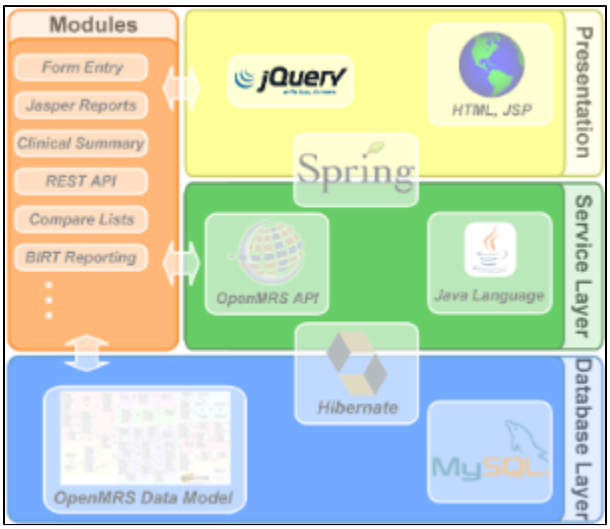
Technical Overview

Overview



OpenMRS has been designed to have a tiered architecture. The real strength of OpenMRS is in its robust and flexible data model. However, not all users want to have to know this data model inside and out. The API layer allows a developer to only have to know Java objects and read/save to them. This layer can be used in a standalone application or, as most people use it, in a J2EE web application.

Context

The backbone of OpenMRS is the core API. This API has methods for all of the basic functions like adding/updating a patient, adding/updating a concept, etc. These methods are provided in services. There are classes named **PatientService**, **ConceptService**, **EncounterService**, **ObsService**, etc. The Data Model groups the database tables into "domains." Each domain is a separate colored box. The breakdown of domains/tables is essentially a visual representation of the service separation.

The Context is a static class to allow the application to save on memory. Only one **PatientService** object, one **ConceptService** object, etc (and of course the associated DAO's) are instantiated. The Context's services are split into two categories: methods for the Services and for Users. The services are kept in the aptly named ServiceContext class. This is instantiated only once and is stored statically in the Context. The getter methods for the services simply pass through Context to the ServiceContext. The StaticContext properties are set via our Spring application Context and Dependency Injection. The UserContext contains methods for acting on users: authentication and authorization, logging in, logging out, etc. A different UserContext is instantiated for every user accessing the system. The "current" UserContext is stored on the current thread. When that user is done, the UserContext is taken off of the thread and put into the user's session variable (in the case of the webapp). When the user accesses the system again, the UserContext is taken off of the user's session and placed onto the thread again. In the webapp, this manipulation is done by the OpenmrsFilter class that wraps around every call to the server. Similar to the services, the methods on the Context class pass through to the current UserContext on the current thread.

Every access to the system must be defined within a "unit of work". This unit is bordered by calls to Context.openSession() and Context.closeSession(). In the webapp, these calls are done in OpenmrsFilter and most developers don't have to worry about making those calls. However, any developer of an external application or a thread spinoff (like HI7InQueueProcessor and FormEntryProcessor) will need to be sure to include open/closeSession calls or risk leaking database connections.

Read more about the API

Spring DI and AOP

PatientService, ConceptService, etc are interfaces. PatientServiceImpl, ConceptServiceImpl, etc are the current default implementations of those interfaces. Which implementation Context serves up is determined by Spring's dependency injection. Each *Impl contains a reference to its DAO (PatientDAO, ConceptDAO etc). The DAOs are also interfaces. The current default implementation of them are for Hibernate (HibernatePatientDAO, HibernateConceptDAO, etc). Which DAO implementation is used at runtime is also determined by Spring's DI.

All of these implementations are described in the Spring applicationContext-services.xml file. This file also controls the transaction and authorization AOP annotations (See next section).

Each *Service (not *DAO) is considered to be an AOP advice point. Because we instantiate and serve only one *Service class out of the Context, this is easy. When Spring starts, each service that has advice points around it gets wrapped with a Spring AOP class. If there are five modules/methods that wrap around a certain class, there will be five different wrappers. Any one of those could exit method execution early (if it's an "around" advice). Most AOP linking will be done by modules.

Also see OpenMRS AOP.

Authorization and Authentication

OpenMRS has a very granulated permissions system. Every action is associated with a Privilege. An action would be "Add Patient", "Update Patient", "Delete Patient", "Add Concept", "Update Concept", etc. A Role contains a collection of Privileges. A Role can also point to a list of inherited roles. The role inherits all privileges from that inherited role. In this way, hierarchies of roles are possible. A User contains only a collection of Roles, not Privileges.

AOP annotations are used to require a privilege for a service method.

```
@Authorized({"Add Patients"})
```

The annotation is placed in the **interface** of the service. If the current user does not possess that privilege an APIAuthenticationException is thrown. (The webapp catches this exception and redirects to the login form).

Each page in the webapp is able to require a certain privilege for the page with the **openmrs:require** taglib. The **openmrs:hasPrivilege** taglib provides support for restricting only certain sections of a page.

