Building the Administration UI using Forge

What Will You Learn Here?

You've just defined the domain model of your application, and all the entities managed directly by the end-users. Now it's time to build an administration GUI for the TicketMonster application using JAX-RS and AngularJS. After reading this guide, you'll understand how to use JBoss Forge to create the JAX-RS resources from the entities and how to create an AngularJS based UI.

We'll round out the guide by revealing the required, yet short and sweet, configuration.

The tutorial will show you how to perform all these steps in JBoss Developer Studio, including screenshots that guide you through.

Setting up Forge

JBoss Developer Studio

Forge is available in JBoss Developer Studio 8. You would have already used Forge in the Introductory chapter.

You can start Forge in JBoss Developer Studio, using the **Ctrl** + **4** (Windows/Linux) or **Cmd** + **4** (Mac OS X) key stroke combination. This would launch the Forge action menu from where you can choose the desired commands to run in a particular context.

Or alternatively, to use the Forge Console, navigate to *Window* → *Show View* → *Other*, locate *Forge Console* and click *OK*. Then click the *Start* button in top right corner of the view.

Getting started with Forge

Forge is a powerful rapid application development (aimed at Java EE 6) and project comprehension tool. It can operate both on projects it creates, and on existing projects, such as TicketMonster. If you want to learn more about Forge, head over to the JBoss Forge site.

Forge can scaffold an entire app for you from a set of existing resources. For instance, it can generate a HTML5 scaffold with RESTful services, based on existing JPA entities. We shall see how to use this feature to generate the administration section of the TicketMonster application.

Generating the CRUD UI

Forge Scripts

Forge supports the execution of scripts. The generation of the CRUD UI is provided as a Forge script in TicketMonster, so you don't need to type the commands everytime you want to regenerate the Admin UI. The script will also prompt you to apply all changes to the generated CRUD UI that listed later in this chapter. This would relieve us of the need to manually type in the changes.

To run the script:

```
run admin_layer.fsh
```

Scaffold the AngularJS UI from the JPA entities

Scaffolding capabilities are available through the "Scaffold: Setup" and "Scaffold: Generate" commands in the Forge action menu. The first command is used to set up the pre-requisites for a scaffold in a project - usually static files and libraries that can be installed separately and are not modified by subsequent scaffolding operations. The second command is used to generate various source files in a project, based on some input files (in this case JPA entities).

In the case of the AngularJS scaffold, an entire CRUD app (a HTML5 UI with a RESTful backend using a database) can be generated from JPA entities.

Forge can detect whether the scaffold was initially setup during scaffold generation and adjust for missing capabilities in the project. Let's therefore go ahead and launch the "Scaffold: Generate" command from the Forge action menu:

