QI `GenericDivView` Tutorial

v1.2 - October 4, 2019

This tutorial shows the basics behind the `GenericDivView` that is part of jPOS-EE's **QI** module.

QI

Some of our products, such as jCard, use a web user interface developed using the Vaadin 8 framework. The **QI module** is part of jPOS-EE and provides a way of configuring the general layout of the web app, with menus, sidebars, permissions, etc. For a quick introduction to QI, refer to the corresponding section in the jPOS-EE manual. Here we do an even briefer introduction to support the explanation of `GenericDivView`.

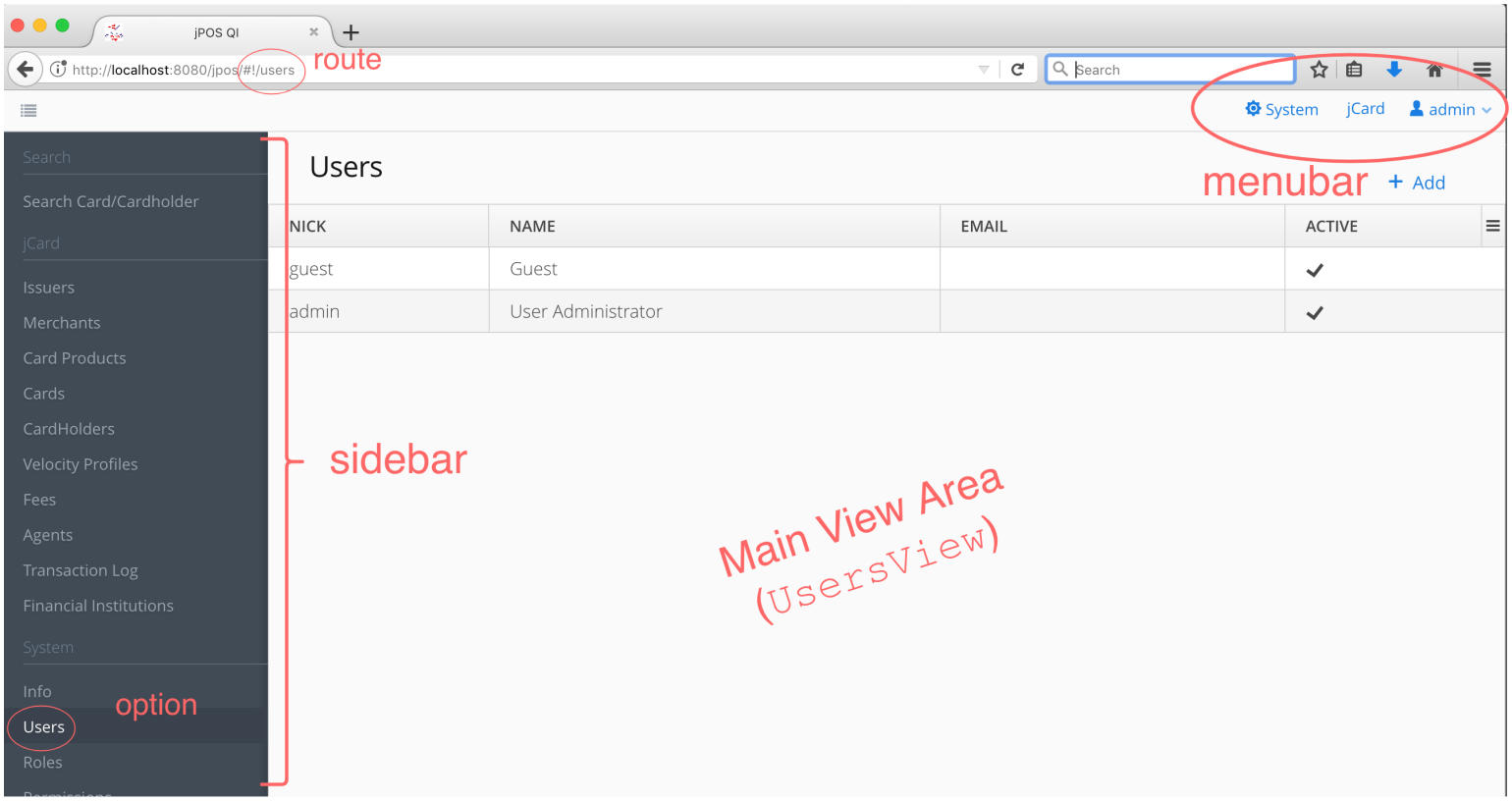
When a QI application is launched (such as `modules/app` in jCard), it will read a deploy file in the `deploy` directory, normally called `00_qi.xml`

