App Framework Step by Step Tutorial

Pre-Requisites

This tutorial will step you through the process of using the App Framework Module to build an app for the Reference Application. It assumed that you are familiar with various other OpenMRS technologies, particularly the UI Framework.

To get started, you'll need a development environment set up for working in the Reference Application. Learn about setting up a Reference Application development environment.

In this tutorial, I will walk through the steps of creating a simple "Find Patient" app within the existing *coreapps* module, and adding it to the home screen of the Reference Application.

Step 0 - Implement Underlying Functionality

The App Framework is largely a layer *on top of* existing functionality, that packages up your functionality to be displayed on the homepage, or linked to other apps. So the first step in building an app should be implementing your actual functionality, as a web page using any technology. In this example I've use the UI Framework, but this could just as easily have been a client-side HTML+JS app.

For this tutorial I implemented, within the "coreapps" module, a simple Find Patient page that uses the webservices.rest module to search for patients, displays results, and on clicking it takes you to a (hardcoded) link to the patient dashboard app in the same module. The two commits are here and here. For those interested in learning more, I have annotated the commits with a few comments relating to building functionality for the Reference Application.

For the rest of this tutorial all you really need to know is that there's a page accessible at (url base)/coreapps/findpatient/findPatient.page.

Step 1 - Package the functionality as an App

To package our functionality as an App, I add a file to the module as /omod/src/main/resources/app/findpatient_app.json, which could potentially hold a list of app descriptors, though we have only one.

findpatient_app.json

```
[
    {
        "id": "coreapps.findPatient",
        "description": "Basic patient search, by ID or Name (using OpenMRS's standard patient search)",
        "order": 2
    }
]
```

We also add a unit test to ensure that this app is being read properly:

AppTest.java

```
public class AppTest {
    @Test
    public void testFindPatientAppIsLoaded() throws Exception {
        AppDescriptor app = AppTestUtil.getAppDescriptor("coreapps.findPatient");
        assertThat(app.getOrder(), is(2));
    }
}
```

The actual work done is in this commit.

The work done in this step won't produce any visible changes, though in a later step you'll see the purpose of this app.json file.

Step 2 - Adding an App Link on the Homepage

Not every app is visible on the homepage. Some examples are (1) the patient dashboard app, which can only be linked to from other apps, and (2) a Manage Users app that you'd reach through a System Administration screen, and not from the homepage directly.

In order to make our app visible on the homepage, we have our app provide an *extension* that links to the *homepageLink extension point*. We do this by adding an extension inside our App's definition.

First, following good TDD practices, we add a unit test for this:

AppTest.java

```
public class AppTest {

    // ...

    @Test
    public void testFindPatientAppHasHomepageExtension() throws Exception {
        AppDescriptor app = AppTestUtil.getAppDescriptor("coreapps.findPatient");
        assertThat(app.getExtensions(), hasSize(1));
        assertThat(app.getExtensions().get(0).getExtensionPointId(), is("org.openmrs.referenceapplication.
homepageLink"));
    }
}
```

Then we add the extension to our app's definition:

findpatient_app.json

```
[
    {
        "id": "coreapps.findPatient",
        "description": "Basic patient search, by ID or Name (using OpenMRS's standard patient search)",
        "order": 2,
        "extensions": [
            {
                "id": "coreapps.activeVisitsHomepageLink",
                "extensionPointId": "org.openmrs.referenceapplication.homepageLink",
                "type": "link",
                "label": "coreapps.findPatient.app.label",
                "url": "coreapps/findpatient/findPatient.page",
                "icon": "icon-search",
                "requiredPrivilege": "App: coreapps.findPatient"
            }
        ]
    }
]
```

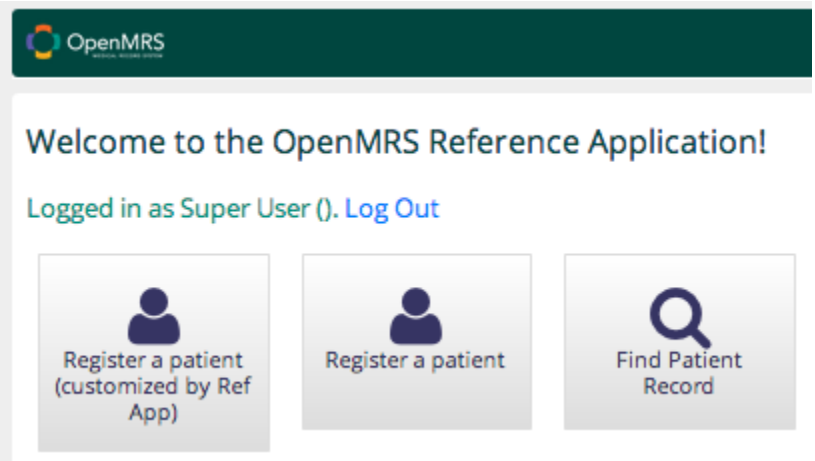
The syntax for this is unnecessarily verbose, and hopefully we improve it at some point. The properties are:

