UI Framework Step By Step Tutorial

- Getting Started
- Create a simple page (also, an intro to Groovy)
- Include an existing fragment
- Create a Fragment
- Fragment Configuration in the Controller
- Fragment Configuration in the View
- Decorating our fragment
- How to Use AJAX in a Fragment
- Inter-fragment Communication Using the JavaScript Event Bus

Getting Started
To get started you need to create your own module that references the uiframework module. Details are at Using the UI Framework in Your Module.

For this entire tutorial I assume that you have created a module, and you have configured things so it runs in development mode.

Create a simple page (also, an intro to Groovy)
Create a file called "helloWorld.gsp" in yourmodule/omod/src/main/webapp/pages/.

Hello, world. Welcome to <b>OpenMRS</b>.

Now browse to the following link:
http://localhost:8080/openmrs/yourmodule/helloWorld.page

Congratulations, you've written your first page! A page that doesn't need to hit the database itself doesn't need a controller--it can be as simple as just a single GSP file.

Now let's make this page a bit more interesting. Change its code to be the following:

```
Hello, world.
<% if (context.authenticated) { %>
   And a special hello to you, $context.authenticatedUser.personName.fullName.
   Your roles are:
   <% context.authenticatedUser.roles.findAll { !it.retired }.each { %>
      $it.role ($it.description)
   <% } %>
<% } else { %>
   You are not logged in.
<% } %>
```

Now refresh the page. It should show you additional information. Also, notice that you did not have to reload anything for that change to take effect. Now let's step through what we did.

First, the GSP file extension tells the UI framework that this page is written as a Groovy template. Groovy templates are very similar to JSPs--they allow us to write regular HTML, with interpolated groovy code. You may insert groovy code using any of these forms:

- `$aVariableToPrint`
- `${ anExpressionToPrint }`
- `<%= "something to print which may include ${ somethingElse }" %>`
- `<% log.debug("Calls a method but prints nothing") %>`

In order to keep our code consistent and readable, please use the dollar-curly-brace construct as much as possible. For very short expressions, you may use just a dollar sign, but if you are going more than one property deep, or calling a method, use curly braces for clarity. Only use the less-than-percent-equals construct if you need to include an inner dollar-curly-brace expression within your outer one.

Also, please put a single space after the opening curly brace and before the closing one.

Note that the dollar sign is a special character in groovy templates. So if you are writing javascript, you need to escape it like `\$`. Since this makes it annoying to use jQuery, as a convenience you probably want to add something like this to the standard includes for all your pages:
You can transform your groovy objects to be used in your javascript:

```javascript
var jq = jQuery;

// pass a list of names of diagnoses from the patientSummary received from the controller into a javascriptFunction, i use '|s|' as my splitter and then remove empty items after the split
javascriptFunction('<% patientSummary.diagnoses.each { d -> %>|s|${d.name}<% } %>'.split('|s|').filter(v=>v!=''));
```

You may be wondering what Groovy code is like. 99% of legal Java code is also legal Groovy code, but Groovy gives you a bunch of extra language constructs which allow the code to be a lot cleaner and shorter.

- object.property notation will automatically find Java bean getters and setters. So "context.authenticated" is equivalent to "context.isAuthenticated()".
- You don't need semi-colons. A line break is sufficient to end a line of code. For style, please avoid using unnecessary semi-colons.
- You don't need return statements. The value of the last line of any code block is returned. For legibility, please use an explicit "return" in long methods, but not for short code blocks.
- Groovy supports Java's loop constructs, but it also adds "closures", and adds several methods to Java's Collection and Map classes which work with closures. There's a lot of complexity behind closures, but for now you may think of them like anonymous functions whose argument is called "it". (See complete details.)
  - In this example, on the collection of roles, we call the "findAll" method and pass it a simple closure. The closure is run on every element of the roles collection, and all the elements for which the closure returns true are collected into a new collection. (Remember that an explicit "return" statement is unnecessary, so "if(it.isRetired())" is equivalent to "return it.isRetired()" in Java.)
  - The resulting collection (i.e. the user's non-retired roles) then has the "each" method called on it. "each" also takes a closure, which it executes on each element in the collection. In this example (because the percent-greater-than takes us from Groovy back to HTML) the closure is effectively doing out.println("%role (%role.description)").
  - Don't worry if closures still seem a bit confusing, they will become clearer over time.