While using the API, you may come to a point where you need to temporarily grant the current user a certain privilege in order to make an API call. This is accomplished using the **Context.addProxyPrivilege(String priv)** and **removeProxyPrivilege(String priv)** methods. Multiple priv objects defining the same string can be proxied. Subsequent calls to removePrivilege will only pop the first one off the stack. Best practice says that you should put your API method calls in a try/catch block and put the **removeProxyPrivilege** call in a finally block.

Hibernate

Hibernate is an excellent Object Relational Mapper. Using just xml files, we are able to describe the relationship between all of our tables and our domain (POJO) objects (like Patient.java, Concept.java, etc). Looking at the concept domain in the datamodel, we see that it consists of tables named concept, concept_answer, concept_set, concept_name. It would be very difficult to keep up with where to store each part of the concept object and the relations between them. Using Hibernate, we only need to concern ourselves with the Concept object, not the tables behind the object. The concept.hbm.xml mapping file does the hard work of knowing that the Concept object contains a collection of ConceptSet objects, a collection of ConceptName objects, etc. To add a new name to a concept:

```
ConceptService conceptService = Context.getConceptService();
Concept concept = conceptService.getConcept(1234);
ConceptName newConceptName = new ConceptName("some name", "some locale");
concept.addName(newConceptName);
conceptService.updateConcept(concept);
```

Hibernate knows what has changed and what needs to be saved into the database. (The long and short of it is that Hibernate wraps the Concept object in its own object and keeps track of what has been added, removed, etc).

Hibernate will not load all associated objects until they are needed – this is called lazy loading. The concept object above never dove into the concept_answer table to get the list of answers for concept 1234. If we had called concept.getConceptAnswers() Hibernate at that point would have made a call to the database to retrieve the answers for us. For this reason, you **must** either fetch/save/manipulate your object in the same session (between one open/closeSession) or you must hydrate all object collections in the object by calling the getters (getConceptAnswers, getConceptNames, getSynonyms, etc).

OpenMRS Source code

OpenMRS lives in a Git repository. See the Code Repositories wiki page for more info.

Building OpenMRS

OpenMRS uses Maven to manage the libraries and build system. See that wiki page for more info.

Modules

OpenMRS has a modular architecture, meaning that "modules" (i.e., add-ons or extensions) can be added to the system to add new behavior or alter existing behavior. OpenMRS Add-ons index hosts publicly downloadable modules; these can be installed directly from within OpenMRS as well. See the Module documentation for developers. Modules are allowed to interact with OpenMRS on every level. They can provide new entries into the Spring Application Context, new database tables, new web pages, and even modify current service layer methods.

Webapp

The OpenMRS organization creates and ships a webapp (openmrs.war) for people to use. This webapp is a consumer of the API. We expect there to be (and there are already) several different consumers of the API by different parties.

Spring MVC

OpenMRS strongly subscribes to the Model-View-Controller pattern. We won't go into the depths of MVC, or even the basics of it. Mediawiki has a complete writeup about everything you may have wanted to know about the MVC programming pattern. Spring has also written a fair amount on how and why to use MVC. OpenMRS, for the most part, uses the domain objects as the model. Most controllers will be SimpleFormControllers and be placed in the org.openmrs.web.controller package. There are some controllers that have been rewritten to use Spring 2.5+ annotations. We recommend using those. The model is set up in the controller's formBackingObject, and processed/saved in the processFormSubmission and onSubmit methods. The jsp views are placed in /web/WEB-INF/view.

Not all files served by the webapp are run through Spring. The /web/WEB-INF/web.xml file maps certain web page extensions to the SpringController. All *.form, *.htm, and *.list pages are mapped. The SpringController then uses the mappings in the openmrs-servlet.xml file to know which pages are mapping to which Controller.

There are no .jsp pages that are accessed directly. If a pages url is /admin/patients/index.htm, the jsp will actually reside in /web/WEB-INF/view/admin/patients/index.jsp. This is necessary so that we can do the redirect with the SpringController. Because the file being accessed ends with .htm, the SpringController is invoked by the web server. When the SpringController sees the url, it simply replaces .htm with .jsp and looks for the file in /web/WEB-INF/view/ according to the jspViewResolver bean in openmrs-servlet.xml. If the page being accessed was patient.form, the mapping in the urlMapping bean would have told spring to use the PatientFormController and the patientForm.jsp file.

That spring descriptor file also contains settings for the max form upload size, locale changing, message names, fieldGen handlers, and name/address templates (to be removed: TRUNK-368).

DWR

DWR is largely being replaced with REST Web Services as of 1.9+.

Direct Web Remoting is a framework that allows us to translate java objects and methods to javascript objects and methods. Together with jquery/DOJO, DWR forms the basis of the AJAX in OpenMRS. The dwr.xml descriptor file describes which classes and methods are translated and made available to javascript calls. Most base DWR-able classes are placed into the org.openmrs.web.dwr package. Modules can add dwr methods/objects and those will go into the already-registered-with-dwr dwr-modules.xml file.

Javascript

jQuery is the recommended JS framework. The included jquery package will be kept up to date with openmrs releases. Import the one you need with

```
<openmrs:htmlInclude file="/scripts/jquery/jquery.min.js" />
```

(but its already done in the header for you as of version 1.7+)