empty) plus a corresponding hidden field submitted with the form unless **also-hidden** is set to false. Use the **format** attribute to specify the Java format string for date/time (SimpleDateFormat), number (DecimalFormat), etc values. For currency formatting specify the field containing the currency Uom.**uomId** in **currency-unit-field**. For HTML output by default encodes the text unless **encode** is set to false.
display-entity	Lookup an entity value for **entity-name** and display the expanded **text** string including the entity field values. This is limited to lookup by a single primary key field, and if the entity's PK field has a name different from field.**name** then specify it with the **key-field-name** attribute. By default this is a cached query, to not use the entity cache set **use-cache** to false. Just like display, this has a corresponding hidden field submitted with the form unless **also-hidden** is set to false. For HTML output by default encodes the text unless **encode** is set to false.

drop-down	A drop-down, or multi-line box if **size** is set to a number greater than 1. To allow selection of multiple values set **allow-multiple** to true. The currently selected value can be the first in the drop-down with a divider from the rest of the options if **current** is set to first-in-list (default) or can be selected from the options with selected. Set **allow-empty** to true to add an empty option to the list. The list of options is assembled using the entity-options, list-options, and/or option subelements, or alternatively the dynamic-options element to get the options with a request to a screen transition. Use entity-options to get options from database records. Specify the entity field to use as the key/value with the **key** attribute, and the field to use as the label text with the **text** attribute. The query constraints and options are specified using the **entity-find** element, the same element used in XML Actions scripts. For options from a List of Maps use the list-options element with a Groovy expression that evaluates to the List in the **list** attribute, and the Map key for the key/value of the option in the **key** attribute and the label text Map key in the **text** attribute. To specify individual options explicitly use an option element with **key** and **text** attributes for each option. For dynamic-options specify the screen **transition** that returns a JSON string containing a List of Maps plus **value-field** and **label-field** attributes for the map keys to get the value and label from in each Map. The main reason to use dynamic options is to change the options when another field changes. To do this use one or more depends-on subelement with the form field name in its **field** attribute. When a referenced field changes new options will be requested from the screen transition, passing all referenced field values as parameters to the request. Set the default option with its key in the **no-current-selected-key** attribute. If that option is not in the existing options specify its description using the **current-description** attribute. By default uses a dynamic drop-down widget that filters options based entered text. To use a plain drop-down set **search** to false. To allow the user to enter a new option to submit that is not already in the drop-down set **combo-box** to true.

`file`	A file upload input box (has a button/link for a file selection popup window) **`size`** (default `30`) characters wide allowing at most **`maxlength`** entered characters. Use the **`default-value`** attribute to specify a value to use if there is no context or parameter value for the field.
`hidden`	A hidden field whose value is passed with the submitted form but nothing is displayed to the user. Use the **`default-value`** attribute to specify a value to use if there is no context or parameter value for the field.
`ignored`	Treats the field as if it was not even defined. Useful when extending another form to eliminate undesired fields.
`password`	A password input box **`size`** (default `30`) characters wide allowing at most **`maxlength`** entered characters. Masks the input for security.
`radio`	Show radio buttons for a list of options from the `entity-options`, `list-options`, and/or `option` subelements (see the `drop-down` description for details). Optionally specify the default option's key using the **`no-current-selected-key`** attribute (used if there is no value or parameter for the field).
`range-find`	Mainly for numeric range find, displays two small input boxes **`size`** (default `10`) characters wide allowing at most **`maxlength`** entered characters in each. Use the **`default-value-from`** attribute for the default value of the from (left) input box, and the **`default-value-thru`** attribute for the thru (right) one.
`reset`	A button to reset the form. The button text comes from the field title.
`submit`	A form submit button. The button text comes from the field title unless the `image` subelement is used to put an image on the button. An icon next to the text can be used with the **`icon`** attribute set to an icon style from the icon library (for the default runtime webroot the Glyphicons for Bootstrap icons are available, for example **`icon`**`="glyphicon glyphicon-plus"` or the Font Awesome icons can be used with something like `"fa fa-search"`). To show a message and ask the user to confirm when the button is pressed put the message in the **`confirmation`** attribute.
`text-area`	A text area **`cols`** characters wide and **`rows`** lines tall allowing at most **`maxlength`** entered characters. Use the **`default-value`** attribute to specify a value to use if there is no context or parameter value for the field. Set **`read-only`** to `true` to make the text area display only, not allow a change to the value.

`text-line`	A simple text input box **`size`** characters wide on a single line allowing at most **`maxlength`** entered characters. Use the **`default-value`** attribute to specify a value to use if there is no context or parameter value for the field. Set **`disabled`** to `true` to make the input box display only, not allow a change to the value. Use the **`format`** attribute to specify the Java format string for date/time (`SimpleDateFormat`), number (`DecimalFormat`), etc values. A `text-line` can have autocomplete by implementing a screen transition to provide the values and specifying the transition name in the **`ac-transition`** attribute. The transition should respond with a JSON string (using `ec.web.`**`sendJsonResponse`**`()`) with a `List` of `Map`s with `value` and `label` fields. Optionally specify the time delay in milliseconds (default `300`) with **`ac-delay`** and the minimum characters to enter before lookup with **`ac-min-length`** (default `1`).
`text-find`	Like `text-line` with **`size`**, **`maxlength`**, and **`default-value`** attributes and also has a checkbox for **`ignore-case`** (defaults to `true`, i.e. checked), and a drop-down for a search operator with a default specified in the **`default-operator`** attribute (can be `equals`, `like`, `contains`, or `empty`). The ignore case checkbox and operator drop-down can also be hidden (defaults passed as hidden parameters, no visible UI widget) using the **`hide-options`** attribute Options for hide are `false` (default, show both), `true` (hide both), `ignore-case` (hide only ignore case checkbox), and `operator` (hide the operator drop-down).

Single Form

Use the `form-single` element to define a single form. These are the attributes of the `form-single` element:

- **`name`**: The name of the form. Used to reference the form along with the XML Screen file location. For HTML output this is the form name and id, and for other output may also be used to identify the part of the output corresponding to the form.
- **`extends`**: The location and name separated by a hash/pound sign (#) of the form to extend. If there is no location it is treated as a form name in the current screen.
- **`transition`**: The transition in the current screen to submit the form to.
- **`map`**: The `Map` to get field values from. Is often a EntityValue object or a `Map` with data pulled from various places to populate in the form. Map keys are matched against field names. This is ignored if the `field.`**`entry-name`** attribute is used, that is evaluated against the context in place at the time each field is rendered. Defaults to `fieldValues`.
- **`focus-field`**: The **`name`** of the field to focus on when the form is rendered.

- **skip-start**: Skip the starting rendered elements of the form. When used after a form with **skip-end**=true this will effectively combine the forms into one.
- **skip-end**: Skip the ending rendered elements of the form. Use this to leave a form open so that additional forms can be combined with it.
- **dynamic**: If true this form will be considered dynamic and the internal definition will be built up each time it is used instead of only when first referred to. This is necessary when auto-fields-* elements have ${} string expansion for service or entity names.
- **background-submit**: Submit the form in the background without reloading the screen.
- **background-reload-id**: After the form is submitted in the background reload the dynamic-container with this id.
- **background-message**: After the form is submitted in the background show this message in a dialog.

To layout fields in a way other than a plain list of fields use the form-single.field-layout element. For HTML output there is an optional **id** attribute to facilitate styling. If the field layout contains field groups set the **collapsible** attribute to true to use an accordion widget to save space, optionally specifying the **active** group index instead of the first to be initially open. Here are the subelements to define a layout:

- field-ref: specifies where to include a field by **name**
- fields-not-referenced: include all fields not referenced elsewhere; if this element is not present fields that are not referenced in the field-layout will not be rendered
- field-row: create a row of fields specified by field-ref subelements; if there are two fields in the row they display in four columns, both with titles; if there are more than two fields only the title of the first field is displayed and the remaining field widgets go side-by-side in the row, wrapping if needed
- field-group: create a group of fields, in an accordion if field-layout.**collapsible** is true, with an optional **title** above the group and for HTML output an optional **style** for the container (div) around the group; use the field-ref, fields-not-referenced, and field-row subelements to specify the fields to include, and optionally put them in rows

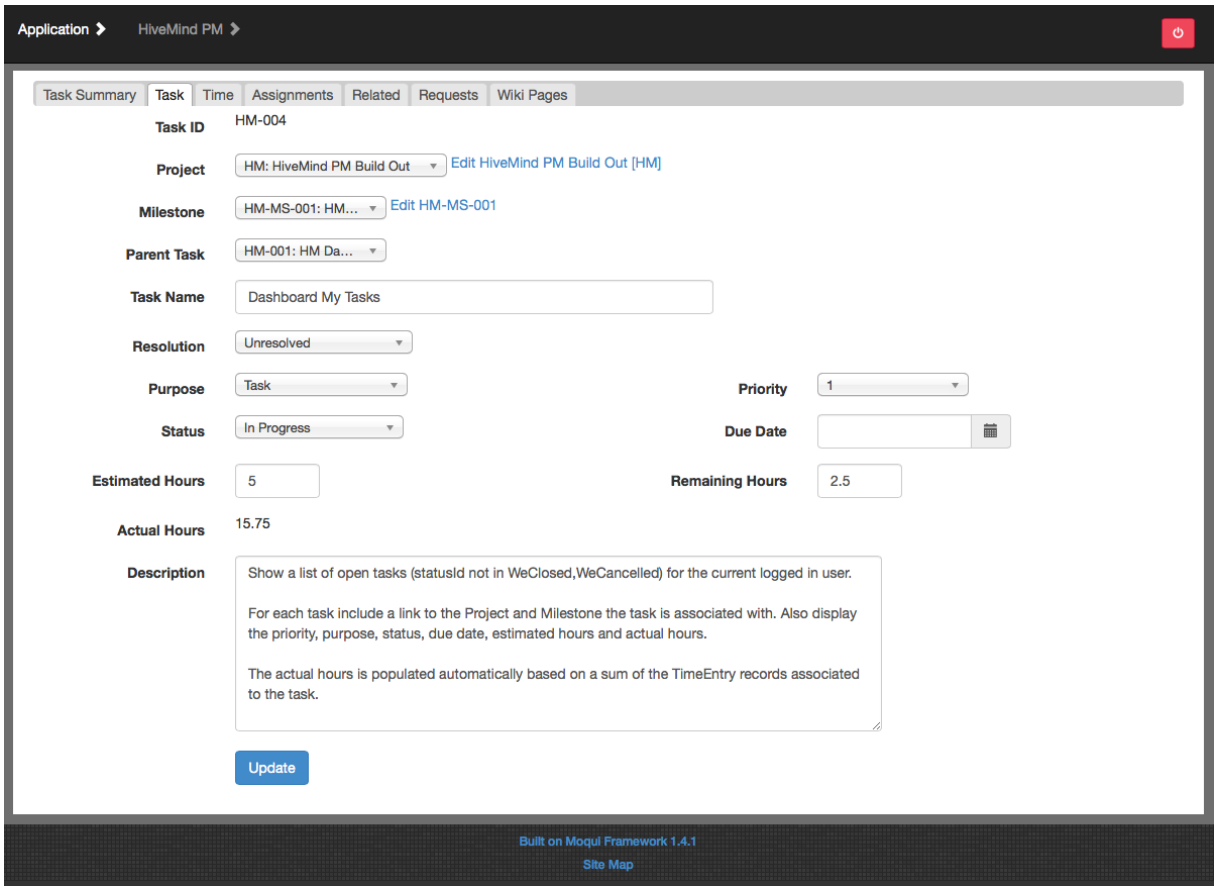
Single Form Example

To get a better idea of the utility of different aspects of a single form let's look at a more complex example. This form is the Edit Task screen from the HiveMind Project Manager application.

This form has examples of the following (see the full source below):

- **Project**: a drop-down populated using entity-options, and a separate link to go to the current project associated with the task

- **Milestone** and **Parent Task**: `drop-down` fields populated with `dynamic-options`, both dependent on the project (**`rootWorkEffortId`**) using the `depends-on` element
- **Task Name**: simple `text-line` input box
- **Resolution** and **Purpose**: standard `Enumeration` `drop-down` fields using the `widget-template-include` element with `set` subelements; Purpose uses a widget template constrained by a parent `Enumeration` (**`parentEnumId`**), whereas Resolution includes all values for an `EnumerationType` (**`enumTypeId`**)
- **Status**: standard status `drop-down` with options based on transitions from the current status using the `StatusFlowTransition` entity
- **Due Date**: simple date-time of type date input box
- **Estimated Hours** and **Remaining Hours**: simple number input boxes
- **Actual Hours**: `display` with a number **`format`** string
- **Description**: simple `text-area`



This form uses `field-layout` to put various fields side-by-side, but otherwise uses the default layout. For an example of a layout with a `field-group` accordion see the Edit Example screen in the Moqui Example app.

Here is the source for the Single Form, and the XML Screen it is part of for context and to see the `transition` definitions, screen `actions` for data preparation, etc:

```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Task" default-menu-index="1">

  <parameter name="workEffortId"/>

  <transition name="updateTask">
    <service-call name="mantle.work.TaskServices.update#Task"
            in-map="context"/>
    <default-response url="."/>
  </transition>
  <transition name="editProject">
    <default-response url="../../Project/EditProject"/></transition>
  <transition name="milestoneSummary">
    <default-response url="../../Project/MilestoneSummary"/>
  </transition>
  <transition name="getProjectMilestones">
    <actions>
      <service-call in-map="context" out-map="context"
          name="mantle.work.ProjectServices.get#ProjectMilestones"/>
      <script>ec.web.sendJsonResponse(resultList)</script>
    </actions>
    <default-response type="none"/>
  </transition>
  <transition name="getProjectTasks">
    <actions>
      <service-call in-map="context" out-map="context"
          name="mantle.work.ProjectServices.get#ProjectTasks"/>
      <script>ec.web.sendJsonResponse(resultList)</script>
    </actions>
    <default-response type="none"/>
  </transition>

  <actions>
    <entity-find-one entity-name="mantle.work.effort.WorkEffort"
            value-field="task"/>
    <entity-find-one entity-name="mantle.work.effort.WorkEffort"
            value-field="project">
      <field-map field-name="workEffortId" from="task.rootWorkEffortId"/>
    </entity-find-one>

    <entity-find entity-name="mantle.work.effort.WorkEffortAssoc"
        list="milestoneAssocList">
      <date-filter/>
      <econdition field-name="toWorkEffortId" from="task.workEffortId"/>
      <econdition field-name="workEffortAssocTypeEnumId"
          value="WeatMilestone"/>
    </entity-find>
    <set field="milestoneAssoc" from="milestoneAssocList?.getAt(0)"/>
    <set field="statusFlowId"
```

```
        from="(task.statusFlowId ?: project.statusFlowId) ?: 'Default'"/>
    </actions>
    <widgets>
      <form-single name="EditTask" transition="updateTask" map="task">
        <field name="workEffortId">
          <default-field title="Task ID"><display/></default-field>
        </field>
        <field name="rootWorkEffortId"><default-field title="Project">
          <drop-down>
            <entity-options key="${workEffortId}"
                text="${workEffortId}: ${workEffortName}">
              <entity-find entity-name="WorkEffortAndParty">
                <date-filter/>
                <econdition field-name="partyId"
                    from="ec.user.userAccount.partyId"/>
                <econdition field-name="workEffortTypeEnumId"
                    value="WetProject"/>
              </entity-find>
            </entity-options>
          </drop-down>
          <link text="Edit ${project.workEffortName} [${task.rootWorkEffortId}]"
              url="editProject">
            <parameter name="workEffortId" from="task.rootWorkEffortId"/>
          </link>
        </default-field></field>
        <field name="milestoneWorkEffortId"
            entry-name="milestoneAssoc?.workEffortId">
          <default-field title="Milestone">
            <drop-down combo-box="true">
              <dynamic-options transition="getProjectMilestones"
                  value-field="workEffortId" label-field="milestoneLabel">
                <depends-on field="rootWorkEffortId"/>
              </dynamic-options>
            </drop-down>
            <link url="milestoneSummary"
text="${milestoneAssoc ? 'Edit ' + milestoneAssoc.workEffortId : ''}">
              <parameter name="milestoneWorkEffortId"
                  from="milestoneAssoc?.workEffortId"/>
            </link>
          </default-field>
        </field>
        <field name="parentWorkEffortId"><default-field title="Parent Task">
          <drop-down combo-box="true">
            <dynamic-options transition="getProjectTasks"
                value-field="workEffortId" label-field="taskLabel">
              <depends-on field="rootWorkEffortId"/>
            </dynamic-options>
          </drop-down>
        </default-field></field>
        <field name="workEffortName"><default-field title="Task Name">
```

```
        <text-line/></default-field></field>
      <field name="priority"><default-field>
        <widget-template-include location="component://HiveMind/template/
            screen/ProjectWidgetTemplates.xml#priority"/>
      </default-field></field>
      <field name="purposeEnumId"><default-field title="Purpose">
        <widget-template-include location="component://webroot/template/
            screen/BasicWidgetTemplates.xml#enumWithParentDropDown">
          <set field="enumTypeId" value="WorkEffortPurpose"/>
          <set field="parentEnumId" value="WetTask"/>
        </widget-template-include>
      </default-field></field>
      <field name="statusId"><default-field title="Status">
        <widget-template-include location="component://webroot/template/
screen/BasicWidgetTemplates.xml#statusTransitionWithFlowDropDown">
          <set field="currentDescription"
            from="task?.'WorkEffort#moqui.basic.StatusItem'?.description"/>
          <set field="statusId" from="task.statusId"/>
        </widget-template-include>
      </default-field></field>
      <field name="resolutionEnumId"><default-field title="Resolution">
        <widget-template-include location="component://webroot/template/
            screen/BasicWidgetTemplates.xml#enumDropDown">
          <set field="enumTypeId" value="WorkEffortResolution"/>
        </widget-template-include>
      </default-field></field>
      <field name="estimatedCompletionDate">
        <default-field title="Due Date">
          <date-time type="date" format="yyyy-MM-dd"/></default-field>
      </field>
      <field name="estimatedWorkTime">
        <default-field title="Estimated Hours">
          <text-line size="5"/></default-field>
      </field>
      <field name="remainingWorkTime">
        <default-field title="Remaining Hours">
          <text-line size="5"/></default-field>
      </field>
      <field name="actualWorkTime"><default-field title="Actual Hours">
        <display format="#.00"/></default-field></field>
      <field name="description"><default-field title="Description">
        <text-area rows="20" cols="100"/></default-field></field>
      <field name="submitButton"><default-field title="Update">
        <submit/></default-field></field>

      <field-layout>
        <fields-not-referenced/>
        <field-row><field-ref name="purposeEnumId"/>
          <field-ref name="priority"/></field-row>
        <field-row><field-ref name="statusId"/>
```

```
          <field-ref name="estimatedCompletionDate"/></field-row>
        <field-row><field-ref name="estimatedWorkTime"/>
          <field-ref name="remainingWorkTime"/></field-row>
        <field-ref name="actualWorkTime"/>
        <field-ref name="description"/>
        <field-ref name="submitButton"/>
      </field-layout>
    </form-single>
  </widgets>
</screen>
```

This screen finds all data based on the single `workEffortId` parameter, which is the ID of the task.

List Form

Use the `form-list` element to define a single form. These are the attributes of the `form-list` element:

- **`name`**: The name of the form. Used to reference the form along with the XML Screen file location. For HTML output this is the form name and id, and for other output may also be used to identify the part of the output corresponding to the form.
- **`extends`**: The location and name separated by a hash/pound sign (#) of the form to extend. If there is no location it is treated as a form name in the current screen.
- **`transition`**: The transition in the current screen to submit the form to.
- **`multi`**: Make the form a multi-submit form where all rows on a page are submitted together in a single request with a `"_${rowNumber}"` suffix on each field. Also passes a **`_isMulti`**=`true` parameter so the Service Facade knows to run the service (a single `service-call` in a `transition`) for each row. Defaults to `true`, so set to `false` to disable this behavior and have a separate form (submitted separately) for each row.
- **`list`**: An expression that evaluates to a list to iterate over.
- **`list-entry`**: If specified each list entry will be put in the context with this name, otherwise the list entry must be a `Map` and the entries in the map will be put into the context for each row.
- **`paginate`**: Indicate if this form should paginate or not. Defaults to `true`.
- **`paginate-always-show`**: Always show the pagination control with count of rows, even when there is only one page? Defaults to `true`.
- **`skip-start`**: Skip the starting rendered elements of the form. When used after a form with **`skip-end`**=`true` this will effectively combine the forms into one.
- **`skip-end`**: Skip the ending rendered elements of the form. Use this to leave a form open so that additional forms can be combined with it.
- **`skip-form`**: Make the output a plain table, not submittable (in HTML don't generate `form` elements). Useful for view-only list forms to minimize output.

- **`dynamic`**: If `true` this form will be considered dynamic and the internal definition will be built up each time it is used instead of only when first referred to. This is necessary when `auto-fields-*` elements have `${}` string expansion for service or entity names.

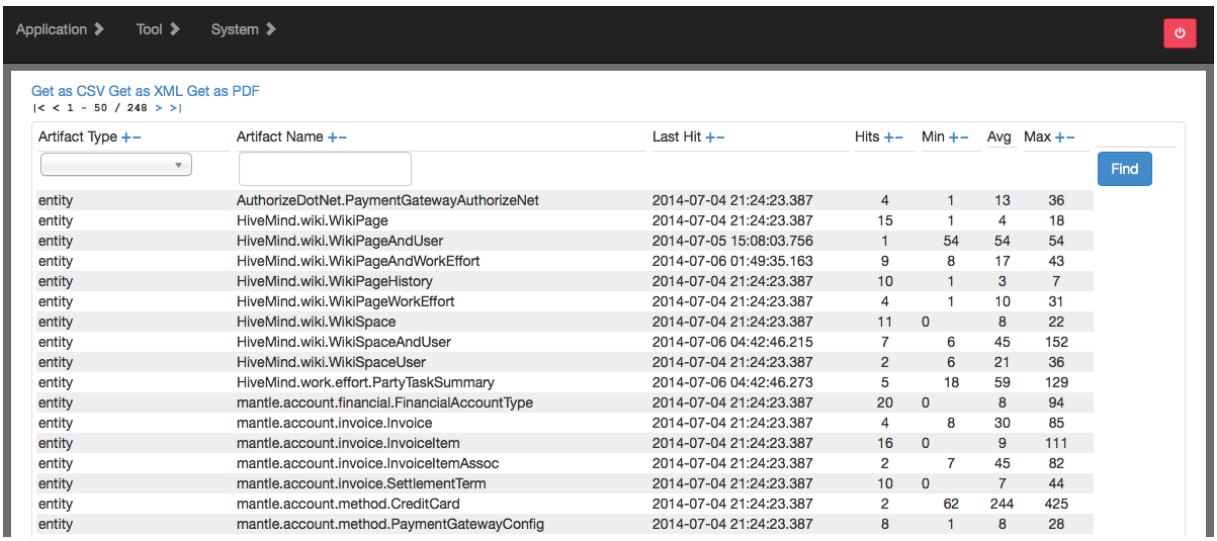
Similar to `field-layout` in a single form there is a `form-list-column` element for list forms. When used there needs to be one element for each column in the list form table, and all fields must be referenced in a column or they will not be rendered. The `form-list-column` element has a single subelement, the same `field-ref` element that is used in the single form `field-layout`.

Data preparation for a form is best done in the `actions` in the XML Screen it is used in but sometimes you need to prepare data for each row in a list form. This can be done by preparing in advance a `List` of `Map` objects that have entries for each list form field. With this approach the logic that prepares the `List` can do additional data lookups or calculations to prepare the data. The other approach is to put XML Actions under the `form-list.row-actions` element. These actions will be run for each row in an isolated context so that any context fields defined will be used only for that row.

List Form View/Export Example

There are two main categories of list forms: those used for searching, viewing, and exporting and those used for editing a number of records in a single screen.

The Artifact Summary screens in the Moqui Tools application is a good example of a screen that is used for searching, viewing data, and exporting results to CSV, XML, and PDF files all using the same screen and form definition. The list form on the screen shows a row for each artifact with a summary of the `moqui.server.ArtifactHitBin` records for that artifact using the `moqui.server.ArtifactHitReport` `view-entity`.



Note the "Get as CSV" link in the upper-left corner (and the similar XML and PDF links). This link goes to the simple `ArtifactHitSummaryStats.csv` transition that goes to the same screen and adds **`renderMode`**=`csv`, **`pageNoLimit`**=`true`, and **`lastStandalone`**=`true` parameters so that the screen renders with csv output instead of html, pagination is disabled (all results are output), and only the last screen is rendered (skipping all parent screens to avoid decoration, i.e. the last screen is "standalone"). See the **XML, CSV and Plain Text Handling** section for more detail.

Below the "Get as" links are the pagination controls which are enabled by default and by default shown when there is more than one page of results to display. In the form header row are the column titles and `"+-"` links for sorting the results in each column, plus a header find form with a `drop-down` for the Artifact Type and a `text-find` box for Artifact Name. These are all defined in the `header-field` elements under each field.

This form uses `form-list.row-actions` element to calculate the `averageTime` for each row, which is then displayed using a form field.

Here is the source for the `ArtifactHitSummary.xml` screen showing the details for the summary above:

```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Artifact Summary">

  <transition name="ArtifactHitSummaryStats.csv">
    <default-response url="."><parameter name="renderMode" value="csv"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>
  <transition name="ArtifactHitSummaryStats.xml">
    <default-response url="."><parameter name="renderMode" value="xml"/>
      <parameter name="pageNoLimit" value="true"/>
      <parameter name="lastStandalone" value="true"/></default-response>
  </transition>
  <transition name="ArtifactHitSummaryStats.pdf">
    <default-response url-type="plain"
        url="${ec.web.getWebappRootUrl(false, null)}/fop/apps/tools/System/
          ArtifactHitSummary">
      <parameter name="renderMode" value="xsl-fo"/>
      <parameter name="pageNoLimit" value="true"/>
    </default-response>
  </transition>

  <actions>
    <entity-find entity-name="moqui.server.ArtifactHitReport"
        list="artifactHitReportList" limit="50">
      <search-form-inputs default-order-by="artifactType,artifactName"/>
    </entity-find>
  </actions>
```

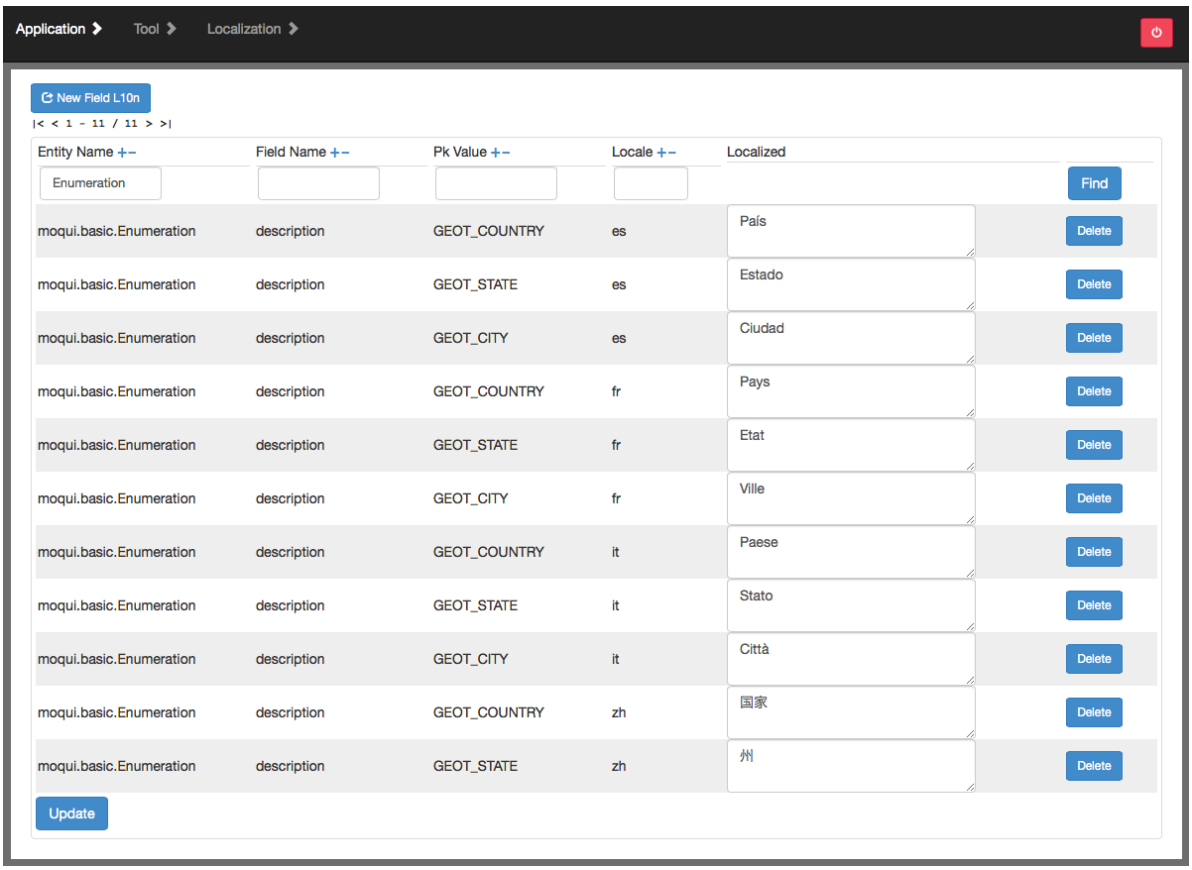
```
<widgets>
  <container>
    <link url="ArtifactHitSummaryStats.csv" text="Get as CSV"
        target-window="_blank" expand-transition-url="false"/>
    <link url="ArtifactHitSummaryStats.xml" text="Get as XML"
        target-window="_blank" expand-transition-url="false"/>
    <link url="ArtifactHitSummaryStats.pdf" text="Get as PDF"
        target-window="_blank"/>
  </container>
  <form-list name="ArtifactHitSummaryList" list="artifactHitReportList">
    <row-actions>
      <set field="averageTime" from="(totalTimeMillis/hitCount as
          BigDecimal).setScale(0,BigDecimal.ROUND_UP)"/>
    </row-actions>

    <field name="artifactType">
      <header-field show-order-by="true">
        <drop-down allow-empty="true">
          <option key="screen"/><option key="screen-content"/>
          <option key="transition"/>
          <option key="service"/><option key="entity"/>
        </drop-down>
      </header-field>
      <default-field><display also-hidden="false"/></default-field>
    </field>
    <field name="artifactName">
      <header-field show-order-by="true">
        <text-find hide-options="true" size="20"/></header-field>
      <default-field><display text="${artifactName}"
          also-hidden="false"/></default-field>
    </field>
    <field name="lastHitDateTime">
      <header-field title="Last Hit" show-order-by="true"/>
      <default-field><display also-hidden="false"/></default-field>
    </field>
    <field name="hitCount">
      <header-field title="Hits" show-order-by="true"/>
      <default-field><display also-hidden="false"/></default-field>
    </field>
    <field name="minTimeMillis">
      <header-field title="Min" show-order-by="true"/>
      <default-field><display also-hidden="false"/></default-field>
    </field>
    <field name="averageTime">
      <default-field title="Avg">
        <display also-hidden="false"/></default-field>
    </field>
    <field name="maxTimeMillis">
      <header-field title="Max" show-order-by="true"/>
      <default-field><display also-hidden="false"/></default-field>
```

```
      </field>
      <field name="find"><header-field title="Find">
        <submit/></header-field></field>
    </form-list>
  </widgets>
</screen>
```

List Form Edit Example

The Entity Fields Localization screen in the Moqui Tools application is a good example of a list form used to update multiple records in a single page. This screen is designed for adding, editing, and deleting `moqui.basic.LocalizedEntityField` records that specify localized text to use instead of an entity record field's actual value.

In the screenshot below there is a button in the upper-left corner to add a new record in a `container-dialog` modal popup. Just below that are the pagination controls which are enabled by default. The header row in the form has the field titles (in this case all generated based on the field name since there are no `header-field.title` attributes), the `"+-"` sorting links (with `header-field.show-order-by=true`), and header widgets for the fields to find only matching records.



The body rows of the list form table have one row for each record with a Delete button, but the Update button is at the bottom and updates all rows in a single form submission to update a number of Localized values at once. Notice that the Find button in the header row is in the same column as the Delete button on each body row. To do this in the form definition the Find button is defined in a subelement of the `header-field` element for the `delete` field.

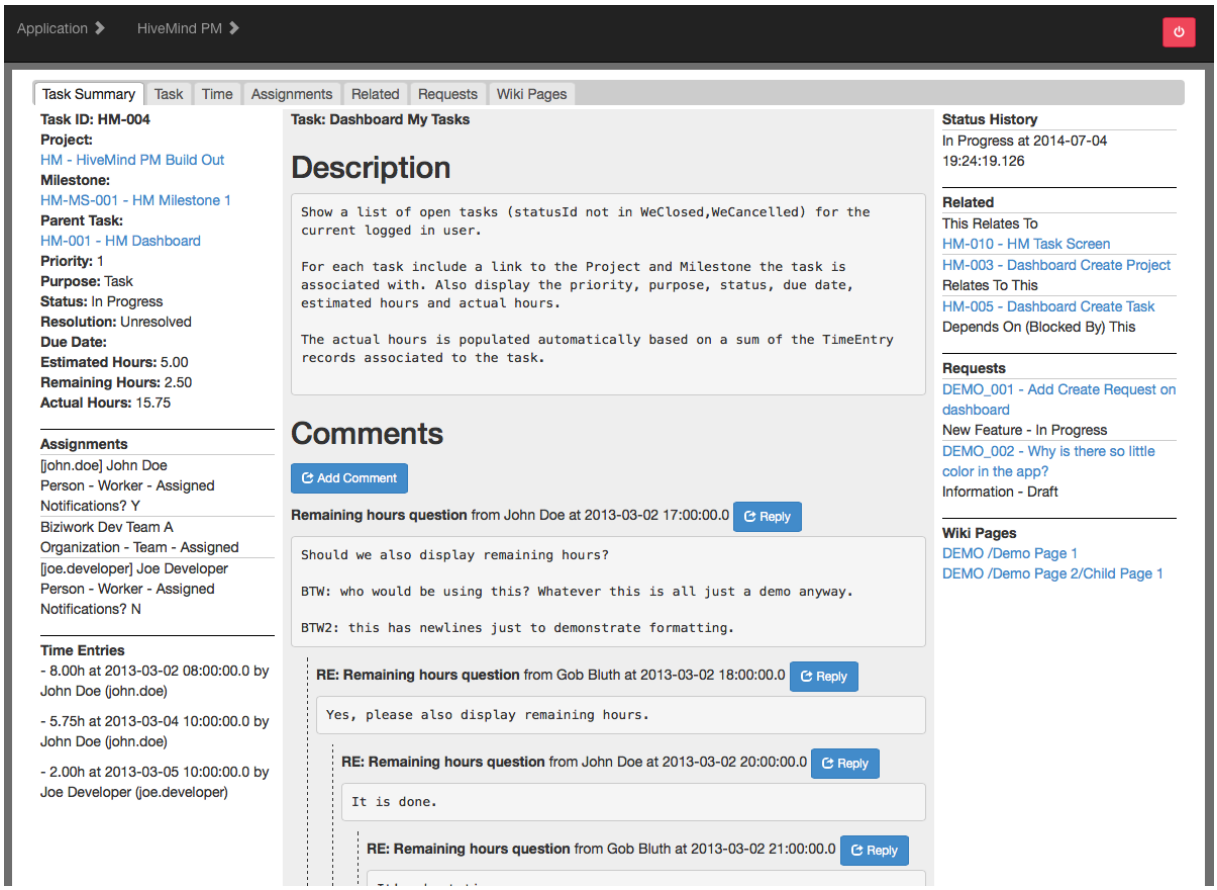
Below is the source for the `EntityFields.xml` screen. The create, update, and delete transitions use implicitly defined entity-auto services so there is no service definition or implementation for them. This functionality relies on only a XML Screen file and the definition of the `LocalizedEntityField` entity.

```
<screen xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://moqui.org/xsd/xml-screen-1.4.xsd"
        default-menu-title="Entity Fields" default-menu-index="2">

  <transition name="createLocalizedEntityField">
    <service-call name="create#moqui.basic.LocalizedEntityField"/>
    <default-response url="."/>
  </transition>
  <transition name="updateLocalizedEntityField">
    <service-call name="update#moqui.basic.LocalizedEntityField"
        multi="true"/>
    <default-response url="."/>
  </transition>
  <transition name="deleteLocalizedEntityField">
    <service-call name="delete#moqui.basic.LocalizedEntityField"/>
    <default-response url="."/>
  </transition>

  <actions>
    <entity-find entity-name="moqui.basic.LocalizedEntityField"
        list="localizedEntityFieldList" offset="0" limit="50">
      <search-form-inputs default-order-by="entityName,fieldName,locale"/>
    </entity-find>
  </actions>
  <widgets>
    <container>
      <container-dialog id="CreateEntityFieldDialog"
          button-text="New Field L10n">
        <form-single name="CreateLocalizedEntityField"
            transition="createLocalizedEntityField">
          <field name="entityName"><default-field>
            <text-line size="15"/></default-field></field>
          <field name="fieldName"><default-field>
            <text-line size="15"/></default-field></field>
          <field name="pkValue"><default-field>
            <text-line size="20"/></default-field></field>
          <field name="locale"><default-field>
```

```
            <text-line size="5"/></default-field></field>
          <field name="localized"><default-field>
            <text-area rows="5" cols="60"/></default-field></field>
          <field name="submitButton"><default-field title="Create">
            <submit/></default-field></field>
        </form-single>
      </container-dialog>
    </container>
    <form-list name="UpdateLocalizedEntityFields"
        list="localizedEntityFieldList"
        transition="updateLocalizedEntityField" multi="true">
      <field name="entityName">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="fieldName">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="pkValue">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="12"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="locale">
        <header-field show-order-by="true">
          <text-find hide-options="true" size="4"/></header-field>
        <default-field><display/></default-field>
      </field>
      <field name="localized"><default-field>
        <text-area rows="2" cols="35"/></default-field></field>
      <field name="update"><default-field title="Update">
        <submit/></default-field></field>
      <field name="delete">
        <header-field title="Find"><submit/></header-field>
        <default-field>
          <link text="Delete" url="deleteLocalizedEntityField">
            <parameter name="entityName"/>
            <parameter name="fieldName"/><parameter name="locale"/></link>
        </default-field>
      </field>
    </form-list>
  </widgets>
</screen>
```

Templates

While a wide variety of screens can be built with XML Forms and the various XML Screen widgets and layout elements. Quite a lot can be done with the OOTB elements. Here is an example of a more complex screen, the Task Summary screen from the HiveMind PM application that is made with only OOTB elements and some custom CSS:



Sometimes you need a more flexible layout, styling, widgets, or custom interactive behavior. For things that will be used in many places, and where you want them to render consistently, add screen and form widgets (including layout elements) using FTL macros to add or extend XML Screen elements. For everything else, especially one-off things, an explicit template is the way to get any sort of HTML output you want.

This is especially useful for custom web site such as corporate or ecommerce sites where custom HTML is needed to get a very specific form and function.

Custom templates also apply to other forms of output like XML, CSS, and XSL-FO. In a XML Screen this is done with the `render-mode` element and one or more `text` subelements for each `render-mode.text.type` to support for the screen. In the current version of Moqui

Framework only text output is supported for screen rendering, but in the future or in custom code other elements under the `render-mode` element could be used to define output for non-text screen rendering such as for GWT or Swing.

If the screen is rendered with a render mode and there is no `text` subelement with a **`type`** matching the active render mode then it will simply render nothing for the block and continue with rendering the screen. The options for the `text.type` attribute match the **`type`** attribute on the `screen-facade.screen-text-output` element in the Moqui Conf XML file where the macro template to use for each output type is defined. Currently supported options include: `csv`, `html`, `text`, `xml`, and `xsl-fo`.

Other attributes (in addition to **`type`**) available on the `text` element include:

- **`location`**: This is the template or text file location and can be any location supported by the Resource Facade including file, http, component, content, etc.
- **`template`**: Interpret the text at the location as an FTL or other template? Supports any template type supported by the Resource Facade. Defaults to `true`, set to `false` if you want the text included literally.
- **`encode`**: If `true` the text will be encoded so that it does not interfere with markup of the target output. Templates ignore this setting and are never encoded. For example, if output is HTML then data presented will be HTML encoded so that all HTML-specific characters are escaped.
- **`no-boundary-comment`**: Defaults to `false`. If `true` won't ever put boundary comments before this (for opening ?xml tag, etc).

The `webroot.xml` screen is the default root screen in the OOTB runtime directory and has a good example of including templates for different render modes:

```
<widgets>
  <render-mode>
    <text type="html"
        location="component://webroot/screen/includes/Header.html.ftl"/>
    <text type="xsl-fo" no-boundary-comment="true"
        location="component://webroot/screen/includes/Header.xsl-fo.ftl"/>
  </render-mode>

  <subscreens-active/>

  <render-mode>
    <text type="html"
        location="component://webroot/screen/includes/Footer.html.ftl"/>
    <text type="xsl-fo"><![CDATA[
        ${sri.getAfterScreenWriterText()}
        </fo:flow></fo:page-sequence></fo:root>
    ]]></text>
  </render-mode>
</widgets>
```

This is an example of a screen with subscreens so it has `render-mode` elements before and after the `subscreens-active` element to decorate (or wrap) what comes from the subscreens. This shows `text` elements with a **`location`** to include a FTL template and inline text in a CDATA block right under the `text` element.

Sending and Receiving Email

The first step to sending and receiving email is to setup an EmailServer with something like this record loaded:

```
<moqui.basic.email.EmailServer emailServerId="SYSTEM"
    smtpHost="mail.test.com" smtpPort="25" smtpStartTls="N" smtpSsl="N"
    storeHost="mail.test.com" storePort="143" storeProtocol="imap"
    storeDelete="N" mailUsername="TestUser" mailPassword="TestPassword"/>
```

Note that these are all example values and should be changed to real values, especially for the **`smtpHost`**, **`storeHost`**, **`mailUsername`** and **`mailPassword`** fields. The **`store*`** fields are for the remote mail store for incoming email. Here are some other common values for the port fields:

- **`smtpPort`**: 25 (SMTP), 465 (SSMTP), 587 (SSMTP)
- **`storePort`** for **`storeProtocol`**=`imap`: 143 (IMAP), 585 (IMAP4-SSL), 993 (IMAPS)
- **`storePort`** for **`storeProtocol`**=`pop3`: 110 (POP3), 995 (SSL-POP)

If you need to work with multiple email servers, just add `EmailServer` records with the settings for each. When sending an email using an email template the `EmailServer` to use is specified on the `EmailTemplate` record with the **`emailServerId`** field.

Speaking of `EmailTemplate`, the next step for sending an email is to create one. Here is an example from HiveMind PM for sending a task update notification email:

```
<moqui.basic.email.EmailTemplate emailTemplateId="HM_TASK_UPDATE"
    description="HiveMind Task Update Notification"
    emailServerId="SYSTEM" webappName="webroot"
    bodyScreenLocation="component://HiveMind/screen/TaskUpdateNotification.xml"
    fromAddress="test@test.com" ccAddresses="" bccAddresses=""
    subject="Task Updated: ${document._id} - ${document.WorkEffort.name}"/>
```

The general idea is to define a screen that will be rendered for the body when the email is sent (**`bodyScreenLocation`**). The email body screen is a little bit different from normal UI screens because there is no Web Facade available when it is rendered as it is not part of a web request. The URL prefixes (domain name, port, etc) are generated based on `webapp` settings in the Moqui Conf XML file, which is why it is necessary to specify a **`webappName`** which is matched against the `moqui-conf.webapp-list.webapp.name` attribute.

The **`subject`** is also a simple template of sorts, it is a Groovy String that is expanded when the email is sent using the same context as rendering the body. The **`fromAddress`** field is required, and you can optionally specify **`ccAddresses`** and **`bccAddresses`**.

Attachments to an EmailTemplate can be added with the EmailTemplateAttachment entity. The filename to use on the email must be specified using the **fileName** field. The attachment itself comes from rendering a screen specified with the **attachmentLocation** field. The **screenRenderMode** field is passed to the ScreenRender to specify the type of output to get from the screen. It is also used to determine the MIME/content type. If empty the content at **attachmentLocation** will be sent over without screen rendering and its MIME type will be based on its extension. This can be used to generate XSL:FO that is transformed to a PDF and attached to the email with by setting **screenRenderMode** to xsl-fo.

Once the EmailServer and EmailTemplate are defined you can send email using the org.moqui.impl.EmailServices.**send#EmailTemplate** service. When calling this service pass in the **emailTemplateId** parameter to identify the EmailTemplate. As mentioned above the EmailServer will be determined based on the EmailTemplate.**emailServerId** field.

The email addresses to send the message to are passed in the **toAddresses** parameter which is a plain String and can have multiple comma-separated addresses. The parameters used to render the email screen are separate from the context of the service and are passed to it in the **bodyParameters** input parameter. By default the **send#EmailTemplate** service saves details about the outgoing message in a record of the EmailMessage entity. To disable this pass in false in the **createEmailMessage** parameter. The output parameters are **messageId** which is the value put in the Message-ID email header field, and **emailMessageId** if a EmailMessage record is created.

The EmailMessage entity is used for both outgoing and incoming email messages. For outgoing messages sent using the **send#EmailTemplate** service the status (**statusId**) starts out as Sent (actually sets it to Ready, sends the email, then sets it to Sent) and may be changed to Viewed if there is open message tracking based on an image request (usually with the **emailMessageId** as a parameter or path element). If the message is returned undeliverable the status may be changed to Bounced.

An EmailMessage may also be sent manually instead of from a template and in that case the status would start out as Draft. Once the user is done with the message they would change the status to Ready, and then when it is actually sent the status would change to Sent. Incoming messages start in the Received status and can be changed to the Viewed status after they are initially opened.

For email threads the EmailMessage entity has **rootEmailMessageId** for the original messages that all messages in the thread are grouped under, and **parentEmailMessageId** for the message the current message was an immediate reply to.

Receiving email follows a very different path. The org.moqui.impl.EmailServices.**poll#EmailServer** service polls a IMAP or POP3 mailbox based on the settings on the EmailServer entity. It takes a single input parameter, the **emailServerId**. Generally this will be run as a scheduled service.

For each message found in the mailbox and not yet marked as seen this service calls the Email ECA (EMECA) rules for it. These are similar to the Entity and Service ECA rules but there is no special trigger, just the receiving of an email. The conditions can be used to only run the actions for a particular to address or tag in the subject like or any other criteria desired.

The context for the condition and actions will include a **`headers`** `Map` with all of the email headers in it (either `String`, or `List` of `String` if there are more than one of the header), and a **`fields`** `Map` with the following: **`toList`**, **`ccList`**, **`bccList`**, **`from`**, **`subject`**, **`sentDate`**, **`receivedDate`**, **`bodyPartList`**. The **`*List`** fields are `List` of `String`, and the **`*Date`** fields are `java.util.Date` objects. For a service that is called directly with this context setup you can implement the `org.moqui.EmailServices.process#EmailEca` interface.

The actions and services they call can do anything with the incoming email. To save the incoming message you can use the `org.moqui.impl.EmailServices.save#EcaEmailMessage` service.

404 - Page Not Found

(not really, this page is intentionally blank for layout reasons; to make it less blank sponsor this book and see your ad here!)

8. System Interfaces

Along with support for user interfaces, Moqui Framework supports various options for interfacing with other systems. There are standards-based options and ways to build more custom system interfaces.

Data and Logic Level Interfaces

System interfaces can generally be divided into two main categories of supporting a step in a process and transferring data (often to keep data updated in another system). For most system integrations a process level one is more flexible and also more focused on a specific part of the system as opposed to transferring all data. Sometimes keeping data consistent between systems is the nature of the integration requirement or the only option available, and then a data level integration is the way to go. Moqui has tools for both logic/process and data level system interfaces.

The best way to trigger outgoing messages is through ECA (event-condition-action) rules, either Service ECA (SECA) rules for a logic level interface or Entity ECA (EECA) rules for a data level interface. See the **Service ECA Rules** and **Entity ECA Rules** sections for details on how to define these.

All ECA rules call actions, typically one or more `service-call` actions, and those actions will call out to whatever system interface is needed. This may be custom code or simply calling an already existing local or remote service. The following sections describe specific tools available in Moqui and with custom code you can implementation any interface and use any additional libraries needed.

XML, CSV and Plain Text Handling

There are various ways to produce and consume XML, CSV, JSON, and other text data using Moqui Framework.

Groovy has a good API for producing and consuming XML with:

- `groovy.util.Node`: The Groovy class that represents a tree node with attributes and child nodes. For XML data each element is represented as a `Node`.
- `groovy.util.XmlNodePrinter`: Print XML text from a tree of `Node` objects.
- `groovy.util.XmlParser`: Read XML text into a tree of `Node` objects.
- `groovy.util.XmlSlurper`: Read XML text into a `GPathResult` object which can be used in Groovy with a syntax similar to XPath expressions to pull out specific parts of a XML element tree.
- `groovy.xml.MarkupBuilder`: Offers a Groovy DSL (domain-specific language) for writing code that has a structure similar to the structure of the XML output. Most useful for scripts that explicitly create and XML tree as opposed to building more dynamically.

There are many other XML libraries written in Java that be used with Moqui such as dom4j and JDOM. If you prefer these just include the JAR files in the Gradle build and code away.

For CSV files Moqui uses the Apache Commons CSV library, and just like with XML files other libraries can be used too. You can see how Moqui uses this in the `org.moqui.impl.entity.EntityDataLoaderImpl.EntityCsvHandler` class.

In Moqui Framework the main tool for repotting and exporting data is the XML Form, especially the list form. XML Screens and Forms can be rendered in various modes including XML, CSV, and plain text. To do this set the **`renderMode`** field in the context either in screen actions or for web requests with a request parameter. This is matched against the `screen-facade.screen-text-output.type` attribute in the Moqui Conf XML file and can be set to any value defined there, including the default Moqui ones (`csv`, `html`, `text`, `xml`, `xsl-fo`) or any that you define in your runtime Moqui Conf XML file.

The XML Form is probably setup for pagination (this is the default). To get all results instead of pagination for an export (or any other reason) set the **`pageNoLimit`** field to `true`. In some cases you will not want to render any of the parent screens that normally decorate the final screen to render, especially for XML files. For CSV files other screen elements are generally ignored. This can be done by setting the **`lastStandalone`** field to `true` meaning that the last screen is rendered standalone and not within parent screens in the screen path. These can be set in screen actions of for web requests as a request parameter.

Just as with other XML Screen and XML Form output modes the FTL macro template used to produce output can be customized by include and override/add. With this approach you can get custom output for a particular screen (including subscreens, so for an entire app or app section, etc) or for everything running in Moqui.

