INTRODUCTION TO MOQUI FRAMEWORK

By David E. Jones

dejc@me.com

www.dejc.com

For Moqui Framework version 1.0 beta 2

Document version 1.0.1

Table of Contents

1. Introduction to the Introduction 1
   - About Moqui Framework 1
   - About This Document 1
   - A Note on Configuration 1
   - The Execution Context 1
2. Tour of a Request 2
   - Web Browser to Database 2
   - Web Services 2
   - Incoming and Outgoing Email 3
3. Tools, API and Libraries 4
   - Execution Context and Web Facade 4
     - Web Parameters Map 4
   - Factory, Servlet & Listeners 4
   - Resource and Cache Facades 6
   - Screen Facade 6
     - Screen Definition 7
     - Screen/Form Render Templates 7
   - Service Facade 7
     - Service Naming 8
     - Parameter Cleaning, Conversion and Validation 8
     - Quartz Scheduler 9
     - Web Services 9
   - Entity Facade 9
     - Multi-Tenant 10
     - Connection Pool and Database 10
     - Database Meta-Data 10
   - Transaction Facade 10
     - Transaction Manager (JTA) 11
   - Artifact Execution Facade 11
     - Artifact Authorization 11
     - Artifact Hit Tracking 11
   - User, L10n, Message, and Logger Facades 12

4. Create a Component (Add-on) 13
What is a Component? 13
Component Directory Structure 13
Install a Component 13
Load the Component 13
Mounting Screen(s) 13
Other Moqui Conf File Changes 14

1. Introduction to the Introduction

About 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.

About This Document

This document is a brief introduction to the functionality and organization of Moqui Framework. It is meant for a technical audience of application developers, the same audience that the framework itself is meant for. If you are not in that audience and are looking for general information about features of Moqui Framework there are better resources available (such as the moqui.org web site).

After reading this document you should know where in the API and tools to find specific functionality and have an understanding of how different parts of the framework interact.

A Note on Configuration

The configuration for Moqui Framework is done in a single XML file with the root element "moqui-conf", and this file will be referred to as the moqui-conf file throughout this document. When running Moqui two locations must be specified: the runtime directory, and the moqui-conf file. Moqui Framework has a default configuration file called MoquiDefaultConf.xml (in framework/impl/conf-default) and at run-time the moqui-conf file specified is "merged" with the MoquiDefaultConf.xml file to make the actual configuration to use.

There are three examples of runtime moqui-conf files: MoquiDevConf.xml, MoquiStagingConf.xml, and MoquiProductionConf.xml (all under runtime/conf/* directories). The default conf file and these three example runtime moqui-conf files are good examples and have various comments to help you get started with configuration of Moqui Framework.

The Execution Context

The central object for Moqui Framework is the Execution Context. Each web page request, service, etc will have an ExecutionContext created for it to act as its special context as it runs. All other parts of the Moqui Framework API can be accessed through the ExecutionContext object, generally in the variable space for screens, scripts, etc as simply "ec".

2. Tour of a Request

Web Browser to Database

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).

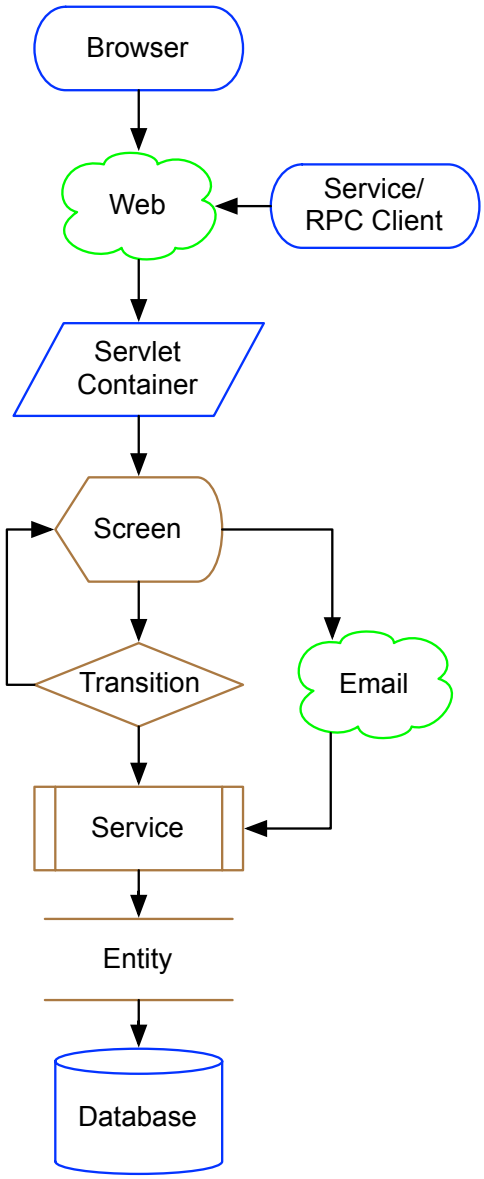

Figure 1: Basic Request Flow

Web Services

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 used 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.

3. Tools, API and Libraries

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.

For an ExecutionContext instance created for 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.