Oh, and where does "context" come from? The UI framework automatically binds several variables to the template, including an org.openmrs.api.context. Context as "context" so that you have a convenient way to access the OpenMRS API. This is incredibly powerful, but remember, "with great power comes great responsibility". To keep our code clean and legible, do not write significant API-related logic in GSP pages. It's fine to do simple things like "context.locationService.allLocations", but if code belongs in a controller, then put it there.

### Include an existing fragment

Now we're going to introduce the most important part of the OpenMRS UI Framework: "fragments". Basically, we want to make every effort to break down our UI into fragments of pages. This has two key benefits:

1. It lets us build a library of fragments that we can reuse all over the UI (and modules!) making application development much faster
2. It allows us to let administrators change the way pages are layed out, and even compose their own pages, just by including fragments

First, we're going to include a fragment that's already written. Change the code of your helloWorld.gsp to

```groovy
${ ui.includeFragment("uiframework", "helloUser") }
```

Now reload the page, and you should see a simple welcome message.

You're probably wondering what "ui" is. Since Groovy templates do not support tag libraries, we have collected a bunch of useful functionality in the "ui" object. (Your module can provide a utility function like this too.)

`ui.includeFragment` tells the UI framework to evaluate the given fragment and return its HTML as a String. Since we want to insert this into the page, we use the dollar-curly-brace construct to wrap the Groovy code.

The first argument to `ui.includeFragment` is a "fragment provider", which 99% of the time is the module id of the module the fragment is defined in. The second argument is the fragment name.

The uiframework module only contains this single helloUser fragment for test purposes. The uilibrary module includes standard helpful fragments.

### Create a Fragment

Now we are going to write our own fragment, listing all encounters in the database whose date is today. Writing a fragment follows the standard MVC pattern. A fragment has a FragmentController, which does business logic, and populates a FragmentModel. The framework then renders a FragmentView, which has access to the model variables. (A fragment is not actually required to have both a controller and a view, though it must have at least one. For example a decorator fragment is unlikely to need a controller. A fragment that does server-side logging would not need its own view.)
Fragment controllers follow specific naming conventions (a lesson learned from Ruby on Rails). The controller for the “encountersToday” fragment must be the controller() method of the class org.openmrs.module.yourmoduleid.fragment.controller.EncountersTodayFragmentController. (Basically take your fragment's name, capitalize the first letter, and put FragmentController after it.)

So let's start out by creating the class mentioned.

```java
package org.openmrs.module.yourmoduleid.fragment.controller;

import java.util.Calendar;
import java.util.Date;
import org.openmrs.api.EncounterService;
import org.openmrs.ui.framework.annotation.SpringBean;
import org.openmrs.ui.framework.fragment.FragmentModel;

/**
 * Controller for a fragment that shows the encounters that have taken place today
 */
public class EncountersTodayFragmentController {

    public void controller(FragmentModel model, @SpringBean("encounterService") EncounterService service) {
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.HOUR_OF_DAY, 0);
        cal.set(Calendar.MINUTE, 0);
        cal.set(Calendar.SECOND, 0);
        cal.set(Calendar.MILLISECOND, 0);
        Date startOfDay = cal.getTime();

        cal.add(Calendar.DAY_OF_MONTH, 1);
        cal.add(Calendar.MILLISECOND, -1);
        Date endOfDay = cal.getTime();

        model.addAttribute("encounters", service.getEncounters(null, null, startOfDay, endOfDay, null, null, null, null, false));
    }
}
```

Controller methods support flexible parameter types, and type conversions, like in Spring MVC. (See Flexible Method Signatures for UI Framework for complete documentation.) We also support the @SpringBean annotation that will cause the indicated Spring bean to be injected into the method call. (If you specify a value to the annotation, the bean is looked up by id, if not, then it's looked up by type.)