For a detailed example of a screen and form that has CSV, XML, and XSL-FO (PDF) output options see the **List Form View/Export Example** section.

Web Service

XML-RPC and JSON-RPC

Moqui has tools for providing and consuming XML-RPC and JSON-RPC services. Any Service Facade service can be exposed as a remote callable service by setting the `service.allow-remote` attribute to `true`.

The Web Facade has methods to receive these RPC calls: `ec.web.handleXmlRpcServiceCall()` and `ec.web.handleJsonRpcServiceCall()`. In the OOTB webroot component there is a `rpc.xml` screen that has `xml` and `json` transitions that call these methods. With the setup the URL paths for the remote service calls are `/rpc/xml` and `/rpc/json`.

Below is an example of a JSON-RPC service call, using `curl` as the client. It calls the `org.moqui.example.ExampleServices.createExample` service with name, type, and status parameters. It also passes in the username and password to use for authentication before running the service (following a pattern that can be used for any Service Facade service call).

The **`id`** field is always something like `1`. This JSON-RPC field is used for multi-message requests Each message in the request would have a different **`id`** value and that value is used in the **`id`** field in the response. To use this the JSON string would have an outer list containing the individual messages like the one in this example.

```
curl -X POST -H "Content-Type: application/json" \
    --data '{"jsonrpc":"2.0",
"method":"org.moqui.example.ExampleServices.createExample", "id":1,
"params": { "authUsername":"john.doe", "authPassword":"moqui",
"exampleName":"JSON-RPC Test 1", "exampleTypeEnumId":"EXT_MADE_UP",
"statusId":"EXST_IN_DESIGN" } }' \
    http://localhost:8080/rpc/json
```

When you run this you will get a response like (the **`exampleId`** value will vary):

```
{"jsonrpc":"2.0","id":1,"result":{"exampleId":"100050"}}
```

The JSON-RPC implementation in Moqui follows the JSON-RPC 2.0 specification available at: http://www.jsonrpc.org/specification.

XML-RPC requests follow a similar pattern. Moqui uses Apache XML-RPC library (http://ws.apache.org/xmlrpc/) which implements the XML-RPC specification available at: http://xmlrpc.scripting.com/spec.html.

While you can write code call remote XML-RPC and JSON-RPC services by directly using a library (or custom JSON handling code like in `RemoteJsonRpcServiceRunner.groovy`), the easiest way to call remote services is to use a proxy service definition. To do this:

- define a service

- use `remote-xml-rpc` or `remote-json-rpc` for the `service.type` attribute
- set `service.location` to the URL of the RPC server and path (such as `http://localhost:8080/rpc/json`), or to a value matching a service location name in the Moqui Conf XML file (i.e. `service-facade.service-location.name`); there are two OOTB service locations for the purpose of calling remote services: main-xml and main-json; these and additional desired one can be configured in the runtime Moqui Conf XML file and then used in your service locations to simplify configuration, especially when you have different URLs for test and production environments
- set `service.method` to the name of the remote service to call; in JSON-RPC this maps to the `method` field; in XML-RPC this maps to the `methodName` element; when calling another Moqui server this is the name of the service that will be called
- the service can have parameters to define that match the remote service definition, or can be setup to not validate input; you can also define parameters with defaults and specify types for type conversion which are done before the remote service is called

When you call this service locally the Service Facade will call the remote service and return the results. In other words, you call a local service that is a configured proxy to the remote service.

Sending and Receiving Simple JSON

Sometimes an API spec calls for a particular JSON structure or something other than the envelope structure of JSON-RPC. There are some feature in the Web Facade that make this easier.

When a HTTP request is received (really when the Web Facade is initialized) if the `Content-Type` (MIME type) of the request is `application/json` it will parse the JSON string in the request body and if the outer element is a `Map` (in JSON an object) then the entries in that `Map` will be added to the web parameters (`ec.web.parameters`), and web parameters are automatically added to the context (`ec.context`) with a screen is rendered or a screen transition run. If the outer element is a List (in JSON an array) then it is put in a `_requestBodyJsonList` web parameter, and again from there available in the context.

This makes it easy to get at the JSON data in a web request. It also resolves issues with getting the request body after the Web Facade automatically looks for multi-part content in the request body (which the Web Facade always does) because the Servlet container may not allow reading the request body again after this.

For a JSON response you can manually put together the response by setting various things on the `HttpServletResponse` and using the Groovy `JsonBuilder` to produce the JSON text. For convenience the `ec.web.sendJsonResponse(Object responseObj)` method does all of this for you.

To go in the other direction, doing a request to a URL that accepts and responds with JSON, there are special tools because the Groovy and other utilities make this pretty simple. For example, this a variation on the actual code that remotely calls a JSON-RPC service:

```
Map jsonRequestMap = [jsonrpc:"2.0", id:1, method:method,
        params:parameters]
JsonBuilder jb = new JsonBuilder()
jb.call(jsonRequestMap)
String jsonResponse = StupidWebUtilities.simpleHttpStringRequest(location,
        jb.toString(), "application/json")
Object jsonObj = new JsonSlurper().parseText(jsonResponse)
```

This uses the `JsonBuilder` and `JsonSlurper` classes from Groovy and the `StupidWebUtilities.simpleHttpStringRequest()` method which internally uses the Apache HTTP Client library.

RESTful Interface

A RESTful service uses a URL pattern and request method to identify a service instead of a method name like JSON-RPC and XML-RPC. The general idea is to have things like a record represented by URL with the type of record (like an entity or table) as a path element and the ID of the record as one or more path elements (often one for simplicity, i.e., a single field primary key).

When interacting with this record as a web resource the HTTP request method specifies what to do with the record. This is much like the create, update, and delete service verbs for Moqui entity-auto services. The GET method generally does a record lookup. The POST method generally maps to creating a record. The PUT method generally maps to updating a record. The DELETE method does the obvious, a delete.

For examples, such as the one below, see the `ExampleApp.xml` file.

To support RESTful web services we need a way for transitions to be sensitive to the HTTP request method when running in a web-based application. This is handled in Moqui Framework using the `transition.method` attribute, like this:

```
<transition name="ExampleEntity" method="put">
    <path-parameter name="exampleId"/>
    <service-call name="org.moqui.example.ExampleServices.updateExample"
                  in-map="ec.web.parameters" web-send-json-response="true"/>
    <default-response type="none"/>
</transition>
```

To test this transition use a `curl` command something like this to update the **`exampleName`** field of the `Example` entity with an **`exampleId`** of `100010`:

```
curl -X PUT -H "Content-Type: application/json" \
    -H "Authorization: Basic am9obi5kb2U6bW9xdWk=" \
    --data '{ "exampleName":"REST Test - Rev 2" }' \
```

```
http://.../apps/example/ExampleEntity/100010
```

There are some important things to note about this example that make it easier to create REST wrappers around internal Moqui services:

- uses HTTP Basic authentication (john.doe/moqui), which Moqui automatically recognizes and uses for authentication
- uses the automatic JSON body input mapping to parameters (the JSON string must have a Map root object)
- the **`exampleId`** is passed as part of the path and treated as a normal parameter using the `path-parameter` element
- uses the `ec.web.parameters` `Map` as the **`in-map`** to explicitly pass the web parameters to the service (could also use `ec.context` for the entire context which would also include the web parameters, but this way is more explicit and constrained)
- sends a JSON response with the `service-call.web-send-json-response` convenience attribute and a type `none` response

There are various other examples of handling RESTful service requests in the `ExampleApp.xml` file.

Enterprise Integration with Apache Camel

Apache Camel (http://camel.apache.org) is a tool for routing and processing messages with tools for Enterprise Integration Patterns which are described here (and other pages on this site have much other good information about EIP): http://www.eaipatterns.com/toc.html

Moqui Framework has a Message Endpoint for Camel (`MoquiServiceEndpoint`) that ties it to the Service Facade. This allows services (with **`type`**=`camel`) to send the service call as a message to Camel using the `MoquiServiceConsumer`. The endpoint also includes a message producer (`MoquiServiceProducer`) that is available in Camel routing strings as `moquiservice`.

Here are some example Camel services from the `ExampleServices.xml` file:

```
<service verb="localCamelExample" type="camel"
    location="moquiservice:org.moqui.example.ExampleServices.targetCamelExample">
  <in-parameters><parameter name="testInput"/></in-parameters>
  <out-parameters><parameter name="testOutput"/></out-parameters>
</service>
<service verb="targetCamelExample">
  <in-parameters><parameter name="testInput"/></in-parameters>
  <out-parameters><parameter name="testOutput"/></out-parameters>
  <actions>
    <set field="testOutput" value="Here's the input: ${testInput}"/>
    <log level="warn"
        message="targetCamelExample testOutput: ${result.testOutput}"/>
  </actions>
</service>
```

When you call the **`localCamelExample`** service it calls the **`targetCamelExample`** service through Apache Camel. This is a very simple example of using services with Camel. To get an idea of the many things you can do with Camel the components reference is a good place to start:

http://camel.apache.org/components.html

The general idea is you can:

- get message data from a wide variety of sources (file polling, incoming HTTP request, JMS messages, and many more)
- transform messages (supported formats include XML, CSV, JSON, EDI, etc)
- run custom expressions (even in Groovy!)
- split, merge, route, filter, enrich, or apply any of the other EIP tools
- send message(s) to endpoint(s)

Camel is a very flexible and feature rich tool so instead of trying to document and demonstrate more here I recommend these books:

- Instant Apache Camel Message Routing by Bilgin Ibryam
  - http://www.packtpub.com/apache-camel-message-routing/book
  - This book is a quick introduction that will get you going quickly with lots of cool stuff you can do with Camel.
- Apache Camel Developer's Cookbook by Scott Cranton and Jakub Korab
  - http://www.packtpub.com/apache-camel-developers-cookbook/book
  - This book has hundreds of tips and examples for using Camel.
- Camel in Action by Claus Ibsen and Jonathan Anstey
  - http://manning.com/ibsen/
  - This is the classic book on Apache Camel. It covers general concepts, various internal details, how to apply the various EIPs, and a summary of many of the components. The web site for this book also has links to a bunch of useful online resources.

404 - Page Not Found

(not really, this page is intentionally blank for layout reasons; to make it less blank sponsor this book and see your ad here!)

9. Security

Authentication

The main code path for user authentication starts with a call to the `UserFacade.loginUser()` method. This calls into Apache Shiro for the actual authentication. This is basically what the code looks like to authenticate using the Shiro SecurityManager that the ExecutionContextFactoryImpl keeps internally:

```
UsernamePasswordToken token = new UsernamePasswordToken(username, password)
Subject currentUser = eci.getEcfi().getSecurityManager()
    .createSubject(new DefaultSubjectContext())
currentUser.login(token)
```

Shiro is configured by default to use the `MoquiShiroRealm` so this ends up in a call to the `MoquiShiroRealm.getAuthenticationInfo()` method, which authenticates using the `moqui.security.UserAccount` entity and handles things like disabled accounts, keeping track of failed login attempts, etc. Here are the lines from the `shiro.ini` file where this is configured:

```
moquiRealm = org.moqui.impl.MoquiShiroRealm
securityManager.realms = $moquiRealm
```

Shiro can be configured to use other authentication realms such as the CasRealm, JdbcRealm, or JndiLdapRealm classes that come with Shiro. You can also implement your own, or even modify the `MoquiShiroRealm` class to better suit your needs. Shiro has documentation for writing your own realm, and each of these classes has documentation on configuration, such as this JavaDoc for JndiLdapRealm to use it with an LDAP server:

http://shiro.apache.org/static/1.2.3/apidocs/org/apache/shiro/realm/ldap/JndiLdapRealm.html

Back to the `MoquiShiroRealm` that is used by default, here is its default configuration from the `MoquiDefaultConf.xml` file that can be overridden in your runtime Moqui Conf XML file:

```
<user-facade>
    <password encrypt-hash-type="SHA-256" min-length="6" min-digits="1"
        min-others="1" history-limit="5" change-weeks="26"
        email-require-change="true" email-expire-hours="48"/>
    <login max-failures="3" disable-minutes="5" history-store="true"
        history-incorrect-password="true"/>
</user-facade>
```

The `login` element configures the max number of login failures to allow before disabling a `UserAccount` (**`max-failures`**), how long to disable the account when the max failures is reached (**`disable-minutes`**), whether to store a history of login attempts in the `UserLoginHistory` entity (**`history-store`**) and whether to persist incorrect passwords in the history (**`history-incorrect-password`**).

The `password` element is used to configure the password constraints that are checked when creating an account (`org.moqui.impl.UserServices.create#UserAccount`) or updating a password (`org.moqui.impl.UserServices.update#Password`).

Settings include the hash algorithm to use for passwords before persisting them and before comparing an entered password (**`encrypt-hash-type`**; MD5, SHA, SHA-256, SHA-384, SHA512), the minimum password length (**`min-length`**), the minimum number of digit characters in the password (**`min-digits`**), the minimum number of characters other than digits or letters (**`min-others`**), how many old passwords to remember on password change to avoid use of the same password (**`history-limit`**), and how many weeks before forcing a password change (**`change-weeks`**).

The main way to reset a forgotten password is by an email that includes a randomly generated password. The **`email-require-change`** attribute specifies whether to require a change on the first login with the password from the email, making it a temporary password. The **`email-expire-hours`** attribute specifies how many hours before the password in the email expires.

Simple Permissions

The most basic for of authorization (authz) is a permission explicitly checked by code. Artifact-aware authz (covered in the next section) is generally more flexible as it is configured external to the artifact (screen, service, etc) and is inheritable to avoid issues when artifacts (especially services) are reused.

The API method to check permissions is the `ec.user.hasPermission(String userPermissionId)` method. A user has a permission if the user is a member (`UserGroupMember`) of a group (`UserGroup`) that has the permission (`UserGroupPermission`). The **`userPermissionId`** may point to a `UserPermission` record, but it may also be any arbitrary text value as the UserGroupPermission has no foreign key to UserPermission.

Artifact-Aware Authorization

The artifact-aware authorization in Moqui enables external configuration of access to artifacts such as screens, screen transitions, services, and even entities. With this approach there is no need to add code or configuration to each artifact to check permissions or otherwise see if the current user has access to the artifact.

Artifact Execution Stack and History

The `ArtifactExecutionFacade` is used by all parts of the framework to keep track of each artifact as it executes. It keeps a stack of the currently executing artifacts, each one pushed on the stack as it begins (with one of the **`push()`** methods) and popped from the stack as it ends (with the **`pop()`** method). As each artifact is pushed on to the stack it is also added to a history of all artifacts used in the current `ExecutionContext` (i.e., for a single web request, remote service call, etc).

Use the `ArtifactExecutionInfo` **`peek()`** method to get info about the artifact at the top of the stack, `Deque<ArtifactExecutionInfo>` **`getStack()`** to get the entire current stack, and `List<ArtifactExecutionInfo>` **`getHistory()`** to get a history of all artifacts executed.

This is important for artifact-aware authorization because authz records are inheritable. If an artifact authz is configured inheritable then not only is that artifact authorized but any artifact it uses is also authorized.

Imagine a system with hundreds of screens and transitions, thousands of services, and hundreds of entities. Configuring authorization for every one of them would require a massive effort to both setup initially and to maintain over time. It would also be very prone to error, both incorrectly allowing and denying access to artifacts and resulting in exposure of sensitive data or functionality, or runtime errors for users trying to perform critical operations that are a valid part of their job.

The solution is inheritable authorization. With this you can setup access to an entire application or part of an application with authz configuration for a single screen that all sub-screens, transitions, services, and entities will inherit. To limit the scope sensitive services and entities can have a deny authz that overrides the inheritable authz, requiring special authorization to those artifacts. With this approach you have a combination of flexibility, simplicity, and granular control of sensitive resources.

This is also used to track performance metrics for each artifact. See the **Artifact Execution Runtime Profiling** section for details.

Artifact Authz

The first step to configure artifact authorization is to create a group of artifacts. This involves a `ArtifactGroup` record and a `ArtifactGroupMember` record for each artifact, or artifact name pattern, in the group.

For example here is the artifact group for the Example app with the root screen (`ExampleApp.xml`) as a member of the group:

```
<moqui.security.ArtifactGroup artifactGroupId="EXAMPLE_APP"
    description="Example App (via root screen)"/>
<moqui.security.ArtifactGroupMember artifactGroupId="EXAMPLE_APP"
    artifactTypeEnumId="AT_XML_SCREEN" inheritAuthz="Y"
    artifactName="component://example/screen/ExampleApp.xml"/>
```

In this case the **`artifactName`** attribute has the literal value for the location of the screen. It can also be a pattern for the artifact name (with **`nameIsPattern`**`="Y"`), which is especially useful for authz for all services or entities in a package. Here is an example of that for all services in the `org.moqui.example` package, or more specifically all services whose full name matches the regular expression `"org\.moqui\.example\..*"`:

```
<moqui.security.ArtifactGroupMember artifactGroupId="EXAMPLE_APP"
    artifactName="org\.moqui\.example\..*" nameIsPattern="Y"
    artifactTypeEnumId="AT_SERVICE" inheritAuthz="Y"/>
```

The next step is to configure authorization for the artifact group with a `ArtifactAuthz` record. Below is an example of a record that gives the `ADMIN` group always (`AUTHZT_ALWAYS`) access for all actions (`AUTHZA_ALL`) to the artifacts in the `EXAMPLE_APP` artifact group setup above.

```
<moqui.security.ArtifactAuthz artifactAuthzId="EXAMPLE_AUTHZ_ALL"
    userGroupId="ADMIN" artifactGroupId="EXAMPLE_APP"
    authzTypeEnumId="AUTHZT_ALWAYS" authzActionEnumId="AUTHZA_ALL"/>
```

The always type (**`authzTypeEnumId`**) of authorization overrides deny (`AUTHZT_DENY`) authorizations, unlike the allow authz (`AUTHZT_ALLOW`) which is overridden by deny. The other options for the authz action (**`authzActionEnumId`**) include view (`AUTHZA_VIEW`), create (`AUTHZA_CREATE`), update (`AUTHZA_UPDATE`), and delete (`AUTHZA_DELETE`) in addition to all (`AUTHZA_ALL`).

For example here is a record that grants only view authz with the type allow (so can be denied) of the same artifact group to the `EXAMPLE_VIEWER` group:

```
<moqui.security.ArtifactAuthz artifactAuthzId="EXAMPLE_AUTHZ_VW"
    userGroupId="EXAMPLE_VIEWER" artifactGroupId="EXAMPLE_APP"
    authzTypeEnumId="AUTHZT_ALLOW" authzActionEnumId="AUTHZA_VIEW"/>
```

Entity artifact authorization can also be restricted to particular records using the `ArtifactAuthzRecord` entity. This is used with a view entity (**`viewEntityName`**) that joins between the **`userId`** of the currently logged in user and the desired record. If the name of the

field with the **userId** is anything other than **userId** specify its name with the **userIdField** field. The record level authz is checked by doing a query on the view entity with the current **userId** and the PK fields of the entity the operation is being done on. To add constraints to this query you can add them to the `view-entity` definition, use the **filterByDate** attribute, or use `ArtifactAuthzRecordCond` records to specify conditions.

If authorization fails when an artifact is used the framework creates a `ArtifactAuthzFailure` record with relevant details.

Artifact Tarpit

An artifact tarpit limits the velocity of access to artifacts in a group. Here is an example of an artifact group for all screens and a `ArtifactTarpit` to restrict access for all users to each screen for 60 seconds (**tarpitDuration**) if there are more than 120 hits (**maxHitsCount**) within 60 seconds (**maxHitsDuration**).

```
<moqui.security.ArtifactGroup artifactGroupId="ALL_SCREENS"
    description="All Screens"/>
<moqui.security.ArtifactGroupMember artifactGroupId="ALL_SCREENS"
    artifactName=".*" nameIsPattern="Y"
    artifactTypeEnumId="AT_XML_SCREEN"/>
<moqui.security.ArtifactTarpit userGroupId="ALL_USERS"
    artifactGroupId="ALL_SCREENS" maxHitsCount="120"
    maxHitsDuration="60" tarpitDuration="60"/>
```

When a particular user (**userId**) exceeds the configured velocity limit for a particular artifact (**artifactName**) or a particular type (**artifactTypeEnumId**) the framework creates a `ArtifactTarpitLock` record to restrict access to that artifact by the user until a certain date/time (**releaseDateTime**).

404 - Page Not Found

(not really, this page is intentionally blank for layout reasons; to make it less blank sponsor this book and see your ad here!)

10. Performance

Performance Metrics

Artifact Hit Statistics

Moqui keeps statistics about use (hits) and timing for artifacts according to the configuration in the `server-stats.artifact-stats` elements in the Moqui Conf XML file. Here is the default configuration (in `MoquiDefaultConf.xml`) that you can override in the runtime conf file. The default development runtime conf file (`MoquiDevConf.xml`) has settings that record even more than this.

```
<server-stats bin-length-seconds="900" visit-enabled="true"
    visitor-enabled="true">
  <artifact-stats type="screen" persist-bin="true" persist-hit="true"/>
  <artifact-stats type="screen-content" persist-bin="true"
      persist-hit="true"/>
  <artifact-stats type="transition" persist-bin="true" persist-hit="true"/>
  <artifact-stats type="service" persist-bin="true" persist-hit="false"/>
  <artifact-stats type="entity" persist-bin="false"/>
</server-stats>
```

These settings create a `ArtifactHit` record for each hit to a `screen`, `screen-content` (content under a screen), and screen `transition`. They also create `ArtifactHitBin` records for those plus `service` calls.

Here are a couple of examples of `ArtifactHit` records, the first for a hit to the `FindExample.xml` screen and the second for a hit to the `EntityExport.xml` transition in the `DataExport.xml` screen in the tools application. The hit to the `EntityExport.xml` transition has parameters which are recorded in the **`parameterString`** attribute.

```
<moqui.server.ArtifactHit hitId="100030" visitId="100000"
    userId="EX_JOHN_DOE" artifactType="screen" artifactSubType="text/html"
    artifactName="component://example/screen/ExampleApp/Example/FindExample.xml"
    startDateTime="1406670788608" runningTimeMillis="1,499" wasError="N"
    requestUrl="http://localhost:8080/apps/example/Example/FindExample"
    serverIpAddress="172.16.7.38" serverHostName="DEJCMBA3.local"
    lastUpdatedStamp="1406670790120"/>
```

```
<moqui.server.ArtifactHit hitId="100037" visitId="100001"
    userId="EX_JOHN_DOE" artifactType="transition"
    artifactName="component://tools/screen/Tools/Entity/
        DataExport.xml#EntityExport.xml"
    parameterString="moquiFormName=ExportData,output=file,filterMap=
        [artifactType:"screen"],entityNames=moqui.server.ArtifactHit"
    startDateTime="1406674728129" runningTimeMillis="45" wasError="N"
    requestUrl="http://localhost:8080/apps/tools/Entity/DataExport/
        EntityExport.xml"
    serverIpAddress="172.16.7.38" serverHostName="DEJCMBA3.local"
    lastUpdatedStamp="1406674728195"/>
```

In a web application there is a `Visit` record for each session that has details about the session and ties together `ArtifactHit` records by the **`visitId`**. The `Visit` will keep track of the logged in **`userId`** once a user is logged in, but even before that visits are tied together using a **`visitorId`** that is tracked on the service in a `Visitor` record and in a browser/client with a cookie to tie sessions together, even if no user is logged in during a session.

```
<moqui.server.Visit visitId="100000" visitorId="100000"
    userId="EX_JOHN_DOE" sessionId="749389362bac39c39de3c77769b9b485"
    serverIpAddress="172.16.7.38" serverHostName="DEJCMBA3.local"
    webappName="ROOT" initialLocale="en_US"
    initialRequest="http://localhost:8080/" initialUserAgent="Mozilla/5.0
        (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.77.4 (KHTML,
        like Gecko) Version/7.0.5 Safari/537.77.4"
    clientIpAddress="0:0:0:0:0:0:0:1" clientHostName="0:0:0:0:0:0:0:1"
    fromDate="1406670784083" lastUpdatedStamp="1406670784396"/>
<moqui.server.Visitor visitorId="100000" createdDate="1406670784353"
    lastUpdatedStamp="1406670784363"/>
```

There is a performance impact for creating a record for each hit on an artifact, and on busy servers the database size can get very large. This can be mitigated by using a low-latency insert database such as OrientDB or other NoSQL databases. If you just want statistics of performance over a time period and don't need the individual hit records for auditing or detailed analysis the `ArtifactHitBin` records will do the trick.

These records have a summary of hits for an artifact during a time period, between **`binStartDateTime`** and **`binEndDateTime`**. The length of the bin is configured with the `server-stats`**`.bin-length-seconds`** attribute and defaults to 900 seconds (15 minutes).

Here is an example of a hit bin for the **`create#moqui.entity.EntityAuditLog`** service. In this example it has been hit/used 77 times with a total (cumulative) run time of 252ms which means the average run time for the artifact in the bin is 3.27ms.

```
<moqui.server.ArtifactHitBin hitBinId="100010" artifactType="service"
    artifactSubType="entity-implicit"
    artifactName="create#moqui.entity.EntityAuditLog"
    serverIpAddress="172.16.7.38" serverHostName="DEJCMBA3.local"
    binStartDateTime="1406268616369" binEndDateTime="1406268636249"
    hitCount="77" totalTimeMillis="252" minTimeMillis="1"
```

```
    maxTimeMillis="61" lastUpdatedStamp="1406268636290"/>
```

These can be used directly from the database and with the **Artifact Bins** and **Artifact Summary** screens in the Tools application.

Artifact Execution Runtime Profiling

Java profilers such as JProfiler are great tools for analyzing the performance of Java methods but know nothing about Moqui artifacts such as screens, transitions, services, and entities. The Moqui Artifact Execution Facade keeps track of performance details of artifacts in memory for each instance (each `ExecutionContext`, such as a web request, etc) as they run.

This data is kept in with the `ArtifactExecutionInfo` objects that are created as each artifact runs and are pushed onto the execution stack and kept in the execution history. You can access these using the `ec.artifactExecution.getStack()`, and `ec.artifactExecution.getHistory()` methods.

From the `ArtifactExecutionInfo` instance you can get its own runtime (long `getRunningTime()`), the artifact that called it (`ArtifactExecutionInfo getParent()`), the artifacts it calls (`List<ArtifactExecutionInfo> getChildList()`), the running time of all artifacts called by this artifact (`long getChildrenRunningTime()`), and based on that the running time of just this artifact (`long getThisRunningTime()`, which is `getRunningTime() - getChildrenRunningTime()`). You can also print a report with these stats for the current artifact info and optionally its children recursively using the `print(Writer writer, int level, boolean children)` method.

For a complex code section like placing an order that does dozens of service calls this can be a lot of data. To make it easier to track down the parts that are taking the most time have this method on the ArtifactExecutionInfoImpl class to generate a list of hot spots:

```
static List<Map> hotSpotByTime(List<ArtifactExecutionInfoImpl> aeiiList,
    boolean ownTime, String orderBy)
```

This goes through all ArtifactExecutionInfoImpl instances in the execution history and sums up stats to create a Map for each artifact with the following entries: time, timeMin, timeMax, count, name, actionDetail, artifact type, and artifact action.

Another situation where you'll have a LOT of data is when running a process many times to get better average statistics. In this case you could have hundreds or thousands of artifact execution infos in the history. To consolidate data from multiple runs into a single tree of info about the execution of each artifact and its children use this method:

```
List<Map> consolidateArtifactInfo(List<ArtifactExecutionInfoImpl> aeiiList)
```

Each `Map` has these entries: time, thisTime, childrenTime, count, name, actionDetail, childInfoList, key (which is: `name + ":" + typeEnumId + ":" + actionEnumId + ":" + actionDetail`), type, and action. With that result you can print the tree with indentation in plain text (best displayed with a fixed width font) with this method:

```
String printArtifactInfoList(List<Map> infoList)
```

One example of using these methods is the `TestOrders.xml` screen in the POP Commerce application. It is used with a URL like this and display a screen with the performance profile results of the code that places and ships the specified number of orders:

```
http://localhost:8080/popc/TestOrders?numOrders=10
```

Here is a snippet from the screen actions script that runs the test code and gets the performance statistics using the methods described above:

```
def artifactHistory = ec.artifactExecution.history
ownHotSpotList = ArtifactExecutionInfoImpl.hotSpotByTime(artifactHistory,
        true, "-time")
totalHotSpotList = ArtifactExecutionInfoImpl.hotSpotByTime(artifactHistory,
        false, "-time")

List<Map> consolidatedList =
    ArtifactExecutionInfoImpl.consolidateArtifactInfo(artifactHistory)
String printedArtifactInfo =
    ArtifactExecutionInfoImpl.printArtifactInfoList(consolidatedList)
```

Here is an example of the top few rows in the **Artifacts by Own Time** section of the output on that screen for the placing and shipping of 25 orders:

Time	Time Min	Time Avg	Time Max	Count	Name	Type	Action	Action Detail
1838	0	2.29	25	801	mantle.order.OrderItem	Entity	View	list
1093	0	1.32	26	825	mantle.ledger.account. GlAccountOrgTimePeriod	Entity	Update	
1025	0	1.08	10	950	moqui.entity.EntityAuditLog	Entity	Create	
844	7	11.25	33	75	mantle.product.PriceServices. get#ProductPrice	Service	All	
686	0	3.43	12	200	mantle.order.OrderPart	Entity	Update	

From these results we can see that the most time is spent doing an Entity View (find) list operation on the `OrderItem` entity. In this run the transaction cache for the **`place#Order`** and **`ship#OrderPart`** services was disabled, and the `OrderItem` entity is not cached using the entity cache so it is doing that query 801 times during this run. The transaction cache is a write-through cache that will cache written records and reads like this. With that enabled overall the orders per second goes from around 0.8 to 1.4 (on my laptop with a Derby database) and the output for **Artifacts by Own Time** looks very different:

Time	Time Min	Time Avg	Time Max	Count	Name	Type	Action	Action Detail
3449	72	137.96	222	25	mantle.shipment. ShipmentServices. ship#OrderPart	Service	All	
1284	0	1.60	10	801	mantle.order.OrderItem	Entity	View	list

Time	Time Min	Time Avg	Time Max	Count	Name	Type	Action	Action Detail
679	6	9.05	14	75	mantle.product.PriceServices.get#ProductPrice	Service	All	
614	14	24.56	51	25	mantle.order.OrderServices.place#Order	Service	All	
561	0	0.68	5	825	mantle.ledger.account.GlAccountOrgTimePeriod	Entity	View	one

Below is some sample output from the **Consolidated Artifacts Tree** section. It shows the hierarchy of artifacts consolidated across runs and within each run to show the data for each artifact in the context of parent and child artifacts. When interpreting these results note that the total counts and times for each artifact are not just the values for that artifact running as a child of the parent artifact shown, but all runs of that artifact. The main value is tracking down where the busiest artifacts are used, and understanding exactly what is actually done at runtime, especially for specific services.

In this output each line is formatted as follows:

```
[${time}:${thisTime}:${childrenTime}][${count}] ${type} ${action} ${actionDetail} ${name}
```

Here is the sample output, note that certain artifact names have been shortened with ellipses for better formatting:

```
[   16:  3: 13][  2] Screen   View         component://webroot/screen/webroot.xml
| [   13:-41: 54][  3] Screen   View         component://PopCommerce/…/PopCommerceRoot.xml
| | [  165:165:  0][126] Entity   View   one   mantle.product.store.ProductStore
| | [    0:-31263:31263][  3] Screen   View         component://PopCommerce/…/TestOrders.xml
| | | [    3:  3:  0][  3] Entity   View   one   moqui.security.UserAccount
| | | [    5:  5:  0][  1] Entity   View   one   moqui.server.Visit
| | | [    6:  1:  5][  1] Service  Create       create#moqui.security.UserLoginHistory
| | | | [    5:  5:  0][  1] Entity   Create       moqui.security.UserLoginHistory
| | | [ 4700:269:4431][ 75] Service  All          …OrderServices.add#OrderProductQuantity
| | | | [  632:632:  0][300] Entity   View   list  mantle.order.OrderPart
| | | | [  497:497:  0][375] Entity   View   one   mantle.order.OrderPart
| | | | [  165:165:  0][126] Entity   View   one   mantle.product.store.ProductStore
| | | | [  195:195:  0][ 25] Entity   View   list  mantle.order.OrderHeaderAndPart
| | | | [  328: 21:307][ 25] Service  Create       mantle.order.OrderServices.create#Order
| | | | | [  146: 12:134][ 25] Service  Create       create#mantle.order.OrderHeader
| | | | | | [  134: 97: 37][ 25] Entity   Create       mantle.order.OrderHeader
| | | | | | | [ 1564:406:1158][950] Service  Create       create#moqui.entity.EntityAuditLog
| | | | | | | | [   83: 83:  0][ 30] Entity   View   one   moqui.entity.SequenceValueItem
| | | | | | | | [   90: 90:  0][ 30] Entity   Update       moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:  0][950] Entity   Create       moqui.entity.EntityAuditLog
| | | | | [  161: 11:150][ 25] Service  Create       create#mantle.order.OrderPart
| | | | | | [  632:632:  0][300] Entity   View   list  mantle.order.OrderPart
| | | | | | [  134: 99: 35][ 25] Entity   Create       mantle.order.OrderPart
| | | | | | | [ 1564:406:1158][950] Service  Create       create#moqui.entity.EntityAuditLog
| | | | | | | | [   83: 83:  0][ 30] Entity   View   one   moqui.entity.SequenceValueItem
| | | | | | | | [   90: 90:  0][ 30] Entity   Update       moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:  0][950] Entity   Create       moqui.entity.EntityAuditLog
| | | | [ 1838:1838:  0][801] Entity   View   list  mantle.order.OrderItem
| | | | [  882:844: 38][ 75] Service  All          …PriceServices.get#ProductPrice
| | | | | [   38: 38:  0][150] Entity   View   list  mantle.product.ProductPrice
| | | | [ 2324: 83:2241][ 75] Service  Create       …OrderServices.create#OrderItem
| | | | | [  430:430:  0][575] Entity   View   one   mantle.product.Product
| | | | | [ 2747: 64:2683][100] Service  Create       create#mantle.order.OrderItem
```

```
| | | | | | [ 1838:1838:  0][801] Entity   View   list  mantle.order.OrderItem
| | | | | | [ 2482:384:2098][100] Entity   Create       mantle.order.OrderItem
| | | | | | | [ 1564:406:1158][950] Service  Create       create#moqui.entity.EntityAuditLog
| | | | | | | | [   83: 83:  0][ 30] Entity   View   one   moqui.entity.SequenceValueItem
| | | | | | | | [   90: 90:  0][ 30] Entity   Update       moqui.entity.SequenceValueItem
| | | | | | | | [ 1025:1025:  0][950] Entity   Create       moqui.entity.EntityAuditLog
| | | | | | | [ 1784: 89:1695][100] Service  Update       …OrderServices.update#OrderPartTotal
| | | | | | | | [ 1838:1838:  0][801] Entity   View   list  mantle.order.OrderItem
| | | | | | | | [  322:127:195][250] Service  All          …OrderServices.get#OrderItemTotal
| | | | | | | | | [ 1838:1838:  0][801] Entity   View   list  mantle.order.OrderItem
| | | | | | | | [  497:497:  0][375] Entity   View   one   mantle.order.OrderPart
| | | | | | | | [ 1204:686:518][200] Entity   Update       mantle.order.OrderPart
| | | | | | | | | [  224:224:  0][200] Entity   View   refresh mantle.order.OrderPart
| | | | | | | | | [ 1564:406:1158][950] Service  Create       create#…EntityAuditLog
| | | | | | | | | | [   83: 83:  0][ 30] Entity   View   one   moqui.entity.SequenceValueItem
| | | | | | | | | | [   90: 90:  0][ 30] Entity   Update       moqui.entity.SequenceValueItem
| | | | | | | | | | [ 1025:1025:  0][950] Entity   Create       moqui.entity.EntityAuditLog
| | | | | | | | [  629: 56:573][100] Service  Update       …update#OrderHeaderTotal
| | | | | | | | | [  632:632:  0][300] Entity   View   list  mantle.order.OrderPart
| | | | | | | | | [  349:349:  0][450] Entity   View   one   mantle.order.OrderHeader
| | | | | | | | | [  884:592:292][175] Entity   Update       mantle.order.OrderHeader
| | | | | | | | | | [  181:181:  0][175] Entity   View   refresh mantle.order.OrderHeader
| | | | | | | | | | [ 1564:406:1158][950] Service  Create       create#…EntityAuditLog
| | | | | | | | | | | [   83: 83:  0][ 30] Entity   View   one   …SequenceValueItem
| | | | | | | | | | | [   90: 90:  0][ 30] Entity   Update       …SequenceValueItem
| | | | | | | | | | | [ 1025:1025:  0][950] Entity   Create       moqui.entity.EntityAuditLog
```

Improving Performance

Once an artifact or code block has been identified a taking up a lot of execution time the next step is to review it and see if any part of it can be improved. Sometimes operations just take time and there isn't much to be done about it. Even in those cases parts can be made asynchronous or other approaches used to at least minimize the impact on users or system resources.

The slowest operations typically involve database or file access and in-memory caching can help a lot with this. The Moqui Cache Facade is used by various parts of the framework and can be used directly by your code for caching as needed. By default Moqui uses ehcache for the actual caching, and the configuration settings in the Moqui Conf XML file are passed through to it. Other cache configuration is ehcache specific and can be setup using its files (mainly `ehcache.xml`). This is especially true for setting up things like a distributed caching in an app server cluster.

In the runtime configuration for development (`MoquiDevConf.xml`) the caches for artifacts such as entities, service definitions, XML Screens, scripts, and templates have a short timeout so that they are reloaded frequently for testing after changing a file. In the production configuration (`MoquiProductionConf.xml`) the caches are all used fully to get the best performance. When doing performance testing make sure you are running with the caches

fully used, i.e. with production settings, so that numbers are not biased by things that are quite slow and won't happen in production.

The Resources Facade does a lot of caching. The `getLocationText(String location, boolean cache)` method uses the `resource.text.location` cache is the cache parameter is set to true. Other caches are always used including scripts and templates in their compiled form (if possible with the script interpreter or template renderer), and even the Groovy expressions and string expansions done by the Resource Facade. As mentioned above these are never "disabled" but to facilitate runtime reloading the easiest approach is to use a timeout on the desired caches.

Another common cache is the entity cache managed by the Entity Facade. There are caches for individual records, list results, and count results. These caches are cleared automatically when records are created, updated, or deleted through the Entity Facade. Both simple entities that correspond to a single table and view entities can be cached, and the automatic cache clearing works for both. To make cache clearing more efficient it uses a reverse association cache by default to lookup cache entries by the entity name and PK values of a record. In other cases (such as when creating a record) it must do a scan of the conditions on cache entries to find matching entries to clear, especially on list and count caches. For more details see the **Data and Resources** chapter.

In addition to the entity read cache there is a write-through per-transaction cache that can be enabled with the `service.transaction` attribute by setting it to `cache` or `force-cache`. The implementation of this is in the `TransactionCache.groovy` file.

The basic idea is that when creating, updating, or deleting a record it just remembers that in an object that is associated with the transaction instead of actually writing it to the database. When the transaction is committed, but before the actual commit, it writes the changes to the database. When find operations are done it uses the values in cache directly or augments the query results from the database with values in the cache.

It is even smart enough to know when finding with a constraint that could only match values in the TX cache (created through it) that there is no need to go to the database at all and the query is handled fully in memory. For example if you create a `OrderHeader` record and then various `OrderItem` records and then query all `OrderItem` records by `orderId` it will see if the `OrderHeader` record was created through the transaction cache and if so it will just get the `OrderItem` records from the TX cache and not query the database at all for them.

For entity find operations another valuable tool is the auto-minimize of view entities. When you do a find on a large view-entity, such as the `FindPartyView` entity, just make sure to specify the fields to select and limit those to only the fields you need. The Entity Facade will automatically look at the fields selected, used in conditions, and used to order/sort the results and only include the aliased fields and member entities necessary for those fields. With this approach there is no need to use a dynamic view entity (`EntityDynamicView`) to conditionally add member entities and aliased fields. Back to the FindPartyView example,

the **`find#Party`** service (implemented in `findParty.groovy`) uses this to provide a large number of options with very minimal code.

A general guideline when querying tables with a very large number of records is to not ask the database to do more than is absolutely necessary. Joining in too many member entities in a view entity is a dramatic form of this as creating large temporary tables is a very expensive operation.

Along these lines another common scenario is doing a find that may return a very large number of results and then showing those results one page (like 20 records) at a time. It is best to not select all the data you'll display for each record in the main query as this makes the temporary table for joins much larger, and you are asking the database to get that data for all records instead of just the 20 or so you will be displaying. A better approach is to just query the field or fields sufficient to identify the records, then query the data to display for just those keys in a separate find. This is usually much faster, but in some rare cases it is not so it is still good to test these and other query variations with real data to see which performs best for your specific scenario.

In high volume production ecommerce and ERP systems another common problem is synchronization and locking delays. These can happen within the app server with Java synchronization, or in a database with locks and lock waiting. You may also find deadlocks, but that is another issue (i.e., separate from performance). The only way to really find these is with load testing, especially load testing that uses the same resources as much as possible like a bunch of orders for the same product as close to the same time as possible.

There are a few ways to improve these. On the Java synchronization level using non-blocking algorithms and data structures can make a huge difference, and many libraries are moving this way. Java Concurrency in Practice by Brian Goetz is a good book on this topic.

Beyond these basic things to keep in mind there are countless ways to improve performance. For really important code, especially highly used or generally performance sensitive functionality, within reasonable constraints the only limit to how much faster it can run is often a matter of how much time and effort can be put into performance testing and optimization.

Sometimes this involves significant creativity and using very different architectures and tools to handle things like a large number of users, a very large amount of data, data scattered in many places, and so on. For some of these issues distributed processing or data storage tools such as Hadoop and OrientDB (and really countless others these days) may be just what you need, even if using them requires significantly more effort and it only makes sense to do so for very specific functionality.

When doing Java profiling with a tool like JProfiler you are usually looking for different sorts of things that impact performance than when looking at Moqui artifact execution performance data. To optimize Java methods (and classes for memory use) there are different tools and guidelines to use than the ones above which are more for optimizing business logic at a higher level.

11. The Tools Application

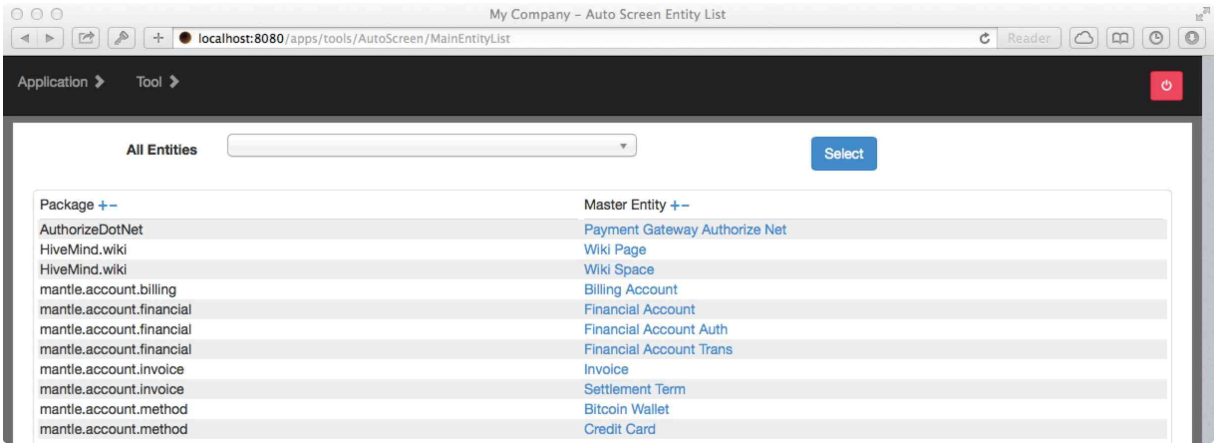
The Tools application is part of the default Moqui runtime and lives in the component at `moqui/runtime/component/tools`. It has screens for technical administration of systems built on Moqui Framework such as viewing and editing data, running services, managing jobs, managing caches, and viewing statistics about server use.

Auto Screen

Auto screens are based on entity definitions and use the default forms generated by a XML Form with auto form fields based on the fields for a given entity. There are screens to find and create values, edit exiting values, and view related values for an entity.

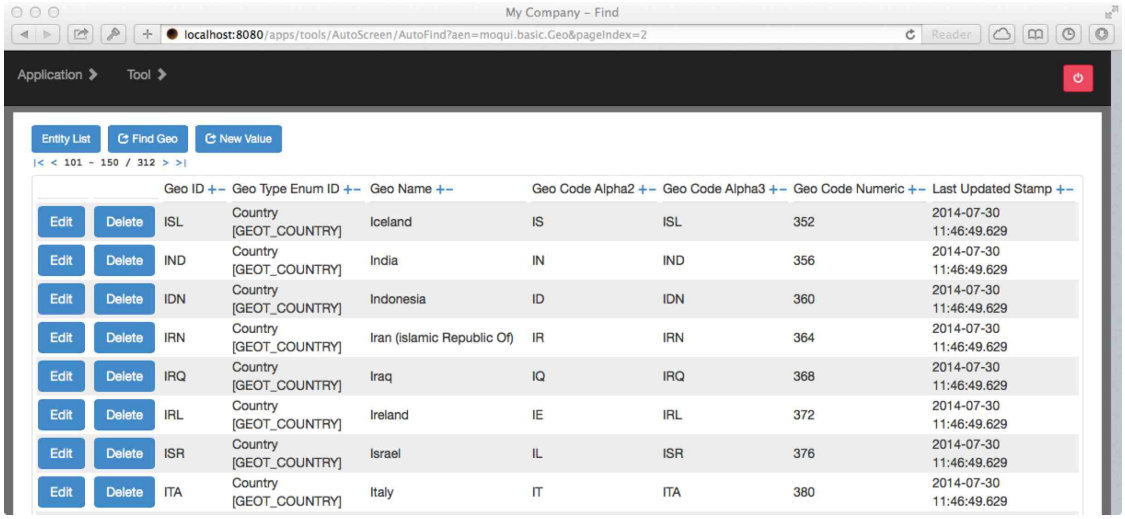
Entity List

The main entity list for auto screens has a drop-down at the top with all entities plus a list of the master entities to select from. Master entities are entities with dependents and are the most useful to find and view with a tab set for their dependent and related entities, though any entity can be used with the auto screens. Select an entity to go to its find page.

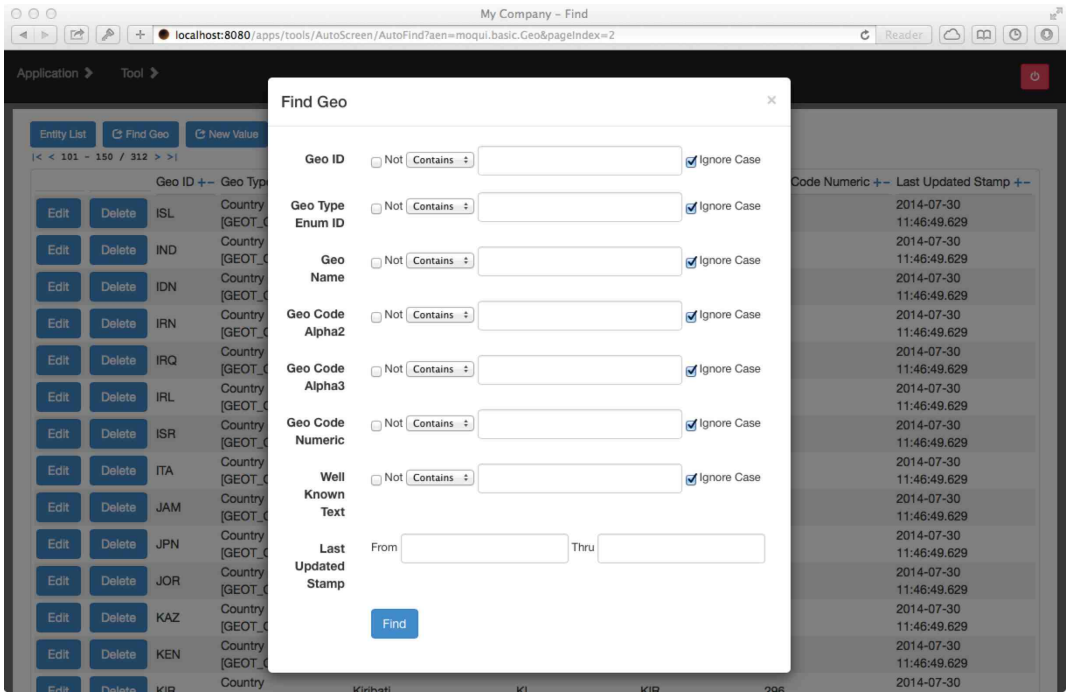


Find Entity

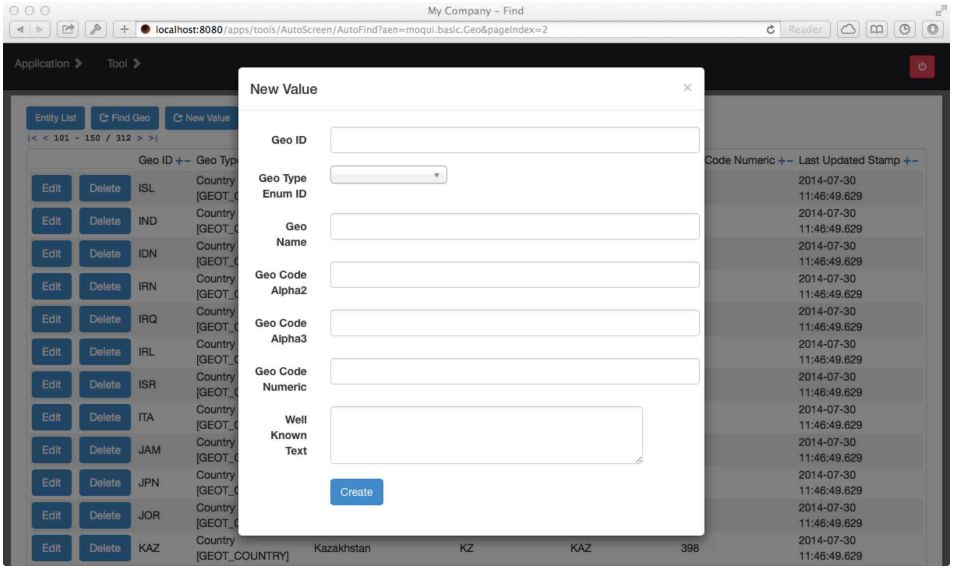
The find screen has a paginated list of records for the selected entity with Edit and Delete buttons for each, the Edit button going to the Edit Entity screen. The table has auto generated view fields based on the entity fields in a `form-list`. The Entity List button goes back to the list of master and all entities. The Find button pops a form with filter inputs for each entity field, and the New Value button pops up a form to create a new record.



