API

OpenMRS is intentionally built with multiple layers in mind. One of the layers is a java API that can be used in other projects just as easily as it can be inside of OpenMRS.

The API allows developers to interact with the complex OpenMRS Data Model with common Java objects. This provides greater data integrity as well as a simple to use approach.

Also see:

- the API in Javadoc form.
- How To Use the OpenMRS API

Java Objects vs Tables

Most of the tables in the data model have a one-to-one relationship with a Java object in the OpenMRS API.

Compare the person table (in the data model) with the person java object:

- There is a "gender" column and "getGender/setGender" methods.
- A "birthdate" column and "get/setBirthdate" methods.
- The "person_name" table contains any number of names of a person. You can find a list of PersonName objects on the Person object.

Persisting and Retrieving Objects

To get data into and out of the database you go through the objects and services. If you want a person with id #1, you ask the PersonService for the Person object:

```
PersonService personService = Context.getPersonService();

Person p = personService.getPerson(1);
```

If you want to change data on this person and persist that in the database, you again use the personService:

```
PersonService personService = Context.getPersonService();

Person p = personService.getPerson(1);

p.setGender("M");

personService.savePerson(p);
```

The Services in the OpenMRS API make sure that only the correct columns are saved/updated in the database.

The Context Singleton

All services are accessed in a static way from the org.openmrs.api.context.Context object. You do not have to instantiate a new "Context" object and you do not ever call "new" on a service.

The primary usage of the Context is to get the services so you can fetch and persist things in the database:

```
UserService userService = Context.getUserService();
List<User> bobObjects = userService.getUsers("bob");
```

The other usage is to access the currently logged in user and their locale settings:

```
User u = Context.getAuthenticatedUser();
```

The currently logged in user's current locale:

```
Locale loc = Context.getLocale();
```

The currently logged in user's date pattern layout:

```
String dataPatternString = Context.getDatePattern;
```

Programming to Interfaces

All of our services are interfaces. The default implementation of these services are named *ServiceImpl.java. The implementations can be found in the impl directory of the api package.

In order to create a new implementation of a service (for whatever unknown reason that would be in the future), we would only need to change the file specified in Spring's `/api/src/main/resources/applicationContext-services.xml` file.

Authentication and Authorization

Users are authenticated against the Context.

```
Context.authenticate("bob", "password");
```

The current user's information is stored on the current thread. For the webapp, we have a special problem. Every request to the server is potentially on a new thread. Our custom OpenmrsFilter class wraps every request. It stores the user's UserContext on the user's session. Before each request the userContext is taken off the session and placed on the thread. After the entire request is complete, the userContext is removed from the thread (and placed back on the session).

Authorization is done through annotations on the **interface**:

```
public interface UserService
...
@Authorized({"View Users"})
void List<User> getUsers(Integer userId);
...
```

Sessions and Transactions

Spring manages our transactions.

All calls within a session are considered to be on one transaction. If an error is encountered all calls are rolled back and an error is returned.

Within the webapp, a session is determined by Spring's OpenSessionInViewFilter. Every request is wrapped with an open session and close session, see the OpenmrsFilter class.

Outside the webapp a session should be wrapped with calls to Context.openSession() and Context.closeSession(). See How To Use the OpenMRS API

To wrap a service within a transaction we use annotations on the **interface**:

```
@Transactional
public Interface BensNewService {
 ...
```

and/or

```
...
@Transactional(readOnly=true)
public List<BensObject> getBensObjects();
...
```