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

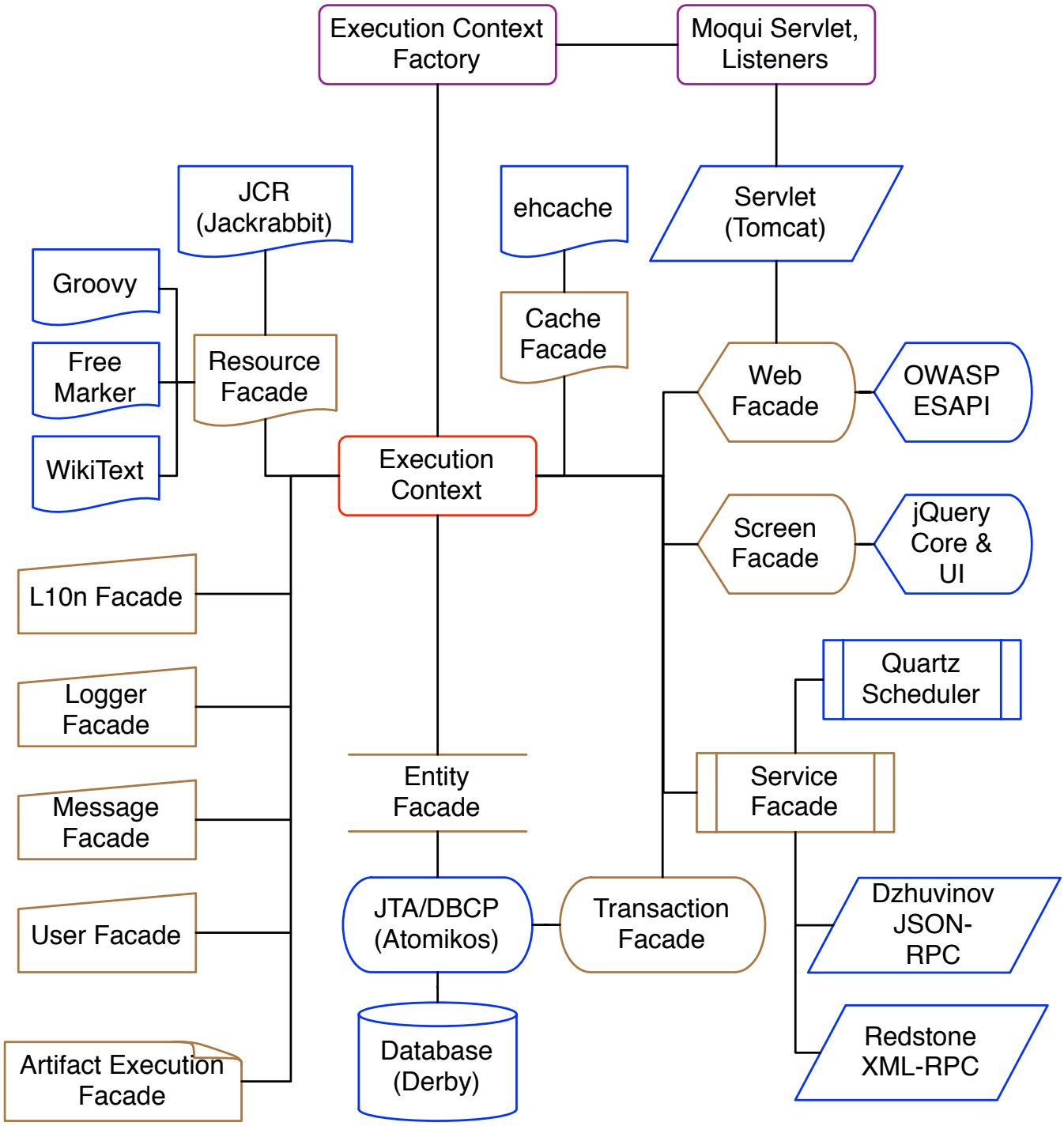

Figure 2: Tool Layout

Servlet Container. In either case the Moqui root webapp is loaded and the WEB-INF/web.xml file tells the servlet container to load the MoquiServlet, the MoquiSessionListener, and the MoquiContextListener. These are default classes included in the framework, and you can certainly create your own if you want to change the lifecycle of the ExecutionContextFactory and ExecutionContext.

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

Resource and Cache Facades

The Resource Facade is used to access and execute resource such as scripts, templates, and content. The Cache Facade is used to do general operations on caches, and to get a reference to a cache as an implementation of the Cache interface. 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 runScriptInCurrentContext method), and compiled FreeMarker (FTL) templates also by location (for the renderTemplateInCurrentContext method).

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

For more generic access to resources the getLocationReference() method returns an implementation of the ResourceReference interface. This can be used to read resource contents (for files and directories), and get information about them such as content/MIME type, last modified time, and whether 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 default implementations exist for the following protocols/schemes: http, https, file, ftp, jar, classpath, component, and content (JCR, 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.

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.

For more details about these see the example screens in the runtime/component/example component, and the annotations (inline documentation) in the xml-screen-1.0.xsd, xml-form-1.0.xsd, and xml-actions-1.0.xsd files.

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" and 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. 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.

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).

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/impl/conf-default directory that may be overridden on the classpath).

Web Services

For Web Services the Service Facade uses Redstone 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.

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.

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, and Oracle.

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 being run. For example if a screen calls a subscreen and that calls a service which does a find on an entity the stack will have (bottom to top) the first screen, then the second screen, then the service and then the entity.

Artifact Authorization

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

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

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

Artifact Hit Tracking

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

User, L10n, Message, and Logger Facades

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

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

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

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

4. Create a Component (Add-on)

What is a 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 make it easier to use in 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 in each Moqui webapp
- 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 in this directory and have the dual extension ".secas.xml"; Email ECA files must be in this directory and have the extension ".emecas.xml"

Install 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 to 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 "_NA_" in the userId field for it to apply to all users, or an actual userId for it to apply to just that user

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.

Other Moqui Conf File Changes

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.