OpenECOMP Portal SDK Documentation

Copyright © 2017 AT&T Intellectual Property.
All rights reserved.
Licensed under the Creative Commons License, Attribution 4.0 Intl. (the "License"); you may not use this documentation except in compliance with the License.
You may obtain a copy of the License at

https://creativecommons.org/licenses/by/4.0/

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

ECOMP and OpenECOMP are trademarks and service marks of AT&T Intellectual Property

---

- Authentication
    - Using External Credentials
- Authorization
    - Managing Users/Roles
    - Restrictions on Menus and URLs
    - Access Control via Roles and Functions
        - Role management
            - Listing available roles
            - Activating/deactivating roles
            - Removing roles
            - Creating new roles
        - Role functions management
            - Listing available role functions
            - Editing role functions
            - Removing role functions
            - Creating new role functions
- Functional Menus
    - Description
    - Implementation
- Session Management
    - Goal:
    - Challenge(s):
    - Solution:
- Utilities
    - Cache Manager
    - AppUtils
- Application Configuration
    - System Properties
- Data Access Utilities
    - Data Access Service
    - Transactions
- Sample Pages / Controllers
    - Spring Based Controllers
    - Angularized Controllers
- REST APIs Exposed for Admin User Management
    - Service APIs
        - Examples:
- Visualization & Productivity Tools
    - Widget Development API
        - Key widget concepts
        - Widgets on the portal dashboard
- SDK Database Schema

Authentication

Using External Credentials

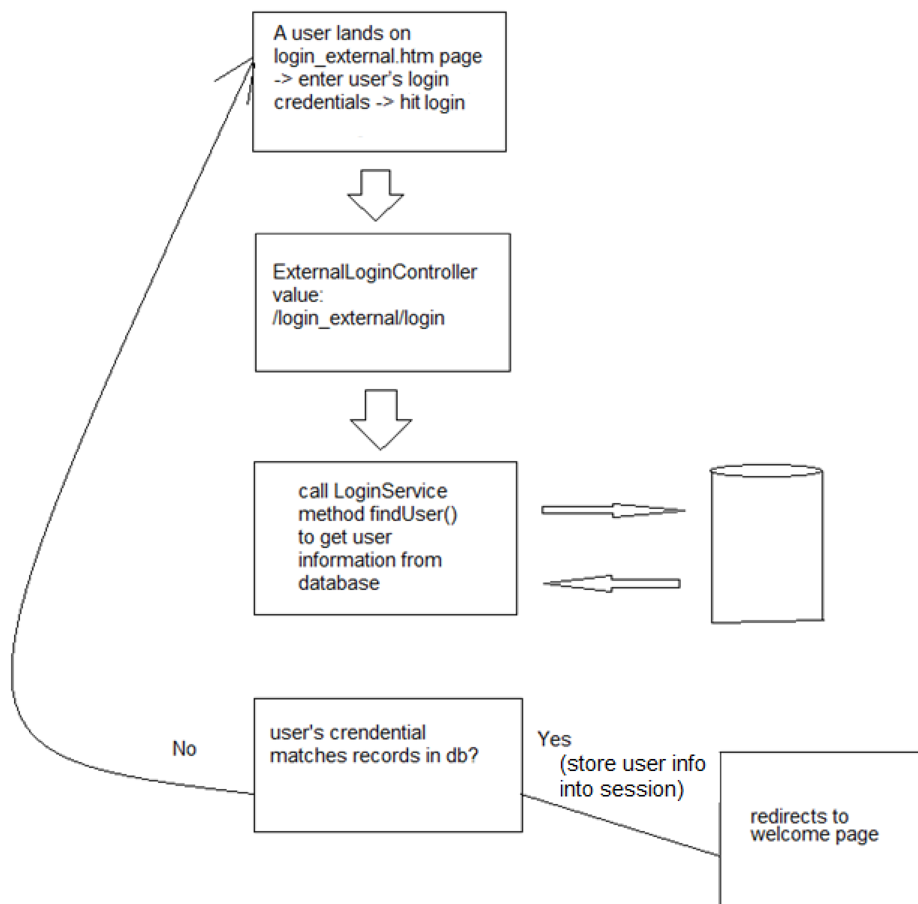
Authentication using external credentials. This is the *back door* ***only used by developers***.

Files: *none*

Front end: **login_external.jsp**

Back end: ExternalLoginController

Workflow:



