Making Apps with Moqui

Holistic Enterprise Applications Made Easy

by David E. Jones

NOTE: This is an early draft and is meant for review purposes only. Please do not distribute.

Table of Contents

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 help manage processes and automate information management.

Like any craftsman an application developer needs a good set of tools, and my quest for a 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 very difficult and expensive to customize and maintain after initial implementation. In a word, it was 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 a lot of 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 concepts 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.

In addition to 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 concepts 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 more and more difficult to change fundamental aspects of the system.

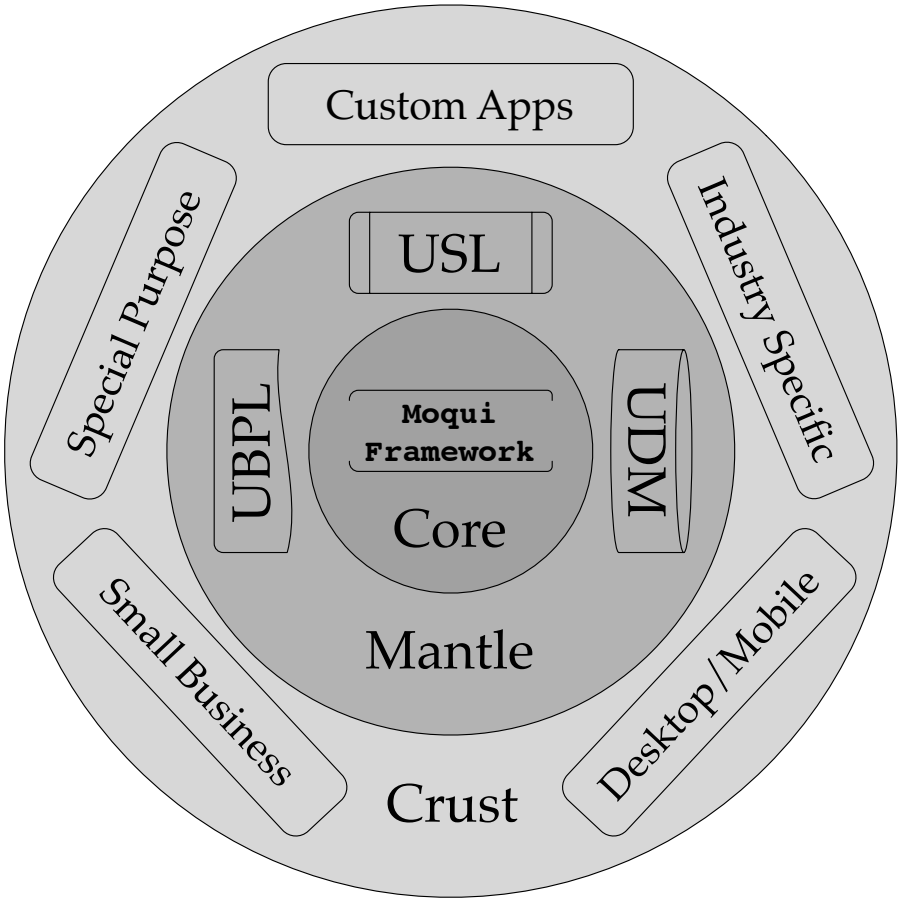
I kept a list of dozens of great ideas that constituted major changes to improve and expand the framework. As the list got longer I knew a different approach would be necessary to enter the next phase of my aforementioned quest for the best toolset possible. The result was the birth of the Moqui Framework 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 provide a reference over 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 around a common framework and universal business artifacts. The central packages (in the Core and Mantle) are intentionally 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**: facilitate efficient, secure and flexible development
- **Mantle Business Artifacts**: universal business artifacts to act as a foundation for your diverse business applications, including:
  - Universal Business Process Library (UBPL)
  - Universal Data Model (UDM)
  - Universal Service Library (USL)
- **Crust**: themes, tool integrations, and applications for different industries, company sizes, business areas, etc

The focus of this books is the Moqui Framework, but the last chapter is an overview of the 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 it 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 design and written by the very person who founded that project. Many of the concepts 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 only about 15% 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 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 actually 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, ie 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 a 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 actually 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 actually 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 of 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.