Like in Spring MVC, the fact that our controller method returns null means that we want to render the default view for this fragment. (Returning a non-null String would specify a different view name.) While a view could even live in a different module than the controller, and might use a non-groovy view provider, in most cases your fragment's view should be at fragmentName.gsp in the conventional location in the same module as its controller. So, create yourmodule/omod/src/main/webapp/fragments/encountersToday.gsp.
<table>
<tr>
<th>${ui.message("Encounter.type")}</th>
<th>${ui.message("Encounter.datetime")}</th>
<th>${ui.message("Encounter.location")}</th>
<th>${ui.message("Encounter.provider")}</th>
</tr>
</table>

We've introduced two more ui methods:

1. `ui.message` is used to display a message from the messages.properties files:
   
   ```java
   String message(String code, String... args)
   ```

2. `ui.format` is used to generate a standard display for an object

Notice that in Groovy, an empty collection is false when treated as a boolean. So we can simply say "if (encounters) ...".

Also note that we're using the "each" method on the encounters collection, and passing it a closure to run on each element. We could also use a for loop, but the each/closure syntax is cleaner and more legible once you get used to it.

In order to see this fragment, we need to include it in our helloWorld page:

```groovy
${ui.includeFragment("uiframework", "helloUser")}
${ui.includeFragment("yourmoduleid", "encountersToday")}
```

If you refresh your [http://localhost:8080/openmrs/yourmoduleid/helloWorld.page](http://localhost:8080/openmrs/yourmoduleid/helloWorld.page), you will probably not see any encounters, because your development database is unlikely to have any encounters today. To show off another feature of the UI Framework, let's temporarily change the date that our fragment uses. Go ahead and do a sql query like "select max(encounter_datetime) from encounter" to find a date you have an encounter on. Now, change the code in EncountersTodayFragmentController to manually set a date, maybe by adding something like the following:

```groovy
try {
    startOfDay = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").parse("2011-02-16 00:00:00.000");
    endOfDay = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").parse("2011-02-16 23:59:59.999");
} catch (Exception ex) { }
```

Once you've saved your changes, refresh [http://localhost:8080/openmrs/yourmoduleid/helloWorld.page](http://localhost:8080/openmrs/yourmoduleid/helloWorld.page) again, and notice that since you are in development mode, changes you make to fragment controllers are immediately recompiled without any action on your part.

If this feature doesn't seem to be working, make sure:

- Make sure you've set up your run configuration to support development mode, as described here: Using the UI Framework in Your Module
- in IntelliJ you actually have to do Build -> Compile on your class
- in Eclipse, your project is set to build automatically

**Fragment Configuration in the Controller**

The whole point behind fragments is to make them configurable. Each time you include a fragment, you can specify a `FragmentConfiguration`. So let's go ahead and make our fragment configurable in two ways:

1. allow the date range to be overridden (we'll handle this in the controller)
2. allow the columns to be chosen and reordered (we'll handle this in the view)
You may access fragment configuration properties in a controller method in either of two ways. If you declare a method parameter to be of type FragmentConfiguration, the framework will pass you the fragment's configuration. Or more conveniently, you can use the @FragmentParam annotation, which works just like Spring's @RequestParam.