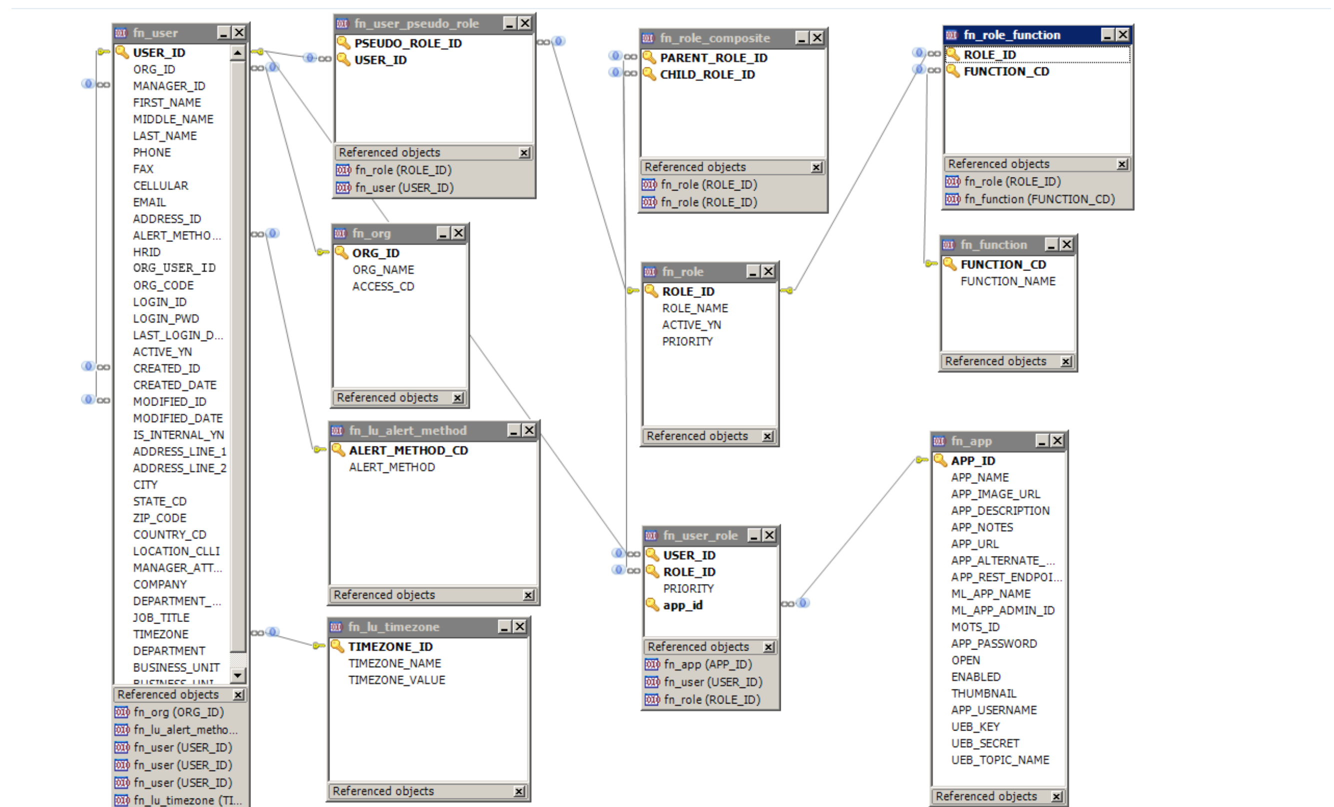
Authorization

Managing Users/Roles

The following tables are used. Using Hibernate, these tables are managed using the role screens.

- **FN_USER**
- **FN_USER_ROLE**
- **FN_ROLE_FUNCTION**
- **FN_FUNCTION**

When user is assigned a role, he would have access to the appropriate resources (menus / urls). Below is a Entity-Relation diagram showing the relations among the tables used to manage users and roles:

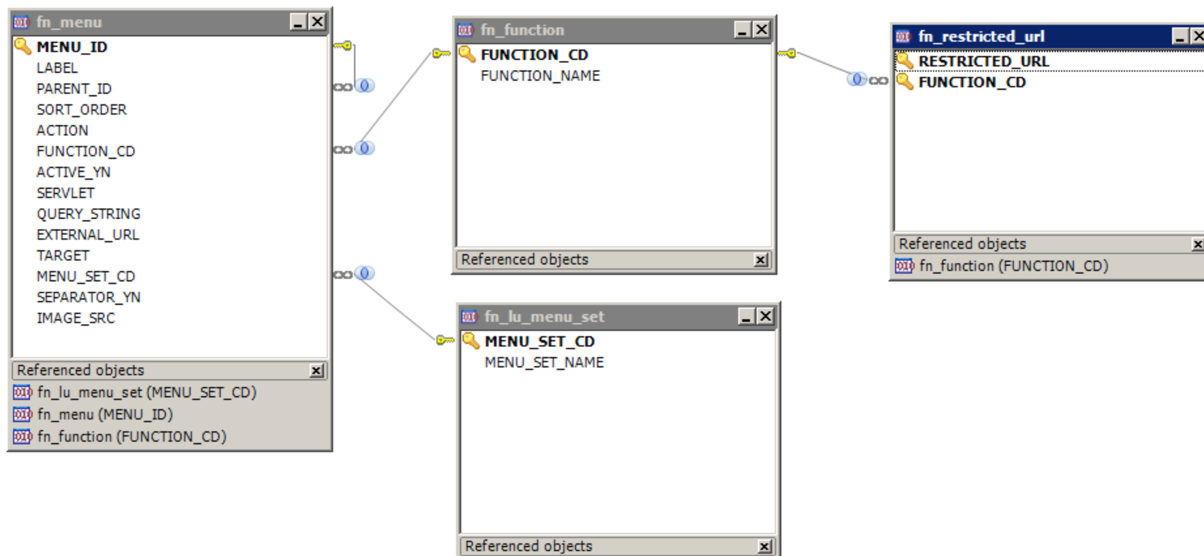


Restrictions on Menus and URLs

The menus and URLs can be restricted by associating it with a 'function'. See the following tables