The general format is

```
<xml-config name="QI" logger="Q2">
  <locale>en-US</locale>
  <locale>es-ES</locale>

  <messages>some_properties_file</messages>
  <!--  ... etc ...
        the label translations, per locale
  -->

  <!-- The menu bar runs at the top of the screen. Each menu item has an "action". -->
  <menubar>
        <menu name="TopMenu1_label" action="sidebar1"  />
        <menu name="TopMenu1_label" action="sidebar2" />
  </menubar>

  <!-- A sidebar runs down the left side of the screen. Each "option" has an "action". -->
  <sidebar id="sidebar1">
      <!-- A section is just a label used as a separator -->
      <section name="First_Section" />
      <option name="option1_label"  action="option1" />
      <option name="option2_label"  action="option2" />
      ...
      ...
      <section name="Second_Section" />
      <option name="users_label"  action="users" />
      ...
  </sidebar>

  <sidebar id="sidebar2">
      ...
  </sidebar>

  <!-- Each "action" from a menubar "menu" or sidebar "option" maps to a Vaadin View implemented by a specific class.
       The "route" is the URL part generated when navigating to that view.
       Each "view" also identifies a sidebar that will be shown when the view is activated.
   -->
```

```
  <view route="home"   class="org.jpos.qi.views.DefaultView" perm="*"      sidebar="system" />
  <view route="about"  class="org.jpos.qi.system.AboutView"  perm="login" sidebar="system"/>

  <!-- some views may have additional "properties" and other custom XML (configured by configuration polling by the
       class itself). The one below is given just as an example, and the meaning is beyond the scope of this tutorial.
  -->
  <view route="users" class="org.jpos.qi.eeuser.UsersView" perm="users.write" sidebar="system">
    <property name="entityName" value="user"/>
    <attribute name="id" field="false" required="true"/>
    <attribute name="nick" length="64" regex="&WORD_PATTERN;" required="true" expand-ratio="1"/>
    ...
  </view>

  ...
</xml-config>
```

Here's a sample screen from jCard. The *Users* `option` and `view` from the xml example above would be equivalent to what is marked and shown on the screen shot below.



When the user clicks on a `menu` item from the top `menubar` or an `option` item from the displayed `sidebar`, its corresponding `action` is mapped into a `route` URL fragment. The browser navigates to that route, and the `view` element that declares said `route` attribute is displayed in the Main View Area. Also, if the `view` element has a `sidebar` attribute, the sidebar section is changed to the `sidebar` with that `id`. The displayed view is an instance of `com.vaadin.navigator.View` and implemented by the class given in the `class` attribute.

GenericDivView

As we've seen, the main area of the screen is rendered by an instance of `com.vaadin.navigator.View` using the usual

Vaadin API. However, there's a special kind of view called `GenericDivView` that provides an "empty" `<div>` that can be dynamically populated by external HTML files, loading JavaScript and CSS dependencies. This allows us to "bridge" the world of server-side Vaadin development in Java, and the world of traditional client-side web development.

The general form to specify a `GenericDivView` is:

```
<xml-config>
...

<sidebar id="sidebar1">
    <option name="MyView"  action="myview" />
    ...
</sidebar>
...
<view route="myview" class="org.jpos.qi.views.GenericDivView" perm="*" sidebar="sidebar1">
    <!--
        this is a *filesystem* path, relative to the "current working dir" of the process
        the content of this file will be used as the html content of the GenericDivView wrapping div
    -->
    <property name="src" value="path/to/div.html" />

    <!--
        External dependencies on script and css files.
        They are URL paths that will be resolved by Vaadin and used by the browser to request
        the dependency.
        The path could be relative to the root of the "webapp", or use a "vaadin:" protocol in
        order to be resolved into the VAADIN 8 special directory.
    -->
    <property name="script" value="vaadin://js/some_other_script.js" />
    <property name="script" value="webapps/js/some_script.js" />
    <property name="css"    value="webapps/css/my_style.css" />
    ...
</view>
```

`GenericDivView` example

As a "quick and dirty" example, we'll use the jPOS-EE module `testbed` and add a new sidebar option with a sample script based on the Vue.js framework. This section assumes you are familiar with usual jPOS development, such as using `gradle`, the `devel.properties` file, etc.