Here is the Find form for the Geo entity that pops up.

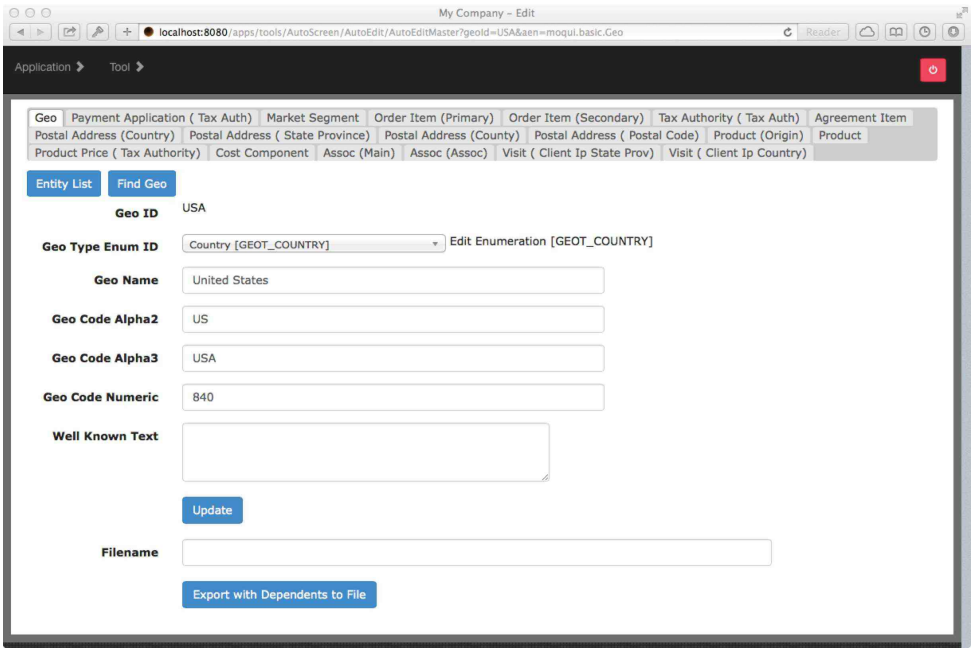


Here is the New Value form that pops up for the Geo entity.



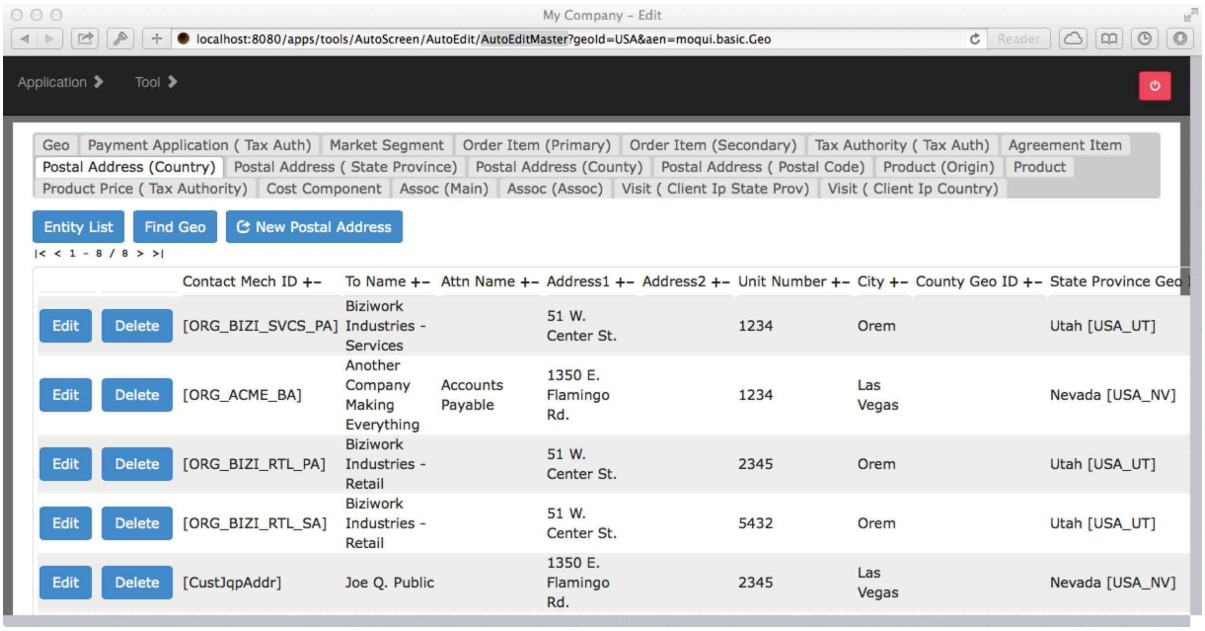
Edit Entity

The edit entity screen has tabs for the current entity and all related entities. It has an auto-generated edit form (`form-single`) based on the entity definition, including drop-downs for fields that are foreign keys to other records. There is also a simple form at the bottom to export the record and its dependent records to a file (like the Entity Export screen). Here is an example for the USA Geo record:



Edit Related

When you click on a tab for a related entity from the edit screen you get a list of the related records with Edit and Delete links for each just like the Entity Find screen. It is a `form-list` with fields auto generated from the entity fields. You also get Entity List, Find, and New value buttons like the find screen. This example shows the Postal Address records with the same Geo (USA) set as the Country.

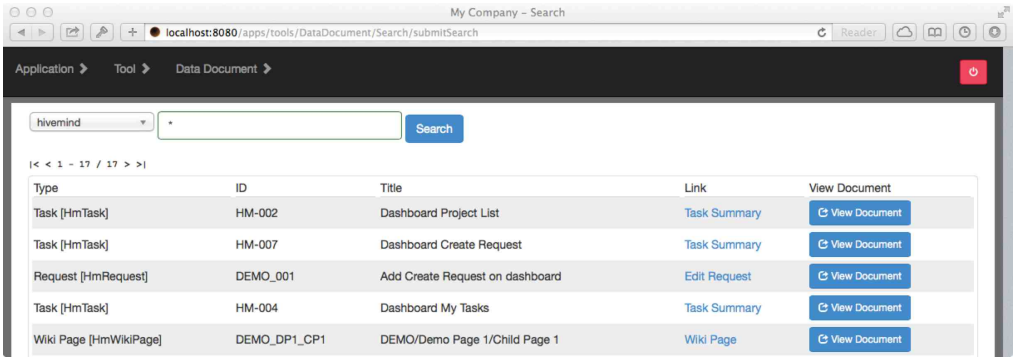


Data Document

Entity data documents are covered in the **Data Document** section of the **Data and Resources** chapter. These screens in the Tools application allow you to search documents, index documents for defined data feeds, and export data documents as JSON files.

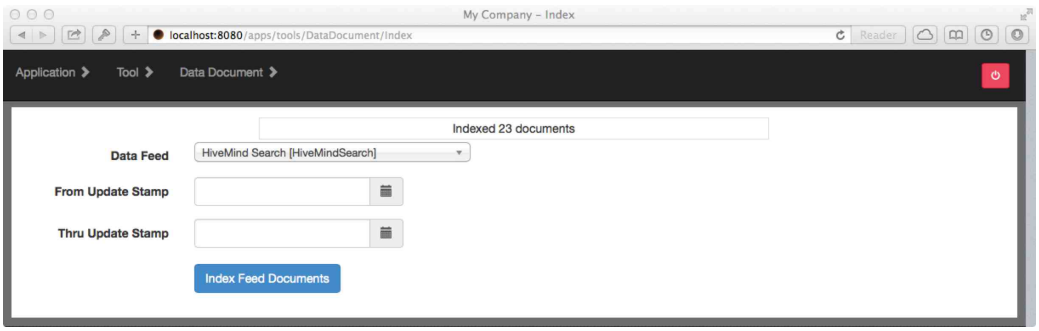
Search

Use the search screen to find data documents in an index, such as the hivemind index in this example. The links are based on the `DataDocumentLink` record to go a screen associated with a document in the corresponding application. The View Document button pops up a window with the full document in JSON text and a print of the flattened map for the document.



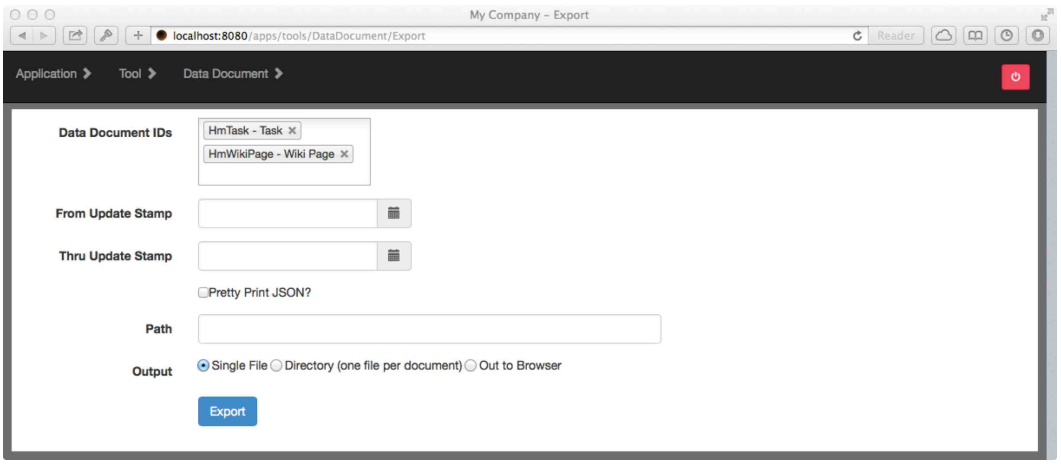
Index

With the data document index screen you can select a Data Feed and optionally specify from and thru timestamps to limit the documents by the **`lastUpdateStamp`** field automatically added by the entity facade, and then index all data documents associated with the feed.



Export

Use this screen to export data documents from the specified IDs and within the from/thru **`lastUpdateStamp`** range to a single file, directory of doc files, or out to the browser.

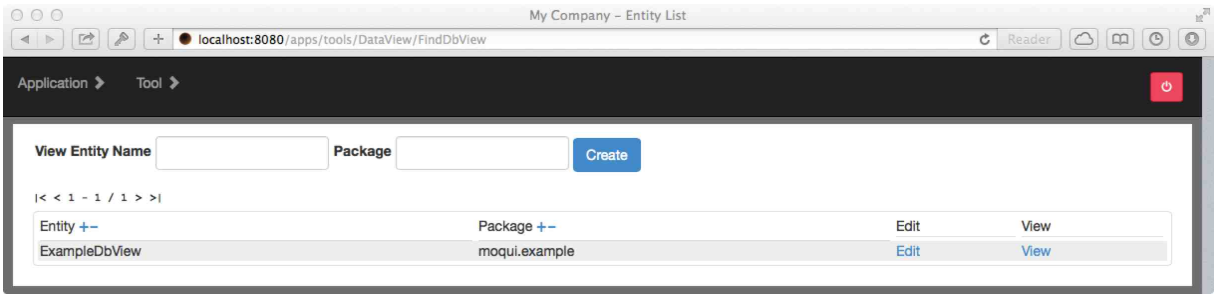


Data View

The data view screens are used to define a simple view entity stored in the database (using the `DbViewEntity` and related entities) and then view the results and export them as a CSV file. These screens are a simple form of ad-hoc report and data export that leverage the concept of master and dependent entities and allow for easy aliasing of fields on a master entity and all directly related dependents with an optional function. More elaborate DB view entities can be defined and viewed/exported from these screens, but the Edit DB View screen only supports a master entity and the entities directly related to it.

Find DB View

The find screen has a form at the top to create a `DbViewEntity` and then table with all existing DB view entities and links to Edit or View them.

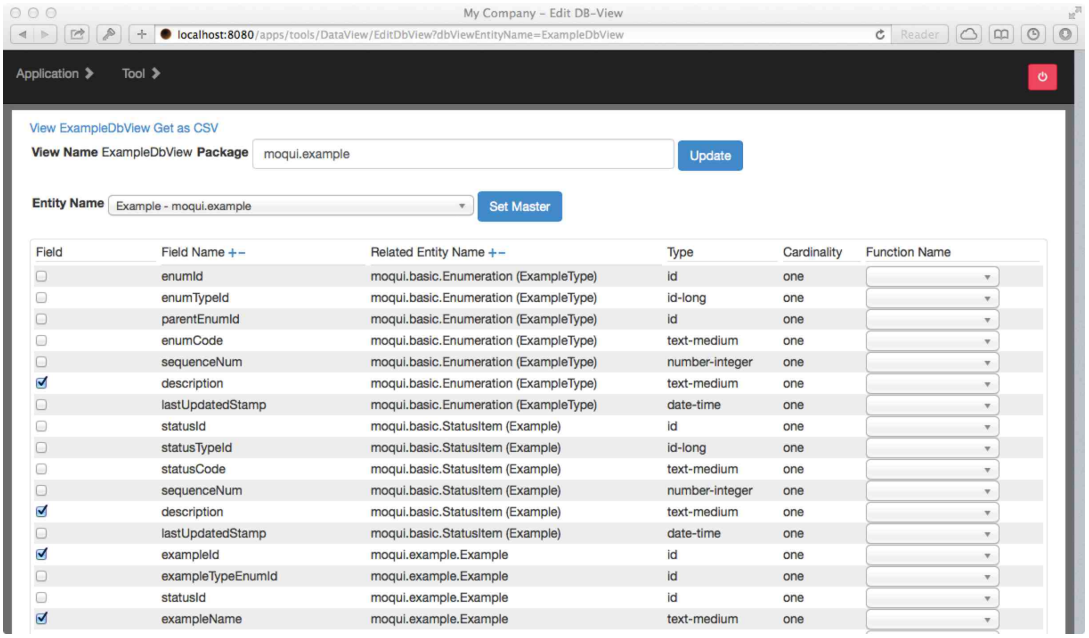


Edit DB View

The screen to edit a DB view entity has a form at the top to change the package the entity is in. Note that view entities defined in `DbViewEntity` can be used in the Entity Facade just like any other entity or view entity.

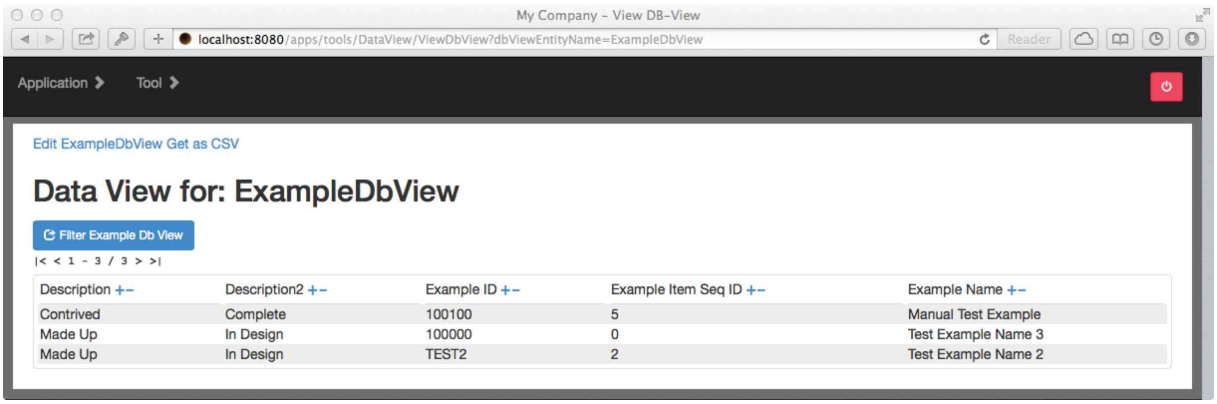
Next on the screen is a form to set the master entity, or the main entity in the view that all other entities will be related to. Once this is set the list form below shows all of the fields on that entity and directly related entities. In this screenshot below the master entity is the `Example` entity and the fields shown are for it and the `ExampleType` `Enumeration`, and `Example` `StatusItem`. The screen is cut off partway down and if you view the full screen you'll also see fields further down for the `ExampleContent`, `ExampleFeatureAppl`, and `ExampleItem` entities (which all have a cardinality of many).

The fields selected to include in the view are the `Enumeration.description` and `StatusItem.description` fields, the **`exampleId`** and **`exampleName`** from the `Example` entity (the master entity), and further off screen the `ExampleItem.exampleItemSeqId` field is selected with a count function to get a count of items on the example.



View DB View

This screen displays the results of querying the defined DB view entity, paginated if needed, and with a Filter button that pops up a form with filter options for the fields on the view entity (using the default auto fields in a `form-single`). There is a link to go back to the Edit DB View screen, and a link to get the results in a CSV file.



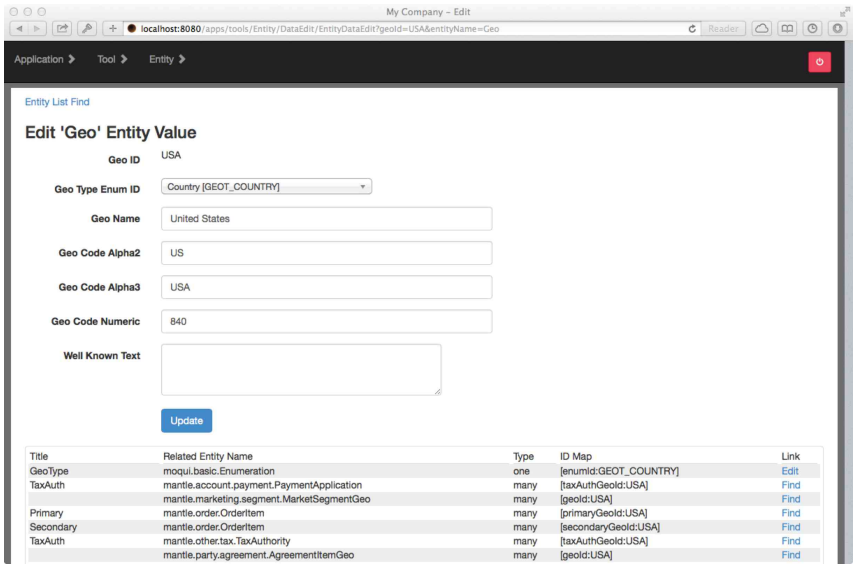
Here is a sample of the CSV export from the same ExampleDbView results as the screenshot:

```
Description,Description2,Example ID,Example Item Seq ID,Example Name
Contrived,Complete,100100,5,Manual Test Example
Made Up,In Design,100000,0,Test Example Name 3
Made Up,In Design,TEST2,2,Test Example Name 2
```

Entity Tools

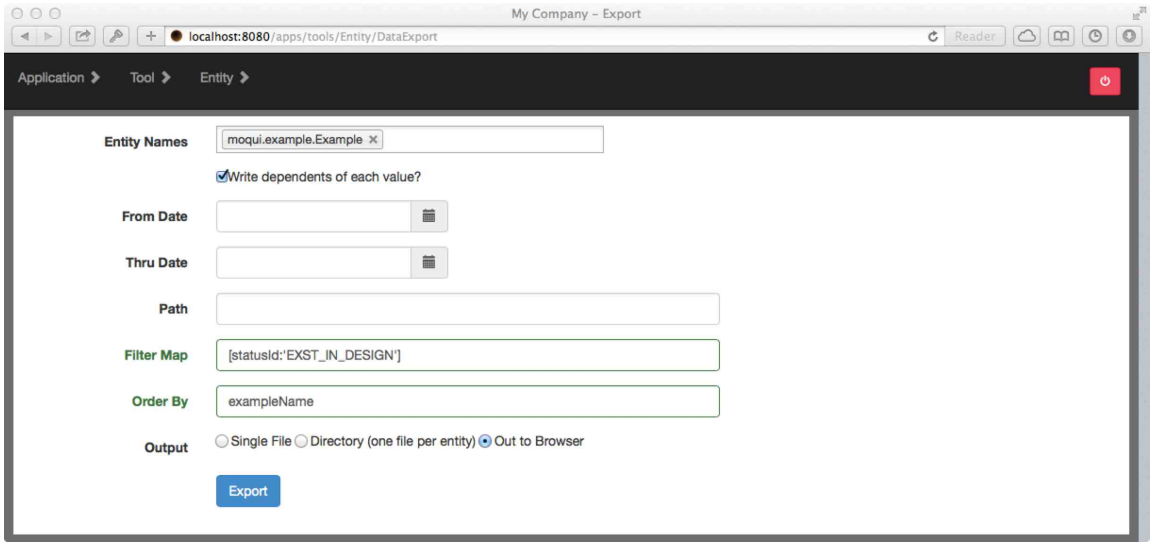
Data Edit

The data edit screens are somewhat similar to the Auto Screens, but without the tab sets and instead on the entity edit screen a list of related entities with a link to find records related to the current record, as you can see here. These screens still have their uses but are mostly superseded by the Auto Screens.



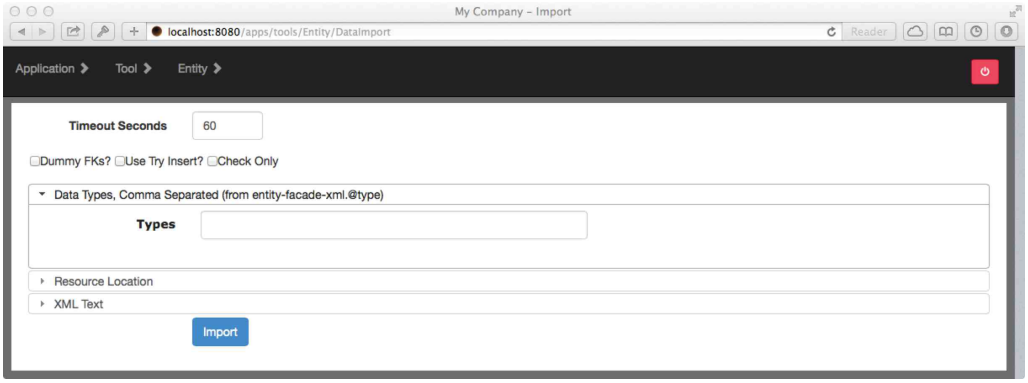
Data Export

This screen is used to export entity data in one or more entity XML files, or out to the browser. Select one or more entity names, from/thru dates to filter by the **`lastUpdatedStamp`**, the output path or filename (leave empty for Out to Browser), an optional `Map` in Groovy syntax to filter by (filter fields only applied to entities with matching field names, otherwise ignored), and optional comma-separated order by field names (also only applies to entities with matching field names).



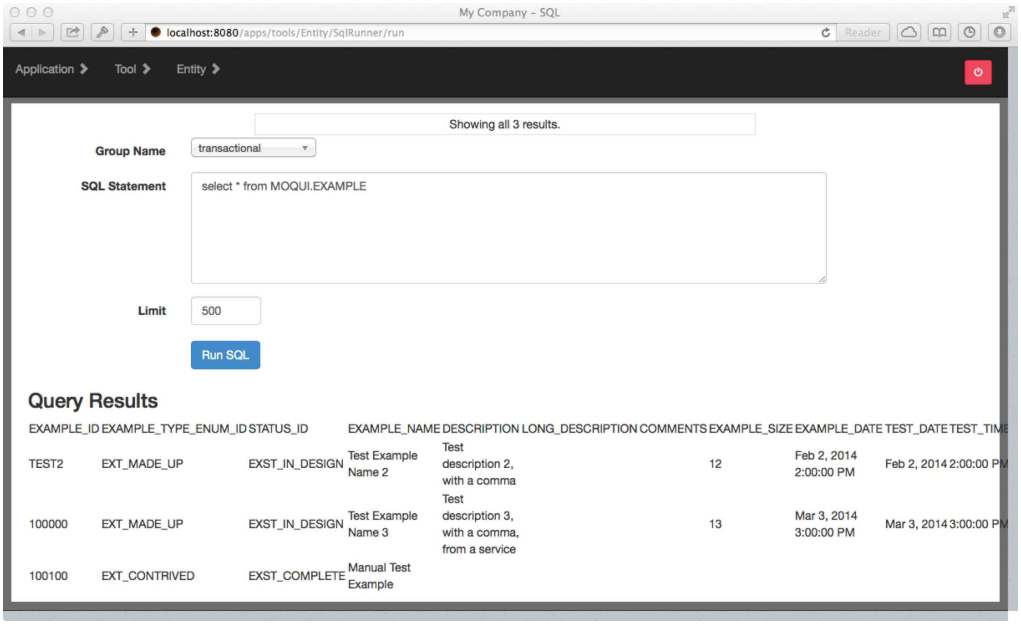
Data Import

Use this screen to import data from entity XML or CSV text. There are 3 options for the text itself: comma-separated data types (matching the entity-facade-xml.type attribute), a resource location that can be a local filename or any location supported by the Resource Facade, or text pasted right into the browser in a textarea. Dummy FKs checks each record's foreign keys and if a record doesn't exist adds one with only PK fields populated. Use Try Insert is meant for data that is expected to not exist and instead of querying each record to see if it does it just tries an insert and if that fails does an update (slower for lots of updates). Check Only doesn't actually load the data and instead checks each record and reports the differences.



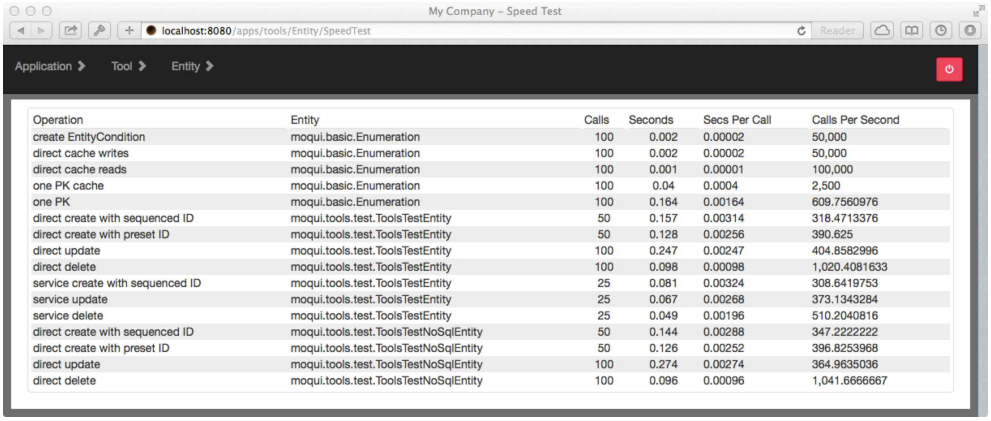
SQL Runner

Use this screen to run arbitrary SQL statements against the database for a given entity group and view the results.



Speed Test

This screen runs a series of cache and entity operations to report timing results. It is most useful to see comparative performance between different databases and server configurations. The screen accepts a baseCalls parameter which defaults to 100 (as seen below). Note that this screen shot uses the default configuration with the "nosql" entity group in the Derby database along with all the others. When using OrientDB or some other NoSQL datasource you'll see fairly different results.

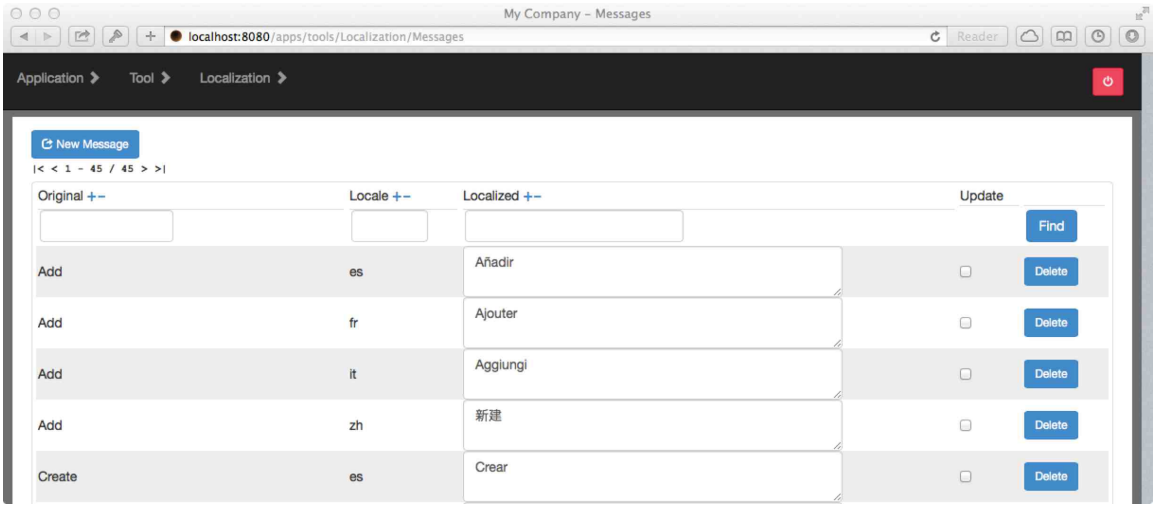


Operation	Entity	Calls	Seconds	Secs Per Call	Calls Per Second
create EntityCondition	moqui.basic.Enumeration	100	0.002	0.00002	50,000
direct cache writes	moqui.basic.Enumeration	100	0.002	0.00002	50,000
direct cache reads	moqui.basic.Enumeration	100	0.001	0.00001	100,000
one PK cache	moqui.basic.Enumeration	100	0.04	0.0004	2,500
one PK	moqui.basic.Enumeration	100	0.164	0.00164	609.7560976
direct create with sequenced ID	moqui.tools.test.ToolsTestEntity	50	0.157	0.00314	318.4713376
direct create with preset ID	moqui.tools.test.ToolsTestEntity	50	0.128	0.00256	390.625
direct update	moqui.tools.test.ToolsTestEntity	100	0.247	0.00247	404.8582996
direct delete	moqui.tools.test.ToolsTestEntity	100	0.098	0.00098	1,020.4081633
service create with sequenced ID	moqui.tools.test.ToolsTestEntity	25	0.081	0.00324	308.6419753
service update	moqui.tools.test.ToolsTestEntity	25	0.067	0.00268	373.1343284
service delete	moqui.tools.test.ToolsTestEntity	25	0.049	0.00196	510.2040816
direct create with sequenced ID	moqui.tools.test.ToolsTestNoSqlEntity	50	0.144	0.00288	347.2222222
direct create with preset ID	moqui.tools.test.ToolsTestNoSqlEntity	50	0.126	0.00252	396.8253968
direct update	moqui.tools.test.ToolsTestNoSqlEntity	100	0.274	0.00274	364.9635036
direct delete	moqui.tools.test.ToolsTestNoSqlEntity	100	0.096	0.00096	1,041.6666667

Localization

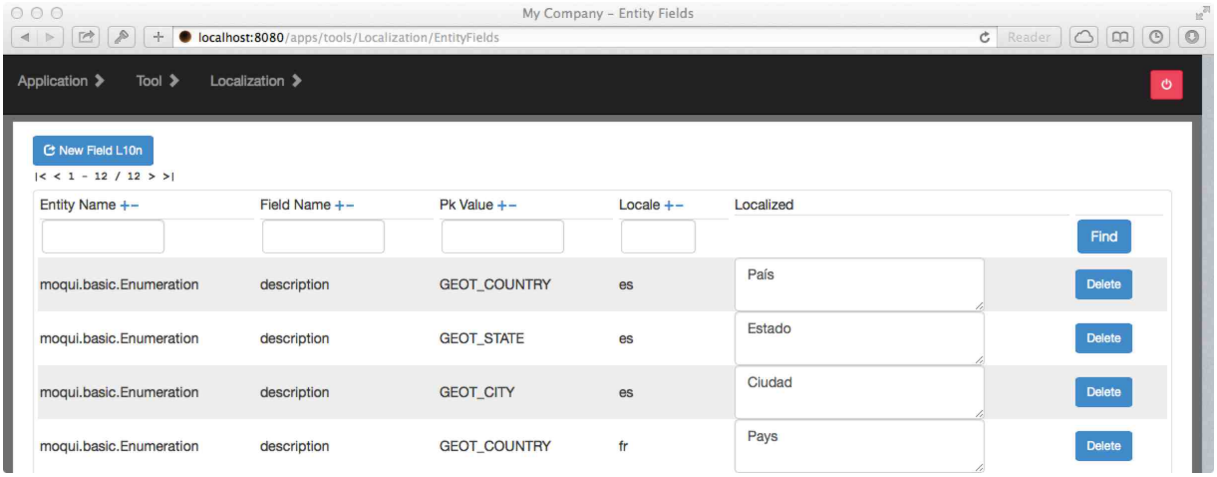
Messages

Moqui uses database records instead of property or XML files for localized messages and labels. Use this screen to administer localized messages that are used by the `L10nFacade.getLocalizedMessage()` method, which is in turn used by the Resource Facade before string expansion and in XML Screens and Forms for titles, etc.



Entity Fields

The `EntityValue.get()` method supports localized entity fields for any entity by simply setting the `field.enable-localization` attribute to `true` and adding records here (which are recorded with the `LocalizedEntityField` entity). Each record had the entity name, the field name to localize, the value of the single field primary key (only entities with single field PKs can use this), the locale for the value, and the localized value.

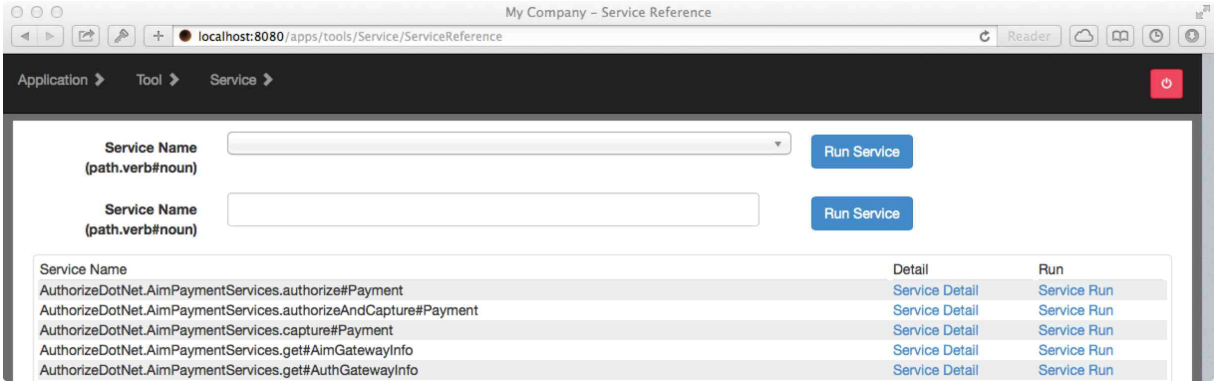


Service

Service Reference

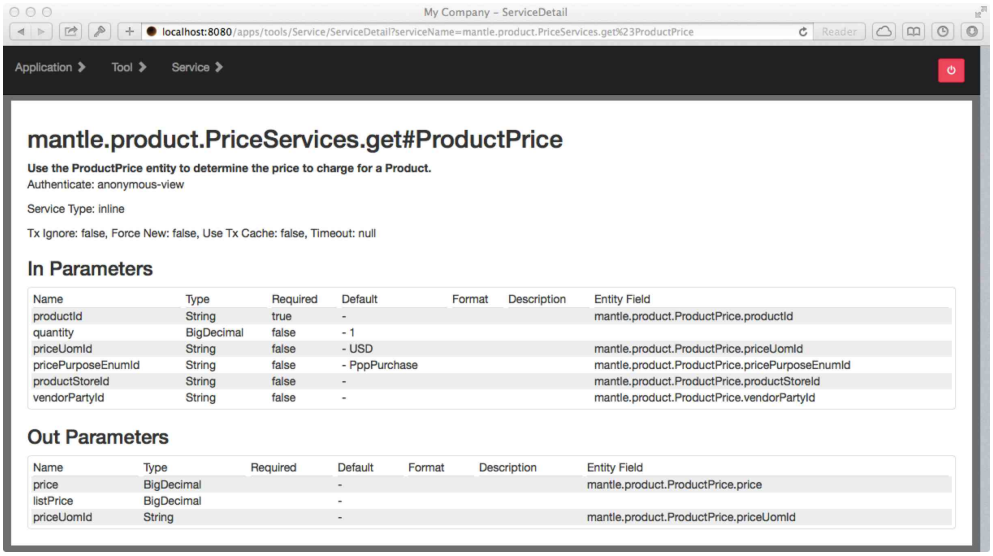
Service List

With the service reference you can see a list of existing services, details of each, and go to a screen to run them as well.



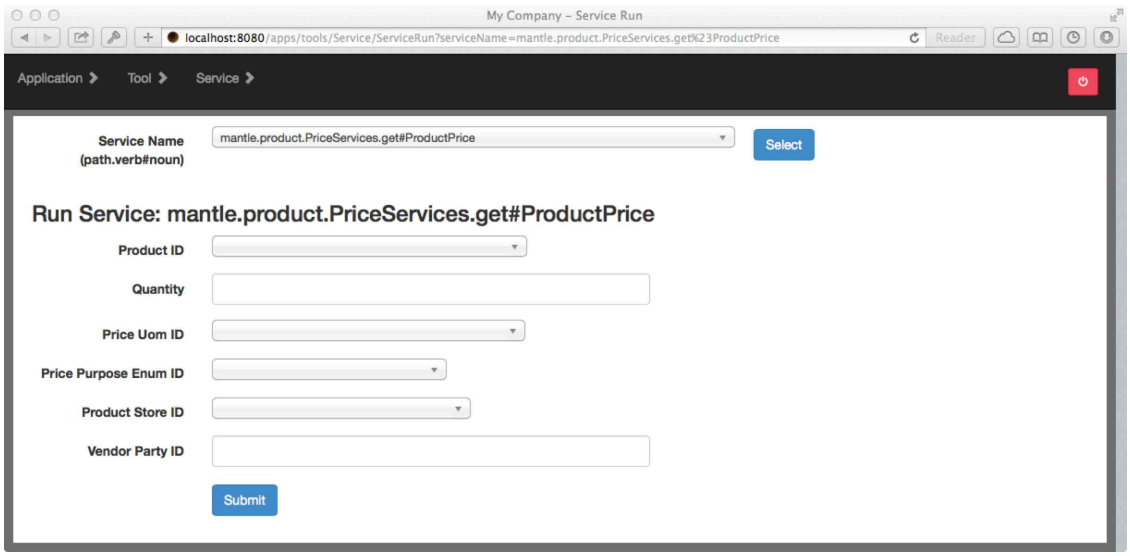
Service Detail

The detail screen for a service shows the service description and general information about the service, plus the in and out parameters with details for each. This is useful for a general reference and to see how a service expands at runtime when it implements interfaces, etc.



Service Run

The service run screen shows a XML single form with fields auto generated based on the service definition, which works best when the service in parameters are associated with entity fields (to get drop-downs for related entity values and such). Simply enter/select values and submit to run the service and see the results.

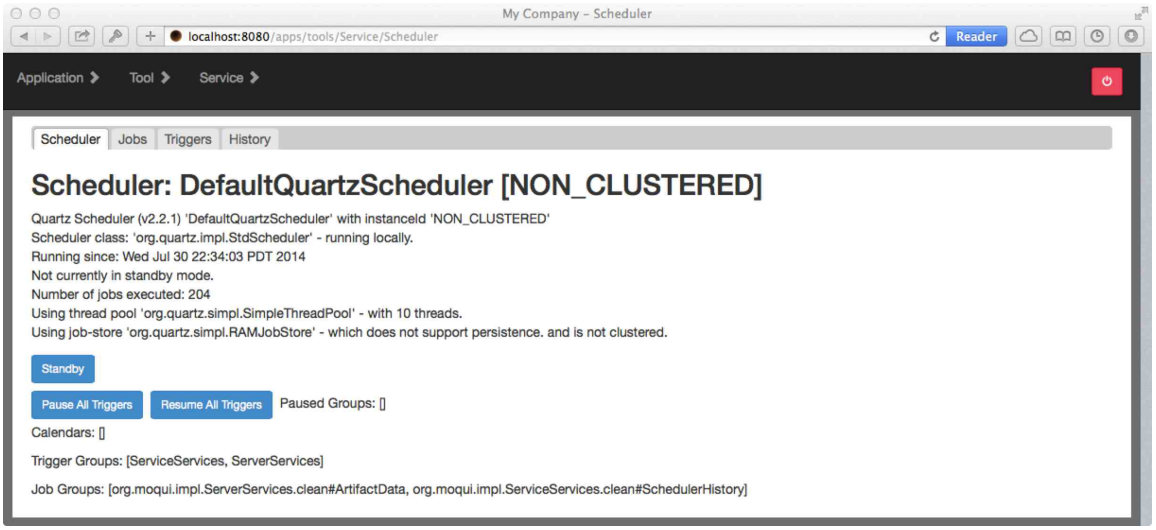


Scheduler

Moqui Framework uses Quartz Scheduler to run scheduled and asynchronous services and jobs. These screens are used to see information about the scheduler and scheduled jobs and perform administration such as pausing and resuming jobs and triggers.

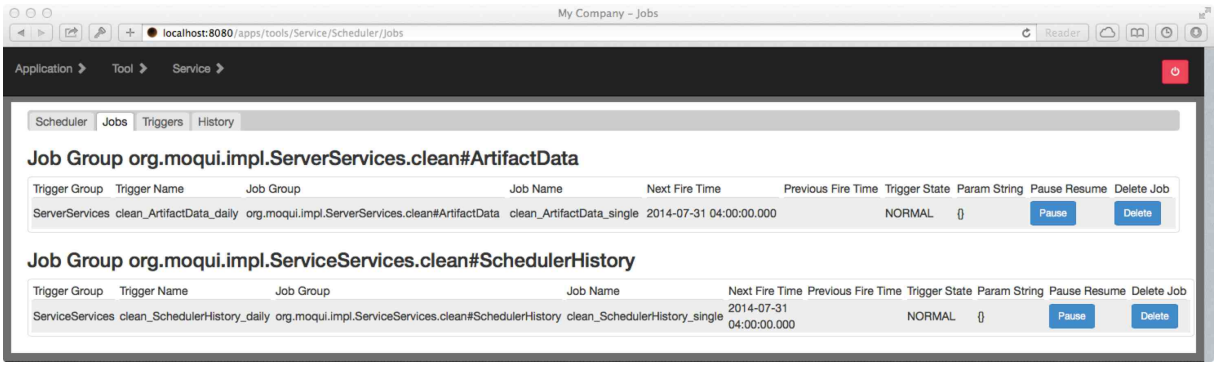
Scheduler Status

This screen shows the status of Quartz Scheduler and has buttons to put the entire scheduler on standby, and to pause and resume all triggers.



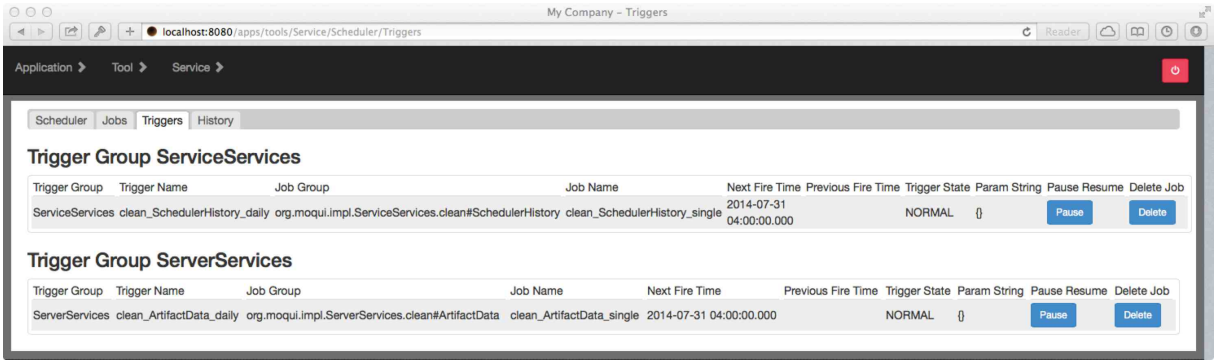
Jobs

The jobs tab shows currently active jobs, organized by job group which for Moqui service jobs is the name of the service. In addition to details about the job is has buttons to Pause the job, or when paused to Resume the job, and to Delete the job. When pausing a job it pauses all triggers associated with the job.



Triggers

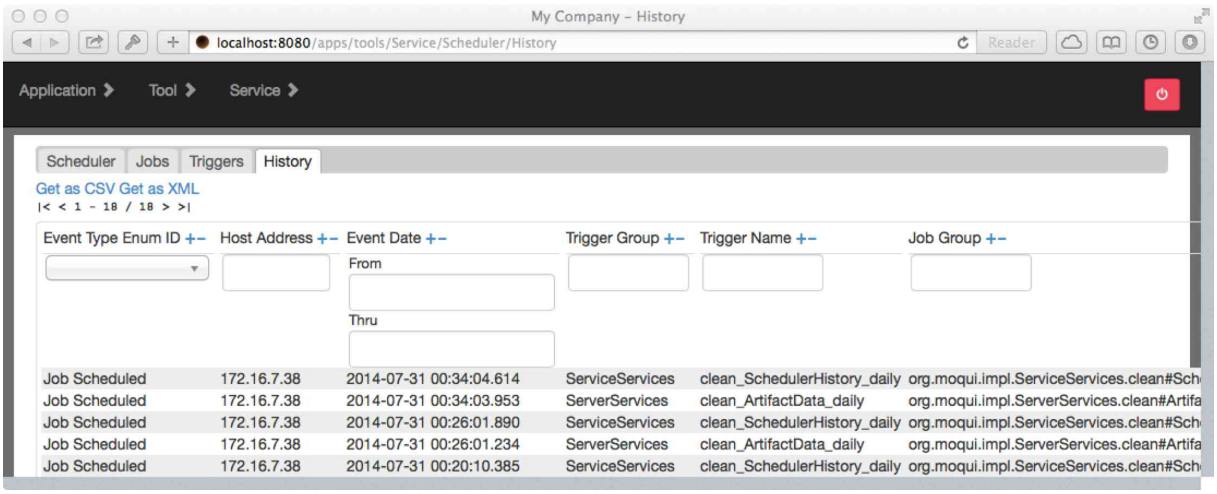
Much like the Jobs tab this tab shows the triggers associated with jobs and has the same options to pause/resume and delete. A job may have more than one trigger and from this screen you can pause/resume certain triggers for a job while leaving the others as-is.



History

The history tab for the scheduler shows a history of jobs run including scheduled services and any other custom jobs you might have running. There are links to get the data as a CSV or XML file. The header of the list form has options to filter the results which are also paginated as there may be a large number of jobs.

This data comes from the `SchedulerHistory` entity, which is managed by the `ServiceFacadeImpl.HistorySchedulerListener` class which implements the Quartz `SchedulerListener` interface.

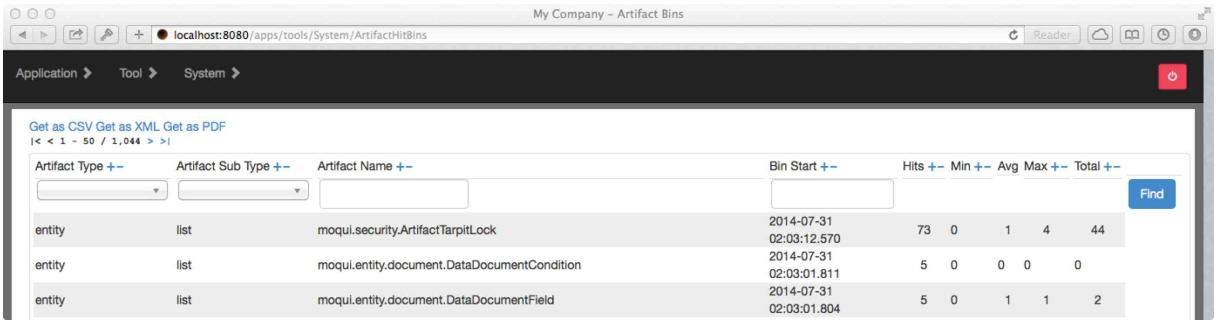


System Info

Artifact Statistics

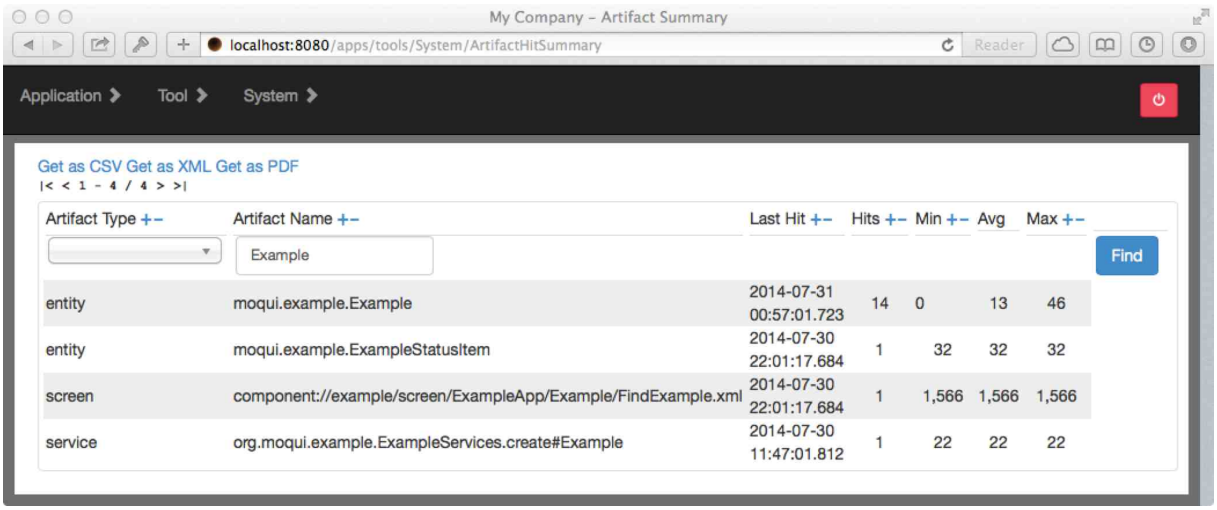
Hit Bins

This screen shows records from the `ArtifactHitBin` entity and has options for filtering, sorting, and exporting to CSV, XML, and PDF. Use this screen to see artifact hit data about specific artifacts in a specific date/time range.