- **FN_MENU**
- **FN_RESTRICTED_URL**

This applies to users via the User's role. Role is associated with a list of functions. If a user has access to a function, he/she will be able to access that menu and url. The ED diagram below shows the relations among the tables involved with the authorization process:



In order to protect a controller and add it to the restricted URL list, we just need to extend the base controller **RestrictedBaseController.java**. Also look at **AppConfig** that applies an interceptor **ResourceInterceptor.java** to all requests. The interceptor will check if the logged-in user is allowed to see the page or not.

See the following:

Menu: **LoginService.findUser**

FN_RESTRICTED_URL (All the pages inside this table would be checked against user's role under **ResourceInterceptor**. Define **function_cd** for **fn_role_function** table)

RESTRICTED_URL	FUNCTION_CD
profile.htm	menu_profile_creat

FN_USER (Maintain user information)

<IMAGE redacted pending creation of Open Source version>

FN_USER_ROLE (bind **user_id** and **role_id**)

USER_ID	ROLE_ID	PRIORITY
1	1	

FN_ROLE

ROLE_ID	ROLE_NAME	ACTIVE_YN	PRIORITY
1	System Administrator	Y	1

FN_ROLE_FUNCTION (Define **role_id**'s right to access the **function_cd**)

ROLE_ID	FUNCTION_CD
1	doclib
1	doclib_admin
1	login
1	menu_admin
1	menu_ajax
1	menu_customer
1	menu_customer_create
1	menu_doclib
1	menu_feedback
1	menu_help
1	menu_hiveconfig
1	menu_hiveconfig_create
1	menu_hiveconfig_search
1	menu_home
1	menu_itracker
1	menu_itracker_admin
1	menu_job
1	menu_job_create
1	menu_logout
1	menu_mapreduce
1	menu_mapreduce_create
1	menu_mapreduce_search
1	menu_notes
1	menu_process
1	menu_profile
1	menu_profile_create
1	menu_profile_import
1	menu_reports
1	menu_sample
1	menu_tab
1	menu_test
1	quantum_bd

Access Control via Roles and Functions

Role management

This section is a user guide to the management of the roles in a given system, including listing available roles, creating new roles, activating/deactivating roles, and removing roles

Listing available roles

The JSP or Java files associated with these features are described below:

Front-end: **role_list.jsp**

Back-end: **RoleListController.java**

Workflow: The user clicks on “Admin->Roles” from menu items to view the list of available roles.

<IMAGE redacted pending creation of Open Source version>

Activating/deactivating roles

The user clicks on the toggle button under “Active?” column to activate or deactivate the role.

Removing roles

The user clicks on the trash button under “Delete?” column to delete the role.

Creating new roles

Front-end: **role.jsp**

Back-end: **RoleController.java**

Workflow: The user clicks on the “Create” button as shown in the roles list screen shown below to go to the screen where he/she can create a new role.

<IMAGE redacted pending creation of Open Source version>

Roles list screen showing “Create” button.

Below is the screen to enter a new role name and priority. The user can add available role functions and child roles using same screen. Once a new role is saved, it will appear in the roles list screen.

<IMAGE redacted pending creation of Open Source version>

Role functions management

This section is a user guide to the management of the role functions in a given system, including listing available role functions, creating new role functions, editing role functions, and removing role functions.

Listing available role functions

The JSP or Java files associated to these features described below:

Front-end: **role_function_list.jsp**

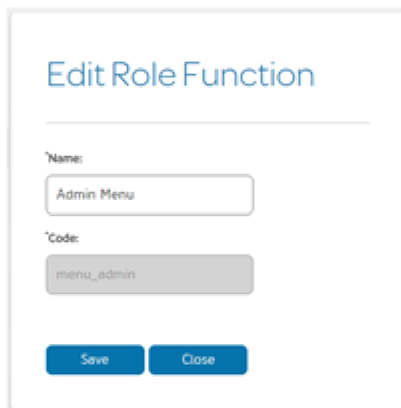
Back-end: **RoleFunctionListController.java**

Workflow: The user clicks on "Admin->Role Functions" from menu items to view the list of available role functions.

: ***<IMAGE redacted pending creation of Open Source version>***

Editing role functions

The user clicks on the edit button under the "Edit?" column to view/edit the role function name and code. Only the role function name can be edited from the edit popup, since editing of the role function code is disabled.