So let's change the EncountersTodayFragmentController class as follows:

```java
package org.openmrs.module.yourmoduleid.fragment.controller;

import org.openmrs.api.EncounterService;
import org.openmrs.ui.framework.annotation.FragmentParam;
import org.openmrs.ui.framework.annotation.SpringBean;
import org.openmrs.ui.framework.fragment.FragmentModel;
import java.util.Calendar;
import java.util.Date;

public class EncountersTodayFragmentController {
    private Date defaultStartDate() {
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.HOUR_OF_DAY, 0);
        cal.set(Calendar.MINUTE, 0);
        cal.set(Calendar.SECOND, 0);
        cal.set(Calendar.MILLISECOND, 0);
        return cal.getTime();
    }

    private Date defaultEndDate(Date startDate) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(startDate);
        cal.add(Calendar.DAY_OF_MONTH, 1);
        cal.add(Calendar.MILLISECOND, -1);
        return cal.getTime();
    }

    public void controller(FragmentModel model,
                           @SpringBean("encounterService") EncounterService service,
                           @FragmentParam(value="start", required=false) Date startDate,
                           @FragmentParam(value="end", required=false) Date endDate) {
        if (startDate == null)
            startDate = defaultStartDate();
        if (endDate == null)
            endDate = defaultEndDate(startDate);

        model.addAttribute("encounters", service.getEncounters(null, null, startDate, endDate, null, null, null, null, null, false));
    }
}
```

As you see we are now getting startDate and endDate from the fragment configuration, but our original values as the defaults if they are not specified. So making this change will have no effect if you include the fragment without specifying config parameters, as we are currently doing in helloWorld.gsp. (At this point if you refresh http://localhost:8080/openmrs/yourmoduleid/helloWorld.page you will probably see no encounters, unless you have some today in your development database. Again, notice that the changes to the controller are picked up automatically. You don't need to restart anything. (In IntelliJ you do have to do Build -> Compile.)

Now, just so that we can see some data for example purposes, let's change the way we include the fragment on the helloWorld page so we get a day that does have encounters. Do something like this:

```groovy
${ ui.includeFragment("uiframework", "welcomeMessage") }

```

We are introducing another Groovy feature here. You can instantiate a Lists and Maps inline in your code.
To include a fragment with custom configuration, you provide a third parameter to the `includeFragment` method, of type Map.

Note that we have given the dates in String format. The OpenMRS UI Framework is capable of converting between most types for you, using Spring's type converters. (Many are provided out of the box, but if you need to add your own, see `Type Converters`.) The UI Framework expects dates to be in "yyyy-MM-dd [HH:mm:ss.SSS]" (i.e. year-month-day hour24:minute:second.millisecond, and neither the time-of-day or milliseconds are required), which differs from how the main OpenMRS web application does things.

**Fragment Configuration in the View**

Now we want to support another config parameter, i.e. allowing the user to specify which properties of the encounter are displayed in columns. Through a particular Groovy trick, we can do this purely in the fragment's view.

Let's change the way we include the fragment in `helloWorld.gsp` as follows:

```groovy
```

(The fragment config is now a Map that also includes a List as a value.)

To implement this functionality, we change the the `encountersToday.gsp` page as follows:

```groovy
<%  def props = config.properties ?: ["type", "datetime", "location", "provider"]%>
<%  if (encounters) {%>
  <table>
    <tr>
      <% props.each { %>
        <th>${ ui.message("Encounter." + it) }</th>
      <% } %>
    </tr>
    <% if (encounters) { %>
      <tr>
        <% encounters.each { enc -> %>
          <td><%= ui.format(enc."${prop}"%></td>
        <% } %>
      </tr>
    <% } else { %>
      <tr><td colspan="4">${ ui.message("general.none") }</td></tr>
    <% } %>
  </table>
<% } %>
```

We've introduced a few new tricks here. First, the easy ones:

- the UI Framework exposes the FragmentConfiguration as "config" in the view (so the "properties" that we specified while including the fragment are accessible from the fragment view as `config.properties`)
- Groovy allows loosely-typed variables. Saying "def x = 5" is approximately like saying "Object x = 5" in Java, but in Groovy it's actually considered good style.
- Groovy adds a new operator ?: (also called the Elvis operator--look at the hair) which returns the first argument if it is not null, and the second otherwise. So "a ?: b" is equivalent to "a != null ? a : b".

So the first line of code sets the variable "props" to be equal to the "properties" property of the FragmentConfiguration if specified, or our original property list if you don't specify one explicitly. You should use this idiom a lot.