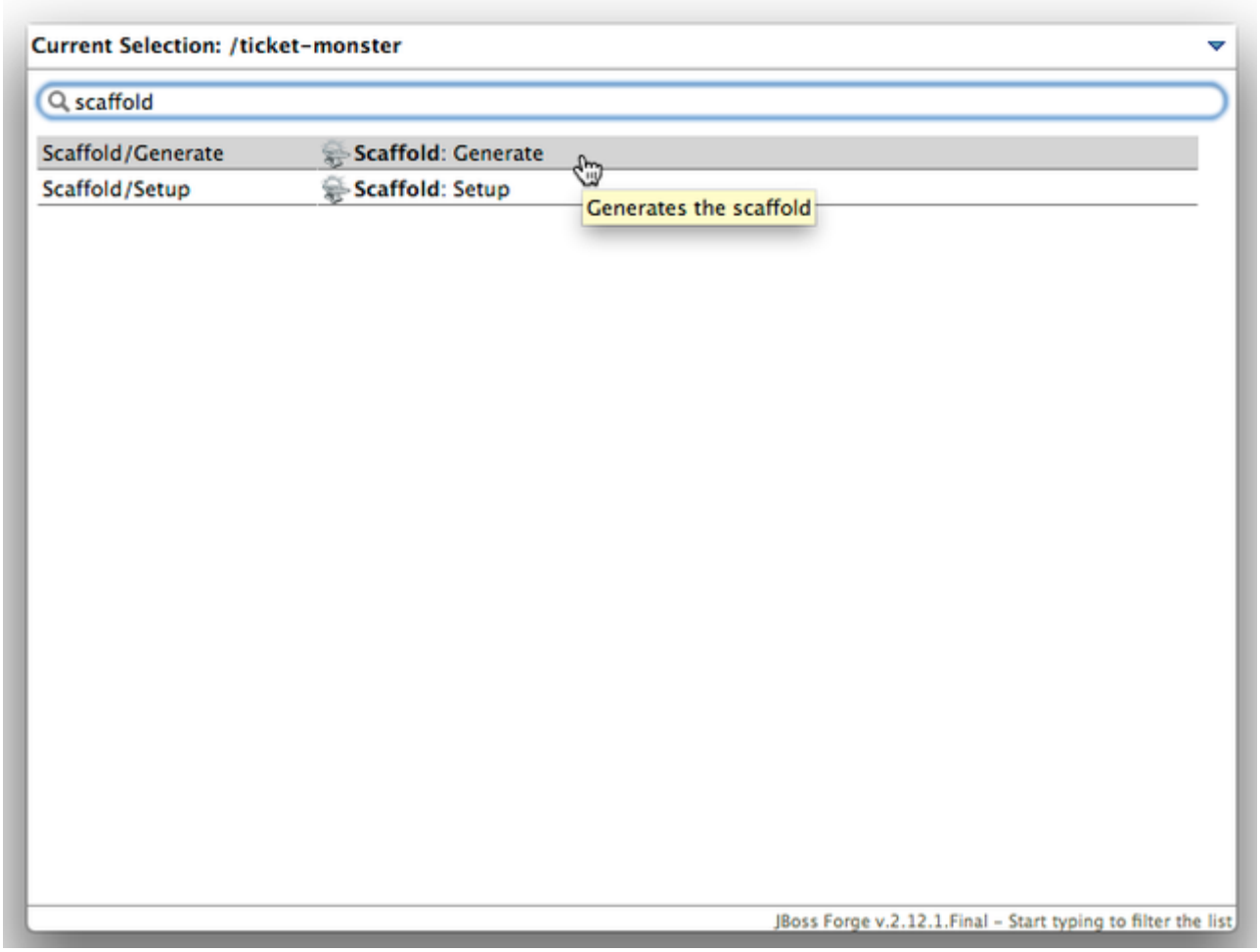

Figure 1. Filter the `Scaffold: Generate` command in the menu

We're now prompted to select which scaffold to generate. Forge supports AngularJS and JSF out of the box. Choose `AngularJS`. The generated scaffold can be placed in any directory under the web root path (which corresponds to the `src/main/webapp` directory of the project). We'll choose to generate the scaffold in the `admin` directory.