Note that this content collaboration tool is generally separate from your code repository, although putting this 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 jonesde/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)

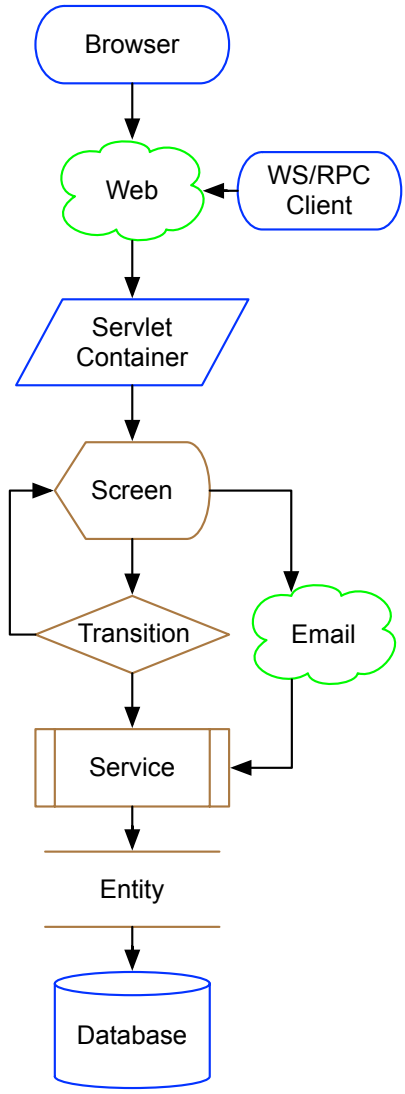
My preferred IDE these days is IntelliJ IDEA from JetBrains. The free Community Edition has excellent XML and Groovy support. For HTML, CSS, JavaScript, and FreeMarker to go beyond a simple text editor you'll have to pay for the Ultimate Edition. I implemented most of Moqui, including the complex FreeMarker 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 really amazingly 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 by all means use what makes you happy and productive.

The Moqui Framework itself is built using Gradle (1.0 or later). 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.

Transitions are used to process input (and not to prepare data for presentation), which is separated from the screen actions which are used to prepare data for presentation (and not to process input). If there is a transition name the service or actions of the transition will be run, a response to the transition selected (based on conditions and whether or not 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 map according to 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 JDK version 6 or later. To build the framework from source you'll need Gradle version 1.0 (final) or later.

You can download Moqui Framework from SourceForge at:

https://sourceforge.net/projects/moqui/files/

Select the folder for the most recent version and then choose either the binary or source distribution archive. The binary release of the framework is named "moqui-<version>.zip" and the source release is named "moqui-<version>-src.zip".

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

https://github.com/jonesde/moqui

Similarly the Mantle Business Artifacts are available on GitHub here:

https://github.com/jonesde/mantle

While you can download Mantle separately through GitHub, there is also a Moqui Framework plus Mantle archive available on SourceForge.

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 project one.

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 Gradle 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 WAR	`$ 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).

3. Framework Tools and Configuration

What follows is an overview 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/jonesde/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 Execution Context 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 basically a hash or map with name/value entries 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 “parent” variable spaces are visible.

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:

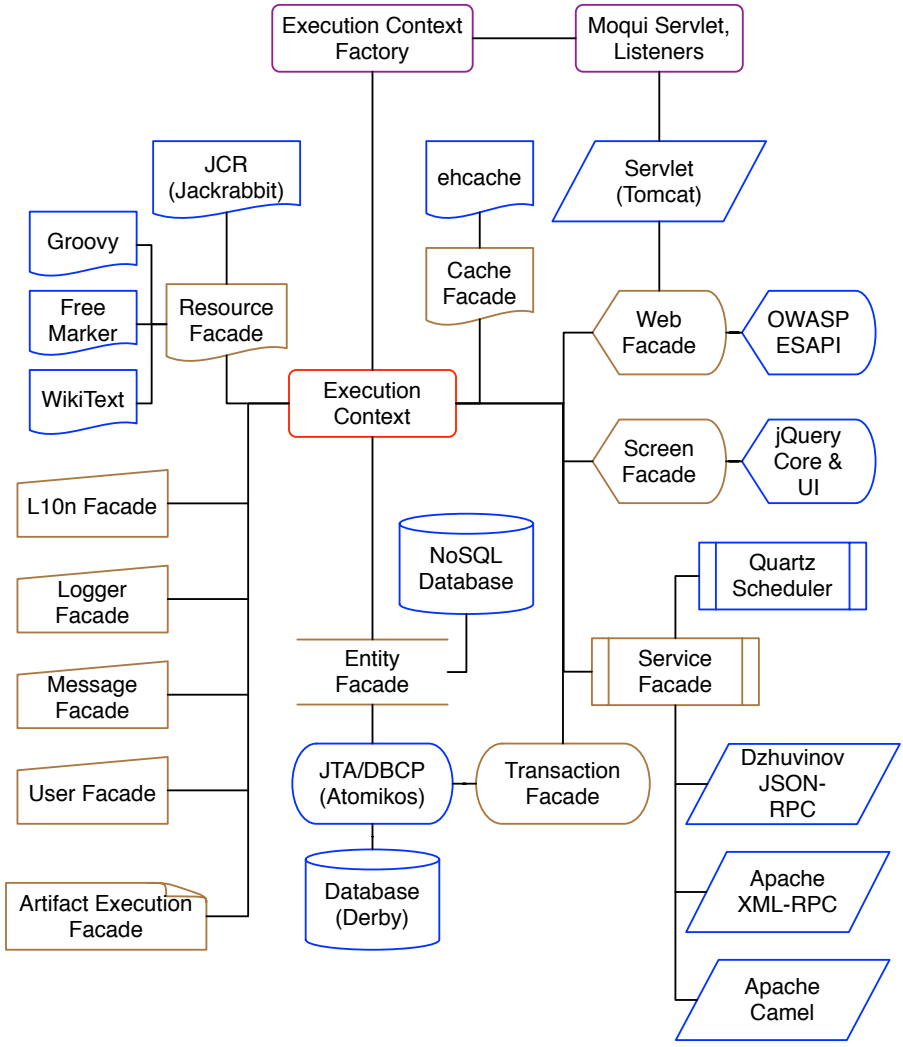
- refer to the current variable context: `ec.context`
- refer to the “exampleId” field from the context: `ec.context.exampleId`
- set the exampleId to “foo”: `ec.context.exampleId = “foo”`
- for inline scripts you can also just do: `exampleId = “foo”`