I previously said you can think of a closure as an anonymous function whose parameter is called "it". You may actually specify the parameter name a closure takes with the syntax:
In our code we now have a closure that contains another closure, so we explicitly specify parameter names of "enc" and "prop" to make it clear what we are doing.

We use one final advanced Groovy trick, which is that:

- `object."property"` is equivalent to `object.property`
- Groovy strings let you insert values, e.g. `${prop}` is basically the same as `prop.toString()`.
- Therefore `enc."${prop}"` means to look up the prop property (interpreted dynamically) of the enc object.
- we also had to use less-than-per cent-equals notation to allow that value to contain a dollar-curly-brace expression.

This trick makes the code quite confusing to read unless you know what it's doing, so make sure you have good variable names, indentation, and spacing if you use this. (In fact we should create a reusable fragment "table" that encapsulates this dynamic-property code in one place, allowing other code to get this behavior without the complexity.)

### Decorating our fragment

Our helloWorld page is actually kind of ugly. We can make it prettier by creating a "decorator" and wrapping it around our encountersToday fragment.

A decorator is just a fragment that, by convention, lives in the decorator subfolder of fragments, which counts on having the framework pass it a "content" config parameter.

Go ahead and create a simple one at yourmodule/omod/src/main/webapp/fragments/decorator/widget.gsp, like:

```gsp
<% ui.decorateWith("yourmoduleid", "widget", [title: "Today's Encounters"]) %>
```

This tells the framework that after it renders the encountersToday fragment, it should wrap it in the "widget" decorator, passing a "title" fragment config parameter to the decorator.

The downside of decorating our fragment this way is that it means our fragment will always be decorated this way. Sometimes this is fine (for example in the emr module every page asks to be decorated with StandardEmrPage), but often we would like the person that is including our fragment to decide whether or not to decorate it, and how. So there's also a way to allow the consumer to decorate a fragment. In helloWorld.gsp we can change the way we include our fragment to be like this:

```gsp
```

Once you start adding lots of configuration to a parameter, especially including nested lists and maps, then you should add line breaks and indentation to keep things maximally legible. Remember, you will not be the only one looking at your code!
We have added three additional configuration parameters to our fragment include, "decoratorProvider" (which module our decorator comes from), "decorator" (the name of a decorator) and "decoratorConfig" (its optional fragment config). Whenever you do a fragment include, the UI framework looks for the "decorator" parameter, and decorates the fragment as requested by its consumer. Note that the fragment is only allowed to be decorated once—if a fragment requests decoration, and its consumer also asks for it to be decorated, the decoration requested by the fragment itself "wins".

How to Use AJAX in a Fragment

Let's say that we want to allow the user to reload the table of encounters by clicking on a "refresh" button, without reloading the entire page. We need to do this using AJAX. (This is a bit of a contrived example. A more realistic example would be to let the user choose a date from a dropdown, and have the table refresh with encounters on that date. The mechanism to support both of these is the same.)

The UI Framework supports "Fragment Actions" which are basically methods in the fragment's controller class that are easy to invoke from views. They aren't required to be asynchronous, but they are particularly helpful for AJAX interactions. So we add a getEncounters method to our fragment controller that fetches the encounters for a given date, and returns them as a JSON list:
package org.openmrs.module.emr.fragment.controller;

import org.openmrs.Encounter;
import org.openmrs.api.EncounterService;
import org.openmrs.ui.framework.SimpleObject;
import org.openmrs.ui.framework.UiUtils;
import org.openmrs.ui.framework.annotation.FragmentParam;
import org.openmrs.ui.framework.annotation.SpringBean;
import org.openmrs.ui.framework.fragment.FragmentModel;
import org.springframework.web.bind.annotation.RequestParam;
import java.util.Calendar;
import java.util.Date;
import java.util.List;