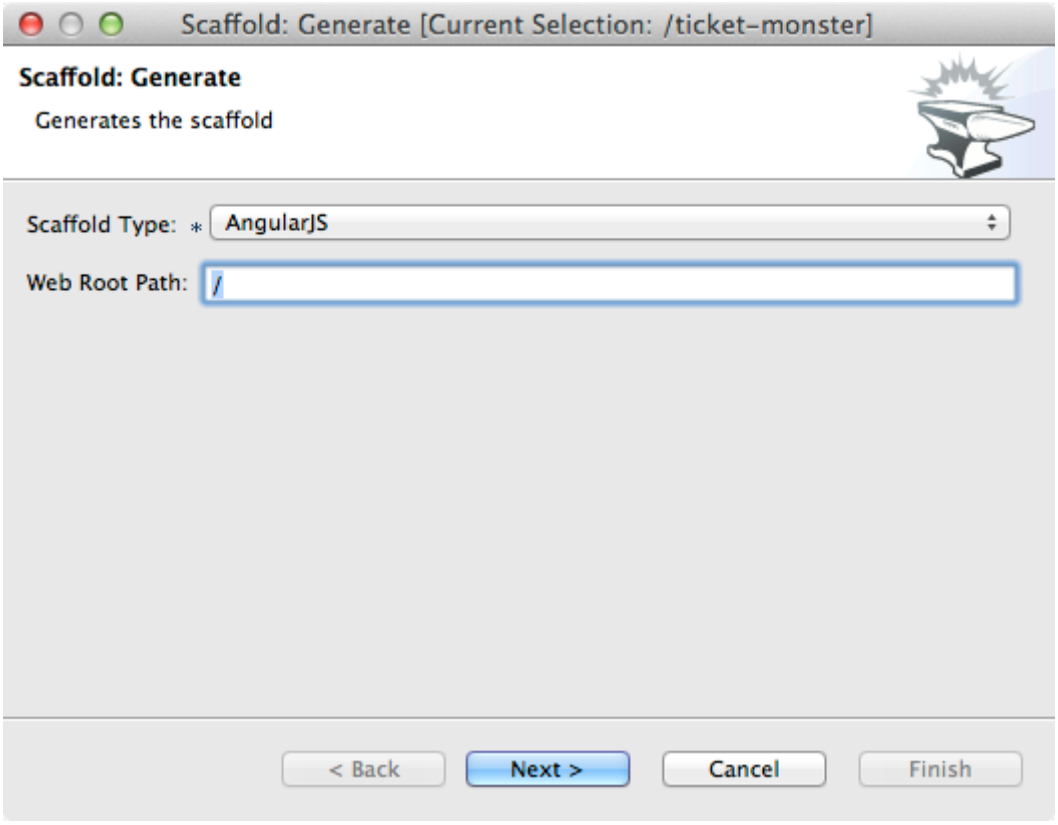

Figure 2. Launch the `Scaffold: Generate` command

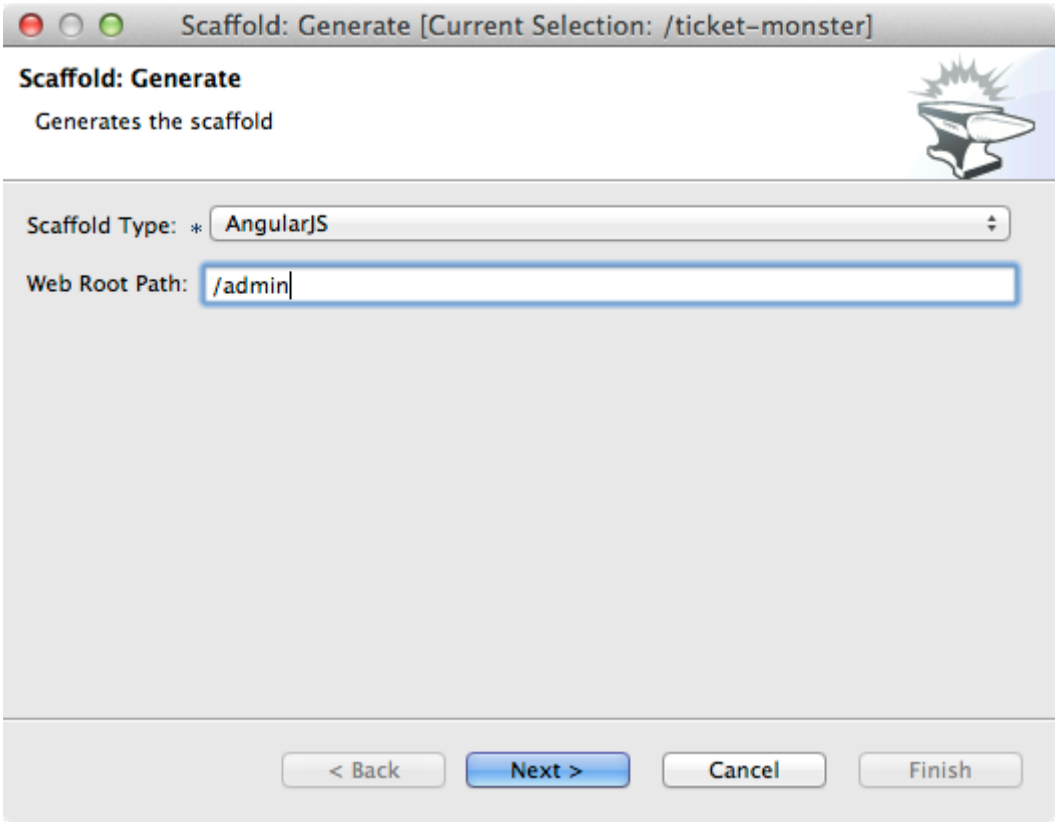

Figure 3. Select the scaffold to generate and the web root path

Click the `Next` button, and proceed to choose the JPA entities that we would use as the basis for the scaffold. You can either scaffold the entities one-by-one, which allows you to control which UIs are generated, or you can generate a CRUD UI for all the entities. We'll do the latter. We'll also choose to generate REST resources for the entities, since the existing REST resources are not suitable for CRUD operations:

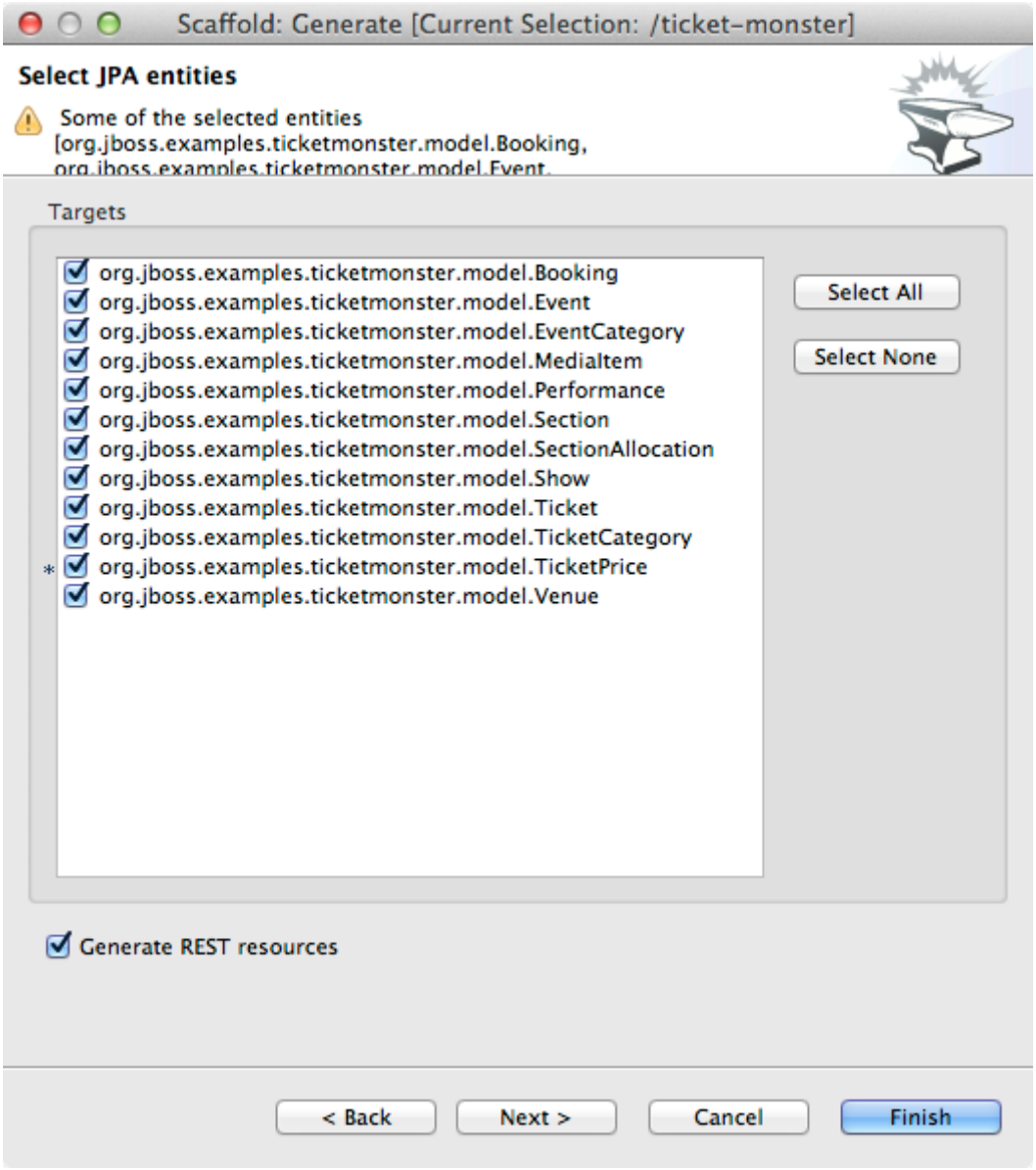

Figure 4. Select the JPA entities to use for generation

Click the `Next` button, to configure the nature of the REST resources generated by the scaffold. Multiple strategies exist in Forge for generating REST resources from JPA entities. We'll choose the option to generate and expose DTOs for the JPA entities, since it is more suitable for the TicketMonster object model. Provide a value of `org.jboss.examples.ticketmonster.rest` as the target package for the generated REST resources, if not already specified. Click `Finish` to generate the scaffold.