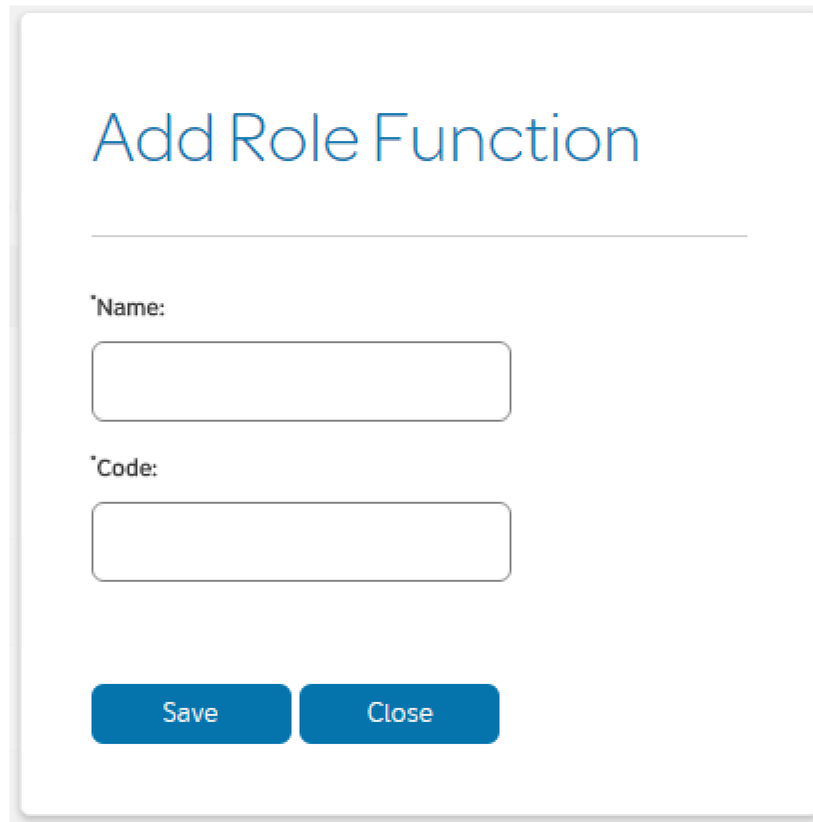
Removing role functions

The user clicks on the trash button under "Delete?" column to delete the role function.

Creating new role functions

The user clicks on the circled "+" button under the tile as shown in below in the role functions list screen to create the popup where he/she can create a new role function.

The user can add a new role function by entering a new name and code in the Add Role Function popup:



Functional Menus

Description

Functional Menus provide seamless application integration. They help users maintain a sense of orientation across different portal applications. Functional menus are a navigation device that categorizes menu items into three major categories: Design, Runtime, and Platform. Important links from each application should reside under these three categories. Users can use this menu to jump between different applications.

The application owner needs to provide menu items that he/she wants to put into this navigation device during the onboarding process. The owner also needs to decide which category each menu item belongs to. Note that the Portal administrator will have final decision on the name/link of the menu items.

The navigation menus should be clear to users. Multiple menu items can be added for the same application. Within the top level menu, the Portal administrator can add application specific menu items and hierarchy up to four levels deep.

Implementation

We have implemented a functional menu code in a directive called a **q-header**. Developers just need to include **header.css**, **portal_ebz_header.css**, and **header.js** in the desired html file, and enter the following code -- the functional menu should the be working:

```
<div q-header></div>
```

Things that need to be modified when embeding the code – in the file **header.js**:

```
$scope.getMenu=function() {
      $http({method: 'GET', url: 'get_menu'})
        .success(function (response) {
            var j = response;
            $scope.parentData = JSON.parse(j.data);
            $scope.childData = JSON.parse(j.data2);
            $scope.userName = j.data3;
            var childItemList = $scope.childData;
            var pageUrl = window.location.href.split('/')[window.location.href.split('/').length-1];
            var parentList =$scope.parentData;

            for (var i = 0; i < parentList.length; i++) {
                $scope.item = {
                    parentLabel : parentList[i].label,
                    parentAction : parentList[i].action,
                    open:pageUrl==parentList[i].action?true:false,
                    childItemList : childItemList[i]
                }
                $scope.menuItems.push($scope.item);
            }

      });
    }
```

We are calling **get_menu** to get the menu data. This can be changed if necessary.

Session Management

Goal:

To provide a positive user experience, a user should be able to jump from one authorized application to another without a session expiration from one application. The applications need to collectively keep their sessions alive during the **idle time - last visited time-stamp** does not equal the **session timeout**.

Challenge(s):

1. When user navigates from app to app, URL Redirect and we loose browser access other apps.
2. Application may have different **session timeouts.**

Solution:

1. OpenECOMP Portal conducts and manages the **session timeouts** on all applications.

2. OpenECOMP Portal will keep track of all **active sessions** across all applications
3. OpenECOMP Portal will retrieve **last-visited-time-stamp** from all applications.
4. Applications will attach the OpenECOMP Portal **session** ID in their application **sessions**.
5. Applications will periodically send the **last-visited-time-stamp** of each session to the OpenECOMP Portal.
6. Before expiring a session, the application will check with the OpenECOMP Portal if there was a recent activity across any other application. If so, the **session** will be extended.

<IMAGE redacted pending creation of Open Source version>

<IMAGE redacted pending creation of Open Source version>

Session management is done via an interceptor. See **AppUtils.addInterceptors**. All urls that are adding the **SessionTimeoutInterceptor** will be checked for a valid session before letting them continue. If the session is expired, the user will be taken back to the login page.