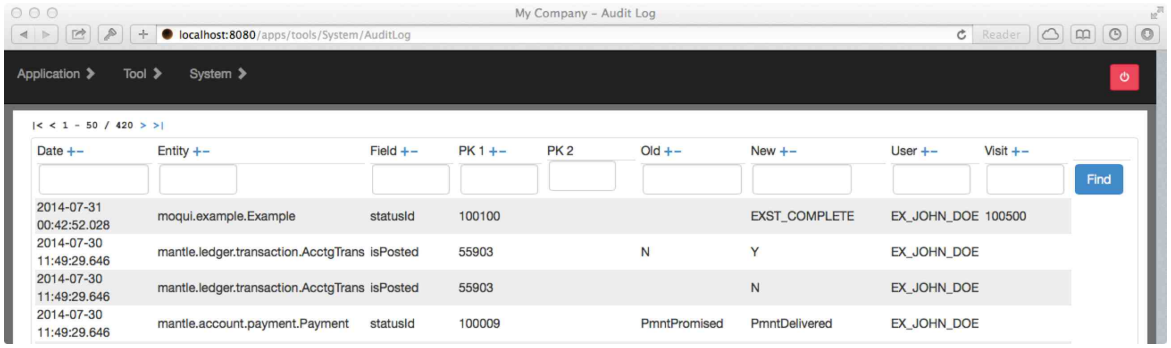
Artifact Summary

The artifact summary screen shows general performance data for each artifact over all time based on `ArtifactHitBin` records using the `ArtifactHitReport` view entity. Just like the hit bins screen this has filter, sort, and export options. The screen shot below shows just the artifacts with "Example" in their name using the header form to filter results.



Audit Log

When the `field.enable-audit-log` attribute is set to `true` the Entity Facade tracks the changes in `EntityAuditLog` records. Use this screen to view those records.

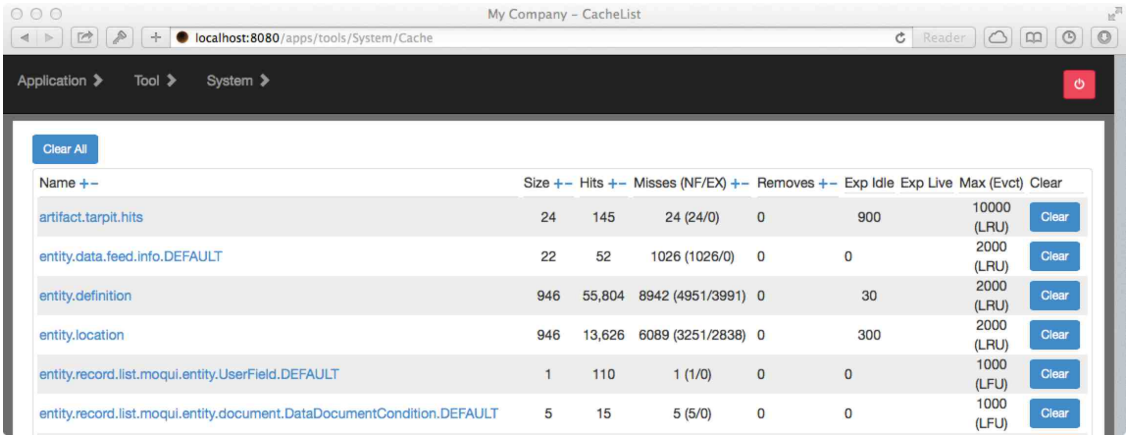


Cache Statistics

Cache List

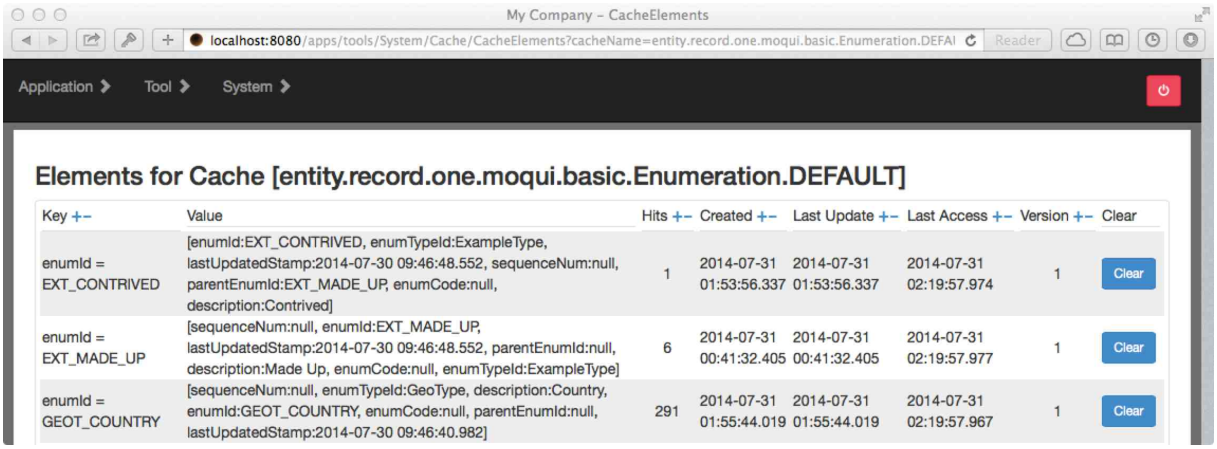
The Moqui Cache Facade is used for caching across the system including resource, entity, and various other caches. Use this list to see a summary of details about each cache. Size is the number of elements in the cache. Hits are the successful cache hits. Misses include general cache misses (unsuccessful gets from the cache) and specifically not found (NF) and expired (EX) miss counts. Removes shows the count of explicit removes from the cache.

There are two expire time that can be configured: idle for expiration after being idle for a certain time and live for the time since the cache element was created. The Max (Evct) column shows the maximum elements for each cache (default is 10,000) and the eviction algorithm to use once the limit is reached. The Clear button for each cache clears just that cache, and the Clear All button at the top clears all caches. Click on the Name to see the elements in the cache.



Cache Elements

When you click on the name of a cache you'll see this screen. It shows the cache entries up to a limit of 500 (use the **`displayLimit`** parameter for a different limit). It has details for each cache element plus a button to Clear (remove) just that element from the cache. This screen shot is for an entity one cache (for the `Enumeration` entity). The text shown for key and value are from calling **`toString()`** on the objects. In this case the key is an `EntityCondition` and the value is an `EntityValue` and they both evaluate to nice text, but not all objects will.

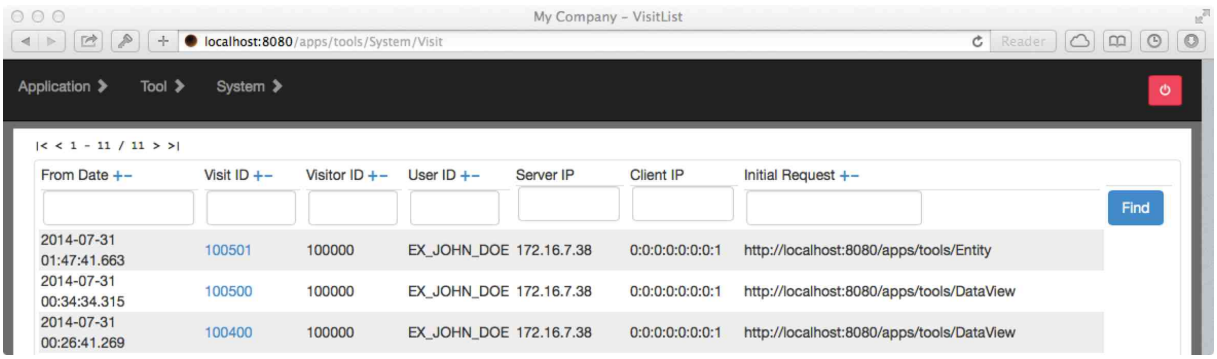


Server Visits

Moqui creates a `Visit` record for each web session to track server access and tie together artifact hits (page requests as screens, content, transitions, services, etc) within a session.

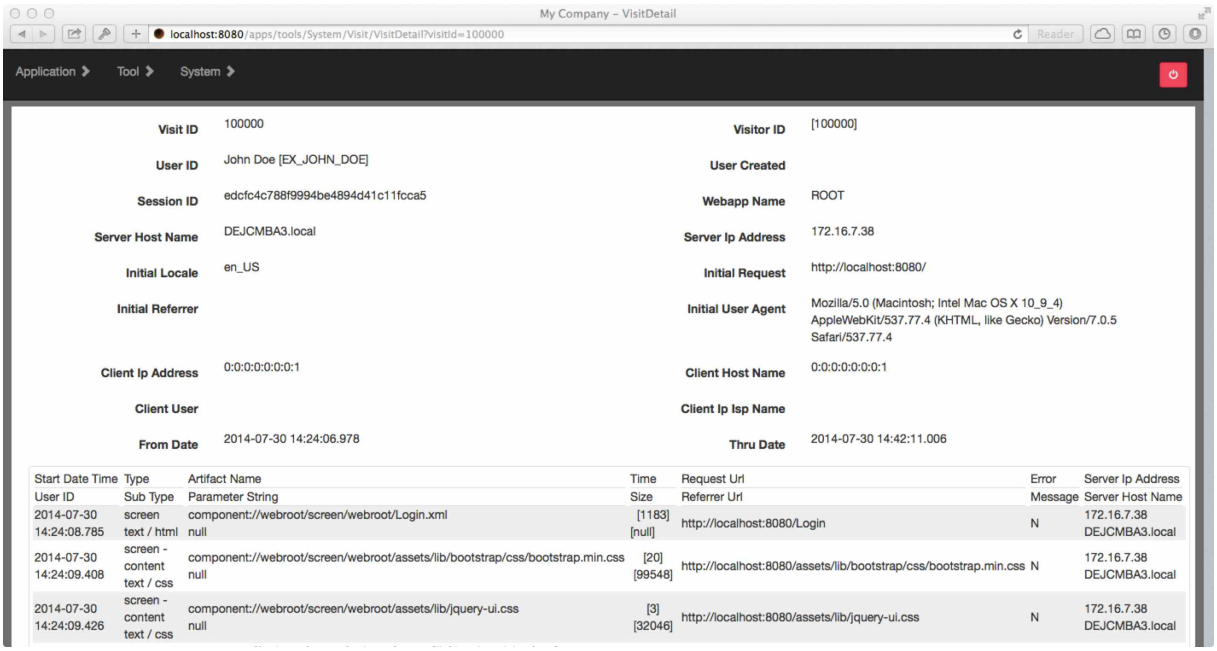
Visit List

This screen shows a list of visits with pagination and options to filter and sort the records because over time there will be a large number of visits. Click on the Visit ID to view details about the visit.



Visit Detail

This screen shows details about the visit (session). The header has fields generally available in a HTTP request plus additional information like the User ID logged in during the visit (if a user logs in). It also shows the artifact hits related to the visit (i.e., page requests and such within a session). This can be used to see a history of activity for specific users for security and service purposes, and the underlying data in `Visit` and `ArtifactHit` can be used for more general analysis for those purposes and marketing too.



12. Mantle Business Artifacts

Mantle Business Artifacts is an open source project separate from and built on Moqui Framework. Moqui Framework is a set of tools to build applications. Mantle Business Artifact is a library of lower-level artifacts that act as a foundation for business applications. The main benefits of using Mantle are cost savings, design and implementation risk reduction, adoption of common and standardized business structures and processes, and consistency with other applications built on Moqui and Mantle.

Mantle has three main parts: Universal Data Model (UDM), Universal Service Library (USL), and Universal Business Process Library (UBPL). This chapter will focus on the data model (UDM) and service library (USL).

UBPL is a set of business process stories and other generic business requirement documents that drive the design of business applications. They are a good source for understanding the business concepts, actors, and processes that the data model and service library are based on. They are also generic enough to be used as a starting point for real-world business and modified as needed.

Mantle is a foundation for building enterprise automation applications such as:

- Enterprise Resource Planning (ERP)
- Project ERP
- Professional Services Automation (PSA)
- Customer Relationship Management (CRM)
- Supply Chain Management (SCM)
- Manufacturing Resource Planning (MRP)
- Enterprise Asset Management (EAM)
- Point-of-Sale (POS)
- eCommerce

Together Moqui Framework and Mantle Business Artifacts form a foundation for an ecosystem of applications that are implicitly integrated. Applications can extend the Mantle data model and will always have their own services, but using the data model and services as intended will make applications work readily with data and services from other applications built on the same.

When such applications are deployed together the data is automatically shared. For example you will have a single structure for customer data that is used across all ecommerce, customer service, fulfillment, project management, and accounting applications and any other types of application that needs it.

NOTE: This chapter uses a large number of business terms. If you run across terms you are not familiar with you may look them up as you go (the internet is a wonderful thing, as is the full text search of the digital version of this book) or just take note of them, move on, and don't worry too much about each one. The **Mantle Structure and UDM** section goes through a lot of terms with only data structures as context. When you get to the **USL Business Processes** section you will see the terms used in context of a process along with examples and they may make more sense, especially if you have spent some time reading about the data structures.

Mantle Structure and UDM

The Mantle data model (UDM) is based on concepts found in The Data Model Resource Book, Revised Edition, Volume 1 and Volume 2 by Len Silverston. In addition to the material in this section these books are a good reference for the data model concepts that make up the foundation for Mantle UDM. UDM is a loose implementation of the data model concepts in these books. UDM has a number of entities that go beyond what is in these books, and consolidates some of them too (like quote and order).

Both the data model (UDM) and the service library (USL) follow the same pattern for organizing artifacts. The directory and file structure of each are based on this pattern.

The sections below are a summary of the structure and the entities in each part. These are in alphabetical order for easy reference and to show the structure. When initially learning about the data model I recommend reading the sections on the more fundamental entities first with an order somewhat like this:

- The **Data Model Patterns** section in the **Data and Resources** chapter
- Party (mantle.party)
- Contact Mechanism (mantle.party.contact)
- Facility (mantle.facility)
- Definition - Product (mantle.product)
- Asset - Asset (mantle.product.asset)
- Account - Invoice (mantle.account.invoice)
- Account - Payment (mantle.account.payment)
- Work Effort (mantle.work.effort)
- Order (mantle.order)
- Shipment (mantle.shipment)

The data model diagrams have only selected entities to illustrate important structures, and only selected fields on those entities. They are not a complete reference of all entities and

fields. In the diagrams the master entities have a blue border, the detail entities a purple border, and the join entities a green border.

Accounting

Account - Billing (mantle.account.billing)

A `BillingAccount` is used to group `Invoice` and `Payment` records for the purposes of tracking how much a customer (**`billToPartyId`**) owes to a vendor (**`billFromPartyId`**). The balance owed on the account is the unpaid invoice total minus the associated payment total. The payment total may be larger than the invoice total, in which case there is a positive balance in the account owed to the customer (**`billToPartyId`**). The `BillingAccount` may have a credit limit in the accountLimit field and its associated currency in **`accountLimitUomId`**.

A `BillingAccount` itself is fairly simple as the "transaction" details in the account are in `Invoice` and `Payment` records. It can have other parties associated with it using `BillingAccountParty`. For terms on the account use `BillingAccountTerm`.

Account - Financial (mantle.account.financial)

A `FinancialAccount` is a singe-entry balance account like a bank account. There are various types of financial account defined with the `FinancialAccountType` entity with settings like **`isRefundable`**, **`requirePinCode`**, automatic replenishment settings, and others. OOTB types include Gift Certificate, Store Credit Account, Service Credit Account, Loan Account, and Bank Account.

A `FinancialAccount` is owned by a `Party` (**`ownerPartyId`**) and an internal organization (**`organizationPartyId`**) is liable for the balance on the account. Other parties may be associated with it using `FinancialAccountParty`. It has a name (**`finAccountName`**), code (**`finAccountCode`**), and may have a PIN number (**`finAccountPin`**). It may be valid only within a date range (**`fromDate`**, **`thruDate`**). It has a status (**`statusId`**) that may be Active, Negative Pending Replenishment, Manually Frozen, or Cancelled.

The **`actualBalance`** of a `FinancialAccount` is the sum of the transactions (`FinancialAccountTrans`) associated with the account. The **`availableBalance`** of an account is the **`actualBalance`** **minus** the total of authorizations (`FinancialAccountAuth`) on the account.

A transaction (`FinancialAccountTrans`) for a given **`amount`** may be a for Deposit, Withdraw, or Adjustment (**`finAccountTransTypeEnumId`**). Transactions requiring approval or for other reasons may have a **`statusId`** of Created, Approved, or Cancelled. They will generally have a reason (**`reasonEnumId`**) such as Purchase, Initial Deposit, Replenishment, or Refund.

A transaction happens at a certain date/time (**transactionDate**) and may be entered at a different time (**entryDate**). It is generally performed or initiated by a party (**performedByPartyId**) and may have **comments** about it. A transaction will also often have a Payment (**paymentId**) and/or OrderItem (**orderId**, **orderItemSeqId**) associated with it.

An authorization (FinancialAccountAuth) is used to reserve an **amount** in advance of a Withdraw transaction. The auth is done on **authorizationDate** and expires on **expireDate**.

Account - Invoice (mantle.account.invoice)

An Invoice or bill is used to request Payment with details about why and is sent from the Party that is owed (**fromPartyId**) to the Party that owes (**toPartyId**). There are a few types of invoices (**invoiceTypeEnumId**) including Sales, Return, Payroll, Commission, and Template. The direction of the invoice is determined by the from and to parties so there is no separate type for purchase versus sales, they are both Sales type invoices with parties going one way or the other.

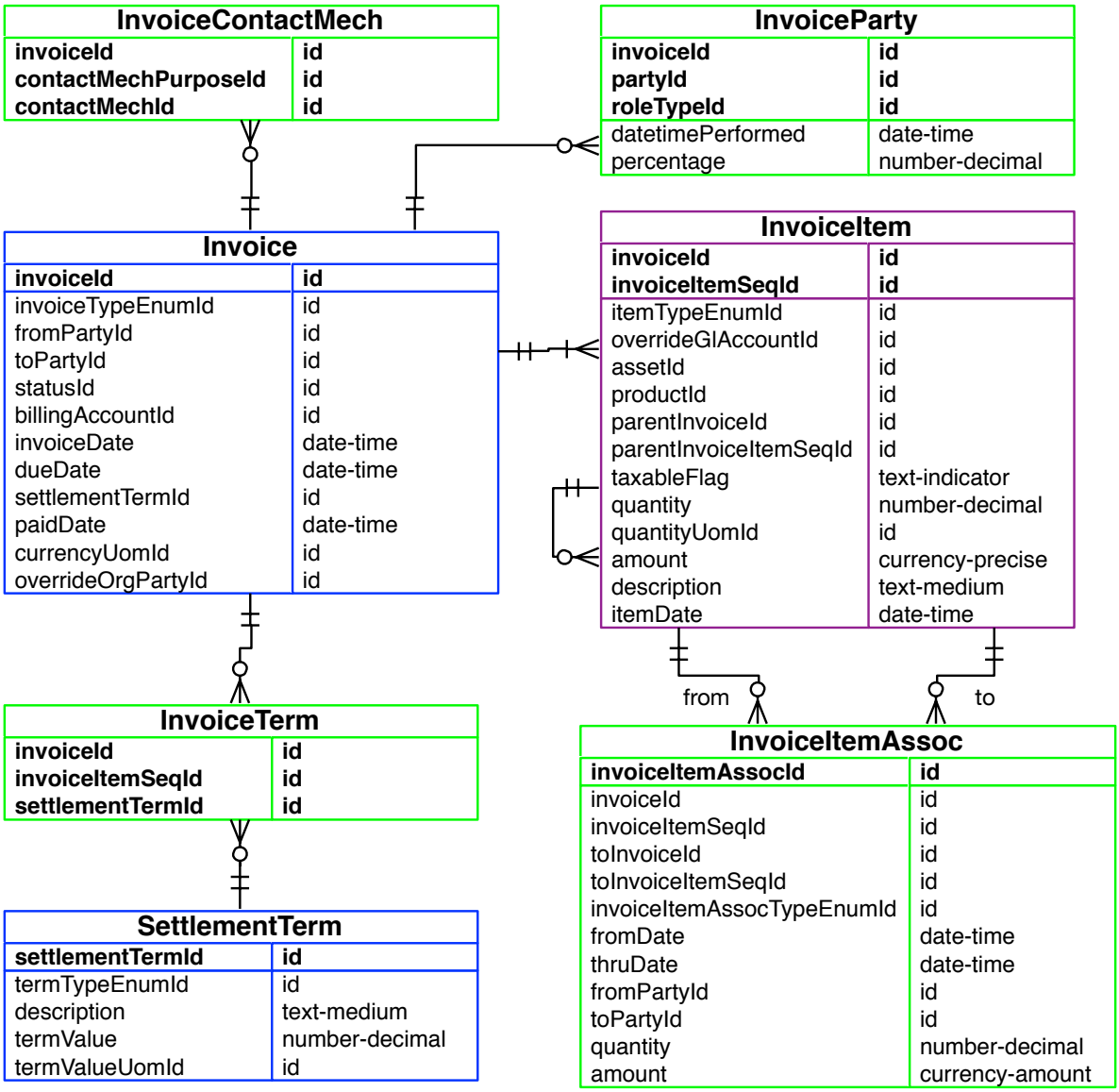
Depending on the direction and which Party is the internal organization there is a different set of statuses (**statusId**). For incoming invoices the statuses are Incoming, Received, Approved, Payment Sent, Billed Through, and Cancelled. For outgoing invoices the statuses are In-Process, Finalized, Sent, Payment Received, Write Off, and Cancelled.

The invoice may be associated with a BillingAccount (**billingAccountId**), see the **Account - Billing (mantle.account.billing)** section for details. Amounts on an invoice are for a single currency specified with the **currencyUomId** field. Each invoice is initiated on a certain date (**invoiceDate**), has a due date (**dueDate**) and for historical reference date when it was paid (**paidDate**). The due date is generally determined by a SettlementTerm record specified with the **settlementTermId** field. Other terms may be associated with the invoice or with invoice items using InvoiceTerm.

Contact details for an invoice are associated with it using InvoiceContactMech. In addition to the from and to parties other parties such as sales reps or accountants may be associated with an invoice using InvoiceParty.

The details of goods, services, shipping, tax, discounts, and so on for an invoice are recorded with InvoiceItem records. Invoice items use the same set of types as other items including mantle.order.OrderItem and mantle.order.return.ReturnItem. These shared item types are defined in the ItemTypeData.xml file. There are a wide variety of types for things like sales, purchase, expenses, commissions, and payroll. For sales orders the most common types are product, time entry, shipping charges, sales taxes, and discounts.

Just like order items, invoice items may have a hierarchical structure using the **parentInvoiceId** and **parentInvoiceItemSeqId** fields. This is used for things like tax items that are for a particular good or service item.



Each item has a **`description`** and will generally have a **`productId`** and possibly an **`assetId`** for more detail about goods and services. Each item has a **`quantity`** and unit for the quantity (**`quantityUomId`**) and an **`amount`** per quantity. The sub-total for an invoice item is: **`quantity * amount`**.

Invoice items may be associated with other items using `InvoiceItemAssoc`. One example of when this is useful is when receiving an invoice with expense items from a service provider and billing those items through to a client.

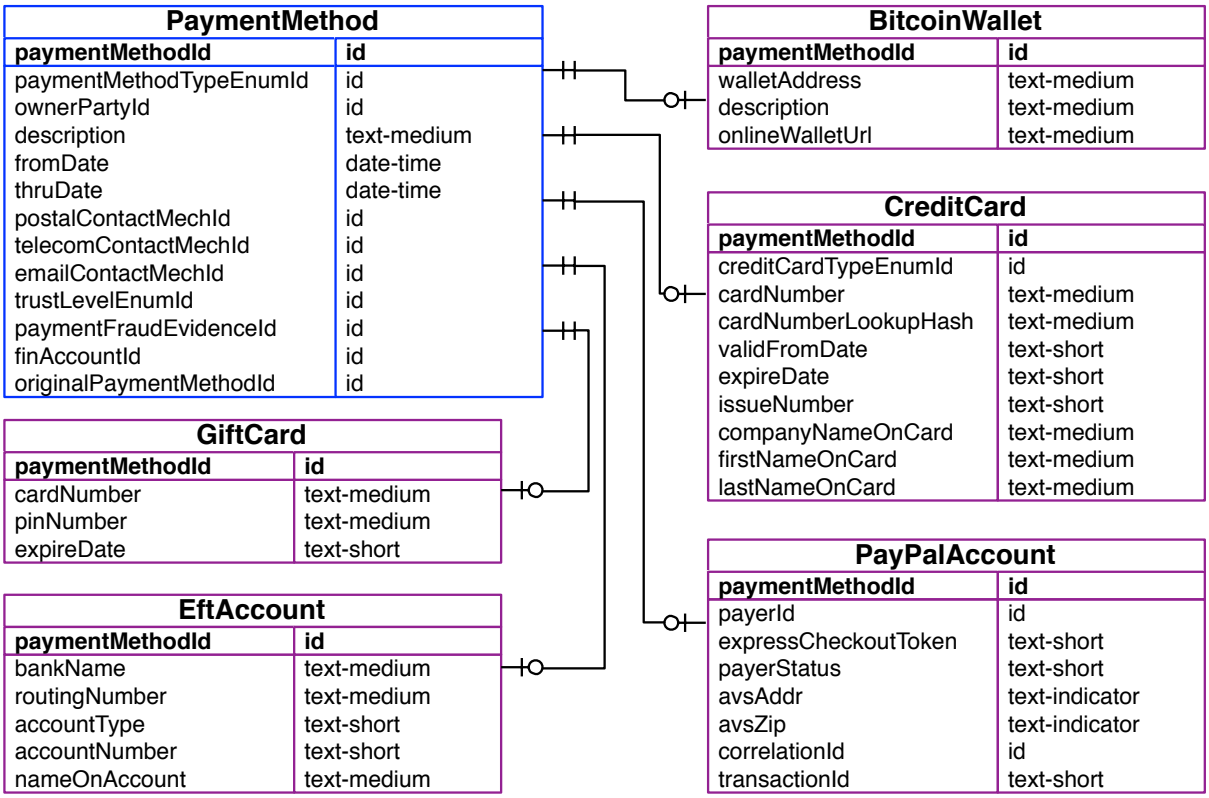
An `Invoice` is a record with financial impact and triggers GL posting when the status changes to Finalized for outgoing invoices and Approved for incoming ones. Note that if both from and to parties on an invoice are internal organizations with accounting settings the

invoice will be posted for both. If the **overrideOrgPartyId** field is populated that Organization will be used instead of the **fromPartyId** or **toPartyId** when posting depending on which is an internal org (this is not generally used if both are internal orgs).

The accounting transaction (AcctgTrans) generated for automated posting of an invoice will have one entry for each invoice item posted to the GL account (GlAccount) configured for the item type, and a balancing transaction entry with the total of the invoice posted to an accounts payable account for incoming invoices and an accounts receivable account for outgoing invoices.

Account - Method (mantle.account.method)

A PaymentMethod is an instrument used for payment and each type has a separate entity with details including BitcoinWallet, CreditCard, EftAccount, GiftCard, and PayPalAccount. A PaymentMethod may be for a FinancialAccount and that is specified with the **finAccountId** field. Some payment method types such as cash, checks, and money orders are used directly on payments, orders, and so on with no PaymentMethod record because the Payment is not processed through a payment method.



A payment method is owned by a Party (**ownerPartyId**), has a **description**, and generally has a **postalContactMechId**, **telecomContactMechId**, and possibly a **emailContactMechId**.

A PaymentMethod is valid in a date range (**fromDate**, **thruDate**). Generally the **thruDate** field is null until the payment method is no longer used, or has been changes. PaymentMethod and related records are considered immutable, so when changed the original record has the **thruDate** set and a new record is created with the modified details. The new record points to the original with the **originalPaymentMethodId** field.

Where fraud is a concern the PaymentMethod should have a **trustLevelEnumId** set. OOTB options include New Data, Valid/Clean (through 3rd party service), Verified (with outbound contact or authorization), Greylisted, and Blacklisted. If the trust level is Greylisted or Blacklisted there should be a **paymentFraudEvidenceId** pointing to a PaymentFraudEvidence record with details about why.

For GiftCard payment methods they are usually purchased from or issue by the organization and details about that are tracked with the GiftCardFulfillment entity.

Certain types of payment method, especially credit cards, commonly have automated payment processing through a payment gateway such as Authorize.net and Cybersource. The integration with the payment processor consists of services for authorize, capture, release, and refund. These services are configured with the PaymentGatewayConfig which is typically associated with a ProductStore using the ProductStorePaymentGateway entity.

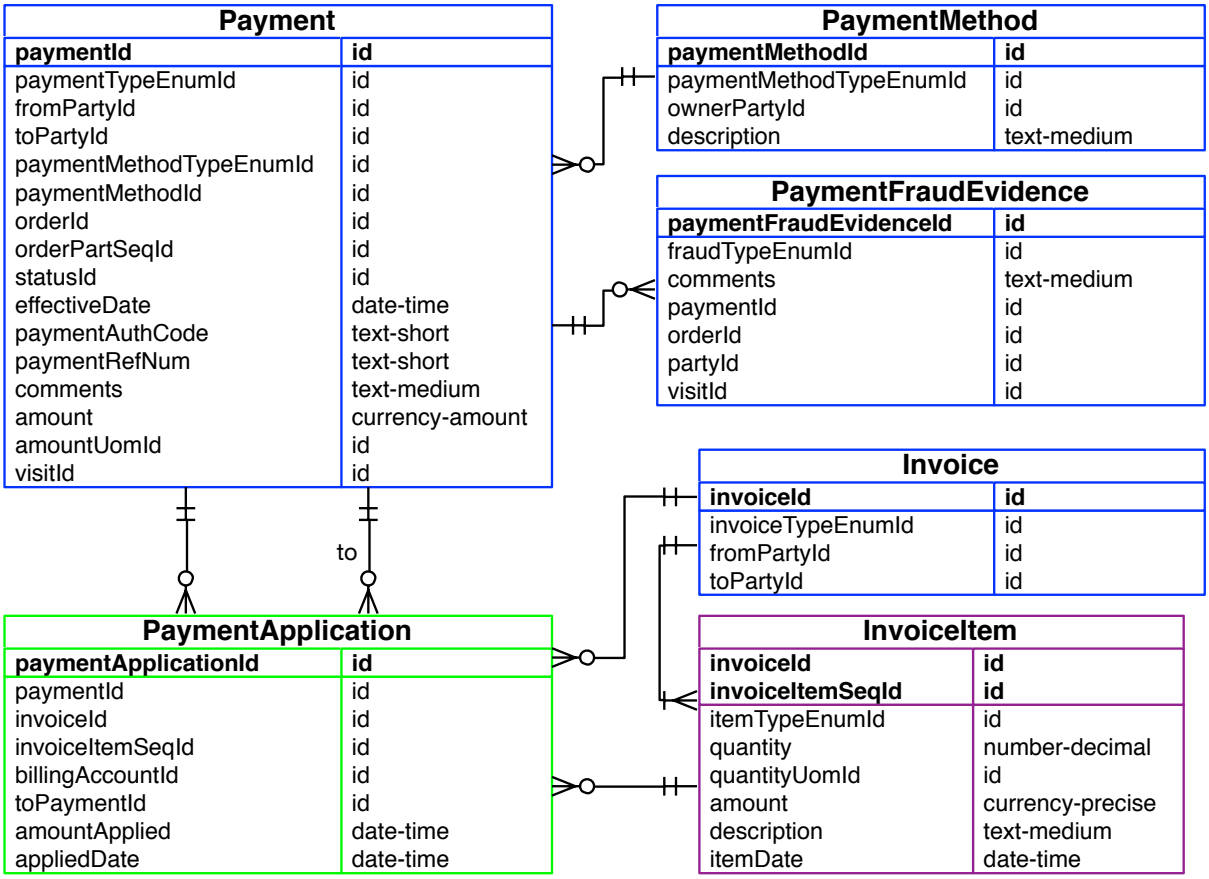
Any time a payment gateway is used the details of the response should be stored with the PaymentGatewayResponse entity. There are generally associated with a Payment (**paymentId**) and have various fields for codes and results from the payment processor.

Account - Payment (mantle.account.payment)

A Payment is generally issued in response to an Invoice and like an invoice goes from one Party (**fromPartyId**) to another (**toPartyId**). The parties on a Payment will be reversed from the parties on an Invoice. Types of payments (**paymentTypeEnumId**) include Invoice Payment, Disbursement, and Refund. A payment always has an **amount** and the currency for it in **amountUomId**.

A Payment should always have a payment method type (**paymentMethodTypeEnumId**) such as cash, check, or credit card and if applicable for the payment method type should also have a payment method (**paymentMethodId**).

If the payment is processed automatically through a payment gateway the gateway used for auth should be recorded in **paymentGatewayConfigId** so that it can be used for subsequent operations like capture or void. For convenience (since these are also on the PaymentGatewayResponse) for automated payments there are **paymentAuthCode** and **paymentRefNum** fields for authorization results and the reference number to use for



subsequent operations. Other fields for details when processing credit card and similar payments through a gateway include **`presentFlag`**, **`swipedFlag`**, **`processAttempt`**, and **`needsNsfRetry`**.

A payment has various statuses (**`statusId`**) including Proposed, Promised, Authorized, Delivered, Confirmed Paid, Cancelled, Void, Declined and Refunded.

Payments to not have items like an invoice, but may have deductions for special cases and these are recorded using the `Deduction` entity.

A `Payment` record may be created very early in an ordering process to specify payment details for an entire order or for a particular order part. There may be multiple `Payment` records for a given `OrderHeader` or `OrderPart`, so they are referred to using the **`orderId`** and if applicable **`orderPartSeqId`** fields on the `Payment` record. Payment details are looked up for an order or part using these fields on the `Payment` entity.

Payments may be associated with a financial account (**`finAccountId`**), and more particularly an authorization and/or transaction on a financial account (**`finAccountAuthId`**, **`finAccountTransId`**).

For fraud sensitive organizations and applications when processing online transactions it is important to associated the `Payment` with a `Visit` using the **`visitId`** field. This tracks the client IP address and other HTTP client and session information. When a fraudulent transaction is identified the evidence should be recorded in a `PaymentFraudEvidence` and this is usually used to change the trust level on the associated payment method (`PaymentMethod`.**`trustLevelEnumId`**) and contact mechs (`ContactMech`.**`trustLevelEnumId`**).

For organizations that deal with multiple currencies the payment may be converted to an internal currency for the organization, or to match the currency on the associated invoice(s). In this case the original amount and currency should be recorded in the **`originalCurrencyAmount`** and **`originalCurrencyUomId`** fields for bank and other reconciliation.

A `Payment` is a record with financial impact and triggers GL posting when the status changes to Delivered. Note that if both from and to parties on a payment are internal organizations with accounting settings the payment will be posted for both. If the **`overrideOrgPartyId`** field is populated that Organization will be used instead of the **`fromPartyId`** or **`toPartyId`** when posting depending on which is an internal org (this is not generally used if both are internal orgs).

The accounting transaction (`AcctgTrans`) generated for automated posting of a payment will have one entry posted to the GL account (`GlAccount`) configured for the cash account for payment method type (unless overrideGlAccountId is populated, then that is used), and a balancing transaction entry posted to an accounts payable account for outgoing payments and an accounts receivable account for incoming payments.

To make things a little more complex payments are explicitly applied to an `Invoice` using the `PaymentApplication` entity so that a single payment can apply to multiple invoices, and an invoice can have multiple payments applied to it. A payment may also be applied to another `Payment` for situations where incoming and outgoing payments between parties cancel one another.

For GL posting purposes a `Payment` can be received without being applied to an invoice, or being partially applied and the unapplied amount will be posted to an unapplied payment account instead of a cash account. When the payment is applied another accounting transaction will be triggered with entries in the unapplied payments account and the cash account to balance things out.

When a `Payment` is part of a budgeted expenditure it can be associated with one or more `BudgetItem` records using `PaymentBudgetAllocation`.

Ledger - Account (mantle.ledger.account)

General ledger accounts (`GlAccount`) make up the chart of accounts for an internal `Organization`. Each account has a class (**`glAccountClassEnumId`**) to determine if the

account balance is add or subtracted to a transaction total and for reporting purposes (especially: Balance Sheet with Asset on one side and Contra Asset, Liability and Equity on the other; and Income Statement with Revenue, Contra Revenue, Cost of Sales, Income and Expense accounts). Here is the structure of the OOTB GL account classes (this can be changed with different `Enumeration` records of type `GlAccountClass`):

- Debit
  - Asset
    - Current Asset
      - Cash and Equivalent
      - Inventory Asset
      - Accounts Receivable
      - Prepaid Expense and Other
    - Long Term Asset
      - Land and Building
      - Equipment
    - Other Asset
  - Expense
    - Cash Expense
    - Interest Expense
    - Sales, General, and Administrative Expense
    - Non-Cash Expense
      - Depreciation
      - Amortization
  - Cost of Sales
    - Cost of Goods Sold
      - Inventory Adjustment
    - Cost of Services Sold
  - Contra Revenue
  - Equity Distribution
    - Return of Capital
    - Dividends
  - Non-Posting
- Credit
  - Income
    - Cash Income
    - Non-Cash Income
  - Revenue
    - Goods Revenue
    - Services Revenue
  - Equity
    - Owners Equity
    - Retained Earnings

    - Liability
        - Current Liability
            - Accounts Payable
            - Accrued Expenses
        - Long Term Liability
    - Contra Asset
        - Accumulated Depreciation
        - Accumulated Amortization
- Resource

`GlAccount` records also have a type (**`glAccountTypeEnumId`**) that is used for automated posting configuration. The available GL account types are in `Enumeration` records of type `GlAccountType`. There are quite a few defined OOTB such as AR, AP, Fixed Asset, Current Liability, Inventory, Finished Good Inventory, Tax, Profit Loss, Cost of Goods Sold, Expense, Customer Deposits, and Commission Expense (plus many others). There is some overlap in GL account classes and types, but they are separate fields because they are used for different things.

GL accounts are hierarchical with the **`parentGlAccountId`** field specifying the parent account. Each account has a code (**`accountCode`**) that is separate from the glAccountId so that it can be changed, a name (**`accountName`**) and a **`description`**. There is a **`postedBalance`** field that is maintained with each posting and derived from `AcctgTransEntry` records associated with the `GlAccount`.

For more general accounting use outside a typical general ledger `GlAccount` has a resource type (**`glResourceTypeEnumId`**) that is generally Money and can be other things such as Raw Material, Labor, and Finished Good. It also has a **`glXbrlClassEnumId`** field to specify the reporting (XBRL) class such as US GAAP and IAP.

To support multi-organization accounting there is a shared chart of accounts in `GlAccount` records and each internal `Organization` that needs it has a subset of the accounts assigned to it using the `GlAccountOrganization` entity. This has a **`postedBalance`** field that is updated with the balance of that account for just that `Organization`. Getting more specific there is a record in `GlAccountOrgTimePeriod` for each `GlAccount`, `Organization`, and `TimePeriod` (a fiscal month, quarter or year period). It has more detailed information about totals: **`postedDebits`**, **`postedCredits`**, **`beginningBalance`**, and **`endingBalance`**. These are all maintained by the GL posting service.

Other parties may be associated with a GL account using the `GlAccountParty` entity. A `GlAccount` may be associated with a budget through a budget item type using the `GlBudgetXref`.

In addition to the inherent hierarchy of GL accounts they may be organized with two other structures: categories and groups. `GlAccountCategory` is used for an arbitrary grouping of GL accounts and has a many-to-many relationship with them through the

GlAccountCategoryMember. This is used for special tracking and reporting purposes such as cost centers.

A GlAccountGroup is a more restricted grouping of GlAccount records for purposes of reporting and populating forms such as tax forms. For example a US IRS Form 1120 (U.S. Corporation Income Tax Return) would be a group type, and groups within the type would be "1a Gross receipts or sales", "1b Returns and allowances", and "4 Dividends". Each GL account can be associated with at most one group of each type (i.e. for each form, etc) through GlAccountGroupMember. This is intentional to avoid applying a GL account more than once and duplicating its value.

Ledger - Config (mantle.ledger.config)

The main entity of accounting preferences for an internal Organization is PartyAcctgPreference. It has fields for the tax filing form to use (**taxFormEnumId**), COGS method (**cogsMethodEnumId**), base currency for accounting (**baseCurrencyUomId**), fields to manage invoice ID sequencing (**invoiceSequenceEnumId**, **invoiceIdPrefix**, **invoiceLastNumber**, **invoiceLastRestartDate**, and **useInvoiceIdForReturns**), order ID sequencing (**orderSequenceEnumId**, **orderIdPrefix**, **orderLastNumber**) and the default PaymentMethod to use for refunds (**refundPaymentMethodId**).

One of the more important fields is **errorGlJournalId**. This is the GlJournal to put transactions (AcctgTrans) in when there is a problem with automatic posting. Transactions in this journal should be reviewed periodically, and most importantly before closing a period, to resolve issues and post the transaction. The most common issue is not finding the configuration for the GlAccount for a particular entry (AcctgTransEntry). Another possible issue is that the debits and credits don't match.

The other entities in this package are for configuration the GlAccount to use for automated posting of various types of records that have a financial impact. The most general are GlAccountTypeDefault and GlAccountTypePartyDefault which are used to configure the default account for different GL account types if no more specific mapping is found.

For **Invoice posting** the various items are mapped by their ItemType (the same item type that is shared among OrderItem, ReturnItem, and InvoiceItem) using ItemTypeGlAccount. If a more specific mapping is found for an InvoiceItem it will be used. This may be for specific products with ProductGlAccount or ProductCategoryGlAccount or for tax items for a specific TaxAuthority with TaxAuthorityGlAccount. The balancing entry for an invoice is generally a debit to the default accounts receivable type account, or a credit to the default accounts payable type account.

For **Payment posting** the PaymentTypeGlAccount entity is used to find the balancing liability or asset (AR, AP, etc) GL account for the payment for an Organization by **paymentTypeEnumId**, **isApplied**, and **isPayable** (i.e., payable versus receivable). The cash

account to post to is found for the payment method using `PaymentMethodTypeGlAccount` unless a more specific mapping is found for the credit card type in `CreditCardTypeGlAccount` or for a financial account type in `FinancialAccountTypeGlAccount`.

For inventory postings the GL account is determined generally with `AssetTypeGlAccount`, but for physical inventory variances the gain or loss is posted according to the variance reason configured with the `VarianceReasonGlAccount` entity.

Ledger - Reconciliation (mantle.ledger.reconciliation)

`GlReconciliation` is used to record results of reconciliation with external sources such as a bank statement. Each `GlReconciliation` record is associated with a `GlAccount` (**`glAccountId`**), and generally for a specific `Organization` (**`organizationPartyId`**) for reconciliation on a certain date (**`reconciledDate`**). It tracks the **`openingBalance`** and **`reconciledBalance`**. The actual records to reconcile are `AcctgTransEntry` and the **`reconciledAmount`** for each is tracked with the `GlReconciliationEntry` entity.

Ledger - Transaction (mantle.ledger.transaction)

An accounting transaction (`AcctgTrans`) is triggered by various things, and is associated with what triggered it or adds detail to what triggered it including asset issuance (**`assetIssuanceId`**), asset receipt (**`assetReceiptId`**), physical inventory (**`physicalInventoryId`**), invoice (**`invoiceId`**), payment (**`paymentId`**), payment application (**`paymentApplicationId`**) financial account transaction (**`finAccountTransId`**), shipment (**`shipmentId`**), and work effort (**`workEffortId`**). Transactions may also be created manually, i.e., not just through automated posting.

There are many types of accounting transaction (**`acctgTransTypeEnumId`**). The most common ones are Sales Invoice (from Org to Customer), Purchase Invoice (from Vendor to Org), Asset Receipt, Sales Inventory, Incoming Payment (Receipt), and Outgoing Payment (Disbursement). More exotic types include Amortization, Capitalization, Period Closing, and Credit Memo.

An `AcctgTrans` happens in the context of an internal `Organization` (**`organizationPartyId`**), happens at a certain date/time (**`transactionDate`**), knows if it is posted yet (**`isPosted`**), and if so the date/time when (**`postedDate`**). It may be in a single journal, such as the organization's error journal, with the **`glJournalId`** field. The currency it is posted in is tracked in the **`amountUomId`** field, and if that is different from the currency of whatever the transaction is based on (such as an order) that currency goes in **`origCurrencyAmountUomId`**.

Each transaction entry (`AcctgTransEntry`) may be a debit or credit (**`debitCreditFlag`** of 'D' or 'C'), has an **`amount`** and if the posting currency is different from the currency of what the transaction is based on the amount in the original currency goes in **`origCurrencyAmount`**.

Each is associated with a specific GL account type (**glAccountTypeEnumId**), and in order to post successfully must be associated with a GL account (**glAccountId**).

An entry may be a summary of transactions from an external system and if so **isSummary** is set to **Y**. For invoice items the **invoiceId** is on the AcctgTrans record and the **invoiceItemSeqId** is on the AcctgTransEntry. The entry may also be associated with a product (**productId**) and/or asset (**assetId**).

Journals (GlJournal) may be used to keep track of specific accounting transactions, usually for transactions with errors or manual transactions and are in progress (**glJournalTypeEnumId**). They are for a particular organization (**organizationPartyId**) and single-use journals may be posted all at once, tracked with **isPosted** and **postedDate**. Transactions are associated with journals using the AcctgTrans.**glJournalId** field.

Other - Budget (mantle.other.budget)

A Budget is generally associated with a TimePeriod (**timePeriodId**) and may be of various types (**budgetTypeEnumId**) such as Capital or Operating. Each BudgetItem has an amount and may have text descriptions of purpose and justification. The item type (**budgetItemTypeEnumId**) is generally something like Required or Discretionary. Parties may be associated with a budget using BudgetParty.

Various other entities point to BudgetItem records to provide detail for them, including: Payment through PaymentBudgetAllocation, EmplPosition, OrderItem, and Requirement through RequirementBudgetAllocation.

When a budget is reviewed by a particular party the results of the review are recorded with the BudgetReview entity. To keep a history of budget revisions use the BudgetRevision and BudgetRevisionImpact entities.

During a budget planning process various scenarios may be discussed and modeled. These can be recorded with the BudgetScenario and details for specific items in BudgetScenarioApplication and more generally for budget item types in BudgetScenarioRule.

Other - Tax (mantle.other.tax)

A TaxAuthority is a government entity (**taxAuthPartyId**) that collects taxes within a geographic boundary (**taxAuthGeoId**). For VAT tax authorities set **includeTaxInPrice** to Y. If a tax ID is required for exemption set **requireTaxIdForExemption** to Y.

Many tax authorities have different tax rates for different types of products. To configure this create a ProductCategory for each type and use TaxAuthorityCategory to associate it with the tax authority. Tax authorities may be associated with other tax authorities using TaxAuthorityAssoc for Exemption Inheritance or as a Collection Agent (**assocTypeEnumId**). For example a US state tax authority may collect taxes on behalf of a

city or county tax authority within that state, and exemption at the state level may exempt at the city or county level.

Parties may be associated with a `TaxAuthority` using `TaxAuthorityParty`. This is useful to represent that an internal organization has a nexus (**`isNexus`**=`Y`) or that a customer is tax exempt (**`isExempt`**=`Y`) and in either case the `Party` may have an ID issued by that tax authority and that is recored in the **`partyTaxId`** field.

Tax may be calculated using an external system or internal services configured using `TaxGatewayConfig` that in either case points to the service (**`calculateServiceName`**) that calculates the taxes or calls out to the external system. There used to be a `TaxAuthorityRate` entity for configuring local tax calculation, but that has been replaced with a Drools decision table which is more flexible. The `TaxGatewayConfig` is associated with a `ProductStore` using the `ProductStore`**`.taxGatewayConfigId`** field.

Facility

Facility (mantle.facility)

A facility is a building, unit, room, land or even floor space. There are also more particular types of facility such as warehouse and office. The primary entity for a facility is what you would imagine (`Facility`) and it is identified by a single PK field (**`facilityId`**). As with many of the main (master) entities a facility has a type (**`facilityTypeEnumId`**), status (**`statusId`**), and name (**`facilityName`**). There are also fields for the owner, size, open/close dates, etc.

Facilities are hierarchical to model things like units within a building and rooms within a unit. The `Facility`**`.parentFacilityId`** field is used to specify the parent for each facility. In theory this could be used for things like warehouse inventory locations but to simplify and flatten that structure the `FacilityLocation` entity is just for inventory locations within a facility. These have a type (such as bulk or pick/primary), locator fields (**`areaId`**, **`aisleId`**, **`sectionId`**, **`levelId`**, and **`positionId`**), and even a **`geoPointId`** for GPS-driven automation.

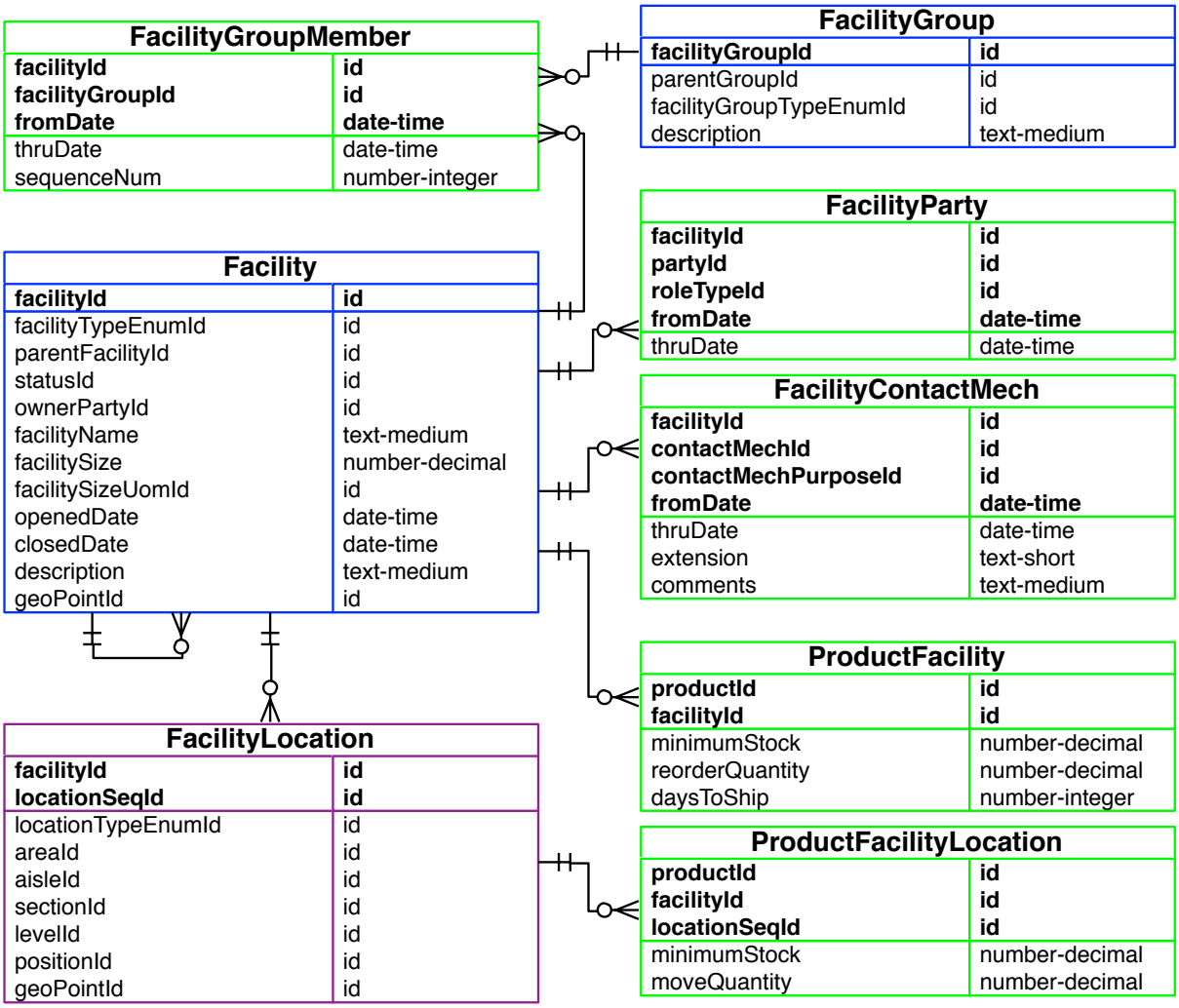
A `Product` may be associated with a `FacilityLocation` using the `ProductFacilityLocation` entity to record which products go in which locations, and to set **`minimumStock`** and **`moveQuantity`** values to use for recommended stock moves (when replenishing pick/primary locations from bulk locations). If you need to track more data about a particular product in a particular location extend this entity.