- id ... An id that is unique across all extensions loaded in the system, *including extensions from other apps*
- extensionPointId ... The unique id of the extension point where we want this extension to be attached
- label ... The label for the app on the application home page. (This is a code into the messages.properties file, though it can also be plain text.)
- url ... The starting page of the app, i.e. where to go when a user clicks on the app from the application homepage
- icon ... The name of an icon (you can see the list of them in the style guide, temporarily available at http://devtest02.openmrs.org:8080/openmrs/uicommons/icons.page)

NB for Font Awesome Icons, you need to add fas of fab extension class. otherwise the don't get to display eg "fas iconname"

- requiredPrivilege ... The link (technically, the extension) will only be available to users with this privilege.

After having made these changes, we can build our module (with the usual mvn clean install), install it in OpenMRS, and we'll see our new app available in the UI (it's the third one here):



The actual work done is in this commit. (Ignore the pom.xml change that I inadvertently included.)

In 80% of cases, we'd be done at this point.

Step 3 (Advanced) - Making a configurable App Template

If we were building an app for one specific purpose, our work would have been done at the end of the last step. The beauty of building OpenMRS applications based on the App Framework is that in 80+% of cases you *should* build an app for your specific purpose, and not prematurely attempt to generalize the app. Each individual app should be small, and if someone else needs to address a related-but-distinct use case, they *should* just build another app for it.

In some cases, however, you do want to make your app configurable and reusable. The App Framework provides a convenient way to do this, that supports letting you define an *App* Template, of which *other apps* can instantiate one or more configurable instances. Our Find Patient app is a good example of an app that *is* worth generalizing a bit. We'll generalize a template for it that allows you to configure the URL that you go to after choosing a patient.

(Aside: if we were considering a more significant change to this app, e.g. because some installations want to search by ID and name, while others want to search based on patients who are currently checked in to the hospital, I would suggest creating a new app for the location-based search. We should aim for our apps to be small, easy to configure and use, and focused on solving one specific use case very well.)

To generalize our app, we create a descriptor for a new AppTemplate. We put this in omod/src/main/resources/apps/coreappsAppTemplates.json, with the contents:

coreappsAppTemplate.json

```
[
   {
       "id": "coreapps.template.findPatient",
       "description": "Basic patient search by ID or Name (using OpenMRS's standard patient search)",
       "contextModel": [ "patientId" ],
       "configOptions": [
           {
               "name": "afterSelectedUrl",
               "description": "URL to go to after the user selects a patient. Supports {{patientId}}",
               "defaultValue": "/coreapps/patientdashboard/patientDashboard.page?patientId={{ patientId }}"
           }
       ]
   }
]
```

This file is allowed to contain multiple AppTemplates, though in this case we just need one. Its key properties are:

- id ... A unique id for this template, unique across all modules installed in the system. I have put "template" in the id for clarity, but this is not needed.
- contextModel ... Documentation for other modules that may want to leverage this template, showing what properties this app could pass off to another
- configOptions ... a list of possible options that may be configured by instances of this template, with default values

At this point we *could* delete the findpatient_app.json file that we created in the previous steps, and have this module purely provide a configurable app template, with no actual instance. However in this case we'd prefer that when you install the coreapps module, you get a basic "Find Patient" app out-of-the-box, with the default configuration. So rather than deleting our app's descriptor json, we modify it slightly just to indicate that it should inherit from the template we've created:

findpatient_app.json

```
[
    {
        "id": "coreapps.findPatient",
        "instanceOf": "coreapps.template.findPatient", // <== ADDED THIS LINE
        "description": "Basic patient search that goes to the patient dashboard",
        "order": 2,
        "extensions": [
            {
                "id": "coreapps.activeVisitsHomepageLink",
                "extensionPointId": "org.openmrs.referenceapplication.homepageLink",
                "type": "link",
                "label": "coreapps.findPatient.app.label",
                "url": "coreapps/findpatient/findPatient.page?app=coreapps.findPatient", // <== CHANGED THIS TOO
                "icon": "icon-search",
                "requiredPrivilege": "App: coreapps.findPatient"
            }
        ]
    }
]
```

The first change we made was to add an "instanceOf" property, indicating the unique id of the app template that we want to inherit from. Secondly, we change the url that the homepage link for this apps points to by having it specify the unique id of *this app*. (We'll get to the importance of this in a moment.)