Also look at **UserUtils** to see logged-in user's details.

Utilities

Cache Manager

The following methods are used to the get access to FUSION's **CacheManager**. The **CacheManager** is an implementation of Java Caching System (JCS) which is a proven caching mechanism that can cache data in memory, to disk, in a clustered server environment, or distributed environment.

- **public static AbstractCacheManager getCacheManager()**
- **public static boolean isCacheManagerAvailable()**

AppUtils

The **AppUtils** class has the following static methods that can be used by developers to manage lookup lists, email notifications, error handling, and feedback messaging.

The following methods are used to retrieve a List of name/value pairs needed for drop-down lists or to decode an **id**. The objects returned in the list are of type **org.openecomp.fusion.domain.Lookup** which has two properties: **value** and **label**. When these methods are called, a SQL statement is constructed and ran based on the passed parameters. The list that is returned is cached in memory unless the **getLookupListNoCache** method is called. To remove a cached list, one needs to call the method **removeLookupListsFromCache** and pass a key representing the SQL that was run.

The key is of the form: **dbTable** + **"|"** + **dbValueCol** + **"|"** + **dbLabelCol** + **"|"** + **dbFilter** + **"|"** + **dbOrderBy.**

- **public static List getLookupList(String dbTable, String dbValueCol, String dbLabelCol, String dbFilter, String dbOrderBy)**
- **public static List getLookupList(String dbTable, String dbValueCol, String dbLabelCol)**
- **public static List getLookupListNoCache(String dbTable, String dbValueCol, String dbLabelCol, String dbFilter, String dbOrderBy)**
- **public static synchronized void removeLookupListsFromCache(String keyStartsWith)**

The following methods go a step further than the ones above. A call to these methods will either find a label in a List for the indicated value, or build a query based on the passed parameters, cache the results, and return a label for the indicated value.

- **public static String lookupValueLabel(String value, List lookupList)**
- **public static String lookupValueLabel(String codeValue, String dbTable, String dbValueCol, String dbLabelCol,String dbFilter, String dbOrderBy)**
- **public static String lookupValueLabel(String codeValue, String dbTable, String dbValueCol, String dbLabelCol**

The following methods are used to send emails from the FUSION platform. These methods vary based on supplied values, whether an attachment needs to be emailed, and MIME type.

- **public static void notify(String message, String to, String from, boolean contentTypeHtml)**
- **public static void notify(String message, String to, String from, String subject, boolean contentTypeHtml)**

- **public static void notify(String message, String to, String from, String subject, String cc, String bcc, boolean contentTypeHtml)**
- **public static void notify(String message, String[] to, String from, String subject, String[] cc, String[] bcc, boolean contentTypeHtml)**
- **public static void notifyWithAttachments(String message, String[] to, String from, String subject, String[] cc, String[] bcc, List mailAttachments, boolean contentTypeHtml)**

The following methods are used to add feedback messaging content, get feedback messages, and check for feedback messages of a certain type.

- **public synchronized static Vector getFeedback(String userSessionId, boolean removeMessages)**
- **public synchronized static Vector getFeedback(String userSessionId) public synchronized static void removeFeedback(String userSessionId)**
- **public synchronized static boolean hasError(String userSessionId)**
- **public synchronized static boolean hasWarning(String userSessionId)**
- **public synchronized static boolean hasSuccess(String userSessionId)**
- **public synchronized static void addFeedback(String userSessionId, FeedbackMessage message)**

The following method is used to the get the current user's session

- **public static HttpSession getSession(HttpServletRequest request)**

The following methods are used to the get access to FUSION's CacheManager. The CacheManager is an implementation of Java Caching System (JCS) which is a proven caching mechanism that can cache data in memory, to disk, in a clustered server environment, or distributed environment.

- **public static AbstractCacheManager getCacheManager()**
- **public static boolean isCacheManagerAvailable()**

Application Configuration

System Properties

We maintain our static values like database information and menu setting in system.properties file, which is located in **quantum/war/WEB-INF/conf**.

In order to get the values from the properties file, we create a java class, **SystemProperties.java**. Simply call the java class to get the values stored in the properties.

For example, in **HibernateConfiguration.java**:

```
dataSource.setDriverClass(SystemProperties.getProperty(SystemProperties.DB_DRIVER));
dataSource.setJdbcUrl(SystemProperties.getProperty(SystemProperties.DB_CONNECTIONURL));
dataSource.setUser(SystemProperties.getProperty(SystemProperties.DB_USERNAME));
dataSource.setPassword(SystemProperties.getProperty(SystemProperties.DB_PASSWOR));
```

If there are multiple properties fields, simply add the path in **SystemProperties.java** by using **@PropertySource** annotation.

Data Access Utilities

Data Access Service

We are using **Hibernate** to access the DB. Hibernate mapping for core SDK mapping is maintained in **Fusion.hbm.xml**. The definition is done in **HibernateCofiguration**

OpenECOMP Portal SDK provides the following services for your use during development:

DataAccessService: is used to execute any HQL, native SQL select or update queries. It also has services to access the hibernate objects.