Similarly a `Product` may be associated with a `Facility` to using the `ProductFacility` entity to specify **`minimumStock`** and **`reorderQuantity`** values for use in simple automated (recommended) replenishment. Other fields related to a particular product in a particular facility can be added to this entity as needed.

The physical location of a facility can be recorded in two ways: through a `GeoPoint` record referenced by the `Facility`.**`geoPointId`** field, or in a `PostalAddress` type of `ContactMech`

with the FacilityContactMech. FacilityContactMech can also be used for more general contact information for the facility including phone/fax/etc (telecom) numbers, email and web addresses, and even multiple postal addresses when there are different ones for things like receiving correspondence, receiving shipments, shipping return address, etc.

For more details about ContactMech see the **Contact Mechanism (mantle.party.contact)** section.



Use the FacilityParty to associate a party (**partyId**) with a facility (**facilityId**) in a particular role (**roleTypeId**) and within an effective date range (**fromDate**, **thruDate**). This can be used for any role and might be used to record who is an owner, tenant, occupant, manager, picker, packer, etc for a particular facility.

To associate Resource Facade content (with a **contentLocation**) with a facility use the FacilityContent entity. This has a content type (**facilityContentTypeEnumId**) such as

internal content (documents, etc) and images, and the ever useful effective date range (**fromDate**, **thruDate**).

To organize facilities for pricing or management purposes, or more generally to keep better track of large numbers of facilities, use FacilityGroup. Facility groups have a **description**, are hierarchical (**parentGroupId**) and have a type (**facilityGroupTypeEnumId**) such as management structure or pricing group. To associate a Facility with a FacilityGroup use the FacilityGroupMember entity. You may also associate a party in a particular role with a facility group with the FacilityGroupParty entity.

Human Resources

Ability (mantle.humanres.ability)

The most general representation of ability is PartyResume which may have the full text in the **resumeText** field or may point to a Resource Facade **contentLocation**.

Getting more structured the PartyQualification entity is used for things like degrees, certifications, and work experience. The types available (**qualificationTypeEnumId**) are Enumeration records of type QualificationType and you can add any needed there. It has a verificationStatusId for tracking verification, and a more general status (**statusId**) that can be Completed, Incomplete, or Deferred for things like degrees and certifications, and Full-time, Part-time, or Contractor for things like work experience.

PartySkill is for more specific skills as opposed to more general qualifications. This would include things like specific programming languages and libraries, equipment operation, and even creative talents. The skill types (**skillTypeEnumId**) are Enumeration records of type SkillType. This has fields about the skill such as **yearsExperience**, **skillLevel**, and **startedUsingDate**.

A PerformanceReview is between a manager (**managerPartyId**) and employee (**employeePartyId**) for a particular position (**emplPositionId**). It has items (PerformanceReviewItem) of various types (**reviewItemTypeEnumId**) such as Responsibility, Attitude, and Job Satisfaction with a rating (**reviewRatingEnumId**) and **comments** for each. Outside the context of a review there may also be performance notes recorded with the PerformanceNote entity.

To track employer sponsored and other training use the TrainingClass entity for classes available and PersonTraining for classes to approve and/or actually completed.

Employment (mantle.humanres.employment)

The Employment entity is used to track employment of an employee (**employeePartyId**) by an employer (**employerPartyId**) in a certain position (**emplPositionId**) within a date range

(**fromDate**, **thruDate**). When employment is terminated it can track a reason (**terminationReasonEnumId**) and type (**terminationTypeEnumId**).

Benefits of BenefitType may be tracked with EmploymentBenefit, the relevant PayGrade with EmploymentPayGrade, and payroll preferences with PayrollPreference.

Before employment there may be an application (EmploymentApplication) by an applicant (**applyingPartyId**) for a position (**emplPositionId**) and optionally associated with a JobRequisition (**jobRequisitionId**).

After employment any unemployment claims would be tracked with UnemploymentClaim.

Position (mantle.humanres.position)

An EmplPosition is a specific position for a single Person (**filledByPartyId**) within an organization (**employerOrganizationPartyId**). For other parties associated with the position such as manager or department use the EmplPositionParty entity. EmplPosition has a pay grade (**payGradeId**), may be part of a budget (**budgetId**, **budgetItemSeqId**) and may be planned for a date range (**estimatedFromDate**, **estimatedThruDate**).

A position is associated with an employment position class (**emplPositionClassId** pointing to EmplPositionClass) like Programmer, Business Analyst, Project Manager, and so on. It is common to have multiple positions for a class, and a class can exist separately and be associated directly with parties (EmplPositionClassParty) for a simplified model for rate determination and such that does not require a EmplPosition record.

Responsibilities such as Finance Management, Inventory Management, and Purchase Management may be associated with a position using EmplPositionResponsibility or with a class using EmplClassResponsibility. A few responsibilities are defined OOTB and additional ones may be defined with Enumeration records of type EmploymentResponsibility.

Rate (mantle.humanres.rate)

Within an organization it is often useful to standardize pay grades. Use the PayGrade entity for pay grades available, and PayGradeSalary for the actual pay **amount** within a date range (**fromDate**, **thruDate**).

For more detailed and structured pay rate information use the RateAmount entity. This can be used for billing rates to clients for services performed, and payment to external vendors if applicable for actually performing services (**ratePurposeEnumId**). Rate types (**rateTypeEnumId**) include Standard, Discounted, Overtime, and On-site Work.

The **rateAmount** (with currency **rateCurrencyUomId** and for time unit **timePeriodUomId**) is valid within a date range (**fromDate**, **thruDate**) and may be restricted to a particular Party (**partyId**), WorkEffort (**workEffortId**), and position class (**emplPositionClassId**).

Recruitment (mantle.humanres.recruitment)

The recruitment process will often begin with creating a `JobRequisition` and one or more `EmplPosition` records for the requisition. Typically `EmploymentApplication` records are next to apply for the position, and then for some of the applications zero to many `JobInterview` records, one for each interview done with the candidate (**`candidatePartyId`**) by an interviewer (**`interviewerPartyId`**). For each position an `Employment` record is created when a candidate is hired.

Marketing

Campaign (mantle.marketing.campaign)

A `MarketingCampaign` is used for general tracking of marketing efforts and may be used for efforts that tracked in the system, or may be used to group other things like `ContactList`, `TrackingCode`, and `SalesOpportunity`.

A campaign has various budget/cost fields including **`budgetedCost`**, **`actualCost`**, and **`estimatedCost`**. It is valid within an optional date range (**`fromDate`**, **`thruDate`**). For campaign results there are fields like **`convertedLeads`**, **`expectedResponsePercent`**, and **`expectedRevenue`**.

A campaign may have various parties like marketers, sales reps, managers, prospects, and leads associated with it using `MarketingCampaignParty`. Use the `MarketingCampaignNote` entity to track notes about the campaign, which are in addition to the **`campaignName`** and **`campaignSummary`** fields on the campaign itself.

Contact (mantle.marketing.contact)

A `ContactList` is used to plan and track mass outgoing communication such as Marketing, Newsletter, and Announcement (**`contactListTypeEnumId`**). This can be by email, phone, postal mail, or any other means of contact (**`contactMechTypeEnumId`**). It may be associated with a `MarketingCampaign` (**`marketingCampaignId`**).

A contact list is generally owned/managed by a particular `Party` (**`ownerPartyId`**). Other parties are associated with it using `ContactListParty`. The main use for this is parties who will receive the outgoing communication and optionally how they should be contacted (**`preferredContactMechId`**). Most emailing lists are opt-in and this is often done with an outgoing email to verify the address and the opt-in with a code, which is tracked for verification with the **`optInVerifyCode`** field.

A `ContactListParty` has a status (**`statusId`**) which may be Pending Acceptance, Accepted, Rejected, In Use, Invalid, Unsubscribe Pending, or Unsubscribed.

To configure outgoing email for the list, including types (**emailTypeEnumId**) such as Subscribe Notification, Unsubscribe Verify, Unsubscribe Notification, and Outgoing Email use the ContactListEmail entity. This points to a Moqui EmailTemplate record (with **emailTemplateId**) to be used with the org.moqui.impl.EmailServices.**send#EmailTemplate** service.

To track actual communication use a CommunicationEvent record associated with the contact list using ContactListCommStatus. Use this to track the Party (**partyId**) and actual ContactMech (**contactMechId**) used, though further details are on the CommunicationEvent record. See the **Communication Event (mantle.party.communication)** section for additional details.

Segment (mantle.marketing.segment)

The MarketSegment and related entities are used to define a group (segment) of Party records by PartyClassification using MarketSegmentClassification, by Geo (geographic boundary) using MarketSegmentGeo, and by Organization parties using MarketSegmentParty for all parties in the organization.

A segment can be used for many purposes such as populating ContactListParty records based on all current Party records in the system that match the segment criteria or recording interest in a set of products in a ProductCategory using the MarketInterest entity.

Tracking (mantle.marketing.tracking)

A TrackingCode can be used for internal path tracking for critical web pages or for AB or other multivariate testing. It can also be used to track incoming links from affiliates for particular orders to pay affiliate commissions.

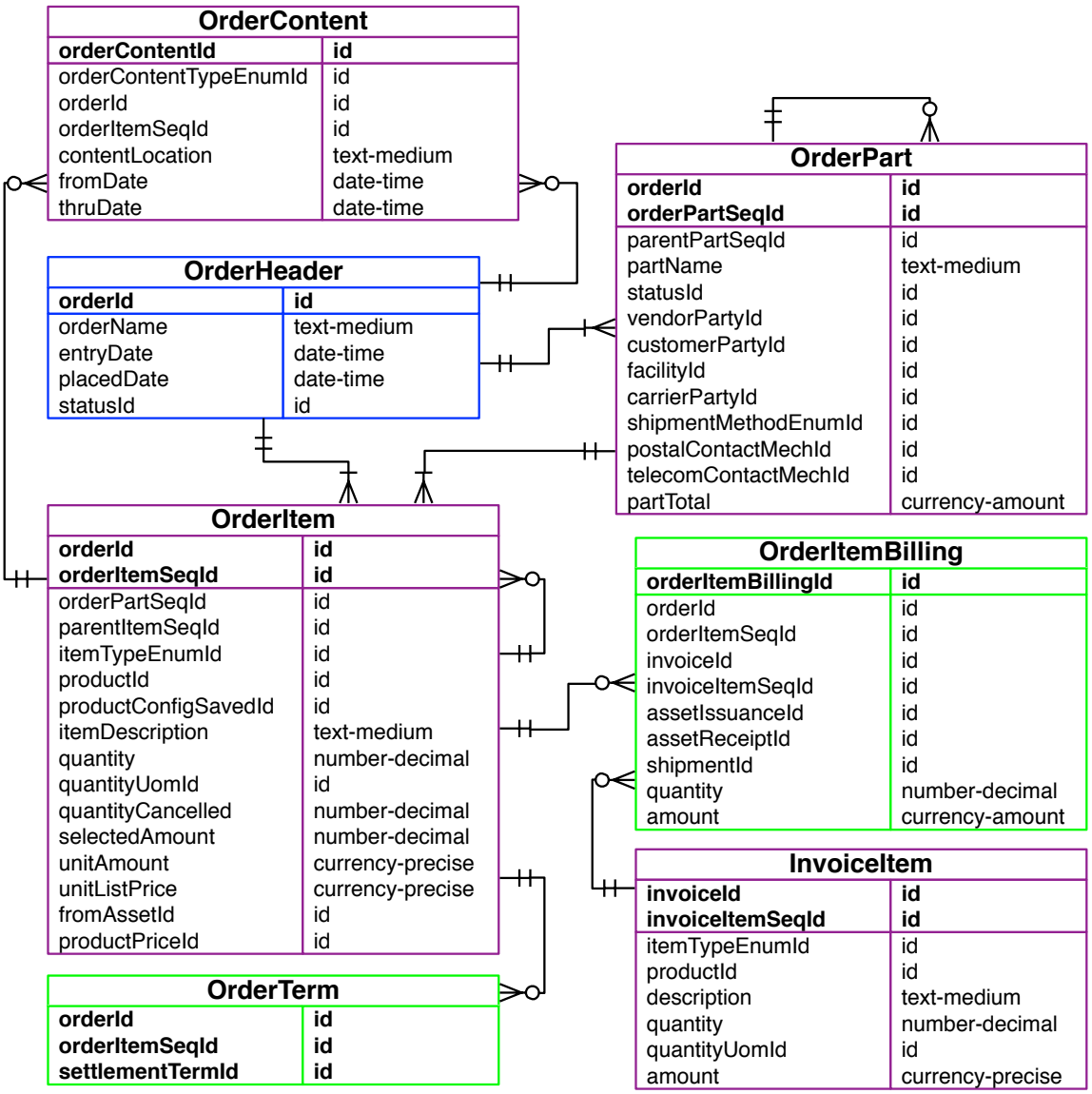
Once a tracking code is in the system it can be associated with a Moqui web Visit using TrackingCodeVisit, with an order (for conversion tracking and affiliate commissions) using TrackingCodeOrder and with returns using TrackingCodeOrderReturn.

For affiliate commissions that follow browser cookie preservation rules the tracking code is generally put in a cookie and then pulled from the cookie when an order is placed as opposed to remembering it through more means. The tracking codes associated with a Visit are different, they are generally all tracking codes used during a Visit and orders can then be tied to these through the **visitId** field on OrderHeader.

Order

Order (mantle.order)

The primary entity for an order is `OrderHeader`. An order can be a purchase or sales order, and in fact with the `OrderPart` structure supports multi-party orders since each order part has a **`customerPartyId`** and a **`vendorPartyId`**. Order parts are used to split the order for other purposes such as shipping to different locations or by different methods, to ship from different locations, and so on. Order parts can have other parties associated with them using the `OrderPartParty` entity. Order parts are also used to split orders by different shipping addresses, shipment options, delivery dates, etc.



The shipping address (a type of contact mechanism) is referenced in the `OrderPart`. **`postalContactMechId`** field and there is an optional **`telecomContactMechId`** field to point to a phone (telecommunications) number. Additional contact mechs can be associated with the order by purpose (such as billing phone, shipping address) using the `OrderContactMech` entity.

With a wide variety of statuses an order can be a shopping cart (Open/Tentative), quote (Proposed by Vendor), or a placed order (Accepted by Customer). There are also statuses so an order can be a wish list, gift registry, and auto reorder (order stays open for automatic recurring orders, each of which is a separate order).

After an order is Placed it can be fulfilled and is eventually either Completed, Cancelled by the customer, or Rejected by the vendor. It can also be Held or put in a special Being Changed status temporarily to avoid automatic calculation of things like shipping and taxes. Both `OrderHeader` and `OrderPart` have **`statusId`** fields to track statuses independently. Order items do not have a **`statusId`** field, their status is determined by looking at quantities on the item and quantities fulfilled, etc.

The items on an order are recorded as `OrderItem` records. For simplicity each `OrderItem` is associated with a single `OrderPart` record. `OrderItem` records are hierarchical so that they can be used for adjusting or detailing a parent item. This is useful for things like sales tax and discounts that apply to a single item. It is also useful for highly complex orders where items are organized under other items, such as specific building materials that are used for different parts of a structure of phases of building it.

Order items use the same set of types as other items including `mantle.account.invoice.InvoiceItem` and `mantle.order.return.ReturnItem`. These shared item types are defined in the `ItemTypeData.xml` file. There are a wide variety of types for things like sales, purchase, expenses, commissions, and payroll. For sales orders the most common types are product, time entry, shipping charges, sales taxes, and discounts.

When an `OrderItem` is billed (invoiced) it is associated with the `InvoiceItem` using the `OrderItemBilling` entity. Often billing of physical goods is done when a `Shipment` is sent (actually packed) or received, so there is a **`shipmentId`** on the `OrderItemBilling` entity. For outgoing shipments there is an inventory issuance modeled as a `AssetIssuance` record and we have the **`assetIssuanceId`** field to point to it. Similarly for incoming shipments there is a `AssetReceipt` record pointed to by the **`assetReceiptId`** field.

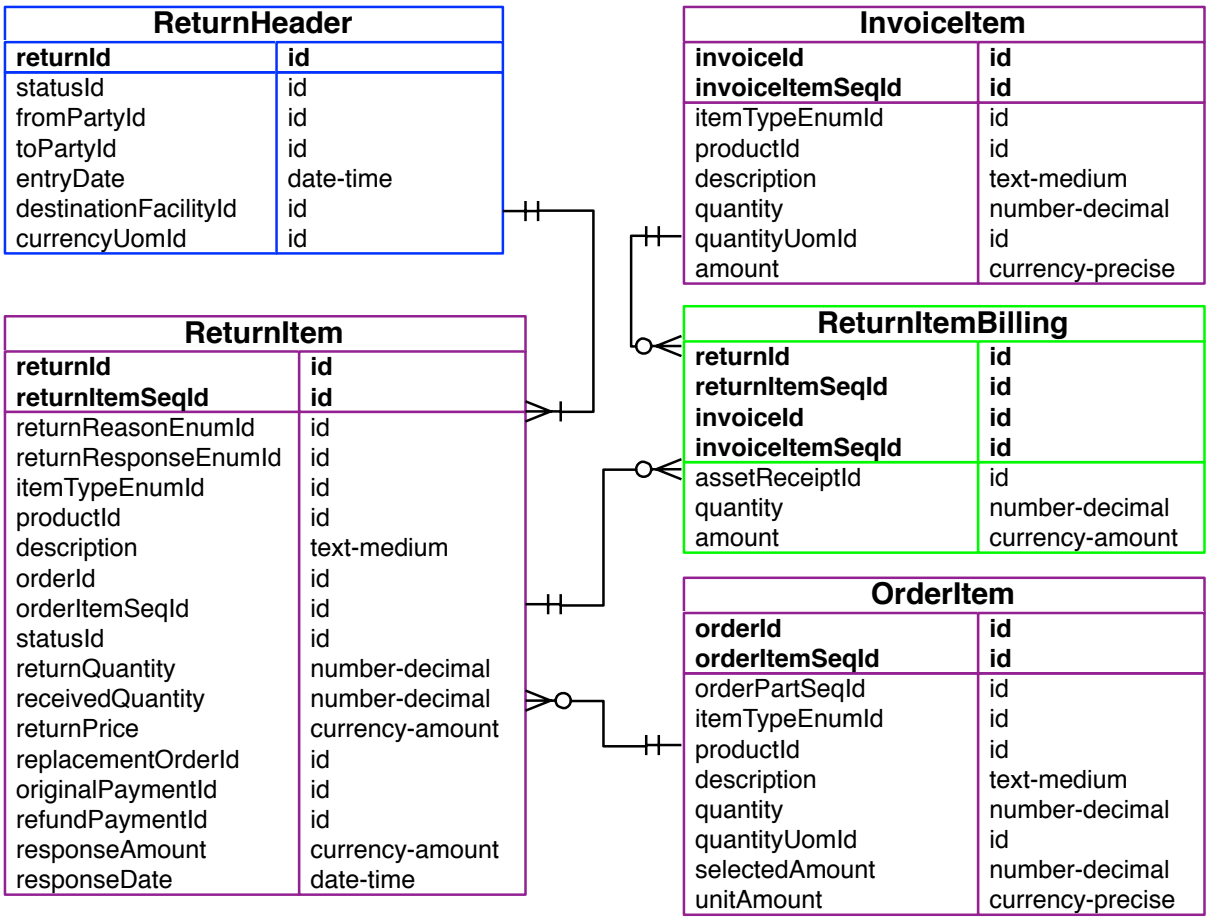
When an `OrderItem` is associated with a task, project or other type of `WorkEffort` (usually for work/service orders) it is associated with it using the `OrderItemWorkEffort` entity.

Orders may have a number of other records associated with them, including communication events (`OrderCommunicationEvent`), content such as documents or images (`OrderContent`), notes (`OrderNote`), and payment or other terms (`OrderTerm`).

Return (mantle.order.return)

A return (`ReturnHeader`) tracks the details of requesting and processing order item returns from the customer (**`fromPartyId`**) to the vendor (**`toPartyId`**). Note that either `Party` may be an internal organization, or in other words the return may be incoming (receiving a return from a customer) or outgoing (sending a return to a supplier).

Each `ReturnItem` record points to a `OrderItem` record and specified the **`returnQuantity`** for that item. There is a separate field, **`receivedQuantity`**, to track the quantity of the item actually received for the return. Each `ReturnItem` also has a **`itemTypeEnumId`** just like the `OrderItem` so that any type of item can be "returned" (including products, taxes, shipping charges, discounts, etc) and considered in the refund or other response.



Each `ReturnItem` has a **`returnReasonEnumId`** (like did not want, defective, mis-shipped, etc) for tracking purpose. Each item also has a **`returnResponseEnumId`** to specify how the organization should respond to the returned item (like refund, store credit, various methods of replacement, etc). There are fields on the item for tracking related records for the response (**`replacementOrderId`**, **`refundPaymentId`**, **`billingAccountId`**, **`finAccountTransId`**).

For refunds there will be an invoice based on the return for financial tracking (which will result in GL posting, etc) and the `ReturnItemBilling` is used to associated each `ReturnItem` with an `InvoiceItem`.

Both `ReturnHeader` and `ReturnItem` have a **`statusId`** field to track the progress of each item, and as major steps are completed the status of the return as a whole. OOTB statuses include: Created, Requested, Approved, Shipped, Received, Completed, Manual Response Required, and Cancelled.

Party

Party (mantle.party)

The term party in this case has a meaning like the legal term of a party to a lawsuit as in an individual or group, not the fun kind of party. There are two types of party and each has its own entity to add applicable detail to the `Party` entity: `Person` representing an individual and `Organization` which is a group and each member of the group may be a person or organization. These entities have the same primary key field as the `Party` entity (**`partyId`**) so that they have a one-to-one relationship.

The name of a `Party` comes from different fields depending on its type. For organizations it comes from the `Organization`**`.organizationName`** field. For persons (people) it comes from multiple fields on the `Person` entity: **`salutation`**`, `**`firstName`**`, `**`middleName`**`, `**`lastName`**`, `**`personalTitle`**`, `**`suffix`**`,` and **`nickname`**. Usually at least first and last names are used, and the others less commonly. There are various other fields on `Party`, `Person`, and `Organization` to specify details about parties, and just like any other entities you can extend these to add any others you might need.

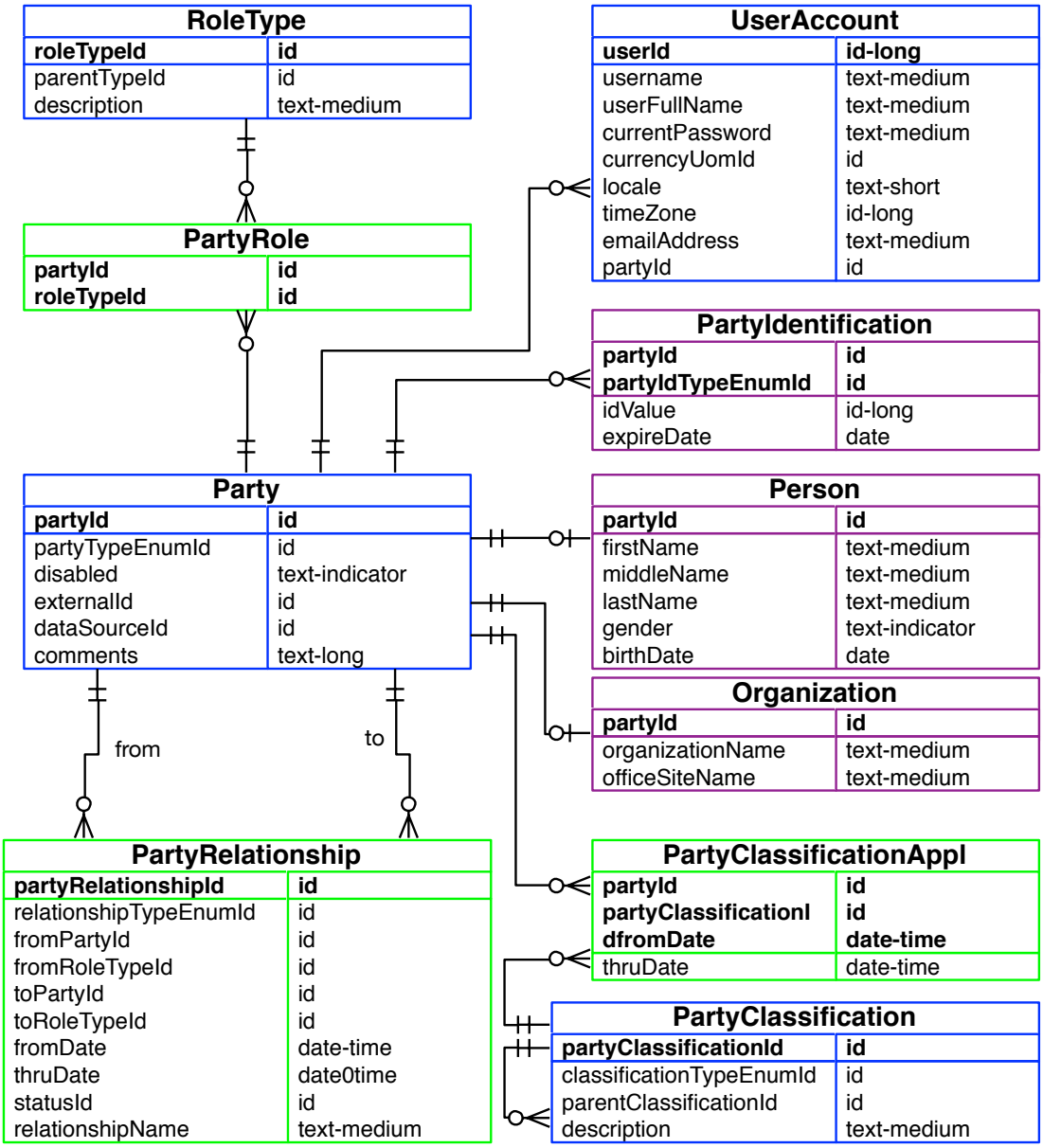
Each party may have zero to many roles that are used to define how a party relates to other structures in the system such as orders, work efforts (tasks, etc), agreements, and even other parties. The available roles are defined using the `RoleType` entity and there is a fairly comprehensive set of them defined OOTB in Mantle. Some examples of roles include: carrier, bill-to customer, ship-from vendor, employee, affiliate, and spouse.

A role can be associated with a party using the `PartyRole` entity. Entities that have a **`partyId`** and a **`roleTypeId`** intentionally have foreign keys just to `Party` and `RoleType` and not to `PartyRole` so that `PartyRole` records are optional. In some cases it is useful to see if a party is in a certain role, and `PartyRole` is the entity you would use for that.

Relationships between parties are recorded with the `PartyRelationship` entity. These include members of a group, employees of an organization, organization hierarchies (rollup), contacts, friends, and so on. There are various OOTB relationship types for the **`relationshipTypeEnumId`** field, and you can add more by adding `Enumeration` records with **`enumTypeId`**`=PartyRelationshipType`. In addition to the relationship type there are from and to party and role type fields that detail the nature relationship. When needed there

are also effective date (**fromDate, thruDate**) fields and a **statusId** field (which like most statuses is has **enable-audit-log**=true for a status history).

A party may have multiple identifiers such as a driver license number, employee number, and external system identifiers for correlation. These are stored with the PartyIdentification entity. This entity has an **idValue**, a **partyIdTypeEnumId** for the type of ID, and an optional **expireDate** for identifiers that expire.



Party classifications are used to classify parties by industry/SIC/NAICS, size, revenue, minority/EEOC, etc. Each PartyClassification (such as the NAICS industry classification

541511 - Custom Computer Programming Services) has a **classificationTypeEnumId** (such as PctNaicsCode) and optional **parentClassificationId** to organize them. Use the PartyClassificationAppl entity to associate a party with a classification in a date range (**fromDate, thruDate**).

A party can be have associated Resource Facade content with PartyContent, geographic points with PartyGeoPoint, and notes with PartyNote. A party may be associated with a Moqui Framework UserAccount with the UserAccount**.partyId** field that Mantle adds to it using extend-entity.

Agreement (mantle.party.agreement)

Agreement is used to track sales, employment, commission, and other types (**agreementTypeEnumId**) of agreements. An agreement is typically between two parties (**fromPartyId**, **toPartyId**) and those parties may be in specific roles (**fromRoleTypeId**, **toRoleTypeId**). Additional parties may be associated with it using the AgreementParty entity. Parties may also be associated with a specific item on the agreement using the AgreementItemParty entity.

An agreement is made on a certain date (**agreementDate**) and is valid within a date range (**fromDate**, **thruDate**). You can record a **description** for the agreement and the full text in **textData** if available.

An agreement is detailed with one or more AgreementItem records that have most of the structure around an agreement. An item may have its own detail text (**itemText**) and its own effective date range (**fromDate**, **thruDate**). An item will typically have a type (**agreementItemTypeEnumId**) such as Sub-Agreement, Pricing, Section, or Commission Rate. When relevant the currency for an item is tracked with the **currencyUomId** field.

If an agreement is changed it should be tracked with an addendum using the AgreementAddendum entity, which can be applied to a particular item or the entire agreement.

An item or the entire agreement may also have terms, recorded with the AgreementTerm entity, such as Payment, Fee, Penalty, Incentive, Termination, Indemnification, Commission, and Purchasing terms. There are various others defined and you can define more by adding Enumeration records with the type TermType. These are also used for BillingAccount and for Invoice through the SettlementTerm entity.

Use AgreementItemGeo to associated an item with a specific geographic boundary (Geo), and AgreementItemWorkEffort for a WorkEffort such as a project. When an agreement (item) is for employment, associated the item with the Employment record using AgreementItemEmployment.

For product pricing agreement items the ProductPrice has **agreementId** and **agreementItemSeqId** fields to point to an AgreementItem. This provides structured detail about the pricing, and can be used for automated price calculation for a particular order.

Communication Event (mantle.party.communication)

Use CommunicationEvent to keep track of communication between parties (**fromPartyId**, **toPartyId**), optionally in particular roles (**fromRoleTypeId**, **toRoleTypeId**), and also optionally with specific contact mechanisms (**fromContactMechId**, **toContactMechId**; see the next section for ContactMech details). Even if there are not specific contact mechs associated with the communication event the type (**contactMechTypeEnumId**) such as phone/telecom number or email address can be.

In addition to the from and to parties other parties, along with a role and contact mech, can be associated with the CommunicationEvent using the CommunicationEventParty entity. This is especially useful for events like meetings and conference calls.

Communication event types are specified with the **communicationEventTypeId** field on the CommunicationEvent entity which points to a CommunicationEventType record. These types correlate to contact mech types. For example the phone comm event type is associated with the telecom number contact mech type using the **contactMechTypeEnumId** attribute on the CommunicationEventType entity.

CommunicationEvent has a status (**statusId**) for both incoming and outgoing events including In Progress, Ready, Sent, Received, Viewed, Resolved, Referred, Bounced, Unknown Party, and Cancelled. These statuses should handle most situations, including inbound email queues that need to be viewed and acted on (resolved). For status history this field use the Entity Facade audit log. The time of special events are tracked on the **entryDate**, **datetimeStarted**, and **datetimeEnded** fields.

Communication events are hierarchical to handle threaded discussions with a **parentCommEventId** to track the previous or parent comm event, and the **rootCommEventId** to tie all comm events to the comm event that initiated the thread.

If available the content of the comm event can be stored with the **subject**, **contentType** (MIME type), and **body** fields with any notes about it in the **note** field. For records from email messages the **Message-ID** header that identifies the email can be recorded with the **emailMessageId** field. Additional content can be saved in a Resource Facade location and associated with the comm event using the CommunicationEventContent entity.

One or more purposes, such as Customer Service and Sales Inquiry, for the comm event can be tracked with the CommunicationEventPurpose entity (separate entity so that multiple purposes can be associated with the comm event). The purpose is specified with the **purposeEnumId** field which points to Enumeration records of type CommunicationPurpose, so use the enum type to add more available purposes.

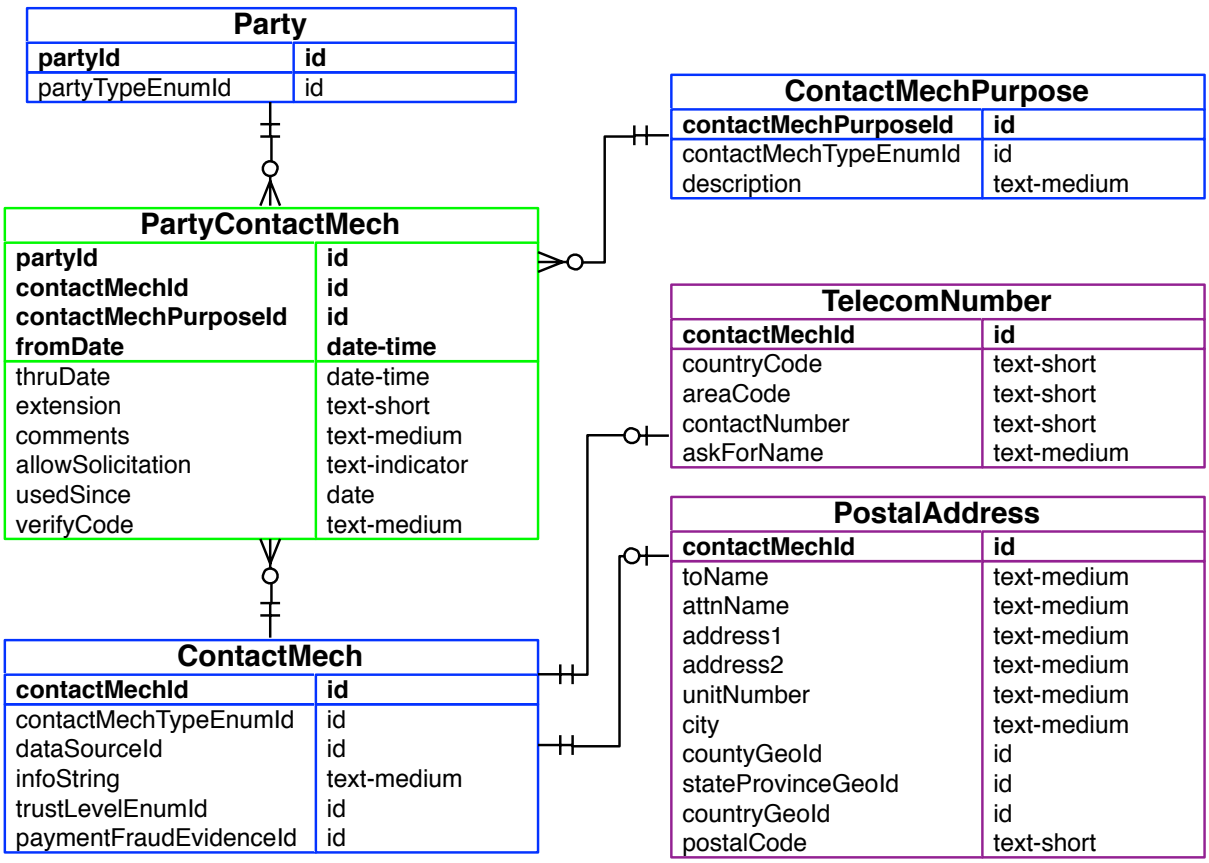
When relevant products may be associated with a comm event using the `CommunicationEventProduct` entity.

Contact Mechanism (mantle.party.contact)

A contact mechanism is a means of contacting a party. The primary entity is `ContactMech` and while there are various types only two have entities with additional fields: `PostalAddress` and `TelecomNumber`. The remaining types (such as email address) use the `ContactMech.infoString` field.

The primary key field of `ContactMech` is `contactMechId`. Like the pattern with `Party`, `Person`, and `Organization` the `ContactMech`, `PostalAddress`, and `TelecomNumber` entities share the same primary key field so they have a one-to-one relationship.

The `PartyContactMech` entity is used to associate a `ContactMech` with a `Party`. A purpose (**`contactMechPurposeId`**) describes what the `ContactMech` is for the `Party` such as destination shipping address or billing phone (telecom) number. The purposes are defined as records in the `ContactMechPurpose` entity. There is a comprehensive set available OOTB and you can add records to define more.



PartyContactMech has effective date (**fromDate, thruDate**) fields to define the date range where the ContactMech is valid for the Party. ContactMech records are immutable (they should never be changed) so that they can be referenced in other places without a change unintentionally effecting other places (and to keep a history of contact information). When one needs to be updated a new record is created and associated with the Party and the **thruDate** is set on the old PartyContactMech record to expire it. See the mantle.party.ContactServices.**update#PartyContactOther** service for details of how this is done (and there are similar services for postal addresses and telecom numbers to handle the additional fields and separate entities).

Where fraud is a concern the ContactMech should have a **trustLevelEnumId** set. OOTB options include New Data, Valid/Clean (through 3rd party service), Verified (with outbound contact or authorization), Greylisted, and Blacklisted. If the trust level is Greylisted or Blacklisted there should be a **paymentFraudEvidenceId** pointing to a PaymentFraudEvidence record with details about why.

Another entity that uses ContactMech similar to Party is mantle.facility.Facility. A facility has contact information just like a party and is a long-lasting record with multiple contact mechs that may change over time. Just like for Party there are services to update contact mechs for a facility (see the mantle.facility.ContactServices services) that expire the old record and create a new one.

There are many entities which refer to contact mechs, and some which use a join entity to associate multiple contact mechs with different purposes. These include InvoiceContactMech, OrderContactMech, ReturnContactMech, ShipmentContactMech, and WorkEffortContactMech. These entities do not have effective date (**fromDate, thruDate**) fields as they are short-lived and if contact information changes the **contactMechId** is simply updated to point to a different record.

Time Period (mantle.party.time)

The TimePeriod entity is for custom time periods, as opposed to calendar time periods, such as fiscal years/quarters/months and sales quarters (**timePeriodTypeId**, references the TimePeriodType entity). They may match calendar time periods, i.e. **fromDate** is the beginning of a calendar period and **thruDate** is the actual end of the calendar period, but are referenced anyway for any functionality that allows the time period to be something other than a calendar period.

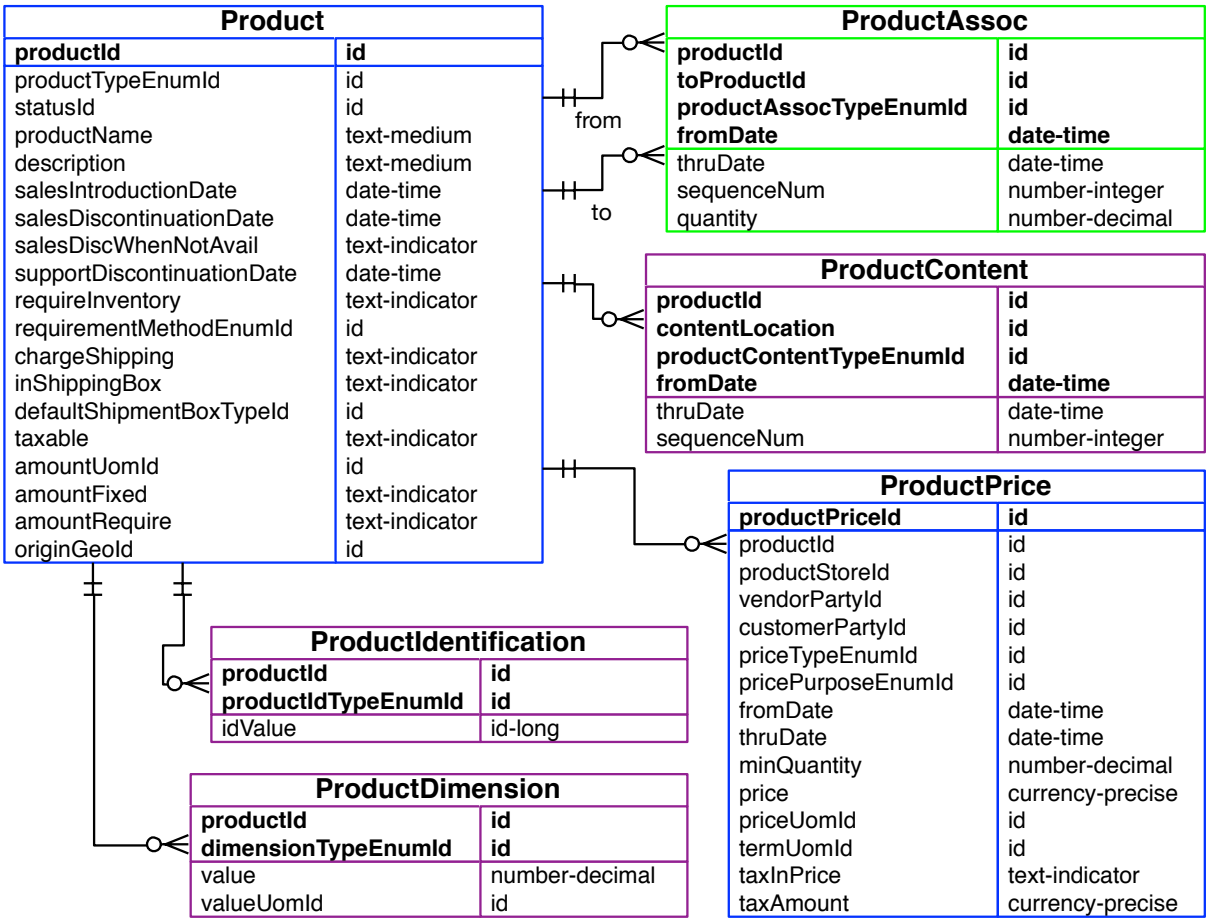
A time period can be linked to its parent (**parentPeriodId**) and previous (**previousPeriodId**) time periods. It can also be associated with a Party (**partyId**) for things such as fiscal time periods that are specific to an organization for accounting purposes. When a TimePeriod is used for the general ledger the **isClosed** field specifies when the period is closed and transactions can no longer be posted to it.

Product

Definition - Product (mantle.product)

A product is a person, place, or thing. Actually, that's a noun, but products are similar. A `Product` is a description of a service, facility use, asset use, or a digital or physical good for sale. For manufacturing a product can represent raw materials, subassemblies, finished goods, and so on. These product types are specified with the **`productTypeEnumId`** and the options available are `Enumeration` records with the type `ProductType`.

An instance of a `Product` is tracked in different places depending on what type of product it is. Physical goods are tracked as inventory using the `Asset` and related entities. Asset use products are tracked as Asset records and have corresponding `WorkEffort` records for their schedule. Facility use products are tracked with `Facility` records, and also use `WorkEffort` for scheduling. Service products are tracked through a variety of `WorkEffort` records for projects, tasks, etc and may also have corresponding `Request` and `Requirement` records. The services will generally have one or more `Party` records associated with them for the people and/or organizations that will perform or have performed the service.



The Product entity has a **statusId**, but this is mostly there for special cases and is not used for certain things that might seem like statuses but are modeled as dates, including **salesIntroductionDate**, **salesDiscontinuationDate**, and **supportDiscontinuationDate**. If you want to know whether a product is available for sale, you check the current date/time against the sales date fields instead of looking at an indicator or status.

For content about the product it has **productName** and **description** fields, and the everything else such as more localized name/description, detailed descriptions, images, instructions, warnings, button/link labels, etc are all recorded with the ProductContent entity. The **contentLocation** points to a Resource Facade location so the content can be in a database (with the DbResource/File entities), a JCR (Java Content Repository, such as Apache JackRabbit), in the local filesystem, or any other location configured OOTB or that you add. See the **Resource Locations** section for more details.

Product has inventory (**requireInventory**, **requirementMethodEnumId**), shipping (**chargeShipping**, **inShippingBox**, **defaultShipmentBoxTypeId**, **returnable**), and tax (**taxable**, **taxCode**) settings. Some products have an amount associated with them, such as a number of cans in a case, or allow the user to enter an amount when purchasing it. Use the **amountUomId**, **amountFixed**, and **amountRequire** fields for this.

The various possible dimensions for a product are recorded with the ProductDimension entity. This would include weight, lengths dimensions, shipping dimensions, quantity and pieces included, and any other dimension you might want to define. To add other dimension types add Enumeration records of type ProductDimensionType. There is a similar structure for identifiers such as UPC, ISBN, EAN, etc: ProductIdentification.

Product has an **originGeoId** field to specify where the product comes from for import/export restrictions or for pure curiosity. For more specific Geo details like shipping and purchase restrictions use the ProductGeo entity.

A product can be associated with other products using ProductAssoc. This is useful for cross/up sell, size/color/etc variants, accessories, and for manufacturing purposes even BOM breakdowns. To associate a Product with a Party use the ProductParty entity. The ProductReview entity is used to record user/customer reviews and ratings.

The ProductPrice entity is used for a wide variety of prices, including: prices from suppliers and prices for customers (via **vendorPartyId** and **customerPartyId**); list, current, max/min, promotional, competitive, etc prices (**priceTypeEnumId**); purchase, recurring, and use prices (**pricePurposeEnumId**). For quantity breaks there is a **minQuantity** field (for any quantity greater than or equal to this, and less than the next highest matching record's **minQuantity**).

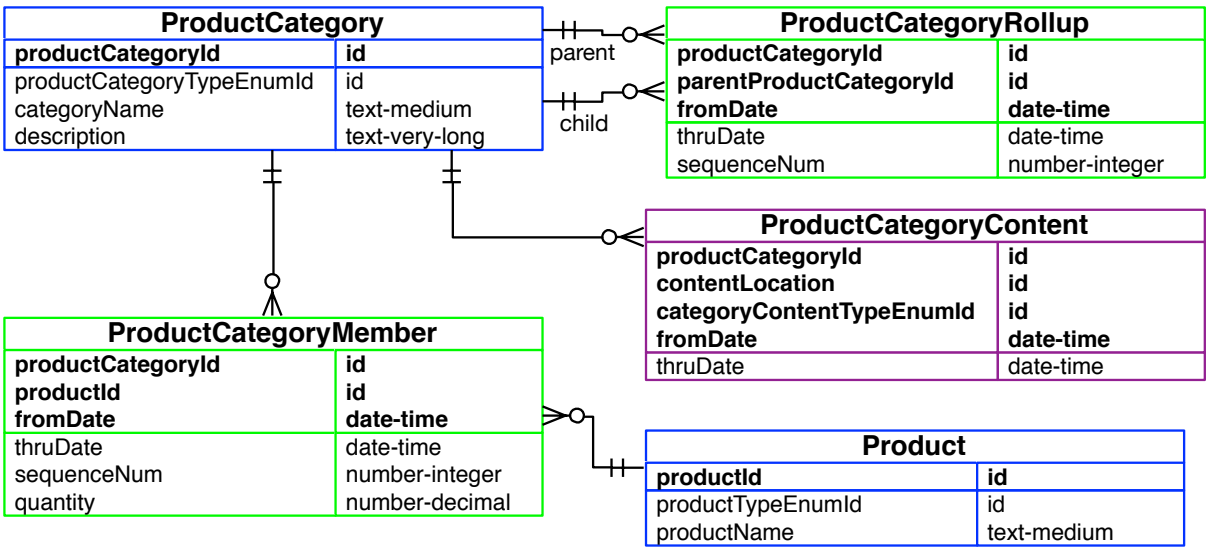
Prices are valid in an effective date range (**fromDate**, **thruDate**) and can be restricted to a particular ProductStore (**productStoreId**). For jurisdictions with VAT taxes the price can have tax included (**taxInPrice**=true), and use the other tax fields to specify details

(**taxAmount**, **taxPercentage**, **taxAuthPartyId**, **taxAuthGeoId**). See the **mantle.other.tax** section for more details about tax calculation.

The actual price goes in the **price** field, and its currency in the **priceUomId** field. See the **Units of Measure** section in the **Data and Resources** chapter for more details on UOMs. For recurring prices the recurrence term goes in **termUomId** and the price is the price per unit (like time, data size, etc).

Definition - Category (mantle.product.category)

The obvious use for a `ProductCategory` is a way to structure products within a catalog, but that is only one of various types (**productCategoryTypeEnumId**). More generally a category is a way to specify a set of products. Other common types include tax, cross sell, industry, search, and best selling.



Products are associated with a category using `ProductCategoryMember`, which is the join entity that supports a many-to-many relationship (products can be in many categories, categories can have many products). Categories are associated with parent/child categories using `ProductCategoryRollup`, which is also a many-to-many relationship so a category can have multiple parent and child categories. Both of these have effective dates (**fromDate**, **thruDate**) and a **sequenceNum** field for sorting products within categories, and subcategories of categories.

`ProductCategoryMember` also has a **quantity** field which can be used when a category represents a set of products that come in a sort of ad-hoc or recommended package.

Content from the Resource Facade can be associated with a category using `ProductCategoryContent`. Similarly, parties can be associated with a category using `ProductCategoryParty`.

Definition - Config (mantle.product.config)

Product configuration entities are used to specify configuration options for products of type Configurable Good (`Product.productTypeEnumId=PtConfigurableGood`). Configuration items are specified with the `ProductConfigItem` entity, and applied to the Configurable Good product using `ProductConfigItemAppl`. The options for a config item are specified with `ProductConfigOption`, and for options that are associated with another `Product` (which has its own inventory, pricing, supplier details, etc) use the `ProductConfigOptionProduct` entity.

To help clarify here is the path between the **configurable** product and the **component** product: `Product ==> ProductConfigItemAppl ==> ProductConfigItem ==> ProductConfigOption ==> ProductConfigOptionProduct ==> Product`.

The `ProductConfigOption` entity has a **`description`** field, and for localized description and other content associated with an option use the `ProductConfigItemContent` entity to reference Resource Facade content locations.