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 of 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 (Tomcat)
Groovy
Free Marker
WikiText
Resource Facade
Cache Facade
Web Facade
OWASP ESAPI
Execution Context
Screen Facade
jQuery Core & UI
L10n Facade
NoSQL Database
Quartz Scheduler
Logger Facade
Entity Facade
Service Facade
Message Facade
JTA/DBCP (Atomikos)
Transaction Facade
Dzhuvinov JSON-RPC
User Facade
Database (Derby)
Apache XML-RPC
Artifact Execution Facade
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 Execu- tionContext 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. In addition to supporting basic get/put/remove/etc operations you can get statistics for each cache, and also modify cache properties such as timeouts, size limit, and eviction algorithm. The default Cache Facade implementation is basically just a wrapper around ehCache, and beyond the cache-facade configuration in the moqui-conf 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 runScriptInCurrent- Context method), and compiled FreeMarker (FTL) templates also by location (for the renderTemplateIn- CurrentContext 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 or not 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 de- fault implementations exist for the following protocols/schemes: http, https, file, ftp, jar, classpath, component, and content (JCR, ie 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 sub- screen 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 (ie 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 file. You can create your own templates that override the default macros, or simply ignore them altogether, and configure them in the moqui-conf 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 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 just 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 actually 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 actually calling the services and configuration in the moqui-conf 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 Dzhuvinov JSON-RPC 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 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/MoquiRoot/rpc.xml screen. The remote URL for these, if MoquiRoot.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 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.

For finding (querying) 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 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 file for the datasource.

Connection Pool and Database

The Entity Facade uses Atomikos TransactionsEssentials for XA-aware database connection pooling by default. To configure Atomikos use the `jta.properties` file. With configuration in the `entity-facade` element of the moqui-conf 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. 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, 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. If not it will create the table, indexes, and foreign keys (for related tables that already exist only) 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 Transaction Facade 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 Atomikos TransactionsEssentials library (also used for a connection pool by the Entity Facade). 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 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 be- ing 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.

4. 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 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 MoquiRoot which is located at "`runtime/component/webroot/screen/MoquiRoot.xml`".

For screens from your component to be available in a screen path under the MoquiRoot 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 MoquiRoot screen. There are two ways to do this (this does not include putting it in the MoquiRoot 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 "MoquiRoot" 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 file for its own default configuration.

5. Create Your First Component

Overview

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.

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:

```
$ ant load
$ ant 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 you component in place just start up Moqui (with "`$ ant run`" or the like).

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

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 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 fo the `hello.html` file is to treat it as screen sub-content.

To do this the `hello.html` file must by in a sub-directory with the same name as the screen, ie 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

With those done your screen XML file should look like:

```
<screen include-child-content="true">
    <widgets>
        <label type="h1" text="Hello world!"/>
        <subscreens-active/>
    </widgets>
</screen>
```

To see the content now you'll need to go to a different URL to tell Moqui that you want the `hello.html` file that is under the `tutorial` screen:

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

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 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 that basically elements that match entity names and attributes that map 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 tutorialId="TestOne"
            description="Test one description."/>
    <Tutorial tutorialId="TestTwo"
            description="Test two description."/>
</entity-facade-xml>
```

To load this just run "`$ ant 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>
    <transition name="findTutorial">
        <default-response url="."/></transition>

    <actions>
        <entity-find entity-name="Tutorial"
                list="tutorialList">
            <search-form-inputs/>
        </entity-find>
    </actions>

    <widgets>
        <form-list name="ListTutorials"
                list="tutorialList"
                transition="findTutorial">
            <auto-fields-entity entity-name="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` very simply just refers back 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 an 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` 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 at 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"
            field-type="find-display"/>
    <field name="description">
        <header-field show-order-by="true">
            <text-find hide-options="true"/>
        </header-field>
        <default-field><display/></default-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 basically the same as what was automatically generated using the `auto-fields-entity` element, and this is how you would do it explicitly.