Data Access Service methods: Gets an object of the domainClass having the indicated **id**. If none exists, a new instance of the domain class is returned.

- **DomainVo getDomainObject(Class domainClass, Serializable id);**

Deletes the instance domainObject passed to the method from the database.

- **void deleteDomainObject(DomainVo domainObject);**

Deletes a set of records from the table represented by domainClass having the specified whereClausecriteria from the database.

- **void deleteDomainObjects(Class domainClass, String whereClause);**

Inserts or updates the corresponding record in the database represented by instance domainObject.

- **void saveDomainObject(DomainVo domainObject);**

the domain service provides that following get list method(s) that can retrieve a list of objects representing rows from a table that domainClass maps to. If no filter is provided, all rows of the

table will be retrieved. In general, Hibernate named queries should be used instead of these methods because they are quicker and cleaner to implement.

- **List getList(Class domainClass, List filter, String orderBy); //List of QueryFilter objects used to filter**
- **List getList(Class domainClass, List filter, int fromIndex, int toIndex, String orderBy); //List of QueryFilter**
- **objects List getList(Class domainClass, String filter, String orderBy); //HQL filter**
- **List getList(Class domainClass, String filter, int fromIndex, int toIndex, String orderBy); //HQL Filter**

The following methods run native SQL queries that return a list of Hibernate domain objects. The executeQuery method runs the SQL specified and returns a list of domainClass objects. A subset of rows can be returned by providing the fromIndex and toIndex parameters. The executeNamedQuery method runs the query specified by queryName defined in a Hibernate mapping file and returns a list of objects specified by the named query. A Map of params or the subset of rows may be passed to the named query if needed.

- **List executeSQLQuery(String sql, Class domainClass, HashMap additionalParams);**
- **List executeSQLQuery(String sql, Class domainClass, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeQuery(String sql, HashMap additionalParams);**
- **List executeQuery(String sql, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeNamedQuery(String queryName, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeNamedQuery(String queryName, Map params, HashMap additionalParams);**
- **List executeNamedQuery(String queryName, Map params, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeNamedQueryWithOrderBy(Class entity, String queryName, Map params, String _orderBy, boolean asc, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeNamedCountQuery(Class entity, String queryName, String whereClause, Map params);**
- **List executeNamedQuery(Class entity, String queryName, String whereClause, Map params, Integer fromIndex, Integer toIndex, HashMap additionalParams);**
- **List executeNamedQueryWithOrderBy(Class entity, String queryName, String whereClause, Map params, String _orderBy, boolean asc, Integer fromIndex, Integer toIndex, HashMap additionalParams);**

- **int executeUpdateQuery(String sql, HashMap additionalParams) throws RuntimeException;**
- **int executeNamedUpdateQuery(String queryName, Map params, HashMap additionalParams) throws RuntimeException;**
- **void synchronize(HashMap additionalParams) //does hibernate flush**

Transactions

The OpenECOMP Portal uses a **HibernateTransactionManager** and declarative transaction management via Spring Annotation to demarcate transactions for the specified FUSION service calls. Put **@Transaction** for the specified FUSION service calls to tell Spring to insert transaction management code into the bytecode. The OpenECOMP Portal uses **@EnableTransactionManager** annotation to enable Spring's annotation-driven transaction management capability. It configured **SessionFactory** and **TransactionManager** in **com/org.openecomp./fusion/core/config/HibernateConfiguration**

```
@Bean
@Autowired
public HibernateTransactionManager transactionManager(SessionFactory s) {
   HibernateTransactionManager txManager = new HibernateTransactionManager();
   txManager.setSessionFactory(s);
   return txManager;
}
```

If you have new methods that require transactions, simply add **@Transactional** annotation above methods. Transaction propagation is handled automatically.

```
@Transactional
public void saveDomainObject(DomainVo vo, HashMap additionalParams) {
```

Sample Pages / Controllers

Spring Based Controllers

Create an **ecomp/jsp** folder under **WEB-INF**

1. Put all the **jsp** files into the **jsp** folder.
2. Then define the page definition to extend **ebz_template** under **definitions.xml** (optional)
3. Put all the new java classes after this package: **org.openecomp.fusionapp.ecomp**
4. Create a controller class.

For pages that require a user's access rights, you need to extend **RestrictedBaseController**. Once the controller extends **RestrictedBaseController**, every time a user tries to access the page, **ResourceInterceptor** will check to see if the user has the right to access the current page. If user does not has right to access the page, it will throw an exception.

For pages that do not require a user's access rights, you need to extend **UnRestrictedBaseController**.

1 Create URL mapping in Spring Controller(UserProfileController) (for example : value = {"/user_profile" }, method = RequestMethod.GET)

2 Add entry to definition.xml for view:

```
<definition extends="ebz_template" name="user_profile">
    <put-attribute name="body" value="/WEB-INF/jsp/user_profile.jsp"></put-attribute>
</definition>
```

URL starts with with "/user_profile" will go to user_profile.jsp which is our main layout, user_profile.jsp is embedded in **ebz_template** which has components like header, footer, left menu and so on.