public class EncountersTodayFragmentController {

    private Date defaultStartDate() {
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.HOUR_OF_DAY, 0);
        cal.set(Calendar.MINUTE, 0);
        cal.set(Calendar.SECOND, 0);
        cal.set(Calendar.MILLISECOND, 0);
        return cal.getTime();
    }

    private Date defaultEndDate(Date startDate) {
        Calendar cal = Calendar.getInstance();
        cal.setTime(startDate);
        cal.add(Calendar.DAY_OF_MONTH, 1);
        cal.add(Calendar.MILLISECOND, -1);
        return cal.getTime();
    }

    public void controller(FragmentModel model,
                           @SpringBean("encounterService") EncounterService service,
                           @FragmentParam(value="start", required=false) Date startDate,
                           @FragmentParam(value="end", required=false) Date endDate) {
        if (startDate == null)
            startDate = defaultStartDate();
        if (endDate == null)
            endDate = defaultEndDate(startDate);

        model.addAttribute("encounters", service.getEncounters(null, null, startDate, endDate, null, null, null, null, null, false));
    }

    public List<SimpleObject> getEncounters(@RequestParam(value="start", required=false) Date startDate,
                                             @RequestParam(value="end", required=false) Date endDate,
                                             @RequestParam(value="properties", required=false) String[] properties,
                                             @SpringBean("encounterService") EncounterService service,
                                             UiUtils ui) {
        if (startDate == null)
            startDate = defaultStartDate();
        if (endDate == null)
            endDate = defaultEndDate(startDate);

        if (properties == null) {
            properties = new String[] { "encounterType", "encounterDatetime", "location", "provider" };}

        List<Encounter> encs = service.getEncounters(null, null, startDate, endDate, null, null, null, null, null, false);
        return SimpleObject.fromCollection(encs, ui, properties);
    }
}

We've added another public method in our controller class. (Any public method on a fragment controller can be accessed as a fragment action.) We know we want to return JSON, but the UI Framework actually takes care of this for us: we just have to return a simple Object, without any hibernate proxies. The last line of this method shows a convenience method that does this for a collection, and I welcome improvements to the SimpleObject class.

The method signature of a fragment action method can be flexible, although the available options are more limited than those on the main controller() method, since a fragment action is just an HTTP request, without the context of a page or a fragment configuration. Most powerfully, we can use Spring's @RequestParam annotation to automatically convert HTTP parameters to our desired datatypes. (See Flexible Method Signatures for UI Framework Controller and Action Methods for complete documentation of allowed parameter types and return types.)

That's all. The server-side definition of our AJAX interaction is quite simple--it relies on the UI Framework to do all the busy work.

Now we need to make our encountersToday.gsp view call this method, and redraw the table when the results come back asynchronously.
<ui.includeJavascript("yourmoduleid", "jquery.js")>

def id = config.id
def props = config.properties ?: ['encounterType', 'encounterDatetime', 'location', 'provider']

<ui.resourceLinks() />

<script>
jq = jQuery;

jq(function() {
    jq('#${ id }_button').click(function() {
        jq.getJSON('${ ui.actionLink("getEncounters") }', {
            'start': '${ config.start }',
            'end': '${ config.end }',
            'properties': [ '<%= props.collect { "' + it + '" }.join(',') %>' ]
        })
            .success(function(data) {
                var tbody = jq('#${ id } > tbody');
                for (index in data) {
                    var item = data[index];
                    var row = '<tr>' + 
                        props.each { prop ->
                            row += '<td>' + item.${ prop } + '</td>'
                        } + '</tr>'
                    tbody.append(row);
                }
            })
            .error(function(xhr, status, err) {
                alert('AJAX error ' + err);
            });
});
</script>

<input id="${ id }_button" type="button" value="Refresh" />

<table id="${ id }">
  <thead>
    <tr>
      %
      props.each { prop ->
        <th>${ ui.message("Encounter." + prop) }</th>
      }%
    </tr>
  </thead>
  <tbody>
    % if (encounters) { %
      %
      encounters.each { enc ->
        <tr>
          %
          props.each { prop ->
            <td>${ ui.format(enc."$prop") }</td>
          }%
        </tr>
      }%
    % } else { %
      <tr>
        <td colspan="4">${ ui.message("general.none") }</td>
      </tr>
    % }%
  </tbody>
</table>
Since we’re going to be updating our table via javascript, we put an "id" attribute on it so we can select it. But since fragments should support being included on a page more than once, we don’t want to hardcode that "id". The UI Framework guarantees that your fragment configuration always has an "id" property, by autogenerating one suitable for use as a DOM id if the fragment request doesn’t provide one.