> *NOTE: To keep things simple and the level of noise low, this example doesn't use any other external tools for web development, frameworks, modern JavaScript/ECMAScript conventions, or other current web development practices. The integration of such tools and practices are very project- and team-dependant, and are left as an excercise for the reader.*

1. Clone jPOS-EE

If you haven't done so, please clone JPOS-EE from https://github.com/jpos/jPOS-EE . The `testbed` module is based on **Jetty** and has the **QI** modules as dependencies. There are other dependencies that we don't care about.

2. Create the main div html file

First, create the file `modules/testbed/src/dist/webapps/myview.html` with the following content:

```
<div id="my_view">
  <span>This is the html from myview.html</span>

  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">Click to Reverse</button>

  <p id="output_text"></p>
</div>

<script>
    var vueApp = new Vue({
        el: '#my_view',

        data: {
            message: 'Hello GenericDivView!'
        } ,

        methods: {
            reverseMessage: function () {
                this.message = this.message.split('').reverse().join('');
                document.getElementById('output_text').innerText= "did it work?";
            }
        }
    });
</script>
```

3. Download Vue.js

We're going to be using a local copy of Vue.js. Follow whatever method you want from https://vuejs.org/v2/guide/installation.html to get a hold of a copy of the library. For example, download it directly from https://cdn.jsdelivr.net/npm/vue/dist/vue.js and put the copy in `modules/testbed/src/main/webapp/VAADIN/js/vue.js` (create any intermediate directories that may be missing)

4. Add some style

Let's add a little external style to our div above.

Create the file `modules/testbed/src/dist/webapps/css/my_style.css`:

```
#my_view {
    background-color: #ffff88;  /* light yellow */
}
#my_view button {
    color: white;
    background-color: red;
}
```

5. Edit the QI layout file

Now, edit the file `modules/testbed/src/dist/deploy/00_qi.xml`.

First, find the `system` sidebar and add a new option at the top

```
...
<sidebar id="system">
```

```
    <section name="system"/>
        <option name="My View" action="myview"/>
...
```

Now, we'll ned to add a `<view>` for the `myview` action/route. Find the `<!-- VIEWS -->` section and add:

```
<view route="myview" class="org.jpos.qi.views.GenericDivView" perm="*" sidebar="system">
    <!-- this is the id of the <div> element that's going to wrap myview.html -->
    <property name="id"  value="myview_wrap"/>

    <!-- this file was taken from the `dist` directory of the source, and copied into
         the build
    -->
    <property name="src" value="webapps/myview.html" />

    <property name="script" value="vaadin://js/vue.js" />
    <property name="css"    value="/css/my_style.css" />    <!-- the leading slash is important! -->
</view>
```

6. Build, create and configure the DB, run it

For `testbed` to work, we need to setup a database that will allow us to login, and other parts of the system to work. The module's DB connection configuration is in file `modules/testbed/src/dist/cfg/db.properties` and it's hardcoded to use MySQL, a db name of `jposee`, and username `jpos`, and password `password`.

> *How to use a different database engine and/or configuration is outside the scope of this tutorial.*

After your database has been created, we need to build the system and initialize the db. The following commands should be run from the root of the cloned jPOS-EE project.

- build with the command `gradle :modules:testbed:installApp`
  NOTE: for reasons beyond the scope of this document, please run the above command twice 😀😀
- run Q2 in "command line mode" `./modules/testbed/build/install/testbed/bin/q2 -cli`
- at the `q2>` prompt, enter the following commands