3 In user_profile.jsp, defined angular controller which includes page behavior and method in script like this:

```
<script>
      app.controller("userProfileSampleController", function ($scope,$http,modalService, $modal) {
             ... ...
      });
</script>
```

Angularized Controllers

New angular single page(No page refresh on page change) work flow

1 Create URL mapping in Spring Controller(**AngularAdminController**) (for example : **@RequestMapping(value = {"/admin" }, method = RequestMethod.GET**)

2 Add entry to definition.xml for view (**<definition name="admin" template="/app/fusion/scripts/view-models/admin-page/admin.html"/>**). All the URL starts with with "/admin" will go to admin.html which is our main layout. ng-view is an Angular directive that will include the template of the current route in the main layout file. In plain words, it takes the file we want based on the route and injects it into our main layout (admin.html)

3 In admin.html, we have a content div **(<div ng-view></div>)** to place our rendered pages when URL changes.

4 Create the angular module and controller in JavaScript (app.js and specific controller js for each page), include these js files in admin.html.

Method	Path	Server side method description
GET	**/roles**	**getAvailableRoles()** RESTful service method to fetch available roles
GET	**/roles**	**getAvailableRoles()** RESTful service method to fetch available roles
GET	**/user/{loginId}/roles**	**getUserRoles(java.lang.String loginId)** RESTful service method to fetch individual user's roles using user's **loginId**
POST	**/user**	**pushUser(java.lang.String userJson)** RESTful service method to save new user - expects user details in json string
POST	**/user/{loginId}**	**editUser(java.lang.String loginId, java.lang.String userJson)** RESTful service method to edit existing user - expects user details in json string
POST	**/user/{loginId}/roles**	**pushUserRole(java.lang.String loginId, java.lang.String rolesJson)** RESTful service method to save user roles using user's login Id and details in roles Json string
GET	**/user/{loginId}**	**getUser(String loginId)**

		RESTful service method to fetch one user
		getUsers()
GET	**/users**	RESTful service method to fetch all users

5 Since we are making a single page application and we don't want any page refreshes, we'll use Angular's routing capabilities. Let's look in our Angular file and add to our application. We will be using [$routeProvider] in Angular to handle our routing. This way, Angular will handle all of the magic required to go get a new file and inject it into our layout.

Create adminController.js which includes routes info,

```
.when('/role_function_list', {
            templateUrl: 'app/fusion/scripts/view-models/profile-page/role_function_list.html',
            controller: 'roleFunctionListController'
      })
      .when('/role/:roleId', {
            templateUrl: 'app/fusion/scripts/view-models/profile-page/role.html',
            controller: 'roleController'
      })
.when('/usage_list', {
            templateUrl: 'app/fusion/scripts/view-models/profile-page/usage_list.html',
            controller: 'usageListController'
      })
.otherwise({
            templateUrl: 'app/fusion/scripts/view-models/profile-page/role_list.html',
            controller : "roleListController"
)
```

As you can see by the configuration, you can specify the **route**, the **template** file to use, and even **controller**. This way, each part of our application will use its own view and Angular controller. By default, we put role_list.html as our home page.

6 Now we can test this single page angular. Just type in the following URLs in your browser and you will see the pages changes without refreshing whole page(change the URL based on your need).

REST APIs Exposed for Admin User Management

Service APIs

Examples:

1. URL: **http://localhost:8080/OpenECOMP_app/api/roles**

   Ouput:

[

{"id":1992,"name":"Document Library Admin","active":true,"priority":2},
{"id":1991,"name":"Document Library Users","active":true,"priority":4},
{"id":16,"name":"Standard User","active":true,"priority":5},
{"id":1,"name":"System Administrator","active":true,"priority":1},
{"id":5012,"name":"Test Role 7","active":false,"priority":7},
{"id":5002,"name":"Test role","active":false,"priority":null},
{"id":250,"name":"iTracker Support","active":true,"priority":null},
{"id":200,"name":"iTracker User","active":true,"priority":null},
{"id":5000,"name":"test role 1","active":false,"priority":10},
{"id":5005,"name":"test rolr 5","active":false,"priority":null}

]

2. URL: **http://localhost:8080/OpenECOMP_app/api/user/mt2061/roles**

   Ouput:

[
{"id":16,"name":"Standard User","active":true,"priority":5},
{"id":1,"name":"System Administrator","active":true,"priority":1}

]

3. URL: **http://localhost:8080/OpenECOMP_app/api/user/mt2061** (Post)

Input:

{

"orgId": null,
"firstName": "DOE",
"lastName": "JOHN",
"phone": "+1 55555555",
"fax": null,
"email": "doe@openecom.org",
"hrid": "9999999",
"orgUserId": "jd9999",
"address1": "100 South Main Street",
"address2": "NA",
"city": "Anytown",
"state": "ST",
"zipCode": "99999",
"managerAttuid": "xy12345",
"locationClli": "XXXXXXXX",
"departmentName": "TECHNOLOGY",
"company": "Open ECOMP Inc.",
"jobTitle": "TECHNICAL ENGINEER",
"loginId": "jd9999",
"active": true

}

Output:

{

"error": "{"response":"edit user success."}"