We’ve also made a few changes to HTML, specifically adding thead and tbody groupings to make it easier to clear and redraw the body, and adding a refresh button.

Most of the code we added to the view is javascript code, done via jQuery. (If you don’t know how jQuery works, you need to read a tutorial on it now, since it is out of scope for this one.)

First, we ensure that jQuery is included on the page by doing ui.includeJavascript("yourmoduleid", "jquery.js"). You also need to put a jquery.js file in your module under omod/src/main/webapp/resources/scripts. You can obtain the latest jQuery file at http://jquery.com/download/. After downloading the file be sure to rename it to jquery.js.

We define jq as a shorthand to refer to jQuery, rather than the traditional $. (This is because the dollar sign has special meaning in Groovy templates, and if we used that, we’d always have to escape it with a \, which makes the code quite hard to read. It gets very confusing to skim code when the dollar sign can either mean jQuery or groovy.)

We then make a standard jQuery getJSON call, giving it a URL and some parameters, and attaching success and error callbacks to it. The success callback is straightforward, though it’s worth noting that we use the same ‘props’ configuration object we defined earlier.

The most interesting thing in this fragment is the call to

```javascript
ui.actionLink("getEncounters")
```

This generates a link to the "getEncounters" action in the current fragment. (If you wanted to call this action from a different fragment, or directly from a page, you’d want to explicitly say actionLink("yourmoduleid", "encountersToday", "getEncounters").)

Besides that, there are a few other interesting things we do in the JSON call. First, we include the start and end config model properties, using a convenience method to format the dates in yyyy-MM-dd format. To turn our ‘props’ Java object into a javascript list, we use a new Groovy collection method, "collect", which applies a closure to all items in the collection, and collects the results into a new list. (In this case we just use that to put single quotes around each property name.) We also use the Groovy collection method "join" which turns a collection into a String, with the given argument as a separator.

Note that unlike in frameworks like JSF and Wicket, fragments in the OpenMRS UI Framework are stateless, and they don’t have any conception of how exactly they’re configured in a particular browser window. So when the browser asks to load the encounters to be shown, it needs to include any necessary configuration parameters in the query. (In this case those are "start", "end", and "properties").

Inter-fragment Communication Using the JavaScript Event Bus

The facts that we are writing reusable page fragments, rather than whole pages, is very powerful, and will lead to lots of flexibility and code reuse across the OpenMRS ecosystem. But it also means we need to introduce a new (more complicated) paradigm for having fragments communicate with each other. For example, we’ve just built a fragment that shows a list of encounters. The UI Library module includes a fragment that can use ajax to load a preview of an encounter that you select on a page. Clearly, these two fragments are very useful when combined. But at the same time, neither one of them should depend on the other, since they are both perfectly functional alone, and they can both be used with other fragments too. So in order to make our fragments as flexible and reusable as possible, we need to “decouple” them.

The UI Framework includes a mechanism (based on the javascript event bus PageBus) that allows fragments to publish and subscribe to messages, and pass content with those messages.

First, you need to download and install version 1.0 of the UI Library module from the OpenMRS module repository here. (Do get version 1.0 for this tutorial, since we make no promises that the widgets it contains won’t change in later module versions. For actual usage, you’ll want the latest version.)

To demonstrate that functionality, we are going to augment our encountersToday fragment so that it publishes an "encounterSelected" message when you click on one of the listed encounters. Later we’ll play around with a couple things we can do with this message.