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

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

Where did the `create#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"
                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` to generate edit fields. The last little detail is to declare a button to submit the form, and it's ready to go. Try it out and see the records appear in the list form that was part of the original screen.

Part 3

Custom Create Service

The `createTutorial` transition from our screen above used the implicit entity-auto service `create#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"
                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:

```
EntityValue tutorial = ec.entity.makeValue("Tutorial")
tutorial.setFields(context, true, null, null)
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 and specifically the doc for the ExecutionContext class which has links to the other major API interface pages.

6. Example Component Walkthrough

7. Data and Content

Resources, Content, and JCR

Accessing Content

Rendering Templates

Running Scripts

Database 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:

`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 same 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`, and `tenantcommon`.
  - `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 go anywhere in any entity definition XML file. This element has basically all 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 an 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 actually 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.

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 also 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 together 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 go along with 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).

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 with 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 records 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 StatusValidChange entity.

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"/>
```

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`). In general you should 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 same 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 actually 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 get 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 of 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.getEntity().makeFind("moqui.example.Example")
      .condition("exampleId", exampleId).one();
```

The `EntityFind` interface has methods on it for:

- conditions (both where and having)
- fields to select
- fields to order the results by
- whether or not to cache the results
- the offset and limit to pass to the datasource to limit results
- database options like distinct, and for update
- JDBC options like result set type and concurrency, fetch size, and maximum number of rows

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 `EntityFacade.getConditionFactory()` method, something like:

```
EntityConditionFactory ecf =
        ec.getEntity().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 a List of EntityValue objects (`list()` method)
- get an EntityListIterator to handle a larger set of results in smaller batches (with the 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. Actually, you can have it operate on multiple entities using the `EntityDynamicView` interface (get an instance using the `EntityFind.makeEntityDynamicView()` method).

To do a query across multiple entities joined together represented by a single entity name you can create a view entity using an XML definition that lives along side of normal entity definitions.

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 if 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.

8. Logic and Services

Service Definition

Service Implementation

Inline Service Logic

Java Class Methods

Service Scripts

Add Your Own Service Runner

Overview of XML Actions

9. User Interfaces

XML Screens

Screens in Moqui are organized in two ways:

- each screen exists in an 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 catch-all `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 a 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"
    location="component://example/screen/ExampleApp.xml"
    menu-title="Example" menu-index="8"/>
```

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
        condition="(ec.web.request.getHeader('User-
Agent')?:'').matches('.*iPad.*')"
        item="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.

Transitions

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 output. 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 go along 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:exampleType
EnumList])
        </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 basically 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.

TODO: describe other attributes in attlist.response (?)

RESTful Transitions

With the popularity of 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.

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 (ie 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 and Pre-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 (ie 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.

Widgets

The elements under the screen.widgets element are the actual 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" location="component://webroot/
screen/includes/Header.xsl-fo.ftl"
        no-boundary-comment="true"/>
</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`:
- `image`:
- `label`:

To structure screens use these widget elements:

- `section`:
- `section-iterate`:
- `container`:
- `container-panel`:
- `container-dialog`:
- `include-screen`:

TODO: more elaborate example with various of these elements used

Conditions and Fail-Widgets

Custom Elements and Macro Templates

XML Forms

Templates

PDF, CSV, XML and Other Screen Output

Standalone Screens

Screen Sub-Content

10. System Interfaces

XML and CSV Output

Web Services

XML-RPC and JSON-RPC

RESTful Interfaces

Simple Sending and Receiving JSON

Enterprise Integration with Apache Camel

Exporting All Records Related to One or More Records

Ad-hoc Data Exports

11. Security

Authentication

Internal Authentication

External Auth with Apache Shiro

Password Options

Login History

Simple Permissions

Artifact-Aware Authorization

Artifact Execution Stack and History

Artifact Authz

Artifact Tarpit

Audit Logging

12. The Tools Application

Auto Screens

Data View

Entity Tools

Data Edit

Data Export

Data Import

Speed Test

Localization

Service Runner

System Info

Artifact Statistics

Audit Log

Cache Statistics

Server Visits

13. Mantle Business Artifacts

Universal Data Model (UDM)

Universal Service Library (USL)

Universal Business Process Library (UBPL)