When a configurable product is configured, usually when added to an order, we need a place to save the configuration and that starts with the `ProductConfigSaved` entity. This is referenced on an order item using the `OrderItem.productConfigSavedId` field. Within the saved configuration the option selected for each item is recorded with the `ProductConfigSavedOption` entity.

Definition - Cost (mantle.product.cost)

`CostComponent` records are used to break down the cost of a `Product`, especially manufactured products. Products purchased from suppliers have a much simpler cost, the price paid to the supplier. Cost components include things like estimated and actual material, supply, equipment usage, and other costs. The various cost components added together for a particular product within a date range make up the cost of a product.

The `CostComponentCalc` entity, and the `ProductCostComponentCalc` to apply it to a Product, are used to specify how a `CostComponent` is to be calculated, or what the cost of a product should be based on.

The `ProductAverageCost` entity is used to keep track of the average cost of a `Product` over time (with a from/thru effective date range) optionally for a particular `Facility` and a particular `Organization`. This is mostly to be used for the purpose of COGS calculations that require an average cost history as opposed to being based on actual cost of an item sold.

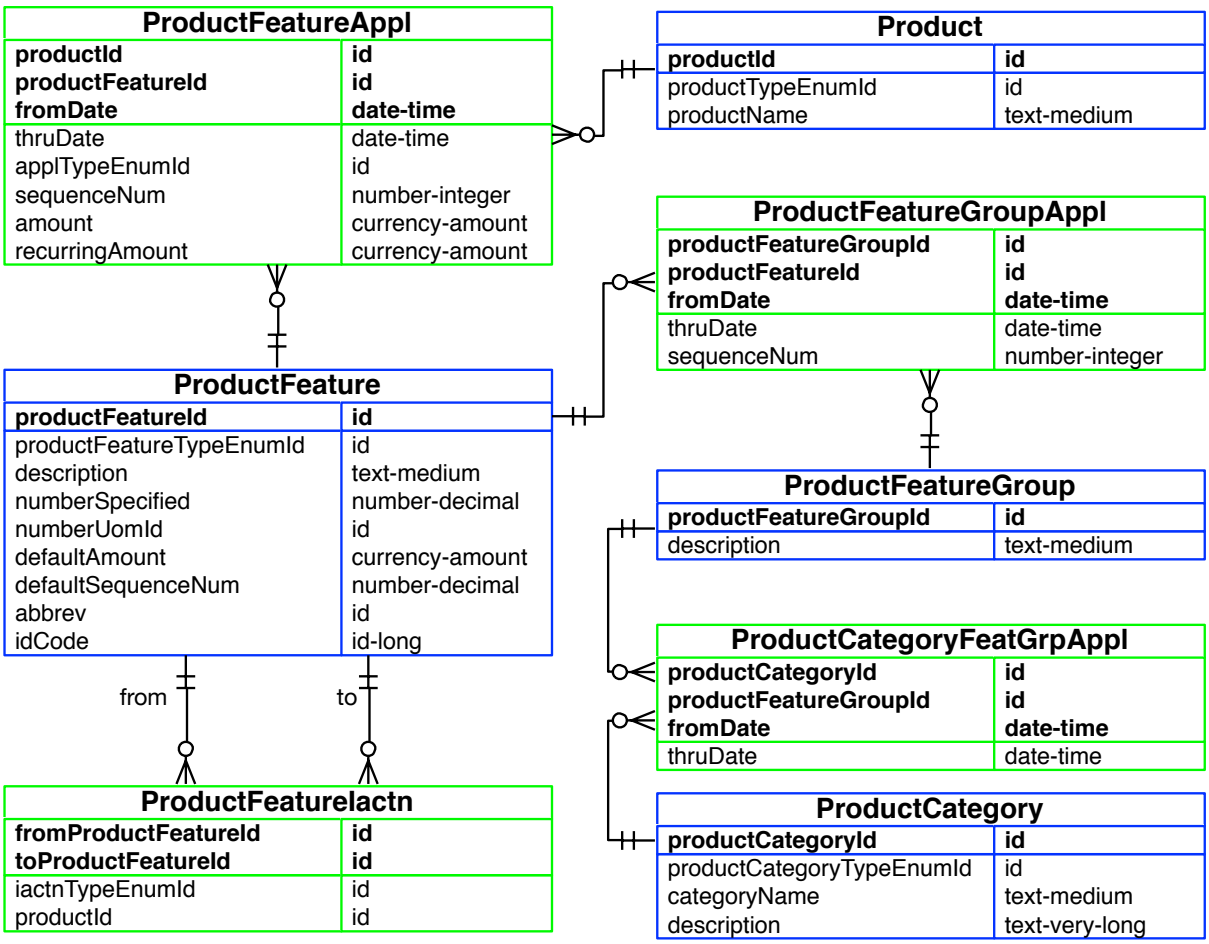
Definition - Feature (mantle.product.feature)

A `ProductFeature` describes a `Product` in a structured way. There are quite a few feature types (**`productFeatureTypeEnumId`**) defined OOTB, like Brand, Color, Fabric, License, and

Size. It is common to add customer feature types using `Enumeration` records of type `ProductFeatureType`.

A feature is applied to a product using the `ProductFeatureAppl` with a **`applTypeEnumId`** to specify what the feature is to the product (Selectable for optional features, Standard for inherent aspects of a product, or Distinguishing to describe variants or a virtual product), and within an effective date range (**`fromDate`**, **`thruDate`**).

Sometimes it is necessary to model features that are incompatible or dependent, use the `ProductFeatureIactn` entity for this.



Features are naturally organized by type, but it is often useful to define sets of features that are used for a particular purpose such as facets for search of certain products or that are used to describe certain types of products (mostly for administrative purposes). The `ProductFeatureGroup` entity does just that. Use `ProductFeatureGroupAppl` to specify which features belong in which groups. For feature groups that are associated with a `ProductCategory` use `ProductCategoryFeatGrpAppl` to tie them together.

Definition - Subscription (mantle.product.subscription)

A `Subscription` is used to record a party's (**`subscriberPartyId`**) access to a `SubscriptionResource` for a specific date range (**`fromDate`**, **`thruDate`**). It is typically associated with the `OrderItem` used to purchase the subscription and for convenience the `Product` that was purchased to create the subscription. In addition to, or as an alternative to, the date range the subscription can be limited by actual use time as opposed to calendar time (**`useTime`**) and/or by use count (**`useCountLimit`**).

To configure a `SubscriptionResource` to be accessible for a **`availableTime`**, **`useTime`**, and/or **`useCountLimit`** when a `Product` is purchased use the `ProductSubscriptionResource` entity.

Use the `SubscriptionDelivery` entity to keep track of `CommunicationEvent` instances that are related to the subscription, especially for delivery of digital subscription resources.

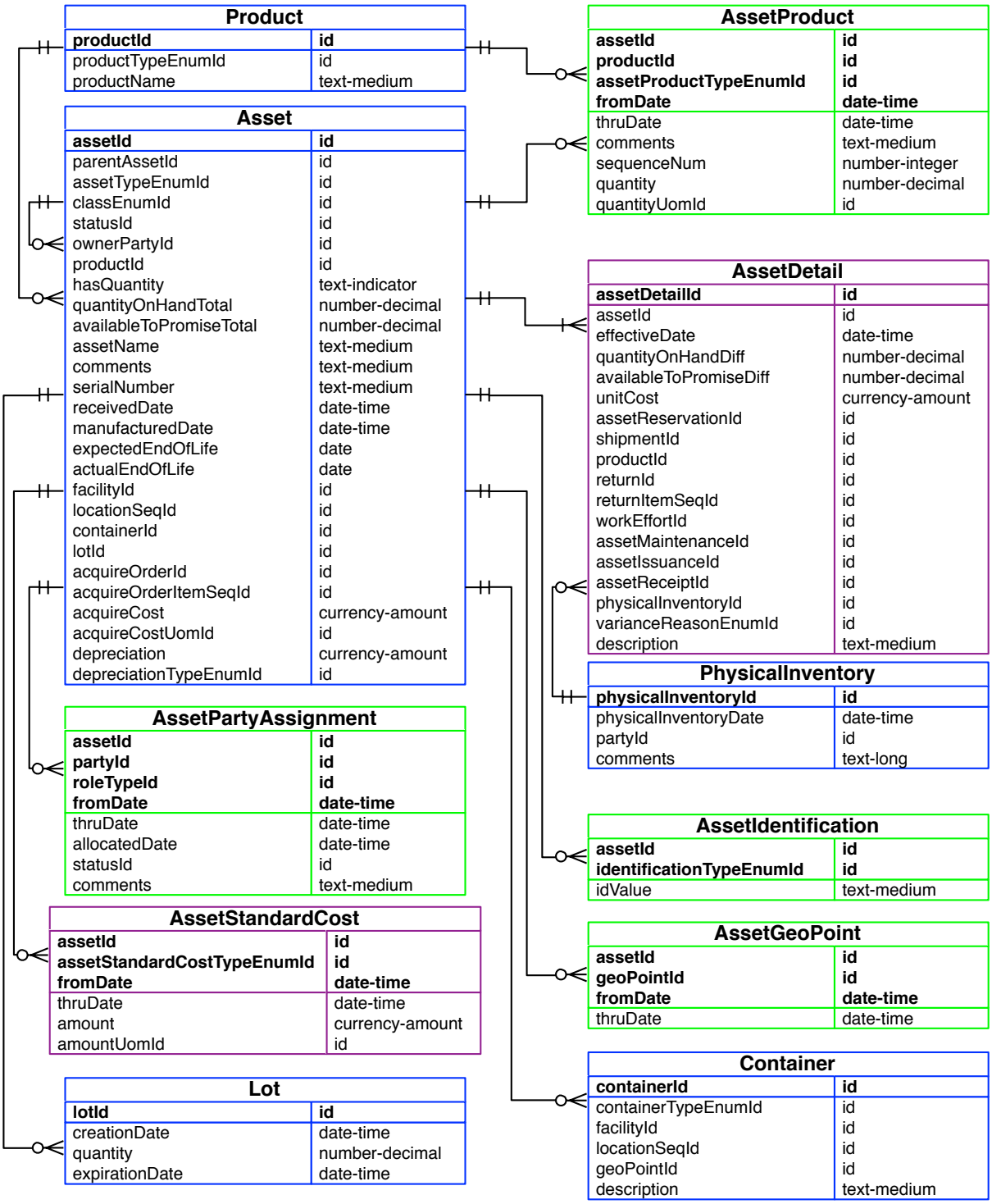
Asset - Asset (mantle.product.asset)

The `Asset` entity is used for inventory, equipment, and anything to be financially tracked as a fixed asset (**`assetTypeEnumId`**). Assets are identified by an **`assetId`**. An asset also has a class (**`classEnumId`**) such as forklift, tractor, laptop computer, or even software that can be used to categorize assets especially for manufacturing purposes to find the equipment needed for specific routes (manufacturing tasks). Add your own asset classes with `Enumeration` records of type `AssetClass`.

An `Asset` commonly represents an instance of a `Product`, or in other words the physical item that the `Product` record describes. The **`productId`** field specified which. An asset will also generally have an **`assetName`** and has a **`comments`** field to track general comments/notes.

Assets have a status (**`statusId`**) with various OOTB statuses for serialized inventory and equipment. A serialized inventory asset represents a single physical item and commonly has a **`serialNumber`**, hence the name.

Non-serialized inventory assets (**`hasQuantity`**=`Y`) represent more than a single quantity to handle simpler inventory needs where the items are all the same and don't need to be individually tracked. The current physical quantity on hand is maintained in the **`quantityOnHandTotal`** field, and the quantity that can be reserved or promised in the **`availableToPromiseTotal`**. Inventory of a product will usually consist of multiple `Asset` records such that all items represented by the record have the same **`receivedDate`**, **`lotId`**, **`facilityId`** and **`locationSeqId`** (for the `FacilityLocation` where the asset is stored), **`ownerPartyId`** (the `Party`, usually internal org for inventory, that owns it), and where applicable **`statusId`**. Typically as each batch of inventory is received and put away a new `Asset` record is created for it.



The quantity fields have the Total suffix because they are derived from the **`quantityOnHandDiff`** and **`availableToPromiseDiff`** fields on the `AssetDetail` entity. Each `AssetDetail` record represents some change to an Asset such as a reservation for a

placed order (**assetReservationId**), issuance on outgoing shipment (**assetIssuanceId**), receipt on incoming shipment (**assetReceiptId**), variance from physical inventory count (**physicalInventoryId**, **varianceReasonEnumId**), and production or consumption in a work effort such as a manufacturing route (**workEffortId**). If a Shipment is involved that is recorded in **shipmentId**, and all details should have their **effectiveDate** recorded.

When a physical inventory count is done it is tracked with a `PhysicalInventory` record, and the details for each inventory variance are recorded in `AssetDetail` records as described above.

An asset may have a number of dates recorded as applicable for the type of asset: **receivedDate**, **acquiredDate**, **manufacturedDate**, **expectedEndOfLife**, and **actualEndOfLife**. To track purchased assets and actual cost data `Asset` has **acquireOrderId**, **acquireOrderItemSeqId**, **acquireCost**, and **acquireCostUomId** fields. For fixed asset depreciation tracking, in adding to the corresponding `AcctgTrans` records, it has **depreciation**, **depreciationTypeEnumId**, and **salvageValue** fields.

Use `AssetGeoPoint` to record where an asset is, and a history of where it has been (with from/thru date fields). Use `AssetIdentification` to ID values for an asset such as a tracking label number, manufacturer serial number, VIN, etc. An `Asset` can be assigned to a `Party` using `AssetPartyAssignment` in a particular role and with an effective date range (**fromDate**, **thruDate**) for purposes such as use, management, maintenance, etc.

While an `Asset` is an instance of a `Product`, additional products may be associated with the asset to represent things such as rental or sale of the asset. Use the `AssetProduct` entity to keep track of these associated products.

While an inventory `Asset`, and sometimes other types of asset, are generally located in a `FacilityLocation` with the **facilityId** and **locationSeqId** fields it can also be located in a `Container` to more easily track movement of a set of assets that are in the container. In this case the **facilityId** and **locationSeqId** fields will be null and the `Asset.containerId` field will be populated. In that case the actual location will be found using the **facilityId**, **locationSeqId**, and **geoPointId** fields as applicable. These fields are audit logged to keep a history of their changes as a container moves.

Asset - Issuance (mantle.product.issuance)

Because competition for specific inventory items is common, such as when sales orders are placed for products with limited inventory, it is necessary to track reservations with `AssetReservation` records that are created when an item is promised. Later on when the physical item is fulfilled a `AssetIssuance` is created and the `AssetReservation` is deleted as it is no longer valid.

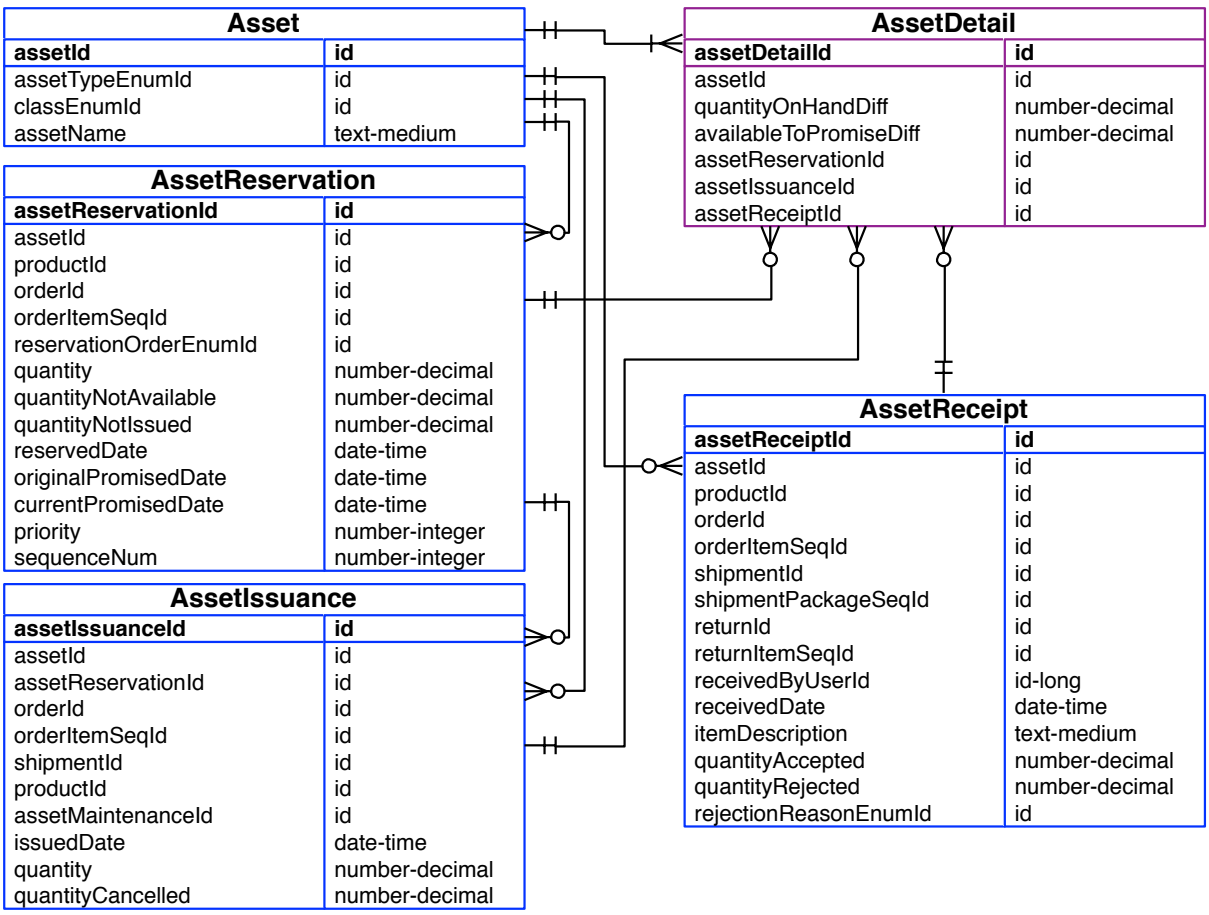
A reservation is associated with the `Asset` (**assetId**), `Product` (**productId**) for convenience, and `OrderItem` (**orderId**, **orderItemSeqId**). It has the **quantity** reserved, and for when the

reservation goes beyond on hand inventory the quantity reserved that was not available to promise is tracked with the **quantityNotAvailable** field.

An issuance is associated with the Asset (**assetId**), the AssetReservation (**assetReservationId**) if applicable, Product (**productId**) for convenience, and OrderItem (**orderId**, **orderItemSeqId**), and the Shipment (**shipmentId**) or AssetMaintenance (**assetMaintenanceId**) the asset is issued to. The issuance has a **issuedDate**, the **quantity** issued, and when applicable the **quantityCancelled** from the issuance.

The issuance may have parties associated with it in particular roles using AssetIssuanceParty.

When an AssetIssuance is created it triggers a general ledger accounting transaction to deduct it from the value of inventory on hand. This is part of the standard set of accounting transactions for inventory sales.



Asset - Receipt (mantle.product.receipt)

When an `Asset` is received, especially an inventory asset, that receipt is tracked with the `AssetReceipt`. For convenience the **`productId`** of the asset and an itemDescription are recorded on this. The receipt may be associated with an `OrderItem` (**`orderId`**, **`orderItemSeqId`**), `ShipmentPackage` (**`shipmentId`**, **`shipmentPackageSeqId`**), `ShipmentItem` (**`shipmentId`**, **`productId`**), and `ReturnItem` (**`returnId`**, **`returnItemSeqId`**).

There are fields to track the user who received the asset (**`receivedByUserId`**), the date it was received (**`receivedDate`**), the **`quantityAccepted`** and **`quantityRejected`**, and if there is a rejected quantity the reason for it (**`rejectionReasonEnumId`**).

When an `AssetReceipt` is created it triggers a general ledger accounting transaction to add it to the value of inventory on hand. This is part of the standard set of accounting transactions for inventory purchasing.

Asset - Maintenance (mantle.product.maintenance)

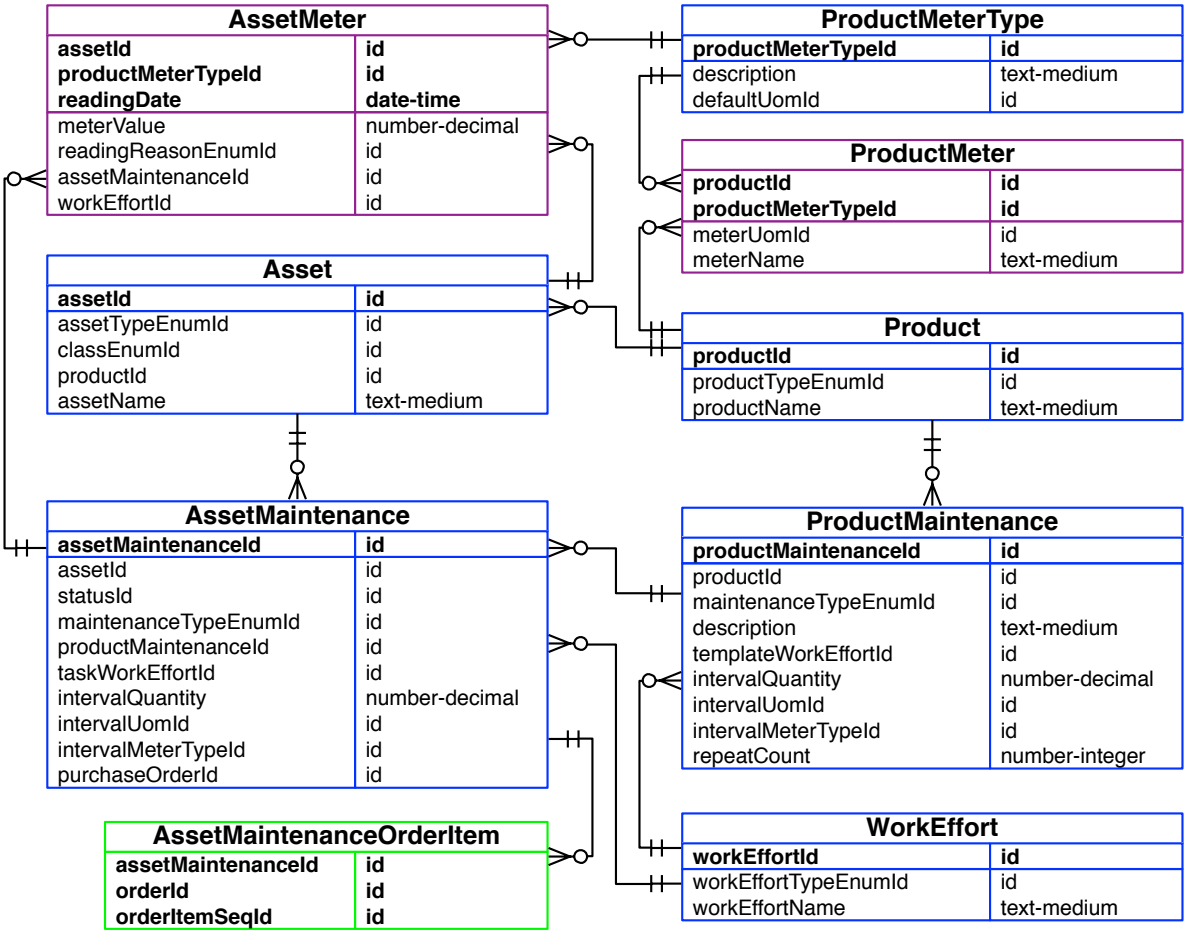
Following the pattern of `Asset` being an instance of a `Product`, the product describes the asset including the maintenance schedule associated with the product in the form of the `ProductMaintenance` entity. There are many types of maintenance, specified with the **`maintenanceTypeEnumId`** field, such as oil change and cleaning. You can add more by creating `Enumeration` records of type `MaintenanceType`.

The maintenance is to be done each **`intervalQuantity`** with the unit **`intervalUomId`**. The interval may be measured by a meter on the asset of `ProductMeterType` identified by **`intervalMeterTypeId`**. The meter should also be associated directly with the Product using `ProductMeter`. If a **`repeatCount`** is specified on the `ProductMaintenance` record the maintenance would be done only that many times.

The maintenance may be tracked with a `WorkEffort` and to simplify this a predefined work effort can be used as a template and copied from the maintenance schedule (`ProductMaintenance.`**`templateWorkEffortId`**) to the actual maintenance record (`AssetMaintenance.`**`taskWorkEffortId`**) where it would be assigned, the status updated, and so on.

For an actual maintenance effort for a particular Asset the `AssetMaintenance` entity has similar fields like the **`intervalQuantity`** value and related fields when the maintenance is actually performed. This has a status (**`statusId`**) to track planning and completion of the maintenance. It also has a **`purchaseOrderId`** for when the work is hired out to track the corresponding order. The maintenance may be purchased or sold and the relevant order item or items are tracked with `AssetMaintenanceOrderItem`.

It is typical to read meters on the asset when maintenance is done, and the meter values for each meter associated with the product (`ProductMeter`) are recorded with `AssetMeter`. Other meter readings may be done outside the context of maintenance and also recorded



with `AssetMeter`. This may be done when fueling, at route waypoints, before/after production tasks, etc (**`readingReasonEnumId`**) and these records are often very important for financial management and tax liability.

Use `AssetRegistration` to record details when an `Asset` is registered with a government authority (**`govAgencyPartyId`**). These may include a **`licenseNumber`** and **`registrationNumber`**. The registration will happen on a certain date (**`registrationDate`**) and be valid within a date range (**`fromDate`**, **`thruDate`**).

Store (mantle.product.store)

For sales order processing on an eCommerce site or in a POS (point of sale) system we need a way to keep track of all of the relevant settings. The `ProductStore` and related entities are used for this.

A store has a name (**`storeName`**) and is owned/run by an internal organization (**`organizationPartyId`**). While a store may support various languages and currencies each

store is typically best focused on a single country/area with a single language (**defaultLocale**) and currency (**defaultCurrencyUomId**).

including:

- **Products available**: The ProductStoreCategory entity associates ProductCategory records with a store for browse root, default search, purchase allow, etc and the products in those categories or their sub-categories make up the products available in the store.
- **Notification emails**: Use ProductStoreEmail to associated Moqui EmailTemplate records with the store for notification emails such as registration, order confirmation, order change, return completion, password update, and so on.
- **Inventory reservation**: The most common case is to have a single inventory Facility for a store, and this is specified in the ProductStore.**inventoryFacilityId** field. When more than one is needed use the ProductStoreFacility entity. Inventory is reserved in the order specified with the ProductStore.**reservationOrderEnumId**, such as FIFO or LIFO by received date or expiration date. For automatic replenishment requirements set the ProductStore.**requirementMethodEnumId** field.
- **Payment processing**: Use ProductStorePaymentGateway to configure the PaymentGatewayConfig to use for each **paymentMethodTypeEnumId**.
- **Shipping options and rate calculation**: The ProductStoreShippingGateway entity is used to configure the ShippingGatewayConfig to use for each **carrierPartyId**.
- **Tax calculation**: The ProductStore.**taxGatewayConfigId** points to the TaxGatewayConfig record to use for this store for sales/VAT tax calculation.

The ProductStoreParty entity is used for general needs to associate parties in a particular role with a store. One of many uses for this is if ProductStore.**requireCustomerRole** is set to Y then only parties associated with the store in the Customer role can access the store.

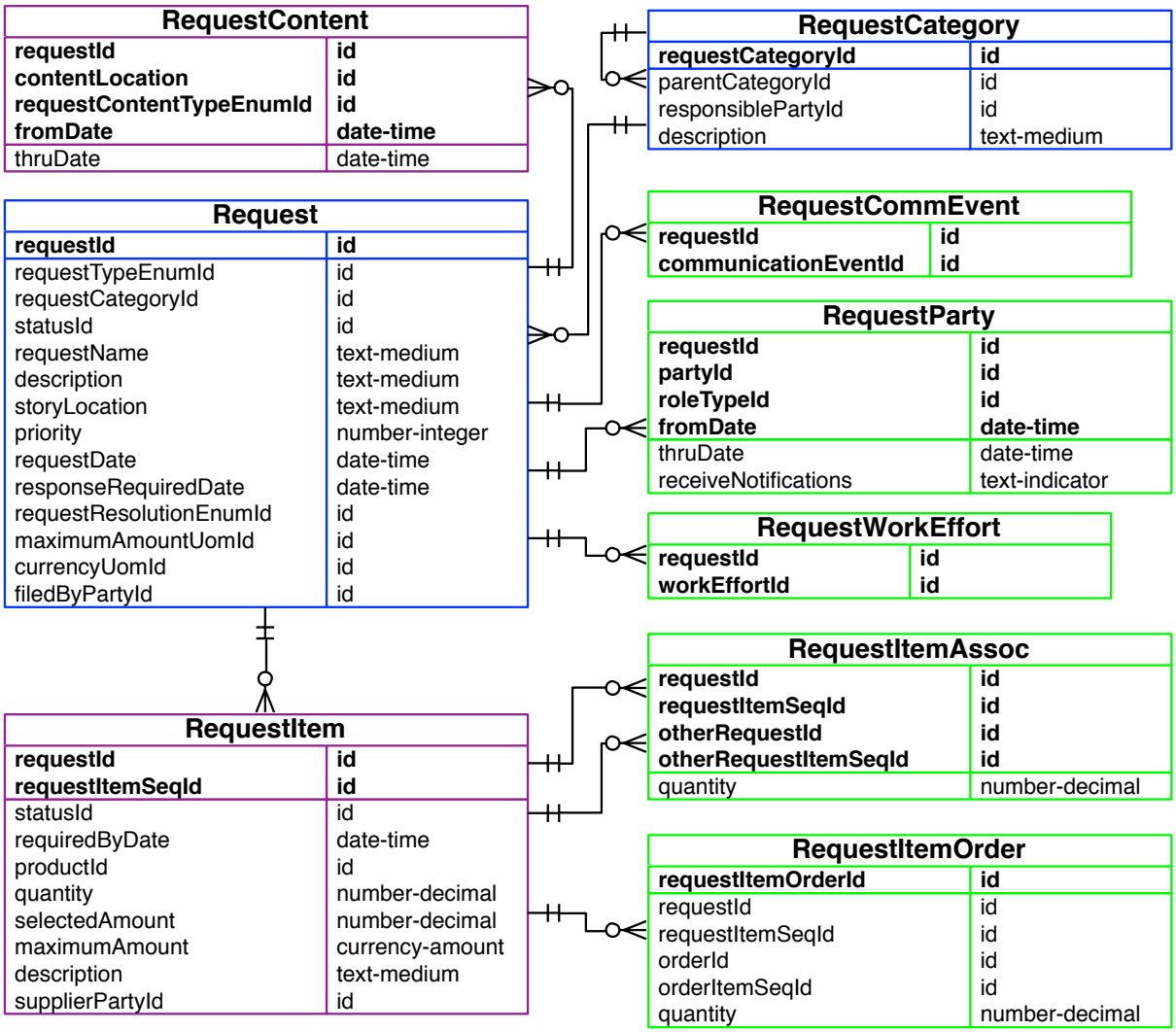
When managing a large number of stores or to automate based on specific sets of stores use the ProductStoreGroup entity to represent a group of stores, and ProductStoreGroupMember entity to associate stores with the group. A store can be associated with multiple groups. Use the ProductStoreGroupParty entity to associate parties with the group.

Request

Request (mantle.request)

A Request can be from a party (**filedByPartyId**) inside an organization such as an employee for things like inventory or general purchases, or outside an organization such as a client or customer for things like a quote, proposal, or in the software world for things like a bug fix or new feature. These are specified in the **requestTypeEnumId** field and while there are a few general ones defined OOTB you may want to define others by adding Enumeration records of type RequestType.

The default Request statuses (**statusId**) include Draft, Submitted, Reviewed, In Progress, Completed, and Cancelled. It also has a resolution (**requestResolutionEnumId**) that is by default Unresolved and default options include Fixed, Can't Reproduce, Won't Fix, Duplicate, Rejected, and Insufficient Information. Additional resolutions can be added as Enumeration records of type RequestResolution. If the result should be sent where to send it is specified with the **fulfillContactMechId** field.



A Request has a name (**requestName**), **description**, and if there is a story with additional details in Resource Facade content it is referred to with the **storyLocation** field. To help determine the order to work on requests and for general information it has **priority**, **requestDate**, and **responseRequiredDate** fields. A request may be associated with a Facility (**facilityId**) and ProductStore (**productStoreId**).

The details for a request are in its RequestItem records. An item can have its own **statusId** (using the same statuses as a request) and **requiredByDate** and typically has a **description**. If the request is for Product use the **productId**, **quantity**, and (if applicable) **selectedAmount** fields to specify details.

For quotes and other similar types of requests where there is a maximum amount/price to pay for the item, specify it in the **maximumAmount** field on the item. The unit for this amount is on the Request record in the **maximumAmountUomId** field. These types of requests also typically result in an order and the RequestItem is associated with an OrderItem using the RequestItemOrder entity.

For manual organization of requests use RequestCategory to specify hierarchical (with **parentCategoryId**) request categories associated with requests using the Request.**requestCategoryId** field.

A request may be associated with CommunicationEvent for communication related to the request (RequestCommEvent), Resource Facade content for additional content or documents (RequestContent), Party for parties working on or otherwise related to the request (RequestParty), and WorkEffort for tasks and other efforts related to handling the request (RequestWorkEffort). A request may also have notes (RequestNote).

As an example a Request may be created for a software bug fix. The request is assigned to someone with a RequestParty record. That person creates a task (WorkEffort) which is associated with the request using a RequestWorkEffort record. That task may be assigned to the same person or someone else, or even a group. Once the task is done its status is updated as is the status on the request.

Requirement (mantle.request.requirement)

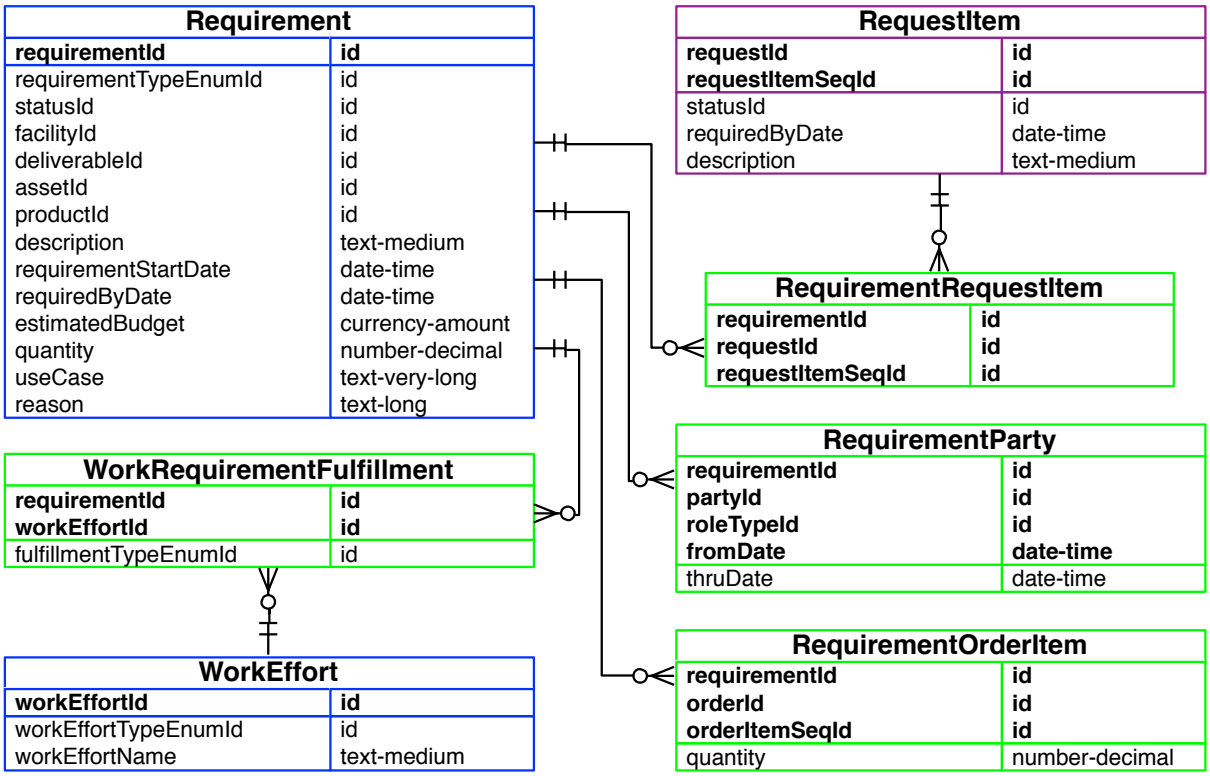
A Requirement may be for work, inventory, general customer or internal requirements, etc (**requirementTypeEnumId**). Add your own types with Enumeration records of type RequirementType. Its statuses (**statusId**) include Proposed, Created, Approved, Ordered, and Rejected. Inventory requirements and other types as applicable may be for a specific Facility (**facilityId**), and Product (**quantity**).

A requirement will typically have a **requirementStartDate** and a **requiredByDate**. To describe the requirement in detail, especially for software requirements, the **useCase** and **reason** fields are there for you. Parties may be associated with the requirement using the RequirementParty entity.

For automatic inventory replenishment inventory requirements can be created based on the ProductStore **requirementMethodEnumId** setting. Common options include creating a requirement based on every order, when ATP or QOH fall below the level configured on the relevant ProductFacility record, or for drop-ship third party ordering purposes. After requirements are created they can be summarized by Product and Facility then after a

supplier is selected an order with the total quantity can be created and associated with the `RequirementOrderItem` entity.

Work requirements follow a different path. They may have an order associated with them for the labor, but more commonly result in a specific `RequestItem` (associated with `RequirementRequestItem`) or directly to a `WorkEffort` (with `WorkRequirementFulfillment`). The work effort can be for Implements, Fixes, Deploys, Tests, or Delivers (**`fulfillmentTypeEnumId`**).



The `Requirement` entity has a simple **`estimatedBudget`** field, and for more complex budgeting requirements or to include it in a larger budget plan it can be associated with a `BudgetItem` using the `RequirementBudgetAllocation` entity.

Sales

Opportunity (mantle.sales.opportunity)

As part of sales force automation (SFA) use the `SalesOpportunity` to keep track of opportunities. An opportunity is typically associated with a certain sales stage (`SalesOpportunityStage`), and you can define any series of stages desired.

There may be many parties associated with an opportunity including the customer/prospect, sales representative, manager, etc. Record these with the `SalesOpportunityParty` entity. You could use this for competitors as well, but generally there is additional information for competitors so use the `SalesOpportunityCompetitor` entity for them.

An opportunity will often be associated with a quote, which may turn into an order. Use `SalesOpportunityQuote` to keep track of these. There may be meetings, other calendar events, or tasks associated with an opportunity and use `SalesOpportunityWorkEffort` to associate it with those.

There are a couple of touch points to marketing records. One is to a `MarketingCampaign` using `SalesOpportunity.marketingCampaignId`. Another is marketing `TrackingCode` records which are associated using the `SalesOpportunityTracking` entity. See the **Marketing** section for more details about these.

Forecast (mantle.sales.forecast)

A `SalesForecast` may be for an entire internal organization (**`organizationPartyId`**) or a specific `Party` within that `Organization` (**`internalPartyId`**). It is associated with a TimePeriod and has amount fields including **`quotaAmount`**, **`forecastAmount`**, **`bestCaseAmount`**, and **`closedAmount`** for the final result.

Details about actual `Product` sold are recorded in `SalesForecastDetail` with a record with the sales **`amount`** and **`quantity`** sold for each `Product` and/or `ProductCategory`.

Need (mantle.sales.need)

To record when a customer or other `Party` needs product (could be internal or external) use the `PartyNeed` entity. It can be for a `Product` and/or `ProductCategory` for needs that may be met by a variety or products, or when the exact product needed is not yet known. It often comes from a `CommunicationEvent` or through a web app with a `Visit` so there are fields for both.

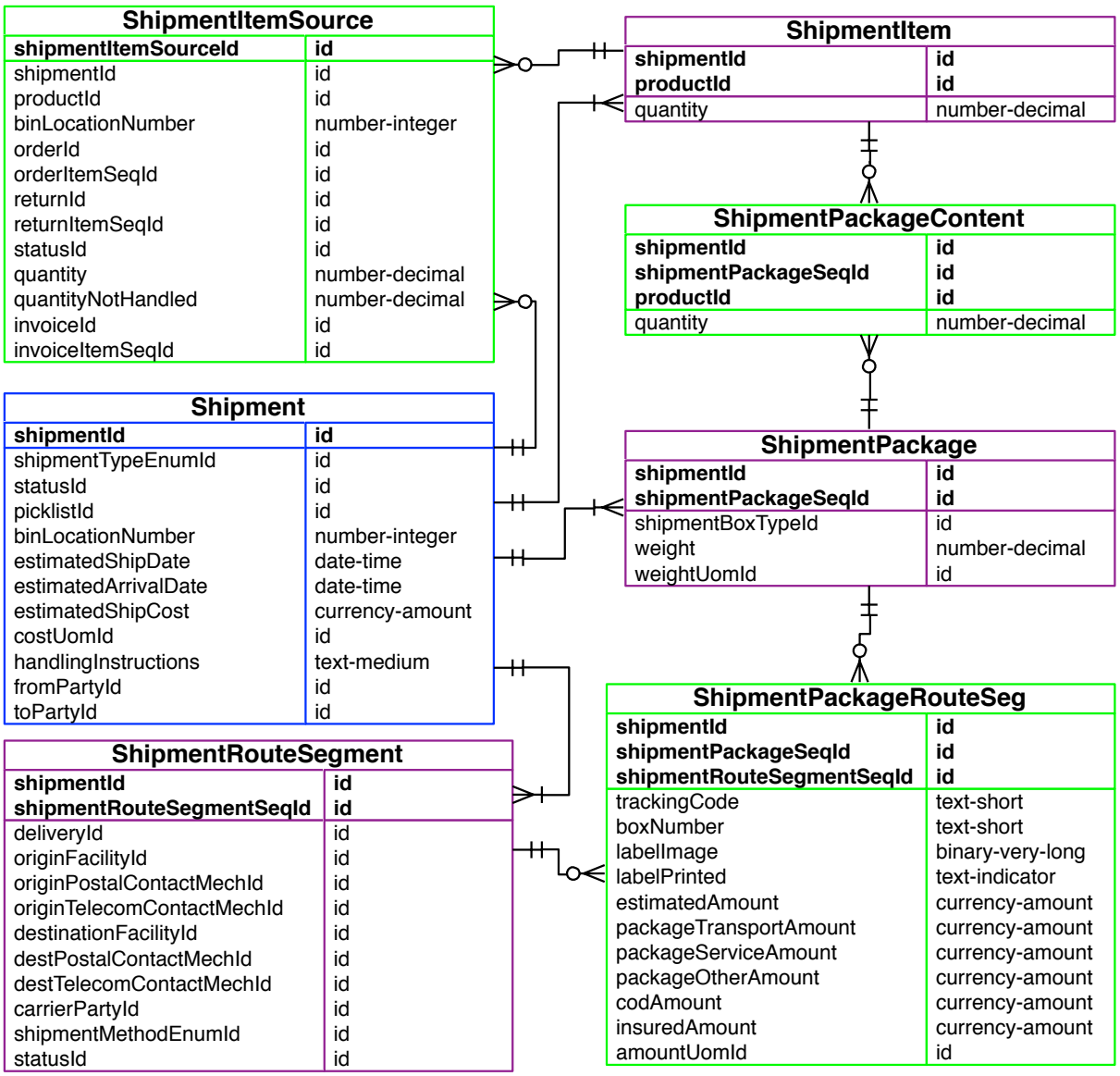
Shipment

Shipment (mantle.shipment)

The `Shipment` and related entities may be used for both Incoming and Outgoing shipments (**`shipmentTypeEnumId`**), and more specifically for Sales Return, Sales Shipment, Purchase Shipment, Purchase Return, Drop Shipment, and Transfer shipments. A `Shipment` is generally from one `Party` (**`fromPartyId`**) and to another (**`toPartyId`**). If needed put special instructions in the **`handlingInstructions`** field.

For planning purposes a shipment may have **estimatedReadyDate**, **estimatedShipDate**, **estimatedArrivalDate**, and **latestCancelDate** values. For further detail or to get the shipment in a calendar as an event use the **shipWorkEffortId** and **arrivalWorkEffortId** fields to point for `WorkEffort` records. There is typically some sort of estimated cost for the shipment, track that in **estimatedShipCost** with its currency in **costUomId**. If the cost is adjusted use the **addtlShippingCharge** field along with a description of the additional charge in **addtlShippingChargeDesc**.

For the entire `Shipment` there is a **statusId** that may be Input, Scheduled, Picked, Packed, Shipped, Delivered, and Cancelled. This field is audit logged for a status history. The **Packed** status is one of the more important as it is the point where the shipment is generally



considered fulfilled for billing purposes. The change to the Packed status is used to trigger Invoice creation for the order(s) on the shipment, and if applicable automated payment processing.

Each shipment has `ShipmentItem` records with a **`quantity`** for each `Product` (**`productId`**) in the shipment.

A `Shipment` always has one or more packages (`ShipmentPackage`) and the **`quantity`** of **`productId`** in each package is recorded with `ShipmentPackageContent`. Each package may have the box used (**`shipmentBoxTypeId`** pointing to a `ShipmentBoxType` record), and the total shipping **`weight`** of the package along with the unit for the weight (**`weightUomId`**).

A `Shipment` also always has one or more route segments (`ShipmentRouteSegment`). Consumer fulfillment and most simple shipments involve a single route segment with a carrier (**`carrierPartyId`**) and shipment method (**`shipmentMethodEnumId`**) going from a certain origin (**`originPostalContactMechId`**, **`originTelecomContactMechId`**) to a destination (**`destPostalContactMechId`**, **`destTelecomContactMechId`**). A shipment may also have other contact information associated with it using the `ShipmentContactMech` entity.

For consumer fulfillment the origin will usually be a warehouse Facility and specified with **`originFacilityId`**. For consumer returns or inventory purchase shipments they will generally go to a Facility, recorded in **`destinationFacilityId`**. There are various dates associated with a route segment including **`estimatedStartDate`**, **`estimatedArrivalDate`**, **`actualStartDate`** and **`actualArrivalDate`**.

Each package will have certain details for each route segment (`ShipmentPackageRouteSeg`) including **`trackingCode`**, **`boxNumber`** (within the shipment, if applicable), and labels/documents including **`labelImage`**, **`labelIntlSignImage`**, **`labelHtml`**, **`labelPrinted`**, and **`internationalInvoice`**.

For billing purposes each package in a route segment (`ShipmentPackageRouteSeg`) has an **`estimatedAmount`** for the estimate before getting a quote or actuals from the carrier, plus **`packageTransportAmount`**, **`packageServiceAmount`**, and **`packageOtherAmount`** for actuals from the carrier, along with **`codAmount`** and **`insuredAmount`** for those special situations. All of these use the currency specified in **`amountUomId`**.

For all packages in a route segment (i.e., on `ShipmentRouteSegment`) there are fields for the totals in **`actualTransportCost`**, **`actualServiceCost`**, **`actualOtherCost`**, and **`actualCost`** with the currency in **`costUomId`**. The route segment also has a total **`billingWeight`** with **`billingWeightUomId`** that includes the billing weight used from all packages for the route segment. A route segment also has a status (**`statusId`**) that is mostly used for keeping track of communication (usually by integration) with the carrier, including: Not Started, Confirmed, Accepted, and Voided.

A `Shipment` is generally based on one or more orders or returns, and generally results in one or more invoices being produced. The `ShipmentItemSource` entity is used to keep track of

these, and there may be more than one ShipmentItemSource for each ShipmentItem record. More specifically a shipment item may be associated with multiple order items (**orderId**, **orderItemSeqId**) or return items (**returnId**, **returnItemSeqId**) and is generally associated with one or more invoice items (**invoiceId**, **invoiceItemSeqId**).

There is a ShipmentItemSource.**quantity** field to specify how much of the ShipmentItem.**quantity** comes from the specified order or return item. There is also a quantityNotHandled field on the source to specify how much of the quantity should have been shipped but was not.

Shipment has **picklistId** and **binLocationNumber** fields, and ShipmentItemSource has **binLocationNumber** and **statusId** fields to use for picking and packing in a warehouse. See the **Picklist (mantle.shipment.picklist)** section below for details.

Carrier (mantle.shipment.carrier)

A carrier is typically a company like UPS or FedEx. Use CarrierShipmentMethod to configure which carriers (**carrierPartyId**) support which shipment methods (**shipmentMethodEnumId**) and the carrier's service code (**carrierServiceCode**) and Standard Carrier Alpha code (**scaCode**) for the method.

Similarly CarrierShipmentBoxType is used to configure the ShipmentBoxType (by **shipmentBoxTypeId**) records for a carrier and their corresponding **packagingTypeCode** and if applicable **oversizeCode**. The method and box codes are all typically used for carrier integrations to specify the service level and boxes using codes that the carrier supports.

If a Party has an account with a carrier track that using the PartyCarrierAccount entity.

The ShippingGatewayConfig entity is used to specify details for an integration with a carrier for purposes of shipping estimates, rate quotes, getting labels, voiding labels, tracking packages, and even validating addresses. To implement a shipping gateway (carrier integration) implement services for each of these and create a record that points to them, then associate that with a ProductStore using the ProductStoreShippingGateway entity.

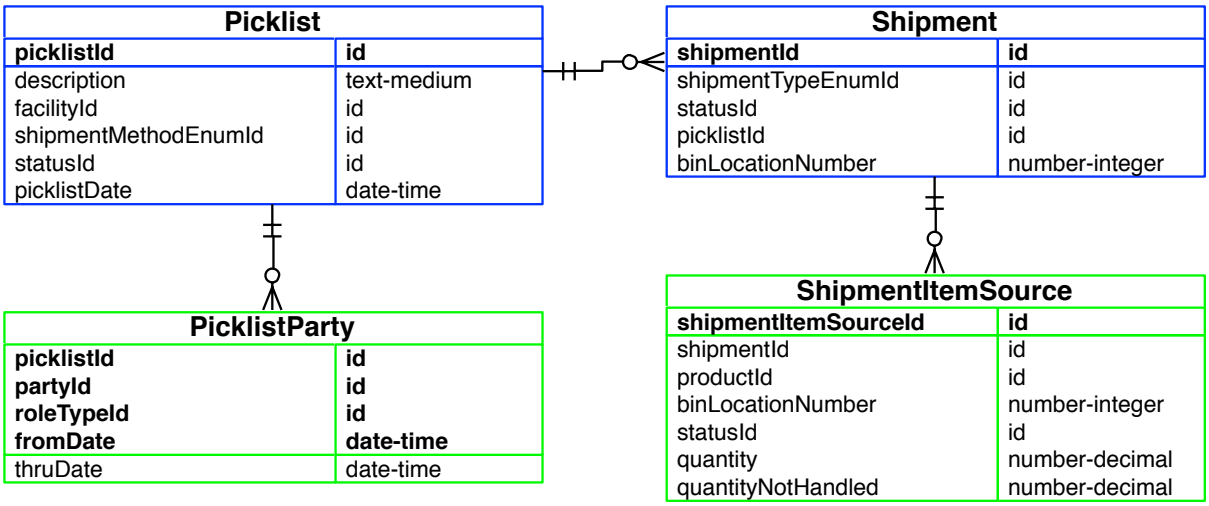
Picklist (mantle.shipment.picklist)

A Picklist is used to organize pending Shipment records for a pick/pack process. There is no separate picklist bin structure, instead Shipment itself is used. Similarly there is no picklist item, the ShipmentItemSource is used to track items in a pick "bin" and details about the order, return, and invoice that the particular quantity of the item are associated with.