Edit yourmodule/omod/src/main/webapp/pages/helloWorld.gsp to add one line to the top:

```groovy
<%= ui.decorateWith("standardPage") %>
```

This decorator is provided by the UI Library module, and it includes standard js and css that the widgets provided by the module depend on.

Now, we can edit the encountersToday.gsp fragment (omitting the refresh button and ajax for simplicity)
We've added a "showSelectButton" config property, which adds an "info" icon at the start of each row. When clicked, these info icons publish encounterSelected messages, giving the id of the encounter selected. Note that the message also includes the (dynamic) id of this fragment. It's possible that multiple fragments on a page will publish an "encounterSelected" message, and we'd like consumers of these fragments to be able to distinguish between them.

Also notice the usage of the "ui.resourceLink(String provider, String path)" method, which allows us to reference content that modules provide in their resources folder. (In this case we are referring to a resource provided by a specific module, which is good style, but if we'd done ui.resourceLink("images/info_16.png"), that would search through all modules until it finds a matching resource.)

We're following good practices here and trying to make our fragment reusable. It can have a select button, but it doesn't require one. In reality we'd want to go further and allow the consumer of the fragment to specify the icon to use, etc.

In order to use this new feature we've added, we need to make some changes to helloWorld.gsp where we define our page. For now, let's just set things up so that when you select an encounter, we open the view encounter page for it.
We've done a couple things here. First as we include the encountersToday fragment we enable the select button, and we explicitly give it an id. (Otherwise we wouldn't be able to connect to its events.) Then, via javascript we subscribe to our fragment's encounterSelected message, providing a function that goes to the encounter page with the given id. (Stylistically I like to put javascript that deals with wiring up fragments by subscribing to messages at the very bottom of the page.)

Another thing to notice is the "ui.pageLink(String pageName)" function, which returns the appropriate link to a page, so you don't have to worry about context paths or anything.

To see this in action, try refreshing http://localhost:8080/openmrs/pages/helloWorld.page and click on one of the info icons. You'll get a 404 error, because we have not actually written an encounter page in this tutorial, but you get the idea...

Now, let's do something a bit different. On the helloWorld page, let's also include the "infobox" fragment, and connect it to our encountersToday fragment such that selecting an encounter show a preview of it on our current page, via ajax.

We've done a couple things here. First as we include the encountersToday fragment we enable the select button, and we explicitly give it an id. (Otherwise we wouldn't be able to connect to its events.) Then, via javascript we subscribe to our fragment's encounterSelected message, providing a function that goes to the encounter page with the given id. (Stylistically I like to put javascript that deals with wiring up fragments by subscribing to messages at the very bottom of the page.)

Another thing to notice is the "ui.pageLink(String pageName)" function, which returns the appropriate link to a page, so you don't have to worry about context paths or anything.

To see this in action, try refreshing http://localhost:8080/openmrs/pages/helloWorld.page and click on one of the info icons. You'll get a 404 error, because we have not actually written an encounter page in this tutorial, but you get the idea...

Now, let's do something a bit different. On the helloWorld page, let's also include the "infobox" fragment, and connect it to our encountersToday fragment such that selecting an encounter show a preview of it on our current page, via ajax.
We've included a second fragment (which is provided by the UI Library module), next to our encountersToday one (using a table to do a quick hacky layout). And we've changed the javascript callback for the encounterSelected event to call a method on the infobox fragment (note that that's the id we included it with).

The key point here is that we've written a fragment that we were able to reuse for two different behaviors (go to encounter page, and preview encounter) without changing the fragment itself, but just by wiring it up to different actions. If you follow these patterns while building new application functionality, you will also build up a reusable library of fragments that you can use to assemble future UIs even more quickly. (Imagine writing a page to quickly collect weights and heights by wiring up a find-patient widget, to an obs-table widget with a form widget.) The same pattern will eventually let system administrators compose pages out of fragments that can be wired together using a (not-yet-ready) XML format.