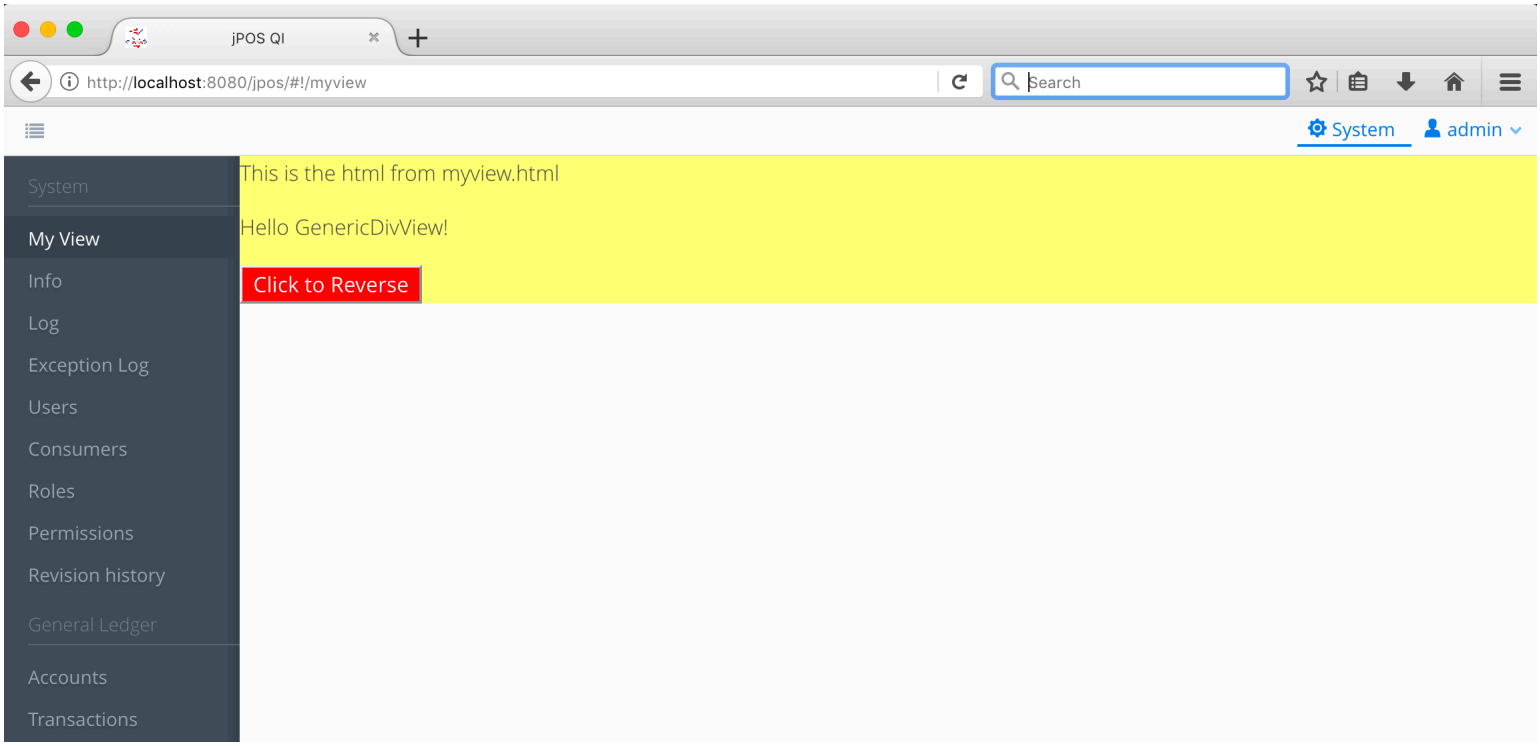
  ```
  createschema - true
  addrole admin sysadmin login sysconfig.read sysconfig.write users.read users.write accounting
  adduser admin -ptest -radmin -n"System Administrator"
  addgluser admin -n"System Administrator" -p"read" -p"write" -p"post" -p"checkpoint"
  sysconfig
  add sys.REMEMBER_PASSWORD_ENABLED true
  add sys.MAX_LOGIN_ATTEMPTS 5
  add sys.PASSWORD_AGE 90
  add perm.login "Can log into the system"
  add perm.sysadmin "Has administrator privileges"
  add perm.sysconfig "Has access to sysconfig records"
  exit

  shutdown
  ```

- run `testbed` with the command `./modules/testbed/build/install/testbed/bin/q2`

7. Test it

If `testbed` launched correctly, you should now be able to point your browser to `http://localhost:8080` and log in with user `admin` and password `test`.

You should get a screen with a menubar at the top that says `System`. Click on it and the sidebar will be displayed. The first item (`<option>` in `00_qi.xml`) of the sidebar will be called `My View`. Clicking on it will navigate to http://localhost:8080/jpos/#!/myview (notice how the last part of the URL is the `action` attribute of the sidebar `option`).



The main area of the screen will now show your `GenericDivView` which is a **Vaadin 8** `View` that has been configured from the options given in the `<view route="myview" ...>` of our example:

- populated by the content of `webapps/myview.html`
- some `script` and `css` dependencies have been **dynamically** added to the current document (you can inspect it with the web development tools of your browser and find those dependencies added into the document's `<head>`.

8. Summary

There are many things not covered here. Some people may choose to develop using tools such as webpack, transpilers, using libraries for calling REST APIs using Ajax or similar methodologies, etc. In order to allow for that, you may need to configure the web server (Jetty at the moment) and add other files to aid you in your toolchain.