For a Shipment in the Input or Scheduled statuses the Shipment.**picklistId** points to the Picklist it is on. A picklist is always associated with a Facility (**facilityId**) and may be associated with a particular shipment method (**shipmentMethodEnumId**) for planning and processing fulfillment by shipment method. For management and historical tracking

purposes `Picklist` has a date/time it was planned in the **`picklistDate`** field. Parties in a particular role such as Picker, Packer, Manager, etc may be associated with the picklist using `PicklistParty`.

In a typical picking process multiple shipments are picked at the same time, with the contents put into a bin. This is tracked with the `Shipment`**`.binLocationNumber`** field unless the shipment is split into multiple bins (like one bin per order on the shipment) and then the `ShipmentItemSource`**`.binLocationNumber`** field is used to override the one on the `Shipment` record.



The `ShipmentItemSource` entity has the `OrderItem` details (**`orderId`**, **`orderItemSeqId`**) to lookup related `AssetReservation` records that have the **`quantity`** (or **`quantityNotIssued`** if used) to pick and the corresponding `Asset` to find the `FacilityLocation` that the asset is stored in for picking.

`ShipmentItemSource` has a status (**`statusId`**) for picking and packing purposes that can be Pending, Picked, Packed, Received, or Cancelled. Note that the Received status goes beyond the typical pick/pack process to track receipt of items when that data is available and needed.

A typical pick sheet will have a list of all facility locations to pick from listed in order of their location for easy walking of the floor to pick all shipments on the list. For each location the product and quantity to pick are listed along with the pick bin number and the quantity for that bin to get the right number of items in the bin for the right shipment (or order). The series of entities above is used to get all of those details.

Work Effort

Work Effort (mantle.work.effort)

The most basic types of `WorkEffort` task and calendar event. More generally `WorkEffort` is used for projects, milestones, tasks, manufacturing routing, meetings, calls, travel, and even time off and work availability.

These are specified with the type (**`workEffortTypeEnumId`**) and purpose (**`purposeEnumId`**). Types have more automation around them and are more limited, currently including Project, Milestone, Task, Event, Available, and Time Off. The purposes are more flexible, there is a much larger set, and you can add more with `Enumeration` records of type `WorkEffortPurpose`.

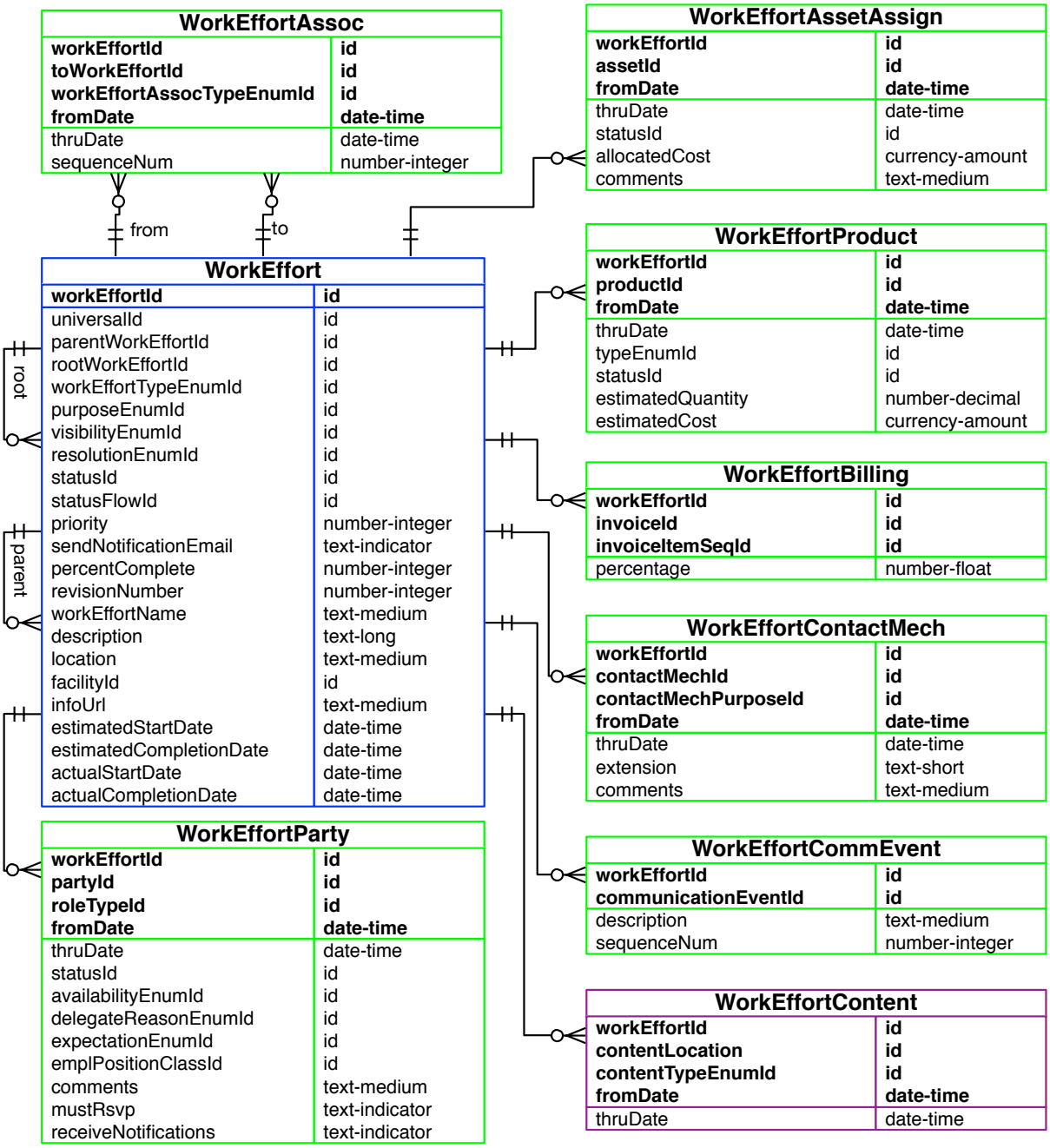
Work efforts are hierarchical with the **`rootWorkEffortId`** to identify the root (such as a project) and **`parentWorkEffortId`** for the immediate parent in the hierarchy. For example with a Project type WorkEffort as the root the top-level tasks are Task type WorkEffort records with the **`rootWorkEffortId`** pointing to the project and no **`parentWorkEffortId`**. Sub-tasks have the same **`rootWorkEffortId`** value and their **`parentWorkEffortId`** field points to the top-level task.

`WorkEffort` has all the basic fields needed for a task or event including name (**`workEffortName`**), **`description`**, **`location`**, **`infoUrl`**, **`estimatedStartDate`**, **`estimatedCompletionDate`**, **`percentComplete`**, and **`priority`**. For iCal files and similar uses the **`workEffortId`** isn't generally a universally unique identifier so there is a **`universalId`** field for that. For historical tracking it also has **`actualStartDate`** and **`actualCompletionDate`** fields.

A work effort may take place in an office, warehouse, or other type of `Facility` and that is tracked with the **`facilityId`** field. For additional location and contact information use the `WorkEffortContactMech` entity to associate contact mechs such a postal addresses, telephone numbers (for conference calls, etc), email addresses, and so on. To keep track of actual communication related to a work effort use the `WorkEffortCommEvent` entity and associated `CommunicationEvent` records.

A `WorkEffort` may be internal, sensitive, or totally public and this is specified with **`visibilityEnumId`**. The OOTB options for it are General (public access), Work Group (group only access), Restricted (private access), and Top Secret (confidential access).

For some types of efforts such as manufacturing tasks more detailed time allowances and tracking are needed. There are a few decimal number fields for this: **`estimatedWorkTime`**, **`estimatedSetupTime`**, **`remainingWorkTime`**, **`actualWorkTime`**, **`actualSetupTime`**, and **`totalTimeAllowed`**. The time unit for these fields is specified in the **`timeUomId`** field.



`WorkEffort` status (**`statusId`**) options include: In Planning, Approved/Scheduled, In Progress, Complete, Closed, On Hold and Cancelled. These are the statuses for the `Default StatusFlow`. To use a different `StatusFlow` use the **`statusFlowId`** field on either a particular `WorkEffort` or (depending on implementation) its root `WorkEffort` pointed to with **`rootWorkEffortId`**.

In addition to status WorkEffort has a resolution (**resolutionEnumId**). OOTB options include Unresolved (default), Completed, Incomplete, Won't Complete, Duplicate, Cannot Reproduce, and Insufficient Information. Additional resolutions can be added with Enumeration records of type WorkEffortResolution.

In addition to the hierarchical structure of work efforts they may be associated with others using the WorkEffortAssoc entity with types such as Depends On, Duplicates, Caused By, Independent Of (Concurrent), Routing Component, and Milestone. Note that milestones are associated with tasks through an association and are not as a parent WorkEffort. This is because a task may be associated with multiple milestones over time so we have a history and forward planning options. Additional association types can be added with Enumeration records of type WorkEffortAssocType.

For equipment or other types of Asset used (but not consumed) for a work effort use the WorkEffortAssetAssign entity. Asset records assigned this way are generally considered busy (otherwise unavailable) for the duration of the WorkEffort. To plan for a type of asset needed by the Product (**assetProductId**) that represents a type of asset, use the WorkEffortAssetNeeded entity. Product records may be associated with a WorkEffort for other reasons using WorkEffortProduct. Assets such as materials and supplies that are used (consumed) for a work effort are tracked with WorkEffortAssetUsed and asset produced by the work effort with WorkEffortAssetProduced.

Sometimes it is useful for organize work efforts by a more general Deliverable. Associate work efforts with it using WorkEffortDeliverableProd.

Use WorkEffortSkillStandard to record the skills (Enumeration of type SkillType from the HR/humanres entities) needed for a WorkEffort, usually as part of selection of parties to assign to the effort.

There are various reasons to associate a Party with a WorkEffort, and the party's involvement with the work effort (just as a party's association with other entities) is determined by the role (**roleTypeId**). This may be Manager, Worker, Operator, or any other role (including Not Applicable). For billing reasons a EmplPositionClass may be specified on the WorkEffortParty with the emplPositionClassId field.

Each Party association with a WorkEffort has a status (**statusId**; Offered, Assigned, Declined, Unassigned), availability (**availabilityEnumId**; Available, Busy, Away), expectation (**expectationEnumId**; For Your Information, Involvement Required, Involvement Requested, Immediate Response Requested) and in the case of delegation a reason for it (**delegateReasonEnumId**; Need Support or Help, My Part Finished, Completely Finished).

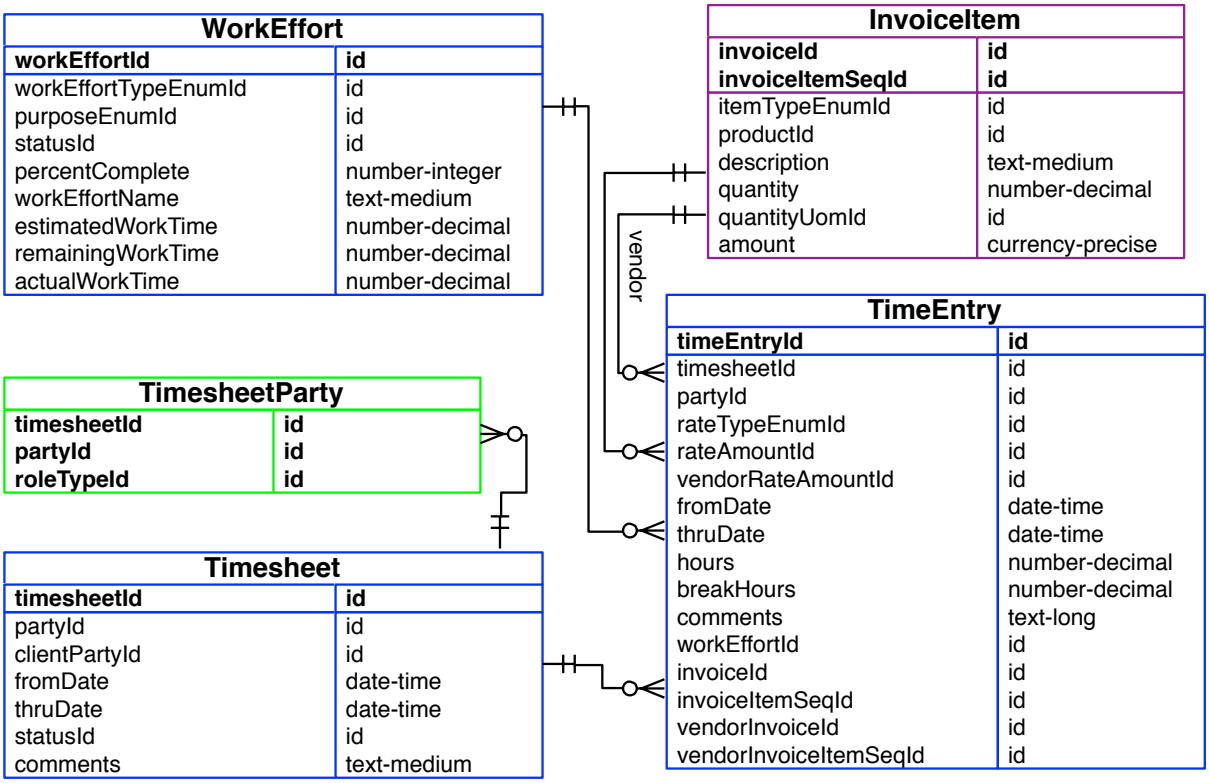
To associated a higher-level WorkEffort (such as a Project) with an Invoice using the WorkEffortInvoice entity. For more detail billing of particular tasks or other lower-level work efforts, or even a percentage of one, use the WorkEffortBilling entity.

General Resource Facade content and documents may be associated with a `WorkEffort` using the `WorkEffortContent`. Notes may be recorded for an effort using `WorkEffortNote`.

Time Entry (mantle.work.time)

Use the `TimeEntry` entity to record the time worked (**`hours`**) on a task or other type of `WorkEffort` (by **`workEffortId`**) by a particular `Party` (**`partyId`**). The working time falls between the **`fromDate`** and **`thruDate`**, and if any time within that range was not spent working it can be recorded in **`breakHours`**. Generally **`hours`** + **`breakHours`**, if both specified, should match the time duration between **`fromDate`** and **`thruDate`**.

For billing purposes a `RateType` will generally be specified in **`rateTypeId`**. Common types include Standard, Discounted, Overtime, and On-site Work. This is used to lookup a `RateAmount` record along with other data applicable (may include **`partyId`**, **`workEffortId`**, **`emplPositionClassId`**, and **`ratePurposeEnumId`** as Client or Vendor). This may be done twice, once for the Client rate (client pays to vendor) and once for the Vendor rate (vendor pays to worker) and recorded in **`rateAmountId`** and **`vendorRateAmountId`**.



Once a `TimeEntry` is billed the relevant `InvoiceItem` is referenced with the **`invoiceId`** and **`invoiceItemSeqId`** fields for the invoice from vendor to client, and with the

vendorInvoiceId and **vendorInvoiceItemSeqId** fields for the invoice from worker to vendor. When these are populated it means the time entry has been billed.

A Timesheet may be used to organize TimeEntry records, or to make time entry easier. There are generally two parties associated with a timesheet, the worker Party (**partyId**) and the client Party (**clientPartyId**). Other parties may be associated with it using TimesheetParty.

A Timesheet is generally used for just a specific date range (**fromDate**, **thruDate**). During its lifecycle a timesheet has a status (**statusId**) which is typically In-Process (work being done, time being recorded), Completed (all relevant work done and time recorded), or Approved (approved for billing).

USL Business Processes

This section contains overviews of the main high-level business processes supported in Mantle. This is an introduction to the business process concepts and the specific services and entities involved with each process. There are other services and entities not covered here, or in other words this is not a complete reference of all services and options available. This will give you a good idea of the general functionality that exists and how it is structured, and from there you can easily review the source or references to find related artifacts.

Mantle Business Artifacts has a wide variety of functionality, including the **procure to pay**, **order to cash**, and **work plan to cash** processes, with:

- Purchase and Sales Orders (for goods, services, materials, etc; POs for inventory and equipment/supplies/etc)
- Project, Task, and Request management with time and expense recording, billable/payable rates by project/task/client/worker/etc
- Incoming and Outgoing Invoices with a wide variety of item types and an XSL:FO template for print or email
- Automatic invoice generation for purchase orders (AP), sales orders (AR), project client time and expenses (AR), project vendor/worker time and expenses (AP)
- Payments, both manually recorded and automatic through payment processing interfaces; applying payments to invoices
- Fulfillment of sales orders (including basic picking and packing) and receiving of purchase orders
- Inventory management including issuance and receipt, and inventory reservation for sales orders
- Automated GL account posting of incoming and outgoing invoices, outgoing and incoming payments, payment application, and inventory receipt and issuance
- General GL functionality for time periods, validation of transactions to post, time period closing
- Balance Sheet and Income Statement reports (and basic posted amounts and account balance by time period summaries)
- Drools rules for product pricing, shipping charge calculation, and tax calculation

Procure to Pay

The Spock test suite for this process is in the `OrderProcureToPayBasicFlow.groovy` file.

Some of the more relevant setup data is shown in the examples below but you can find the rest of it in the `ZzaGlAccountsDemoData.xml`, `ZzbOrganizationDemoData.xml`, and `ZzcProductDemoData.xml` files.

Supplier Product Pricing

Here are some test calls to get pricing for the `DEMO_1_1` product from the external supplier (vendor) `Party` `MiddlemanInc` (**`vendorPartyId`**) for the internal organization `ORG_BIZI_RETAIL` (**`customerPartyId`**) with quantities of 1 and 100 to test quantity breaks:

```
String vendorPartyId = 'MiddlemanInc', customerPartyId = 'ORG_BIZI_RETAIL'
String priceUomId = 'USD', currencyUomId = 'USD'
String facilityId = 'ORG_BIZI_RETAIL_WH'

Map priceMap = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId, quantity:1,
        vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()
Map priceMap2 = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId, quantity:100,
        vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()
```

Here is the demo `Product` record and the demo `ProductPrice` records used to configure these supplier prices:

```
<mantle.product.Product productId="DEMO_1_1"
    productTypeEnumId="PtFinishedGood" chargeShipping="Y"
    returnable="Y" productName="Demo Product One-One" description=""/>
<mantle.product.ProductPrice productPriceId="DEMO_1_1_CS1"
    productId="DEMO_1_1" vendorPartyId="MiddlemanInc"
    pricePurposeEnumId="PppPurchase" priceTypeEnumId="PptCurrent"
    fromDate="2010-02-03 00:00:00" minQuantity="1" price="9.00"
    priceUomId="USD"/>
<mantle.product.ProductPrice productPriceId="DEMO_1_1_CS100"
    productId="DEMO_1_1" vendorPartyId="MiddlemanInc"
    pricePurposeEnumId="PppPurchase" priceTypeEnumId="PptCurrent"
    fromDate="2010-02-03 00:00:00" minQuantity="100" price="8.00"
    priceUomId="USD"/>
```

The results are validated like this, note the 9.00 for quantity of 1 and 8.00 for quantity of 100:

```
priceMap.price == 9.00
priceMap2.price == 8.00
priceMap.priceUomId == 'USD'
```

Place and Approve Purchase Order

For purchase orders there is no ProductStore, so we have no payment, shipping, party, and other settings to use from configuration. In this create#Order call we explicitly set the customer and vendor. Here is a code snippet with service calls to create the order, add product items to the order, add a shipping charge item to the order, set billing and shipping info for the order, place the order, and then approve the order.

```
Map orderOut = ec.service.sync()
    .name("mantle.order.OrderServices.create#Order")
    .parameters([customerPartyId:customerPartyId,
        vendorPartyId:vendorPartyId, currencyUomId:currencyUomId]).call()

purchaseOrderId = orderOut.orderId
orderPartSeqId = orderOut.orderPartSeqId

ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_1_1', quantity:150,
        itemTypeEnumId:'ItemProduct']).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_3_1', quantity:100,
        itemTypeEnumId:'ItemProduct']).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        productId:'EQUIP_1', quantity:1,
        itemTypeEnumId:'ItemAsset', unitAmount:10000]).call()

// add shipping charge
ec.service.sync()
    .name("mantle.order.OrderServices.create#OrderItem")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        unitAmount:145.00, itemTypeEnumId:'ItemShipping',
        itemDescription:'Incoming Freight']).call()
// set billing and shipping info
setInfoOut = ec.service.sync()
    .name("mantle.order.OrderServices.set#OrderBillingShippingInfo")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        shippingPostalContactMechId:'ORG_BIZI_RTL_SA',
        shippingTelecomContactMechId:'ORG_BIZI_RTL_PT',
        shipmentMethodEnumId:'ShMthNoShipping']).call()

// one person will place the PO
ec.service.sync()
```

```
    .name("mantle.order.OrderServices.place#Order")
    .parameters([orderId:purchaseOrderId]).call()
// typically another person will approve the PO
ec.service.sync()
    .name("mantle.order.OrderServices.approve#Order")
    .parameters([orderId:purchaseOrderId]).call()
```

Once this process is done the PO is somehow sent to the supplier (vendor). Below is the entity XML for the order that is created. Note that much of the detail is in the `OrderPart` record including the vendor and customer parties, the payment and shipping info, and so on. Also note that **`effectiveTime`** is set to on `ec.user` as the effective time with a line like this (before the code above runs):

```
long effectiveTime = System.currentTimeMillis()
ec.user.setEffectiveTime(new Timestamp(effectiveTime))
```

Here is the XML for the order:

```
<mantle.order.OrderHeader orderId="${purchaseOrderId}"
    entryDate="${effectiveTime}" placedDate="${effectiveTime}"
    statusId="OrderApproved" currencyUomId="USD" grandTotal="11795.00"/>
<mantle.order.OrderPart orderId="${purchaseOrderId}" orderPartSeqId="01"
    vendorPartyId="MiddlemanInc" customerPartyId="ORG_BIZI_RETAIL"
    shipmentMethodEnumId="ShMthNoShipping"
    postalContactMechId="ORG_BIZI_RTL_SA"
    telecomContactMechId="ORG_BIZI_RTL_PT" partTotal="11795.00"/>
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" orderId="${purchaseOrderId}"
    orderPartSeqId="01" statusId="PmntPromised" amount="11795.00"
    amountUomId="USD"/>

<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="01"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    itemDescription="Demo Product One-One" quantity="150" unitAmount="8.00"
    isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="02"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    itemDescription="Demo Product Three-One" quantity="100"
    unitAmount="4.50" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="03"
    orderPartSeqId="01" itemTypeEnumId="ItemAsset" productId="EQUIP_1"
    itemDescription="Picker Bot 2000" quantity="1" unitAmount="10000"
    isModifiedPrice="Y"/>
<mantle.order.OrderItem orderId="${purchaseOrderId}" orderItemSeqId="04"
    orderPartSeqId="01" itemTypeEnumId="ItemShipping"
    itemDescription="Incoming Freight" quantity="1" unitAmount="145.00"/>
```

Create Incoming Shipment and Purchase Invoice

The code below creates a `Shipment` for the `OrderPart` (and there is just one order part, so we just create one), then marks the `Shipment` as Shipped, and then creates an `Invoice` for the entire `OrderPart`.

In real-world scenarios the invoice received may not match what is expected, or may even be for multiple or partial purchase orders. For this example we'll simply create an invoice automatically from the order to somewhat simulate a real-world scenario. In a real process we would more likely create the `Invoice` in the `InvoiceIncoming` status and then change it to `InvoiceReceived` to allow for manual changes between based on the invoice document received from the suppler (vendor).

```
shipResult = ec.service.sync()
    .name("mantle.shipment.ShipmentServices.create#OrderPartShipment")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        destinationFacilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.ship#Shipment")
    .parameters([shipmentId:shipResult.shipmentId]).call()
invResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#EntireOrderPartInvoice")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId,
        statusId:'InvoiceReceived']).call()
```

The `Shipment` created looks like this:

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpPurchase" statusId="ShipInput"
    fromPartyId="MiddlemanInc" toPartyId="ORG_BIZI_RETAIL"/>
<mantle.shipment.ShipmentPackage shipmentId="${shipResult.shipmentId}"
    shipmentPackageSeqId="01"/>
<mantle.shipment.ShipmentRouteSegment shipmentId="${shipResult.shipmentId}"
    shipmentRouteSegmentSeqId="01"
    destPostalContactMechId="ORG_BIZI_RTL_SA"
    destTelecomContactMechId="ORG_BIZI_RTL_PT"/>
<mantle.shipment.ShipmentPackageRouteSeg
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    shipmentRouteSegmentSeqId="01"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" quantity="150"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    orderId="${purchaseOrderId}" orderItemSeqId="01" statusId="SisPending"
    quantity="150" quantityNotHandled="150" invoiceId=""
    invoiceItemSeqId=""/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" quantity="100"/>
```

```
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    orderId="${purchaseOrderId}" orderItemSeqId="02" statusId="SisPending"
    quantity="100" quantityNotHandled="100" invoiceId=""
    invoiceItemSeqId=""/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="EQUIP_1" quantity="1"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    shipmentId="${shipResult.shipmentId}" productId="EQUIP_1"
    orderId="${purchaseOrderId}" orderItemSeqId="03" statusId="SisPending"
    quantity="1" quantityNotHandled="1" invoiceId="" invoiceItemSeqId=""/>
```

After the **ship#Shipment** call the `Shipment` record looks like this:

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpPurchase" statusId="ShipShipped"
    fromPartyId="MiddlemanInc" toPartyId="ORG_BIZI_RETAIL"/>
```

The XML below is what the `Invoice` looks like. Note that each `InvoiceItem` has a corresponding `OrderItemBilling` record to associated it with the `OrderItem` it is based on.

```
<!-- Invoice created and received, not yet approved/etc -->
<mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="MiddlemanInc"
    toPartyId="ORG_BIZI_RETAIL" statusId="InvoiceReceived"
    invoiceDate="${effectiveTime}"
    description="Invoice for Order ${purchaseOrderId} part 01"
    currencyUomId="USD"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    quantity="150" amount="8.00" description="Demo Product One-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55400"
    orderId="${purchaseOrderId}" orderItemSeqId="01"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="01" quantity="150"
    amount="8.00" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="02" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    quantity="100" amount="4.50" description="Demo Product Three-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55401"
    orderId="${purchaseOrderId}" orderItemSeqId="02"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="02" quantity="100"
    amount="4.50" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="03" itemTypeEnumId="ItemAsset" productId="EQUIP_1"
    quantity="1" amount="10,000" description="Picker Bot 2000"
```

```
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55402"
    orderId="${purchaseOrderId}" orderItemSeqId="03"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="03" quantity="1"
    amount="10,000" shipmentId="${shipResult.shipmentId}"/>

<mantle.account.invoice.InvoiceItem invoiceId="${invResult.invoiceId}"
    invoiceItemSeqId="04" itemTypeEnumId="ItemShipping" quantity="1"
    amount="145" description="Incoming Freight"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55403"
    orderId="${purchaseOrderId}" orderItemSeqId="04"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="04" quantity="1"
    amount="145"/>

<!-- ShipmentItemSource now has invoiceId and invoiceItemSeqId -->
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="01"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="02"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    invoiceId="${invResult.invoiceId}" invoiceItemSeqId="03"/>
```

Receive Shipment

There is a **receive#EntireShipment** service but in this case we want to receive an item at a time to show how to specify more details, and to handle the equipment product telling the system it is equipment and not inventory (**assetTypeEnumId**=AstTpEquipment) and recording the **serialNumber**.

```
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'DEMO_1_1',
        quantityAccepted:150, facilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'DEMO_3_1',
        quantityAccepted:100, facilityId:facilityId]).call()
ec.service.sync()
    .name("mantle.shipment.ShipmentServices.receive#ShipmentProduct")
    .parameters([shipmentId:shipResult.shipmentId, productId:'EQUIP_1',
        quantityAccepted:1, facilityId:facilityId,
        serialNumber:'PB2000AZQRTFP',
        assetTypeEnumId:'AstTpEquipment']).call()
```

This produces quite a bit of data including Asset records, AssetReceipt records to show the inventory and equipment received, and AssetDetail records to show the quantity change on the Asset records and why the quantity changed:

```
<mantle.product.asset.Asset assetId="55400"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_1_1" hasQuantity="Y"
    quantityOnHandTotal="150" availableToPromiseTotal="150"
    assetName="Demo Product One-One" receivedDate="${effectiveTime}"
    acquiredDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"
    acquireOrderId="${purchaseOrderId}" acquireOrderItemSeqId="01"
    acquireCost="8" acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55400" assetId="55400"
    productId="DEMO_1_1" orderId="${purchaseOrderId}" orderItemSeqId="01"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="150"/>
<mantle.product.asset.AssetDetail assetDetailId="55400" assetId="55400"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="150"
    availableToPromiseDiff="150" unitCost="8"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    assetReceiptId="55400"/>

<mantle.product.asset.Asset assetId="55401"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_3_1" hasQuantity="Y"
    quantityOnHandTotal="100" availableToPromiseTotal="100"
    assetName="Demo Product Three-One" receivedDate="${effectiveTime}"
    acquiredDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"
    acquireOrderId="${purchaseOrderId}" acquireOrderItemSeqId="02"
    acquireCost="4.5" acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55401" assetId="55401"
    productId="DEMO_3_1" orderId="${purchaseOrderId}" orderItemSeqId="02"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="100"/>
<mantle.product.asset.AssetDetail assetDetailId="55401" assetId="55401"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="100"
    availableToPromiseDiff="100" unitCost="4.5"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    assetReceiptId="55401"/>

<mantle.product.asset.Asset assetId="55402"
    assetTypeEnumId="AstTpEquipment" statusId="AstInStorage"
    ownerPartyId="ORG_BIZI_RETAIL" productId="EQUIP_1" hasQuantity="N"
    quantityOnHandTotal="1" availableToPromiseTotal="0"
    assetName="Picker Bot 2000" serialNumber="PB2000AZQRTFP"
    receivedDate="${effectiveTime}" acquiredDate="${effectiveTime}"
    facilityId="ORG_BIZI_RETAIL_WH" acquireOrderId="${purchaseOrderId}"
    acquireOrderItemSeqId="03" acquireCost="10,000"
    acquireCostUomId="USD"/>
<mantle.product.receipt.AssetReceipt assetReceiptId="55402" assetId="55402"
    productId="EQUIP_1" orderId="${purchaseOrderId}" orderItemSeqId="03"
    shipmentId="${shipResult.shipmentId}" receivedByUserId="EX_JOHN_DOE"
    receivedDate="${effectiveTime}" quantityAccepted="1"/>
<mantle.product.asset.AssetDetail assetDetailId="55402" assetId="55402"
```

```
    effectiveDate="${effectiveTime}" quantityOnHandDiff="1"
    availableToPromiseDiff="0" unitCost="10,000"
    shipmentId="${shipResult.shipmentId}" productId="EQUIP_1"
    assetReceiptId="55402"/>
```

Two other entities that is updated automatically when records exist are `OrderItemBilling` to have the assetReceiptId, and `ShipmentItemSource` now has quantityNotHandled="0" and statusId is set to `SisReceived`:

```
<mantle.order.OrderItemBilling orderItemBillingId="55400"
    assetReceiptId="55400"/>
<mantle.order.OrderItemBilling orderItemBillingId="55401"
    assetReceiptId="55401"/>
<mantle.order.OrderItemBilling orderItemBillingId="55402"
    assetReceiptId="55402"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55400"
    statusId="SisReceived" quantity="150" quantityNotHandled="0"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55401"
    statusId="SisReceived" quantity="100" quantityNotHandled="0"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55402"
    statusId="SisReceived" quantity="1" quantityNotHandled="0"/>
```

Inventory receipt also triggers accounting transactions with balancing entries for the COGS and inventory accounts:

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55400"
    acctgTransTypeEnumId="AttInventoryReceipt"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="55400" assetReceiptId="55400"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55400"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="1,200"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55400"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="1,200"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55401"
    acctgTransTypeEnumId="AttInventoryReceipt"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="55401" assetReceiptId="55401"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55401"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="450"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
```

```
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55401"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="450"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
```

Next to wrap things up with the order and shipment we record that the Shipment is Delivered and that the OrderPart is Complete:

```
ec.service.sync().name("update#mantle.shipment.Shipment")
    .parameters([shipmentId:shipResult.shipmentId,
        statusId:'ShipDelivered']).call()
ec.service.sync().name("mantle.order.OrderServices.complete#OrderPart")
    .parameters([orderId:purchaseOrderId, orderPartSeqId:orderPartSeqId])
    .call()
```

Because there is only one OrderPart on the order the status is updated on the OrderHeader as well. This data shows that and the updated Shipment status:

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    statusId="ShipDelivered"/>
<mantle.order.OrderHeader orderId="${purchaseOrderId}"
    statusId="OrderCompleted"/>
```

Approve Purchase Invoice and Send Payment

Now that the `Shipment` is received it's time to approve the `Invoice` for payment. Here is the service call to do that, note the pattern of using the implicit entity-auto service to change status (this is how ALL status changes are done to facilitate a consistent place to attach SECA rules):

```
ec.service.sync().name("update#mantle.account.invoice.Invoice")
   .parameters([invoiceId:invResult.invoiceId, statusId:'InvoiceApproved'])
   .call()
```

Here is the updated `Invoice` record:

```
<mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
    statusId="InvoiceApproved"/>
```

When an `Invoice` goes into the `InvoiceApproved` status it triggers the posting of the accounting transaction for the Invoice. The XML below has the `AcctgTrans` record and the corresponding `AcctgTransEntry` records, one for each `InvoiceItem` and the last one (05) for the balancing entry to `GlAccount 210000` which is the Accounts Payable account.

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55402"
    acctgTransTypeEnumId="AttPurchaseInvoice"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
```

```
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="MiddlemanInc"
    invoiceId="${invResult.invoiceId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="1200"
    glAccountId="501000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_1_1" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="450"
    glAccountId="501000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_3_1" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
    acctgTransEntrySeqId="03" debitCreditFlag="D" amount="10,000"
    glAccountTypeEnumId="FIXED_ASSET" glAccountId="171000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="EQUIP_1" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
    acctgTransEntrySeqId="04" debitCreditFlag="D" amount="145"
    glAccountTypeEnumId="" glAccountId="509000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55402"
    acctgTransEntrySeqId="05" debitCreditFlag="C" amount="11795"
    glAccountTypeEnumId="ACCOUNTS_PAYABLE" glAccountId="210000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

The `Payment` was created above with the order as the promised payment (see the **Place and Approve Purchase Order** section). Now we call a service to mark that promised payment as sent. This service will also apply the `Payment` to the `Invoice`, creating a `PaymentApplication` record. Once a `Payment` is applied to a purchase `Invoice` its status gets changed to payment sent (`InvoicePmtSent`).

```
sendPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.send#PromisedPayment")
    .parameters([invoiceId:invResult.invoiceId,
        paymentId:setInfoOut.paymentId]).call()
```

Here is the `PaymentApplication` just created, the `Payment` record with a **`statusId`** of `PmntDelivered` and the **`effectiveDate`** field set, and the `Invoice` updated to the `InvoicePmtSent` status:

```
<mantle.account.payment.PaymentApplication
    paymentApplicationId="${sendPmtResult.paymentApplicationId}"
    paymentId="${setInfoOut.paymentId}" invoiceId="${invResult.invoiceId}"
    amountApplied="11795.00" appliedDate="${effectiveTime}"/>
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    statusId="PmntDelivered" effectiveDate="${effectiveTime}"/>
<mantle.account.invoice.Invoice invoiceId="${invResult.invoiceId}"
    statusId="InvoicePmtSent"/>
```

The `Payment` status change to Delivered triggers its GL posting. Because it is a check received and the automated posting is configured this way the `Payment` comes from (credited to) the General Checking Account GL account (`111100`):

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55403"
    acctgTransTypeEnumId="AttOutgoingPayment"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="MiddlemanInc"
    paymentId="${setInfoOut.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55403"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="11795"
    glAccountId="216000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55403"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="11795"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

Because the `Payment` was posted (from the status update) before it was applied to the `Invoice` it was posted to the Accounts Payable Unapplied Payments account (`216000`). When the `PaymentApplication` is created this triggers another GL posting for the `PaymentApplication` to credit those funds back to the unapplied payments account and debit them from the main Accounts Payable account (`210000`). Here is that transaction:

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55404"
    acctgTransTypeEnumId="AttOutgoingPaymentAp"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="MiddlemanInc"
    paymentId="${setInfoOut.paymentId}"
    paymentApplicationId="${sendPmtResult.paymentApplicationId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55404"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="11795"
    glAccountId="210000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55404"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="11795"
    glAccountId="216000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

With the assets received, the invoice paid, and the everything posted to the general ledger the Procure to Pay process is complete. The net effect of the GL postings is the outgoing payment is credited to the General Checking Account GL account (`111100`), the price of the inventory purchase is debited to the Inventory asset account (`140000`), the price of the equipment is debited to the Equipment asset account (`171000`), and the shipping cost to the

Freight In cost of sales account (509000). The entries posted to all other accounts balance each other out to zero, including the Accounts Payable account (210000).

Order to Cash

The Spock test suite for this process is in the OrderToCashBasicFlow.groovy file and that is the file covered in this section. There are related test suites for placing a sales order for tenant subscription and provisioning in OrderTenantAccess.groovy and for testing the time it takes to place sales orders in OrderToCashTime.groovy.

Some of the more relevant setup data is shown in the examples below but you can find the rest of it in the ZzaGlAccountsDemoData.xml, ZzbOrganizationDemoData.xml, and ZzcProductDemoData.xml files.

Place a Sales Order as a Customer

This process is a basic ecommerce process. The order is placed by a customer (joe@public.com) so the first step in the code below is to login that user, and the last step is to logout that user, then an internal user does the shipping which triggers automated payment processing and so on.

The code below uses the POPC_DEFAULT demo ProductStore, which uses the ORG_BIZI_RETAIL_WH Facility for inventory, a test payment processor, and local services for tax and shipping calculation. Here are the records that define these (from the ZzcProductDemoData.xml file):

```
<mantle.facility.Facility facilityId="ORG_BIZI_RETAIL_WH"
    facilityTypeEnumId="FcTpWarehouse" ownerPartyId="ORG_BIZI_RETAIL"
    facilityName="Biziwork Retail Warehouse"/>
<mantle.product.store.ProductStore productStoreId="POPC_DEFAULT"
    storeName="Biziwork Retail Store" organizationPartyId="ORG_BIZI_RETAIL"
    inventoryFacilityId="ORG_BIZI_RETAIL_WH"
    reservationOrderEnumId="AsResOrdFifoRec" requirementMethodEnumId=""
    defaultLocale="en_US" defaultCurrencyUomId="USD"
    taxGatewayConfigId="LOCAL"/>
<mantle.product.store.ProductStorePaymentGateway
    productStoreId="POPC_DEFAULT" paymentMethodTypeEnumId="PmtCreditCard"
    paymentGatewayConfigId="TEST_APPROVE"/>
<mantle.product.store.ProductStoreShippingGateway
    productStoreId="POPC_DEFAULT" carrierPartyId="_NA_"
    shippingGatewayConfigId="NA_LOCAL"/>
```

In the code below the **get#ProductPrice** service is used to get (calculate) the price for a Product and is called here on its own for demonstration. When adding to the order it calls this service on its own to get the price.

Note that the first call to the **add#OrderProductQuantity** service results in a new order being created, so we get the "cart" orderId from the results of that service call. Subsequent calls pass in an orderId parameter so that the product quantities are added to the same order.

Next it calls the **set#OrderBillingShippingInfo** service to set the billing and shipping info on the order, using a payment method and contact mechs from the customer's profile. Finally it calls the **place#Order** service which is what would happen when a customer does a final order review and confirms the order.

```
ec.user.loginUser("joe@public.com", "moqui", null)
long effectiveTime = System.currentTimeMillis()
ec.user.setEffectiveTime(new Timestamp(effectiveTime))

String productStoreId = "POPC_DEFAULT"
EntityValue productStore =
    ec.entity.makeFind("mantle.product.store.ProductStore")
        .condition("productStoreId", productStoreId).one()
String currencyUomId = productStore.defaultCurrencyUomId
String priceUomId = productStore.defaultCurrencyUomId
String vendorPartyId = productStore.organizationPartyId
String customerPartyId = ec.user.userAccount.partyId

Map priceMap = ec.service.sync()
    .name("mantle.product.PriceServices.get#ProductPrice")
    .parameters([productId:'DEMO_1_1', priceUomId:priceUomId,
        productStoreId:productStoreId, vendorPartyId:vendorPartyId,
        customerPartyId:customerPartyId]).call()

Map addOut1 = ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([productId:'DEMO_1_1', quantity:1,
        customerPartyId:customerPartyId, currencyUomId:currencyUomId,
        productStoreId:productStoreId]).call()

cartOrderId = addOut1.orderId
orderPartSeqId = addOut1.orderPartSeqId

ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:cartOrderId, productId:'DEMO_3_1', quantity:5,
        customerPartyId:customerPartyId, currencyUomId:currencyUomId,
        productStoreId:productStoreId]).call()
ec.service.sync()
    .name("mantle.order.OrderServices.add#OrderProductQuantity")
    .parameters([orderId:cartOrderId, orderPartSeqId:orderPartSeqId,
        productId:'DEMO_2_1', quantity:7, customerPartyId:customerPartyId,
        currencyUomId:currencyUomId, productStoreId:productStoreId])
    .call()

setInfoOut = ec.service.sync()
```

```
    .name("mantle.order.OrderServices.set#OrderBillingShippingInfo")
    .parameters([orderId:cartOrderId, paymentMethodId:'CustJqpCc',
        shippingPostalContactMechId:'CustJqpAddr',
        shippingTelecomContactMechId:'CustJqpTeln', carrierPartyId:'_NA_',
        shipmentMethodEnumId:'ShMthGround']).call()
ec.service.sync().name("mantle.order.OrderServices.place#Order")
    .parameters([orderId:cartOrderId]).call()

ec.user.logoutUser()
```

The **place#Order** service call triggers payment authorization, which then updates the order status to Approved so that is the status at this point. We also have a Payment record with the billing settings **set#OrderBillingShippingInfo** and the rest are on the OrderPart record. There is a PaymentGatewayResponse record from the credit card authorization. To wrap it up we have three OrderItem records, one for each call to **add#OrderProductQuantity** with a different productId.

```
<mantle.order.OrderHeader orderId="${cartOrderId}"
    entryDate="${effectiveTime}" placedDate="${effectiveTime}"
    statusId="OrderApproved" currencyUomId="USD"
    productStoreId="POPC_DEFAULT" grandTotal="145.68"/>
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    paymentTypeEnumId="PtInvoicePayment" paymentMethodId="CustJqpCc"
    paymentMethodTypeEnumId="PmtCreditCard" orderId="${cartOrderId}"
    orderPartSeqId="01" statusId="PmntAuthorized" amount="145.68"
    amountUomId="USD" fromPartyId="CustJqp" toPartyId="ORG_BIZI_RETAIL"/>
<mantle.account.method.PaymentGatewayResponse
    paymentGatewayResponseId="55500" paymentOperationEnumId="PgoAuthorize"
    paymentId="${setInfoOut.paymentId}" paymentMethodId="CustJqpCc"
    amount="145.68" amountUomId="USD" transactionDate="${effectiveTime}"
    resultSuccess="Y" resultDeclined="N" resultNsf="N" resultBadExpire="N"
    resultBadCardNumber="N"/>

<mantle.order.OrderPart orderId="${cartOrderId}" orderPartSeqId="01"
    vendorPartyId="ORG_BIZI_RETAIL" customerPartyId="CustJqp"
    shipmentMethodEnumId="ShMthGround" postalContactMechId="CustJqpAddr"
    telecomContactMechId="CustJqpTeln" partTotal="145.68"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="01"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_1_1"
    itemDescription="Demo Product One-One" quantity="1" unitAmount="16.99"
    unitListPrice="19.99" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="02"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_3_1"
    itemDescription="Demo Product Three-One" quantity="5" unitAmount="7.77"
    unitListPrice="" isModifiedPrice="N"/>
<mantle.order.OrderItem orderId="${cartOrderId}" orderItemSeqId="03"
    orderPartSeqId="01" itemTypeEnumId="ItemProduct" productId="DEMO_2_1"
    itemDescription="Demo Product Two-One" quantity="7" unitAmount="12.12"
    unitListPrice="" isModifiedPrice="N"/>
```

The other main thing that happens when an order is placed is that inventory (in the `Asset` entity) is reserved for the items on the order. Inventory reservations are tracked with the `AssetReservation` entity, so we have 3 records for it (one for each `Product` on the order).

The first 2 Asset records are from demo data in the `ZzcProductDemoData.xml` file. They already have inventory available to the `AssetDetail` records for those that adjust the `Asset.availableToPromiseTotal` using a negative `AssetDetail.availableToPromiseDiff` value.

The last `Asset` record has a sequenced ID because there was no inventory for this product and the `Asset` record was created on the fly with an ATP and QOH of 0. After the `AssetDetail` record is created the `availableToPromiseTotal` is set to "-7" meaning there is a quantity of 7 on backorder. This is also tracked in the `AssetReservation.quantityNotAvailable` field, as this is the quantity "reserved" that is not available to promise.

```
<mantle.product.asset.Asset assetId="DEMO_1_1A"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_1_1" hasQuantity="Y"
    quantityOnHandTotal="100" availableToPromiseTotal="99"
    receivedDate="1265184000000" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55500"
    assetId="DEMO_1_1A" productId="DEMO_1_1" orderId="${cartOrderId}"
    orderItemSeqId="01" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="1" reservedDate="${effectiveTime}" sequenceNum="0"/>
<mantle.product.asset.AssetDetail assetDetailId="55500" assetId="DEMO_1_1A"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-1"
    assetReservationId="55500" productId="DEMO_1_1"/>

<mantle.product.asset.Asset assetId="DEMO_3_1A"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_3_1" hasQuantity="Y"
    quantityOnHandTotal="5" availableToPromiseTotal="0"
    receivedDate="1265184000000" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55501"
    assetId="DEMO_3_1A" productId="DEMO_3_1" orderId="${cartOrderId}"
    orderItemSeqId="02" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="5" reservedDate="${effectiveTime}" sequenceNum="0"/>
<mantle.product.asset.AssetDetail assetDetailId="55501" assetId="DEMO_3_1A"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-5"
    assetReservationId="55501" productId="DEMO_3_1"/>

<mantle.product.asset.Asset assetId="55500"
    assetTypeEnumId="AstTpInventory" statusId="AstAvailable"
    ownerPartyId="ORG_BIZI_RETAIL" productId="DEMO_2_1" hasQuantity="Y"
    quantityOnHandTotal="0" availableToPromiseTotal="-7"
    receivedDate="${effectiveTime}" facilityId="ORG_BIZI_RETAIL_WH"/>
<mantle.product.issuance.AssetReservation assetReservationId="55502"
    assetId="55500" productId="DEMO_2_1" orderId="${cartOrderId}"
```

```
    orderItemSeqId="03" reservationOrderEnumId="AsResOrdFifoRec"
    quantity="7" quantityNotAvailable="7" reservedDate="${effectiveTime}"/>
<mantle.product.asset.AssetDetail assetDetailId="55502" assetId="55500"
    effectiveDate="${effectiveTime}" availableToPromiseDiff="-7"
    assetReservationId="55502" productId="DEMO_2_1"/>
```

Ship Sales Order

There is a single service call that can be used to ship an entire `OrderPart`: **`ship#OrderPart`**.

```
shipResult = ec.service.sync()
    .name("mantle.shipment.ShipmentServices.ship#OrderPart")
    .parameters([orderId:cartOrderId, orderPartSeqId:orderPartSeqId])
    .call()
```

This service does a few things and when implementing a real-world system the services it calls, or even the services they call, will have more granular options and be more useful:

- `mantle.shipment.ShipmentServices.create#OrderPartShipment` (created a Shipment, adds ShipmentItem records for all products on the order, creates a package and route segment, and ties it all together)
- `mantle.shipment.ShipmentServices.pack#ShipmentProduct` (with the productId and quantity from each OrderItem)
- `mantle.shipment.ShipmentServices.pack#Shipment` (the Shipment going to the Packed status triggers invoicing with the `mantle.account.InvoiceServices.create#SalesShipmentInvoices` service and credit card payment capture)
- `mantle.order.OrderServices.checkComplete#OrderPart` (if all items in the order part have been fulfilled change its status to Complete)
- `mantle.shipment.ShipmentServices.ship#Shipment`

Here is the XML for the `Shipment` and related entities. There is a `ShipmentItem` record for each productId, and a `ShipmentItemSource` record to associate it with the `OrderItem` and `InvoiceItem`, and to keep track of pick/pack status (in this case `Packed` as we called the **`pack#ShipmentProduct`** service). There is also a `ShipmentPackage` plus a `ShipmentPackageContent` record for each shipment item to associate it with the package. Finally there is a `ShipmentRouteSegment` record and a `ShipmentPackageRouteSeg` to associate it with the package.

```
<mantle.shipment.Shipment shipmentId="${shipResult.shipmentId}"
    shipmentTypeEnumId="ShpTpSales" statusId="ShipShipped"
    fromPartyId="ORG_BIZI_RETAIL" toPartyId="CustJqp"/>
<mantle.shipment.ShipmentPackage shipmentId="${shipResult.shipmentId}"
    shipmentPackageSeqId="01"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" quantity="1"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55500"
```

```
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
    orderId="${cartOrderId}" orderItemSeqId="01" statusId="SisPacked"
    quantity="1" invoiceId="55500" invoiceItemSeqId="01"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_1_1" quantity="1"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" quantity="5"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55501"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    orderId="${cartOrderId}" orderItemSeqId="02" statusId="SisPacked"
    quantity="5" invoiceId="55500" invoiceItemSeqId="02"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_3_1" quantity="5"/>

<mantle.shipment.ShipmentItem shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" quantity="7"/>
<mantle.shipment.ShipmentItemSource shipmentItemSourceId="55502"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_2_1"
    orderId="${cartOrderId}" orderItemSeqId="03" statusId="SisPacked"
    quantity="7" invoiceId="55500" invoiceItemSeqId="03"/>
<mantle.shipment.ShipmentPackageContent
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    productId="DEMO_2_1" quantity="7"/>

<mantle.shipment.ShipmentRouteSegment shipmentId="${shipResult.shipmentId}"
    shipmentRouteSegmentSeqId="01" destPostalContactMechId="CustJqpAddr"
    destTelecomContactMechId="CustJqpTeln"/>
<mantle.shipment.ShipmentPackageRouteSeg
    shipmentId="${shipResult.shipmentId}" shipmentPackageSeqId="01"
    shipmentRouteSegmentSeqId="01"/>
```

Here is the `OrderHeader` with its status updated based on the Complete `OrderPart`:

```
<mantle.order.OrderHeader orderId="${cartOrderId}"
    statusId="OrderCompleted"/>
```

When an `ShipmentItem` (or more specifically a `ShipmentItemSource`) is **packed** the inventory, usually reserved so having a `AssetReservation` record, is issued to the shipment and recorded in a `AssetIssuance` record plus a `AssetDetail` record with a **`quantityOnHandDiff`** to adjust the `Asset.quantityOnHandTotal`. Here are those records for this shipment:

```
<mantle.product.asset.Asset assetId="DEMO_1_1A" quantityOnHandTotal="99"
    availableToPromiseTotal="99"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55500"
    assetId="DEMO_1_1A" assetReservationId="55500"
    orderId="${cartOrderId}" orderItemSeqId="01"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_1_1"
```

```
    quantity="1"/>
<mantle.product.asset.AssetDetail assetDetailId="55503" assetId="DEMO_1_1A"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-1"
    assetReservationId="55500" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_1_1" assetIssuanceId="55500"/>

<mantle.product.asset.Asset assetId="DEMO_3_1A" quantityOnHandTotal="0"
    availableToPromiseTotal="0"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55501"
    assetId="DEMO_3_1A" assetReservationId="55501"
    orderId="${cartOrderId}" orderItemSeqId="02"
    shipmentId="${shipResult.shipmentId}" productId="DEMO_3_1"
    quantity="5"/>
<mantle.product.asset.AssetDetail assetDetailId="55504" assetId="DEMO_3_1A"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-5"
    assetReservationId="55501" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_3_1" assetIssuanceId="55501"/>

<mantle.product.asset.Asset assetId="55500" quantityOnHandTotal="-7"
    availableToPromiseTotal="-7"/>
<mantle.product.issuance.AssetIssuance assetIssuanceId="55502"
    assetId="55500" assetReservationId="55502" orderId="${cartOrderId}"
    orderItemSeqId="03" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" quantity="7"/>
<mantle.product.asset.AssetDetail assetDetailId="55505" assetId="55500"
    effectiveDate="${effectiveTime}" quantityOnHandDiff="-7"
    assetReservationId="55502" shipmentId="${shipResult.shipmentId}"
    productId="DEMO_2_1" assetIssuanceId="55502"/>
```

Asset issuance is a business activity that has a financial impact, so there are accounting transactions posted to the GL for it. The one exception is there is no `AcctgTrans` for **`assetId`** `55500`, **`productId`** `DEMO_2_1` because it is auto-created and has no **`acquireCost`**. For most organizations you wouldn't want to do this, i.e. the **`acquireCost`** field should always be populated, but for simpler system needs where you don't want to track the cost and inventory value this is what is expected.