In this instantiation of the app, we do *not* override the default value of "afterSelectedUrl" because we want this app to take you to the default patient dashboard after choosing a patient. If another module wanted to reuse our template, but wanted to have clicking on a patient lead to a different page, it might instantiate the app like this:

hypothetical_other_app.json

```
 [
    {
        "id": "myresearchstudy.findPatientAndEditContacts",
        "instanceOf": "coreapps.template.findPatient",
        "description": "Search for a study participant and take user to the Edit Contacts page",
        "order": 7,
        "config": {
            "afterSelectedUrl": "/myresearchstudy/editContacts.page?patientId={{ patientId }}"
        },
        "extensions": [
            {
                "id": "myresearchstudy.findPatientAndEditContactsHomepageLink",
                "extensionPointId": "org.openmrs.referenceapplication.homepageLink",
                "type": "link",
                "label": "Edit Contacts of Study Participant",
                "url": "coreapps/findpatient/findPatient.page?app=myresearchstudy.findPatientAndEditContacts",
                "icon": "icon-group",
                "requiredPrivilege": "App: myresearchstudy.findPatientAndEditContacts"
            }
        ]
    }
]
```

In this hypothetical example we have overridden the default value of "afterSelectedUrl".

We also include a test cases for our changes.

AppTest.java

```
public class AppTest {


    @Test // <== THIS IS NEW
    public void testFindPatientAppTemplateIsLoaded() throws Exception {
        AppTemplate template = AppTestUtil.getAppTemplate("coreapps.template.findPatient");
        assertThat(template.getConfigOptions().get(0).getName(), is("afterSelectedUrl"));
    }

    @Test
    public void testFindPatientAppIsLoaded() throws Exception {
        AppDescriptor app = AppTestUtil.getAppDescriptor("coreapps.findPatient");
        assertThat(app.getOrder(), is(2));
        assertThat(app.getInstanceOf(), is("coreapps.template.findPatient"));  // <== THIS IS NEW
        assertThat(app.getTemplate().getId(), is("coreapps.template.findPatient"));  // <== THIS IS NEW
        String expectedUrl = app.getTemplate().getConfigOptions().get(0).getDefaultValue().getTextValue();  //
<== THIS IS NEW
        assertThat(app.getConfig().get("afterSelectedUrl").getTextValue(), is(expectedUrl));  // <== THIS IS NEW
    }

    // ...
}
```

We introduce a new test case to verify that the App Template is being loaded correctly. We also modify an existing test to verify that the basic App correctly inherits from the new template.

Having made (and tested) these configuration changes, we can now modify our functionality to support this. (In real life I did this first, but for tutorial purposes it's easier to describe second.)

Before, our app's one page did not have a controller (since it's functionality is purely GSP + JS + a REST web service call). But in order to handle configuration we introduce a controller:

FindPatientPageController.java

```
public class FindPatientPageController {
    /**
     * This page is built to be shared across multiple apps. To use it, you must pass an "app" request
parameter, which
     * must be the id of an existing app that is an instance of coreapps.template.findPatient
     */
    public void get(PageModel model,
                    @RequestParam("app") AppDescriptor app) {
        model.addAttribute("afterSelectedUrl", app.getConfig().get("afterSelectedUrl").getTextValue());
    }
}
```

By using @RequestParam("app"), we let the caller of this page give a unique app id, and we fetch its AppDescriptor. (The *appui* module provides a Spring Converter for this.) We then get one particular field out of the AppDescriptor's config property (which is a JsonNode, from the Jackson JSON library), and put it in the page model, so we can use it in our view.

We have intentionally made this page only work if you provide an app's id via request parameter, which is why we added `?app=coreapps.findPatient` to the url in our app's json configuration.

We make two small changes to the view:

(part of) findPatient.gsp

```
...
<script type="text/javascript">
// 1. CHANGED THIS EACH LOOP
_.each(data.results, function(patient) {
    var url = '/' + OPENMRS_CONTEXT_PATH + emr.applyContextModel('${ ui.escapeJs(afterSelectedUrl) }', {
patientId: patient.uuid });
    resultTarget.append(resultTemplate({ patient: patient, url: url }));
});
// ...
</script>

<!-- 2. IN OUR TEMPLATE, CHANGE THE HREF -->
<a class="button" href="{{= url }}">${ ui.message("coreapps.findPatient.result.view") }</a>
```

The first change we made here is that we compute the URL we'd go to if you select a patient and pass it to the template. To comput this, we use a utility function from emr.js called applyContextModel, which will do straightforward substitution of an actual value for {{patientId}}.

Then within the template we just use the url that was passed in, rather than our previous hardcoded value.

In case you're wondering why we actually use the patient's uuid rather than their internal primary key, it's because we implemented the patient search using our REST Web Services module, and this intentionally only exposes UUIDs, not internal primary keys. Since the patient dashboard (like 99% of patient pages) does `@RequestParam("patientId") Patient`, and the UI Framework's StringToPatientConverter handles both ids and UUIDs, this is not a problem.

You can see the actual work for this in this commit.

At this point the coreapps module includes:

1. a "Find Patient" app that takes you to the patient dashboard
2. a template that lets that lets other modules create app instances with the same Find Patient functionality, but leading to different URLs.

Feedback and improvements to this tutorial are welcome. 🙂