}

4. URL: **http://localhost:8080/OpenECOMP_app/api/user/{loginId}/role** (push)

E.g. See the **org.openecom.fusion.core.domain.Role**. **id** is the same as returned from **getAvailableRoles**

Input:

{"16":"Standard User"}

Output:

{

"error": "{"response":"push user role success."}"

}

5. URL: **http://localhost:8080/OpenECOMP_app/api/user/{loginId}/roles** (get via **getUserRoles()**)

Output:

[

{"id":1992,"name":"Document Library Admin"},
{"id":1991,"name":"Document Library Users"},
{"id":16,"name":"Standard User"}

]

6. URL: **http://localhost:8080/OpenECOMP_app/api/user/mt2061** (get via **getUser**())

Output:

{

"orgId": null,
"firstName": "DOE",
"lastName": "JOHN",
"phone": "+1 55555555",
"fax": null,
"email": "doe@openecom.org",
"hrid": "9999999",
"orgUserId": "jd9999",
"address1": "100 South Main Street",
"address2": "NA",
"city": "Anytown",
"state": "ST",
"zipCode": "99999",
"managerAttuid": "xy12345",
"locationClli": "XXXXXXXX",
"departmentName": "TECHNOLOGY",
"company": "Open ECOMP Inc.",
"jobTitle": "TECHNICAL ENGINEER",
"loginId": "jd9999",
"active": true

}

7. URL: **http://localhost:8080/OpenECOMP_app/api/users** (get via **getUsers()**)


```
[
    {
    "orgId": null,
    "firstName": "DOE",
    "lastName": "JOHN",
    "phone": "+1 55555555",
    "fax": null,
    "email": "doe@openecom.org",
    "hrid": "9999999",
    "orgUserId": "jd9999",
    "address1": "100 South Main Street",
    "address2": "NA",
    "city": "Anytown",
    "state": "ST",
    "zipCode": "99999",
    "managerAttuid": "xy12345",
    "locationClli": "XXXXXXXX",
    "departmentName": "TECHNOLOGY",
    "company": "Open ECOMP Inc.",
    "jobTitle": "TECHNICAL ENGINEER",
    "loginId": "jd9999",
    "active": true
    "roles":
        [{
            "id": 16
            "name": "Standard User"
        }]
    }
{
    "orgId": null,
    "firstName": "ROE",
    "lastName": "JANE",
    "phone": "+1 55555556",
    "fax": null,
    "email": "roe@openecom.org",
    "hrid": "9999999",
    "orgUserId": "jr9999",
    "address1": "100 South Main Street",
    "address2": "NA",
    "city": "Anytown",
    "state": "ST",
    "zipCode": "99999",
    "managerAttuid": "xy12345",
    "locationClli": "XXXXXXXX",
```

```
        "departmentName": "TECHNOLOGY",
        "company": "Open ECOMP Inc.",
        "jobTitle": "TECHNICAL ENGINEER",
        "loginId": "jr9999",
        "active": true
        "roles":
        [{
            "id": 1991
            "name": "Document Library Users"
        }
        {
            "id": 1992
            "name": "Document Library Admin"
        }
        {
            "id": 16
            "name": "Standard User"
        }]
    }
]
```

Visualization & Productivity Tools

Widget Development API

Key widget concepts

1. Widgets will be hosted in the source applications.The OpenECOMP Portal dashboard will use iframes to embed widgets hosted and served by source applications. Source applications = OpenECOMP Portal on-boarded applications.
2. Widgets will be able to interact with each other, making it necessary that Widget Registration is also part of the Application Onboarding / Registration process.
3. Widget-to-Widget communication will be based on well-defined contracts managed and defined in the OpenECOMP Portal. Users should be able to define these contracts on the Widgets Registration screen.

Widgets on the portal dashboard

A user logged in will be presented with a dashboard which will have two switchable views: Widgets and Applications

1. Dashboard physical appearance:
   1. **Widgets View:** Shows all widgets grouped by applications and sorted in alphabetical order within each group. A Widget in this view can be interacted with – In other words, it will not be an image but an interact-able UI component. This will be the default selected tab on the landing page / dashboard.
   2. **Applications View:**User can click on a second tab 'Applications' to see all applications he/she have access to. It is based on the user's access to the application. Each application is represented by a thumbnail (I recommend we start calling this a Thumbnail going forward to differentiate it from the Widgets described above). {This is the existing dashboard view}
2. Dashboard preferences: The user should also be able to define various preferences. A few are mentioned below; this item is open for suggestions.
   1. Assign an ordering to each application. If no ordering is specified, a default alphabetical order will be used.
   2. Assign an ordering to each widget within the Application group (this will override the default alphabetical order).
   3. Use an automatic, dynamic ordering based on the user's most recent interactions.
   4. Other more comprehensive orderings can be added later.
3. Other dashboard requirements
   1. One Application can have zero or more widgets. If a user doesn't have access to widgets of any application – the **Widgets** view is empty – the default view should be **Applications**, not **Widgets**.

SDK Database Schema

This document describes the schemata for the tables in the SDK Database. Most of these table are prefixed by a code describing the table's general function.

<SECTION redacted pending creation of Open Source version>