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55500"
    acctgTransTypeEnumId="AttInventoryIssuance"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="DEMO_1_1A" assetIssuanceId="55500"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55500"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="7.5"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55500"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="7.5"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
```

```
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_1_1"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55501"
    acctgTransTypeEnumId="AttInventoryIssuance"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" assetId="DEMO_3_1A" assetIssuanceId="55501"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55501"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="20"
    glAccountTypeEnumId="INVENTORY_ACCOUNT" glAccountId="140000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55501"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="20"
    glAccountTypeEnumId="COGS_ACCOUNT" glAccountId="501000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"
    productId="DEMO_3_1"/>
```

As mentioned above when a `Shipment` goes into the **Packed** status it triggers the creation of an `Invoice` for the order items on the `Shipment`. Here is what that `Invoice` looks like with its `InvoiceItem` records and `OrderItemBilling` records that associate `InvoiceItem` records with their corresponding `OrderItem` records:

```
<mantle.account.invoice.Invoice invoiceId="55500"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="ORG_BIZI_RETAIL"
    toPartyId="CustJqp" statusId="InvoicePmtRecvd"
    invoiceDate="${effectiveTime}"
    description="Invoice for Order ${cartOrderId} part 01 and Shipment
        ${shipResult.shipmentId}" currencyUomId="USD"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="01"
    itemTypeEnumId="ItemProduct" productId="DEMO_1_1" quantity="1"
    amount="16.99" description="Demo Product One-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55500"
    orderId="${cartOrderId}" orderItemSeqId="01" invoiceId="55500"
    invoiceItemSeqId="01" assetIssuanceId="55500"
    shipmentId="${shipResult.shipmentId}" quantity="1" amount="16.99"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="02"
    itemTypeEnumId="ItemProduct" productId="DEMO_3_1" quantity="5"
    amount="7.77" description="Demo Product Three-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55501"
    orderId="${cartOrderId}" orderItemSeqId="02" invoiceId="55500"
    invoiceItemSeqId="02" assetIssuanceId="55501"
    shipmentId="${shipResult.shipmentId}" quantity="5" amount="7.77"/>
```

```
<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="03"
    itemTypeEnumId="ItemProduct" productId="DEMO_2_1" quantity="7"
    amount="12.12" description="Demo Product Two-One"
    itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55502"
    orderId="${cartOrderId}" orderItemSeqId="03" invoiceId="55500"
    invoiceItemSeqId="03" assetIssuanceId="55502"
    shipmentId="${shipResult.shipmentId}" quantity="7" amount="12.12"/>

<mantle.account.invoice.InvoiceItem invoiceId="55500" invoiceItemSeqId="04"
    itemTypeEnumId="ItemShipping" quantity="1" amount="5"
    description="Ground" itemDate="${effectiveTime}"/>
<mantle.order.OrderItemBilling orderItemBillingId="55503"
    orderId="${cartOrderId}" orderItemSeqId="04" invoiceId="55500"
    invoiceItemSeqId="04" shipmentId="${shipResult.shipmentId}"
    quantity="1" amount="5"/>
```

Invoices are records with a financial impact so they also have accounting transactions posted to the GL. There is one transaction entry (`AcctgTransEntry`) per `InvoiceItem` to credit the applicable sales account (or shipping/handling received account), and one balancing entry to debit the Accounts Receivable account.

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55502"
    acctgTransTypeEnumId="AttSalesInvoice"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="CustJqp" invoiceId="55500"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="16.99"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_1_1" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="38.85"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_3_1" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="03" debitCreditFlag="C" amount="84.84"
    glAccountId="401000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" productId="DEMO_2_1" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="04" debitCreditFlag="C" amount="5"
    glAccountId="731200" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55502"
    acctgTransEntrySeqId="05" debitCreditFlag="D" amount="145.68"
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE" glAccountId="120000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

The final operation is to capture the credit card payment resulting in a `PaymentGatewayResponse` record and an update of the `Payment` status to **Delivered**. This also has an `AcctgTrans` record with entries for the cash account and accounts receivable account.

```
<mantle.account.payment.Payment paymentId="${setInfoOut.paymentId}"
    statusId="PmntDelivered"/>
<mantle.account.payment.PaymentApplication paymentApplicationId="55500"
    paymentId="${setInfoOut.paymentId}" invoiceId="55500"
    amountApplied="145.68" appliedDate="${effectiveTime}"/>
<mantle.account.method.PaymentGatewayResponse
    paymentGatewayResponseId="55501" paymentOperationEnumId="PgoCapture"
    paymentId="${setInfoOut.paymentId}" paymentMethodId="CustJqpCc"
    amount="145.68" amountUomId="USD" transactionDate="${effectiveTime}"
    resultSuccess="Y" resultDeclined="N" resultNsf="N"
    resultBadExpire="N" resultBadCardNumber="N"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55503"
    acctgTransTypeEnumId="AttIncomingPayment"
    organizationPartyId="ORG_BIZI_RETAIL"
    transactionDate="${effectiveTime}" isPosted="Y"
    glFiscalTypeEnumId="GLFT_ACTUAL" amountUomId="USD"
    otherPartyId="CustJqp" paymentId="${setInfoOut.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55503"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="145.68"
    glAccountId="120000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55503"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="145.68"
    glAccountId="122000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

With the items shipped, payment received, and everything posted to the general ledger the Order to Cash process is complete.

Work Plan to Cash

The Spock test suite for this process is in the `WorkPlanToCashBasicFlow.groovy` file. This is the main process supported by the HiveMind Project Manager application.

Some of the more relevant setup data is shown in the examples below but you can find the rest of it in the `ZzaGlAccountsDemoData.xml`, `ZzbOrganizationDemoData.xml`, and `ZzcProductDemoData.xml` files.

Vendor

The first thing to setup in a system for a services organization is the vendor, the services organization itself. This is an `Organization` type `Party` with the role of Internal

Organization (`OrgInternal`). It is also in the vendor (`VendorBillFrom`) role. It also has a full accounting configuration copied from the `'DefaultSettings' Party` (defined in the `ZzaGlAccountsDemoData.xml` file) using the **`init#PartyAccountingConfiguration`** service. This also uses the **`create#Account`** service to create a representative (for AR/AP/etc) of the vendor organization that is a `Person` type `Party` with a `UserAccount`.

```
long effectiveTime = System.currentTimeMillis()
ec.user.loginUser("john.doe", "moqui", null)
// set an effective date so data check works, etc
ec.user.setEffectiveTime(new Timestamp(effectiveTime))
effectiveThruDate = ec.l10n.parseTimestamp(
    ec.l10n.formatValue(ec.user.nowTimestamp,'yyyy-MM-dd HH:mm'),
        'yyyy-MM-dd HH:mm')
Map vendorResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Organization")
    .parameters([roleTypeId:'VendorBillFrom',
        organizationName:'Test Vendor']).call()
Map vendorCiResult = ec.service.sync()
    .name("mantle.party.ContactServices.store#PartyContactInfo")
    .parameters([partyId:vendorResult.partyId,
        postalContactMechPurposeId:'PostalPayment',
        telecomContactMechPurposeId:'PhonePayment',
        emailContactMechPurposeId:'EmailPayment', countryGeoId:'USA',
        address1:'51 W. Center St.', unitNumber:'1234', city:'Orem',
        stateProvinceGeoId:'USA_UT', postalCode:'84057',
        postalCodeExt:'4605', countryCode:'+1', areaCode:'801',
        contactNumber:'123-4567', emailAddress:'vendor.ar@test.com'])
     .call()
ec.service.sync().name("create#mantle.party.PartyRole")
    .parameters([partyId:vendorResult.partyId, roleTypeId:'OrgInternal'])
    .call()
ec.service.sync()
    .name("mantle.ledger.LedgerServices.init#PartyAccountingConfiguration")
    .parameters([sourcePartyId:'DefaultSettings',
        organizationPartyId:vendorResult.partyId]).call()

Map vendorRepResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Vendor', lastName:'TestRep',
        emailAddress:'vendor.rep@test.com', username:'vendor.rep',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map repRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtRepresentative',
        fromPartyId:vendorRepResult.partyId, fromRoleTypeId:'Manager',
        toPartyId:vendorResult.partyId, toRoleTypeId:'VendorBillFrom',
        fromDate:ec.user.nowTimestamp]).call()
```

Here are the records for the vendor `Organization` and its contact information:

```
<mantle.party.Party partyId="${vendorResult.partyId}"
    partyTypeEnumId="PtyOrganization"/>
<mantle.party.Organization partyId="${vendorResult.partyId}"
    organizationName="Test Vendor"/>
<mantle.party.PartyRole partyId="${vendorResult.partyId}"
    roleTypeId="OrgInternal"/>
<mantle.party.PartyRole partyId="${vendorResult.partyId}"
    roleTypeId="VendorBillFrom"/>

<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.postalContactMechId}"
    contactMechTypeEnumId="CmtPostalAddress"/>
<mantle.party.contact.PostalAddress
    contactMechId="${vendorCiResult.postalContactMechId}"
    address1="51 W. Center St." unitNumber="1234" city="Orem"
    stateProvinceGeoId="USA_UT" countryGeoId="USA" postalCode="84057"
    postalCodeExt="4605"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.postalContactMechId}"
    contactMechPurposeId="PostalPayment" fromDate="${effectiveTime}"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.telecomContactMechId}"
    contactMechTypeEnumId="CmtTelecomNumber"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.telecomContactMechId}"
    contactMechPurposeId="PhonePayment" fromDate="${effectiveTime}"/>
<mantle.party.contact.TelecomNumber
    contactMechId="${vendorCiResult.telecomContactMechId}" countryCode="+1"
    areaCode="801" contactNumber="123-4567"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorCiResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="vendor.ar@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${vendorResult.partyId}"
    contactMechId="${vendorCiResult.emailContactMechId}"
    contactMechPurposeId="EmailPayment" fromDate="${effectiveTime}"/>
```

Here are the records for the accounting configuration for the vendor. The various configuration records (GlAccountTypeDefault, ItemTypeGlAccount, GlAccountOrganization, PaymentTypeGlAccount, etc) are a small selection and there are many others copied from the `'DefaultSettings' Party`.

```
<mantle.ledger.transaction.GlJournal
    glJournalId="${vendorResult.partyId}Error"
    glJournalName="Error Journal for ${vendorResult.partyId}"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.PartyAcctgPreference
    organizationPartyId="${vendorResult.partyId}"
    taxFormEnumId="TxfUsIrs1120" cogsMethodEnumId="CogsActualCost"
    baseCurrencyUomId="USD" invoiceSequenceEnumId="InvSqStandard"
```

```
    orderSequenceEnumId="OrdSqStandard"
    errorGlJournalId="${vendorResult.partyId}Error"/>
<mantle.ledger.config.GlAccountTypeDefault
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE"
    organizationPartyId="${vendorResult.partyId}" glAccountId="120000"/>
<mantle.ledger.config.GlAccountTypeDefault
    glAccountTypeEnumId="ACCOUNTS_PAYABLE"
    organizationPartyId="${vendorResult.partyId}" glAccountId="210000"/>
<mantle.ledger.config.PaymentMethodTypeGlAccount
    paymentMethodTypeEnumId="PmtCompanyCheck"
    organizationPartyId="${vendorResult.partyId}" glAccountId="111100"/>
<mantle.ledger.config.ItemTypeGlAccount glAccountId="402000" direction="O"
    itemTypeEnumId="ItemTimeEntry"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.ItemTypeGlAccount glAccountId="550000" direction="I"
    itemTypeEnumId="ItemTimeEntry"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.ItemTypeGlAccount itemTypeEnumId="ItemExpTravAir"
    direction="E" glAccountId="681000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.account.GlAccountOrganization glAccountId="120000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.account.GlAccountOrganization glAccountId="210000"
    organizationPartyId="${vendorResult.partyId}"/>
<mantle.ledger.config.PaymentTypeGlAccount
    paymentTypeEnumId="PtInvoicePayment"
    organizationPartyId="${vendorResult.partyId}" isPayable="N"
    isApplied="Y" glAccountId="120000"/>
<mantle.ledger.config.PaymentTypeGlAccount
    paymentTypeEnumId="PtInvoicePayment"
    organizationPartyId="${vendorResult.partyId}" isPayable="Y"
    isApplied="Y" glAccountId="210000"/>
```

Here are the records for the vendor representative `Person` and its contact information. Note that the passwordSalt is randomly generated so the SHA-256 encrypted password will be different from any other run.

```
<mantle.party.Party partyId="${vendorRepResult.partyId}"
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${vendorRepResult.partyId}"
    firstName="Vendor" lastName="TestRep"/>
<moqui.security.UserAccount userId="${vendorRepResult.userId}"
    username="vendor.rep" userFullName="Vendor TestRep"
    passwordHashType="SHA-256" passwordSetDate="${effectiveTime}"
    disabled="N" requirePasswordChange="N"
    emailAddress="vendor.rep@test.com" passwordSalt="{.rqlPt8x"
    partyId="${vendorRepResult.partyId}" currentPassword="32ce60c14d9e72c1
        fb17938ede30fe9de04390409cce7310743c2716a2c7bf89"/>
<mantle.party.contact.ContactMech
    contactMechId="${vendorRepResult.emailContactMechId}"
```

```
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="vendor.rep@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${vendorRepResult.partyId}"
    contactMechId="${vendorRepResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${repRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtRepresentative"
    fromPartyId="${vendorRepResult.partyId}" fromRoleTypeId="Manager"
    toPartyId="${vendorResult.partyId}" toRoleTypeId="VendorBillFrom"
    fromDate="${effectiveTime}"/>
```

Worker and Rates

The code below creates a `Person` type `Party` and `UserAccount` for a worker, i.e. someone to work on tasks. It also creates two `RateAmount` records, one for the $60 rate the vendor (the internal organization, i.e. the org running the system) will bill the client, and another for the $40 rate the worker as an external contractor will bill to the vendor. The worker is related to the vendor as an agent with a `PartyRelationship` record of type `PrtAgent`.

```
Map workerResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Test', lastName:'Worker',
        emailAddress:'worker@test.com', username:'worker',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map workerRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtAgent',
        fromPartyId:workerResult.partyId, fromRoleTypeId:'Worker',
        toPartyId:vendorResult.partyId, toRoleTypeId:'VendorBillFrom',
        fromDate:ec.user.nowTimestamp]).call()
Map clientRateResult = ec.service.sync()
    .name("create#mantle.humanres.rate.RateAmount")
    .parameters([rateTypeEnumId:'RatpStandard',
        ratePurposeEnumId:'RaprClient', timePeriodUomId:'TF_hr',
        emplPositionClassId:'Programmer', fromDate:'2010-02-03 00:00:00',
        rateAmount:'60.00', rateCurrencyUomId:'USD',
        partyId:workerResult.partyId]).call()
Map vendorRateResult = ec.service.sync()
    .name("create#mantle.humanres.rate.RateAmount")
    .parameters([rateTypeEnumId:'RatpStandard',
        ratePurposeEnumId:'RaprVendor', timePeriodUomId:'TF_hr',
        emplPositionClassId:'Programmer', fromDate:'2010-02-03 00:00:00',
        rateAmount:'40.00', rateCurrencyUomId:'USD',
        partyId:workerResult.partyId]).call()
```

Here are the records for the worker `Party` and the billing rates:

```
<mantle.party.Party partyId="${workerResult.partyId}"
```

```
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${workerResult.partyId}" firstName="Test"
    lastName="Worker"/>
<moqui.security.UserAccount userId="${workerResult.userId}"
    username="worker" userFullName="Test Worker" passwordHashType="SHA-256"
    passwordSetDate="${effectiveTime}" disabled="N"
    requirePasswordChange="N" emailAddress="worker@test.com"
    partyId="${workerResult.partyId}" passwordSalt="{.rqlPt8x"
    currentPassword="32ce60c14d9e72c1fb17938ede30fe9de04390409cce7310743
        c2716a2c7bf89"/>
<mantle.party.contact.ContactMech
    contactMechId="${workerResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress" infoString="worker@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${workerResult.partyId}"
    contactMechId="${workerResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${workerRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtAgent" fromPartyId="${workerResult.partyId}"
    fromRoleTypeId="Worker" toPartyId="${vendorResult.partyId}"
    toRoleTypeId="VendorBillFrom" fromDate="${effectiveTime}"/>

<mantle.humanres.rate.RateAmount
    rateAmountId="${clientRateResult.rateAmountId}"
    rateTypeEnumId="RatpStandard" ratePurposeEnumId="RaprClient"
    timePeriodUomId="TF_hr" partyId="${workerResult.partyId}"
    emplPositionClassId="Programmer" fromDate="2010-02-03 00:00:00"
    rateAmount="60.00" rateCurrencyUomId="USD"/>
<mantle.humanres.rate.RateAmount
    rateAmountId="${vendorRateResult.rateAmountId}"
    rateTypeEnumId="RatpStandard" ratePurposeEnumId="RaprVendor"
    timePeriodUomId="TF_hr" partyId="${workerResult.partyId}"
    emplPositionClassId="Programmer" fromDate="2010-02-03 00:00:00"
    rateAmount="40.00" rateCurrencyUomId="USD"/>
```

Client

Below is the code that create the client (`CustomerBillTo`) `Organization`, and a `Person` that is a representative (with a `PartyRelationship` of type `PrtRepresentative`) of the client along with contact information, etc.

```
Map clientResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Organization")
    .parameters([roleTypeId:'CustomerBillTo',
        organizationName:'Test Client']).call()
Map clientCiResult = ec.service.sync()
    .name("mantle.party.ContactServices.store#PartyContactInfo")
    .parameters([partyId:clientResult.partyId,
        postalContactMechPurposeId:'PostalBilling',
```

```
        telecomContactMechPurposeId:'PhoneBilling',
        emailContactMechPurposeId:'EmailBilling', countryGeoId:'USA',
        address1:'1350 E. Flamingo Rd.', unitNumber:'1234',
        city:'Las Vegas', stateProvinceGeoId:'USA_NV', postalCode:'89119',
        postalCodeExt:'5263', countryCode:'+1', areaCode:'702',
        contactNumber:'123-4567',   emailAddress:'client.ap@test.com'])
    .call()

Map clientRepResult = ec.service.sync()
    .name("mantle.party.PartyServices.create#Account")
    .parameters([firstName:'Client', lastName:'TestRep',
        emailAddress:'client.rep@test.com', username:'client.rep',
        newPassword:'moqui1!', newPasswordVerify:'moqui1!',
        loginAfterCreate:'false']).call()
Map repRelResult = ec.service.sync()
    .name("create#mantle.party.PartyRelationship")
    .parameters([relationshipTypeEnumId:'PrtRepresentative',
        fromPartyId:clientRepResult.partyId,
        fromRoleTypeId:'ClientBilling', toPartyId:clientResult.partyId,
        toRoleTypeId:'CustomerBillTo', fromDate:ec.user.nowTimestamp])
    .call()
```

Here are the records for the client, contact info, and client representative:

```
<mantle.party.Party partyId="${clientResult.partyId}"
    partyTypeEnumId="PtyOrganization"/>
<mantle.party.Organization partyId="${clientResult.partyId}"
    organizationName="Test Client"/>
<mantle.party.PartyRole partyId="${clientResult.partyId}"
    roleTypeId="CustomerBillTo"/>

<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.postalContactMechId}"
    contactMechTypeEnumId="CmtPostalAddress"/>
<mantle.party.contact.PostalAddress
    contactMechId="${clientCiResult.postalContactMechId}"
    address1="1350 E. Flamingo Rd." unitNumber="1234" city="Las Vegas"
    stateProvinceGeoId="USA_NV" countryGeoId="USA" postalCode="89119"
    postalCodeExt="5263"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.postalContactMechId}"
    contactMechPurposeId="PostalBilling" fromDate="${effectiveTime}"/>
<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.telecomContactMechId}"
    contactMechTypeEnumId="CmtTelecomNumber"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.telecomContactMechId}"
    contactMechPurposeId="PhoneBilling" fromDate="${effectiveTime}"/>
<mantle.party.contact.TelecomNumber
    contactMechId="${clientCiResult.telecomContactMechId}" countryCode="+1"
    areaCode="702" contactNumber="123-4567"/>
```

```
<mantle.party.contact.ContactMech
    contactMechId="${clientCiResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="client.ap@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${clientResult.partyId}"
    contactMechId="${clientCiResult.emailContactMechId}"
    contactMechPurposeId="EmailBilling" fromDate="${effectiveTime}"/>

<mantle.party.Party partyId="${clientRepResult.partyId}"
    partyTypeEnumId="PtyPerson" disabled="N"/>
<mantle.party.Person partyId="${clientRepResult.partyId}"
    firstName="Client" lastName="TestRep"/>
<moqui.security.UserAccount userId="${clientRepResult.userId}"
    username="client.rep" userFullName="Client TestRep"
    passwordHashType="SHA-256" passwordSetDate="${effectiveTime}"
    disabled="N" requirePasswordChange="N"
    emailAddress="client.rep@test.com"
    partyId="${clientRepResult.partyId}" passwordSalt="{.rqlPt8x"
    currentPassword="32ce60c14d9e72c1fb17938ede30fe9de04390409cce7310743
        c2716a2c7bf89"/>
<mantle.party.contact.ContactMech
    contactMechId="${clientRepResult.emailContactMechId}"
    contactMechTypeEnumId="CmtEmailAddress"
    infoString="client.rep@test.com"/>
<mantle.party.contact.PartyContactMech partyId="${clientRepResult.partyId}"
    contactMechId="${clientRepResult.emailContactMechId}"
    contactMechPurposeId="EmailPrimary" fromDate="${effectiveTime}"/>
<mantle.party.PartyRelationship
    partyRelationshipId="${repRelResult.partyRelationshipId}"
    relationshipTypeEnumId="PrtRepresentative"
    fromPartyId="${clientRepResult.partyId}" fromRoleTypeId="ClientBilling"
    toPartyId="${clientResult.partyId}" toRoleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>
```

Project and Milestone

This code creates a `Project` type `WorkEffort` with the client and vendor set, and assigns the worker `Person` created above as a `Worker`. Note that the `WorkEffortParty` record for the assignment has a **`emplPositionClassId`** of `Programmer` which is used for looking up the `RateAmount` record create above for the billing rate.

```
ec.service.sync().name("mantle.work.ProjectServices.create#Project")
    .parameters([workEffortId:'TEST', workEffortName:'Test Project',
        statusId:'WeInProgress', clientPartyId:clientResult.partyId,
        vendorPartyId:vendorResult.partyId]).call()
ec.service.sync().name("create#mantle.work.effort.WorkEffortParty")
    .parameters([workEffortId:'TEST', partyId:workerResult.partyId,
        roleTypeId:'Worker', emplPositionClassId:'Programmer',
        fromDate:'2013-11-01', statusId:'PRTYASGN_ASSIGNED']).call()
```

Here are the records for the project and the client (`CustomerBillTo`), vendor (`VendorBillFrom`) and worker (`Worker`) associated with it:

```
<mantle.work.effort.WorkEffort workEffortId="TEST"
    workEffortTypeEnumId="WetProject" statusId="WeInProgress"
    workEffortName="Test Project"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="EX_JOHN_DOE" roleTypeId="Manager" fromDate="${effectiveTime}"
    statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${clientResult.partyId}" roleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${vendorResult.partyId}" roleTypeId="VendorBillFrom"
    fromDate="${effectiveTime}"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="1383282000000" statusId="PRTYASGN_ASSIGNED"
    emplPositionClassId="Programmer"/>
```

The `WorkEffort.statusId` field is audit logged and here is the `EntityAuditLog` record for the status change from In Planning to In Progress:

```
<moqui.entity.EntityAuditLog auditHistorySeqId="55911"
    changedEntityName="mantle.work.effort.WorkEffort"
    changedFieldName="statusId" pkPrimaryValue="TEST"
    oldValueText="WeInPlanning" newValueText="WeInProgress"
    changedDate="${effectiveTime}" changedByUserId="EX_JOHN_DOE"/>
```

Next we'll create a couple of milestones for the project:

```
ec.service.sync().name("mantle.work.ProjectServices.create#Milestone")
    .parameters([rootWorkEffortId:'TEST', workEffortId:'TEST-MS-01',
        workEffortName:'Test Milestone 1', estimatedStartDate:'2013-11-01',
        estimatedCompletionDate:'2013-11-30', statusId:'WeInProgress'])
    .call()

ec.service.sync().name("mantle.work.ProjectServices.create#Milestone")
    .parameters([rootWorkEffortId:'TEST', workEffortId:'TEST-MS-02',
        workEffortName:'Test Milestone 2', estimatedStartDate:'2013-12-01',
        estimatedCompletionDate:'2013-12-31', statusId:'WeApproved'])
    .call()
```

Here are the milestone records. They are of type `WetMilestone` and are associated with the project using the `rootWorkEffortId` field.

```
<mantle.work.effort.WorkEffort workEffortId="TEST-MS-01"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetMilestone"
    statusId="WeInProgress" workEffortName="Test Milestone 1"
    estimatedStartDate="2013-11-01 00:00:00.0"
    estimatedCompletionDate="2013-11-30 00:00:00.0"/>
<mantle.work.effort.WorkEffort workEffortId="TEST-MS-02"
```

```
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetMilestone"
    statusId="WeApproved" workEffortName="Test Milestone 2"
    estimatedStartDate="2013-12-01 00:00:00.0"
    estimatedCompletionDate="2013-12-31 00:00:00.0"/>
```

Tasks and Time Entries

These service calls create 3 tasks with their own purpose, status, priority, estimated work time, etc:

```
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:null,
        workEffortId:'TEST-001', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1', estimatedCompletionDate:'2013-11-15',
        statusId:'WeApproved', assignToPartyId:workerResult.partyId,
        priority:3, purposeEnumId:'WepTask', estimatedWorkTime:10,
        description:'Will be really great when it\'s done'])
    .call()
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:'TEST-001',
        workEffortId:'TEST-001A', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1A',
        estimatedCompletionDate:'2013-11-15', statusId:'WeInPlanning',
        assignToPartyId:workerResult.partyId, priority:4,
        purposeEnumId:'WepNewFeature', estimatedWorkTime:2,
        description:'One piece of the puzzle'])
    .call()
ec.service.sync().name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST', parentWorkEffortId:'TEST-001',
        workEffortId:'TEST-001B', milestoneWorkEffortId:'TEST-MS-01',
        workEffortName:'Test Task 1B',
        estimatedCompletionDate:'2013-11-15', statusId:'WeApproved',
        assignToPartyId:workerResult.partyId, priority:4,
        purposeEnumId:'WepFix', estimatedWorkTime:2,
        description:'Broken piece of the puzzle'])
    .call()
```

Here are the records produced by those service calls including a WorkEffort record with a rootWorkEffortId connection it to the product and a WorkEffortAssoc record connecting it to the milestone. There is also a WorkEffortParty record for each task for the worker that is associated with it. Note that the **estimatedCompletionDate** is in the milliseconds since epoch format. This is the case for all entity XML exported data to avoid issues with time zones and such.

```
<mantle.work.effort.WorkEffort workEffortId="TEST-001"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetTask"
    purposeEnumId="WepTask" resolutionEnumId="WerUnresolved"
    statusId="WeApproved" priority="3" workEffortName="Test Task 1"
    description="Will be really great when it's done"
```

```
    estimatedCompletionDate="1384495200000" estimatedWorkTime="10"
    remainingWorkTime="10" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001A"
    parentWorkEffortId="TEST-001" rootWorkEffortId="TEST"
    workEffortTypeEnumId="WetTask" purposeEnumId="WepNewFeature"
    resolutionEnumId="WerUnresolved" statusId="WeInPlanning" priority="4"
    workEffortName="Test Task 1A" description="One piece of the puzzle"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001A"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001A" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001B"
    parentWorkEffortId="TEST-001" rootWorkEffortId="TEST"
    workEffortTypeEnumId="WetTask" purposeEnumId="WepFix"
    resolutionEnumId="WerUnresolved" statusId="WeApproved" priority="4"
    workEffortName="Test Task 1B" description="Broken piece of the puzzle"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty workEffortId="TEST-001B"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.work.effort.WorkEffortAssoc workEffortId="TEST-MS-01"
    toWorkEffortId="TEST-001B" workEffortAssocTypeEnumId="WeatMilestone"
    fromDate="${effectiveTime}"/>
```

This code first updates the status of the 3 tasks to In Progress.

Then there are 3 different examples of recording time worked on a task for common options that a user recording time might use. The first specifies the **hours** worked and the **remainingWorkTime**, and the from and thru dates for the TimeEntry are calculated based on the **thruDate** being set to the current date/time. The second call has **hours** worked and **breakHours**, and again no from/thru dates and in this case the **thruDate** is the current date/time and the **fromDate** is the **thruDate** minus (**hours** + **breakHours**). In the third call it specifies the **breakHours**, the **fromDate** and the **thruDate** and the **hours** are calculated based on that.

Finally it sets the status of all 3 tasks to Completed.

```
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001', statusId:'WeInProgress']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001A', statusId:'WeInProgress']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001B', statusId:'WeInProgress']).call()


ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:3, hours:6,
        fromDate:null, thruDate:null, breakHours:null]).call()

ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001A', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:1, hours:1.5,
        fromDate:null, thruDate:null, breakHours:0.5]).call()

ec.service.sync().name("mantle.work.TaskServices.add#TaskTime")
    .parameters([workEffortId:'TEST-001B', partyId:workerResult.partyId,
        rateTypeEnumId:'RatpStandard', remainingWorkTime:0.5, hours:null,
        fromDate:"2013-11-03 12:00:00", thruDate:"2013-11-03 15:00:00",
        breakHours:1]).call()

ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001A', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:'TEST-001B', statusId:'WeComplete',
        resolutionEnumId:'WerCompleted']).call()
```

Below are the updated `WorkEffort` records with the fields that were changed including resolution, status, and remaining and actual work times. Also below are the `TimeEntry` records for each task. Note that the **`rateAmountId`** field gets filled in automatically based on the most relevant `RateAmount` record for the worker `Party`. That rate is used for displaying the rate and total cost for the `TimeEntry`, and as the amount on the InvoiceItem records later on when they are created for worker and client (as shown in the 2 invoice and payment sections below).

```
<mantle.work.effort.WorkEffort workEffortId="TEST-001"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="10" remainingWorkTime="3" actualWorkTime="6"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="${effectiveThruDate.time-(6*60*60*1000)}"
```

```
    thruDate="${effectiveThruDate.time}" hours="6"
    workEffortId="TEST-001"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001A"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="2" remainingWorkTime="1" actualWorkTime="1.5"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="${effectiveThruDate.time-(2*60*60*1000)}"
    thruDate="${effectiveThruDate.time}" hours="1.5" breakHours="0.5"
    workEffortId="TEST-001A"/>

<mantle.work.effort.WorkEffort workEffortId="TEST-001B"
    resolutionEnumId="WerCompleted" statusId="WeComplete"
    estimatedWorkTime="2" remainingWorkTime="0.5" actualWorkTime="2"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    partyId="${workerResult.partyId}" rateTypeEnumId="RatpStandard"
    rateAmountId="${clientRateResult.rateAmountId}"
    vendorRateAmountId="${vendorRateResult.rateAmountId}"
    fromDate="1383501600000" thruDate="1383512400000" hours="2"
    breakHours="1" workEffortId="TEST-001B"/>
```

Request and Task for Request

This code shows how to create a support request assigned to the worker, update its status from submitted to reviewed, create a task for the request, complete the task, and then complete the request.

```
Map createReqResult = ec.service.sync()
    .name("mantle.request.RequestServices.create#Request")
    .parameters([clientPartyId:clientResult.partyId,
        assignToPartyId:workerResult.partyId, requestName:'Test Request 1',
        description:'Description of Test Request 1', priority:7,
        requestTypeEnumId:'RqtSupport', statusId:'ReqSubmitted',
        responseRequiredDate:'2013-11-15 15:00:00']).call()
ec.service.sync().name("mantle.request.RequestServices.update#Request")
    .parameters([requestId:createReqResult.requestId,
        statusId:'ReqReviewed']).call()

Map createReqTskResult = ec.service.sync()
    .name("mantle.work.TaskServices.create#Task")
    .parameters([rootWorkEffortId:'TEST',
        workEffortName:'Test Request 1 Task',
        estimatedCompletionDate:'2013-11-15', statusId:'WeApproved',
        assignToPartyId:workerResult.partyId, priority:7,
        purposeEnumId:'WepTask', estimatedWorkTime:2,
        description:'']).call()
```

```
ec.service.sync().name("create#mantle.request.RequestWorkEffort")
    .parameters([workEffortId:createReqTskResult.workEffortId,
        requestId:createReqResult.requestId]).call()
ec.service.sync().name("mantle.work.TaskServices.update#Task")
    .parameters([workEffortId:createReqTskResult.workEffortId,
        statusId:'WeComplete', resolutionEnumId:'WerCompleted']).call()

ec.service.sync().name("mantle.request.RequestServices.update#Request")
    .parameters([requestId:createReqResult.requestId,
        statusId:'ReqCompleted']).call()
```

Here is the `Request` record and the `RequestParty` records to associate it with worker and client (customer). Here is also the task `WorkEffort`, the `WorkEffortParty` record for the worker, and the `RequestWorkEffort` record to associate it with the `Request`.

```
<mantle.request.Request requestId="${createReqResult.requestId}"
    requestTypeEnumId="RqtSupport" statusId="ReqCompleted"
    requestName="Test Request 1"
    description="Description of Test Request 1" priority="7"
    responseRequiredDate="1384549200000"
    requestResolutionEnumId="RrUnresolved" filedByPartyId="EX_JOHN_DOE"/>
<mantle.request.RequestParty requestId="${createReqResult.requestId}"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}"/>
<mantle.request.RequestParty requestId="${createReqResult.requestId}"
    partyId="${clientResult.partyId}" roleTypeId="CustomerBillTo"
    fromDate="${effectiveTime}"/>

<mantle.work.effort.WorkEffort
    workEffortId="${createReqTskResult.workEffortId}"
    rootWorkEffortId="TEST" workEffortTypeEnumId="WetTask"
    purposeEnumId="WepTask" resolutionEnumId="WerCompleted"
    statusId="WeComplete" priority="7" workEffortName="Test Request 1 Task"
    estimatedCompletionDate="1384495200000" estimatedWorkTime="2"
    remainingWorkTime="2" timeUomId="TF_hr"/>
<mantle.work.effort.WorkEffortParty
    workEffortId="${createReqTskResult.workEffortId}"
    partyId="${workerResult.partyId}" roleTypeId="Worker"
    fromDate="${effectiveTime}" statusId="PRTYASGN_ASSIGNED"/>
<mantle.request.RequestWorkEffort requestId="${createReqResult.requestId}"
    workEffortId="${createReqTskResult.workEffortId}"/>
```

Worker Invoice and Payment

The `Invoice` from the worker to the services vendor (the internal organization running the system) has both expenses and time entries. The **`create#ProjectExpenseInvoice`** service gets most of the settings for the `Invoice` (including the vendor, bill-to, party) from the project `WorkEffort` (ID: `TEST`) and specifies the worker as the fromPartyId.

Once the invoice is created the next two service calls add expense invoice items and then call the **create#ProjectInvoiceItems** service to add invoice items for all time entries for the worker party in the TEST project, with **ratePurposeEnumId** of RaprVendor so that the rates and other details are for a worker to vendor invoice (as opposed to a vendor to client invoice). Next we mark the invoice as Received. This is something that would be done by a representative of the vendor organization, i.e., the bill-to party for the invoice.

The last service call, to **create#InvoicePayment**, records a delivered check payment for the invoice.

```
expInvResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectExpenseInvoice")
    .parameters([workEffortId:'TEST', fromPartyId:workerResult.partyId])
    .call()
ec.service.sync().name("create#mantle.account.invoice.InvoiceItem")
    .parameters([invoiceId:expInvResult.invoiceId,
        itemTypeEnumId:'ItemExpTravAir', description:'United SFO-LAX',
        itemDate:'2013-11-02', quantity:1, amount:345.67]).call()
ec.service.sync().name("create#mantle.account.invoice.InvoiceItem")
    .parameters([invoiceId:expInvResult.invoiceId,
        itemTypeEnumId:'ItemExpTravLodging',
        description:'Fleabag Inn 2 nights', itemDate:'2013-11-04',
        quantity:1, amount:123.45]).call()

ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectInvoiceItems")
    .parameters([invoiceId:expInvResult.invoiceId,
        workerPartyId:workerResult.partyId, ratePurposeEnumId:'RaprVendor',
        workEffortId:'TEST',
        thruDate:new Timestamp(effectiveTime + 1)]).call()

ec.service.sync().name("update#mantle.account.invoice.Invoice")
    .parameters([invoiceId:expInvResult.invoiceId,
        statusId:'InvoiceReceived']).call()

Map expPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.create#InvoicePayment")
    .parameters([invoiceId:expInvResult.invoiceId,
        statusId:'PmntDelivered', amount:'849.12',
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        effectiveDate:'2013-11-10 12:00:00', paymentRefNum:'1234',
        comments:'Delivered by Fedex']).call()
```

Here are the records created for the invoice, including the expense items and three time entry items (one for each of the task time entries):

```
<mantle.account.invoice.Invoice invoiceId="${expInvResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="${workerResult.partyId}"
    toPartyId="${vendorResult.partyId}" statusId="InvoicePmtSent"
    invoiceDate="${effectiveTime}" currencyUomId="USD"/>
```

```
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="01" itemTypeEnumId="ItemExpTravAir" quantity="1"
    amount="345.67" description="United SFO-LAX" itemDate="1383368400000"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="02" itemTypeEnumId="ItemExpTravLodging" quantity="1"
    amount="123.45" description="Fleabag Inn 2 nights"
    itemDate="1383544800000"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="03" itemTypeEnumId="ItemTimeEntry" quantity="6"
    amount="40" itemDate="${effectiveThruDate.time-(6*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="03"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="04" itemTypeEnumId="ItemTimeEntry" quantity="1.5"
    amount="40" itemDate="${effectiveThruDate.time-(2*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="04"/>
<mantle.account.invoice.InvoiceItem invoiceId="${expInvResult.invoiceId}"
    invoiceItemSeqId="05" itemTypeEnumId="ItemTimeEntry" quantity="2"
    amount="40" itemDate="1383501600000"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    vendorInvoiceId="${expInvResult.invoiceId}"
    vendorInvoiceItemSeqId="05"/>
```

This is the accounting transaction for the GL posting of the invoice with one entry for each invoice item, and the balancing entry to the accounts payable account:

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55900"
    acctgTransTypeEnumId="AttPurchaseInvoice"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${workerResult.partyId}"
    invoiceId="${expInvResult.invoiceId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="345.67"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="123.45"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="03" debitCreditFlag="D" amount="240"
    glAccountId="550000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="04" debitCreditFlag="D" amount="60"
    glAccountId="550000" reconcileStatusId="AES_NOT_RECONCILED"
```

```
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="05" debitCreditFlag="D" amount="80"
    glAccountId="550000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="05"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55900"
    acctgTransEntrySeqId="06" debitCreditFlag="C" amount="849.12"
    glAccountTypeEnumId="ACCOUNTS_PAYABLE" glAccountId="210000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
<mantle.work.effort.WorkEffortInvoice invoiceId="${expInvResult.invoiceId}"
    workEffortId="TEST"/>
```

Here is the payment record for the check from the vendor (internal organization) to the worker, the payment application to apply it to the invoice, and the accounting transition for the payment:

```
<mantle.account.payment.Payment paymentId="${expPmtResult.paymentId}"
    paymentTypeEnumId="PtInvoicePayment"
    fromPartyId="${vendorResult.partyId}"
    toPartyId="${workerResult.partyId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" statusId="PmntDelivered"
    effectiveDate="1384106400000" paymentRefNum="1234"
    comments="Delivered by Fedex" amount="849.12" amountUomId="USD"/>
<mantle.account.payment.PaymentApplication
    paymentApplicationId="${expPmtResult.paymentApplicationId}"
    paymentId="${expPmtResult.paymentId}"
    invoiceId="${expInvResult.invoiceId}" amountApplied="849.12"
    appliedDate="${effectiveTime}"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55901"
    acctgTransTypeEnumId="AttOutgoingPayment"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${workerResult.partyId}"
    paymentId="${expPmtResult.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55901"
    acctgTransEntrySeqId="01" debitCreditFlag="D" amount="849.12"
    glAccountId="210000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55901"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="849.12"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```

Client Invoice and Payment

With everything setup already, including the worker expenses and project settings, call the **`create#ProjectInvoiceItems`** service to add invoice items for all time entries for the

worker party in the `TEST` project, with **`ratePurposeEnumId`** of `RaprClient` so that the rates and other details are for a vendor to client invoice (as opposed to a worker to vendor invoice). The **`thruDate`** passed to the service tells it to get all expenses and time entries for the project that are not yet billed up to that date/time. Next we mark the invoice as Finalized, which triggers GL posting for the invoice.

```
clientInvResult = ec.service.sync()
    .name("mantle.account.InvoiceServices.create#ProjectInvoiceItems")
    .parameters([ratePurposeEnumId:'RaprClient', workEffortId:'TEST',
        thruDate:new Timestamp(effectiveTime + 1)]).call()
ec.service.sync().name("update#mantle.account.invoice.Invoice")
    .parameters([invoiceId:clientInvResult.invoiceId,
        statusId:'InvoiceFinalized']).call()
```

Below are the records for the vendor to client invoice with the time entry and expense invoice items, and `InvoiceItemAssoc` records to associate the expense items on this vendor to client invoice with the expense items as originally recorded on the worker to vendor invoice (which is how expenses are recorded, and this is how they are marked as billed through).

```
<mantle.account.invoice.Invoice invoiceId="${clientInvResult.invoiceId}"
    invoiceTypeEnumId="InvoiceSales" fromPartyId="${vendorResult.partyId}"
    toPartyId="${clientResult.partyId}" statusId="InvoiceFinalized"
    invoiceDate="${effectiveTime}" currencyUomId="USD"
    description="Invoice for projectTest Project [TEST] "/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="01"
    itemTypeEnumId="ItemTimeEntry" quantity="6" amount="60"
    itemDate="${effectiveThruDate.time-(6*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55900"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="01"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="02"
    itemTypeEnumId="ItemTimeEntry" quantity="1.5" amount="60"
    itemDate="${effectiveThruDate.time-(2*60*60*1000)}"/>
<mantle.work.time.TimeEntry timeEntryId="55901"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="02"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="03"
    itemTypeEnumId="ItemTimeEntry" quantity="2" amount="60"
    itemDate="1383501600000"/>
<mantle.work.time.TimeEntry timeEntryId="55902"
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="03"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="04"
    itemTypeEnumId="ItemExpTravAir" quantity="1" amount="345.67"
    description="United SFO-LAX" itemDate="1383368400000"/>
<mantle.account.invoice.InvoiceItemAssoc invoiceItemAssocId="55900"
    invoiceId="${expInvResult.invoiceId}" invoiceItemSeqId="01"
    toInvoiceId="${clientInvResult.invoiceId}" toInvoiceItemSeqId="04"
```

```
    invoiceItemAssocTypeEnumId="IiatBillThrough" quantity="1"
    amount="345.67"/>
<mantle.account.invoice.InvoiceItem
    invoiceId="${clientInvResult.invoiceId}" invoiceItemSeqId="05"
    itemTypeEnumId="ItemExpTravLodging" quantity="1" amount="123.45"
    description="Fleabag Inn 2 nights" itemDate="1383544800000"/>
<mantle.account.invoice.InvoiceItemAssoc invoiceItemAssocId="55901"
    invoiceId="${expInvResult.invoiceId}" invoiceItemSeqId="02"
    toInvoiceId="${clientInvResult.invoiceId}" toInvoiceItemSeqId="05"
    invoiceItemAssocTypeEnumId="IiatBillThrough" quantity="1"
    amount="123.45"/>
```

These are the records for the accounting transaction posted to the GL for the invoice, with one entry for each invoice item and the balancing entry in the accounts receivable account. Note the different **glAccountId** values for the time entry and expense entries.

```
<mantle.ledger.transaction.AcctgTrans acctgTransId="55902"
    acctgTransTypeEnumId="AttSalesInvoice"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${clientResult.partyId}"
    invoiceId="${clientInvResult.invoiceId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="360"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="01"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="02" debitCreditFlag="C" amount="90"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="02"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="03" debitCreditFlag="C" amount="120"
    glAccountId="402000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="03"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="04" debitCreditFlag="C" amount="345.67"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="04"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="05" debitCreditFlag="C" amount="123.45"
    glAccountId="681000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N" invoiceItemSeqId="05"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55902"
    acctgTransEntrySeqId="06" debitCreditFlag="D" amount="1,039.12"
    glAccountTypeEnumId="ACCOUNTS_RECEIVABLE" glAccountId="120000"
    reconcileStatusId="AES_NOT_RECONCILED" isSummary="N"/>
```

This is the service call to record a delivered payment by company check for the invoice, which automatically makes it from the client to the vendor:

```
Map clientPmtResult = ec.service.sync()
    .name("mantle.account.PaymentServices.create#InvoicePayment")
    .parameters([invoiceId:clientInvResult.invoiceId,
        statusId:'PmntDelivered', amount:1039.12,
        paymentMethodTypeEnumId:'PmtCompanyCheck',
        effectiveDate:'2013-11-12 12:00:00', paymentRefNum:'54321'])
    .call()
```

The first record here shows the status update on the invoice to payment received. Then we have the payment record and the application of the payment to the invoice. After that is the accounting transaction to post the payment to the general ledger.

```
<mantle.account.invoice.Invoice invoiceId="${clientInvResult.invoiceId}"
    statusId="InvoicePmtRecvd"/>
<mantle.account.payment.Payment paymentId="${clientPmtResult.paymentId}"
    paymentTypeEnumId="PtInvoicePayment"
    fromPartyId="${clientResult.partyId}"
    toPartyId="${vendorResult.partyId}"
    paymentMethodTypeEnumId="PmtCompanyCheck" statusId="PmntDelivered"
    effectiveDate="1384279200000" paymentRefNum="54321" amount="1,039.12"
    amountUomId="USD"/>
<mantle.account.payment.PaymentApplication
    paymentApplicationId="${clientPmtResult.paymentApplicationId}"
    paymentId="${clientPmtResult.paymentId}"
    invoiceId="${clientInvResult.invoiceId}" amountApplied="1,039.12"
    appliedDate="${effectiveTime}"/>

<mantle.ledger.transaction.AcctgTrans acctgTransId="55903"
    acctgTransTypeEnumId="AttIncomingPayment"
    organizationPartyId="${vendorResult.partyId}"
    transactionDate="${effectiveTime}" isPosted="Y"
    postedDate="${effectiveTime}" glFiscalTypeEnumId="GLFT_ACTUAL"
    amountUomId="USD" otherPartyId="${clientResult.partyId}"
    paymentId="${clientPmtResult.paymentId}"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55903"
    acctgTransEntrySeqId="01" debitCreditFlag="C" amount="1,039.12"
    glAccountId="120000" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
<mantle.ledger.transaction.AcctgTransEntry acctgTransId="55903"
    acctgTransEntrySeqId="02" debitCreditFlag="D" amount="1,039.12"
    glAccountId="111100" reconcileStatusId="AES_NOT_RECONCILED"
    isSummary="N"/>
```