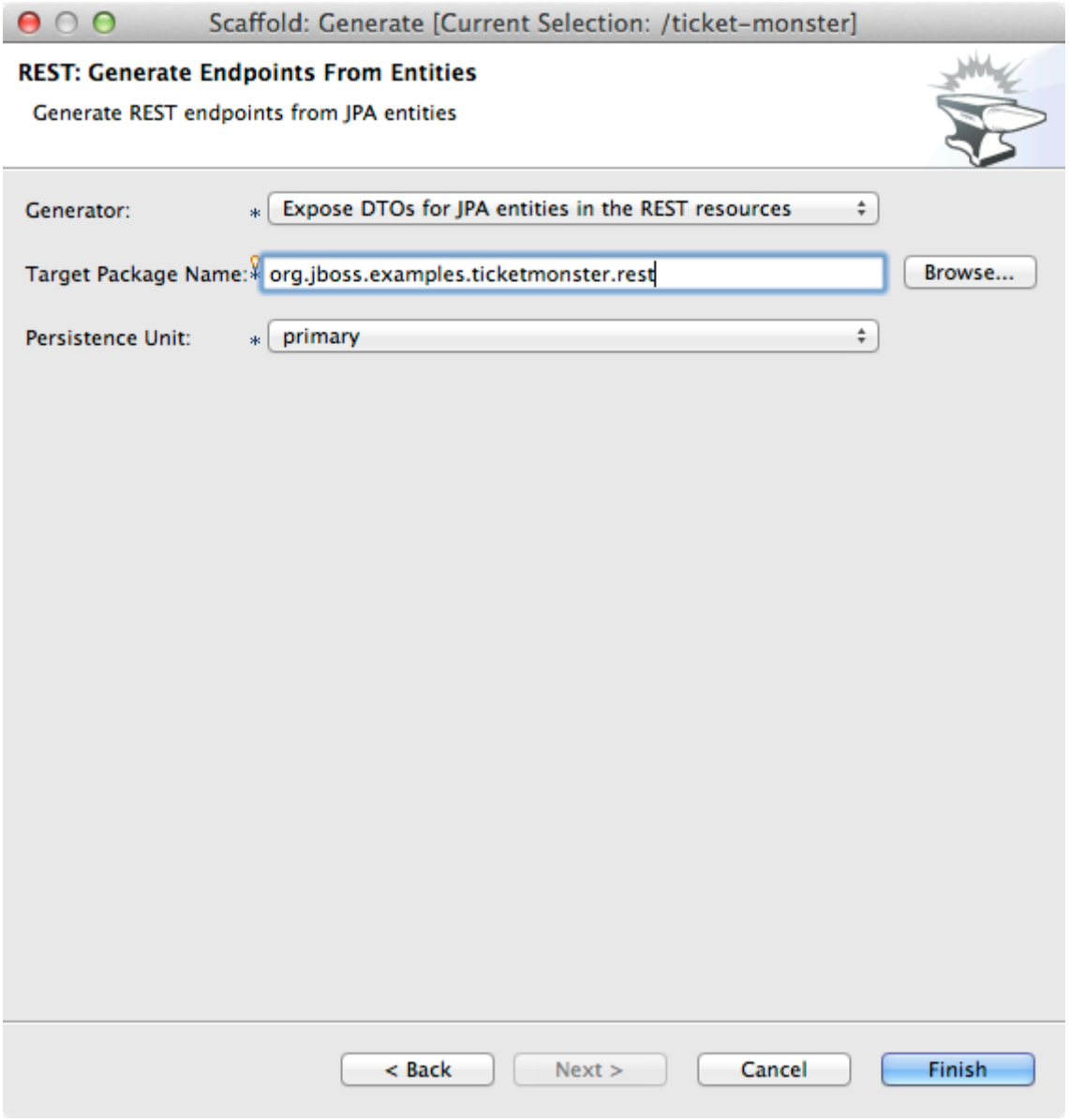

Figure 5. Choose the REST resource generation strategy

NOTE

The `Root and Nested DTO` resource representation enables Forge to create REST resources for complex object graphs without adding Jackson annotations to avoid cycles in the graph. Without this constrained representation, one would have to add annotations like `@JsonIgnore` (to ignore certain undesirable object properties), or `@JsonIdentity` (to represent cycles in JSON without succumbing to StackOverflowErrors or similar such errors/exceptions).

The scaffold generation command performs a multitude of activities, depending on the previous state of the project:

- It copies the css, images and JavaScript libraries used by the scaffold, to the project. It does this if you did not setup the scaffold in a separate step (this is optional; the generate command will do this for you).
- It generates JAX-RS resources for all the JPA entities in the project. The resources would be represented in JSON to enable the AngularJS-based front-end to communicate with the backend services. Each resource representation is structured to contain the representation of the corresponding JPA entity (the root) and any associated entities (that are represneted as nested

objects).

- It generates the AngularJS-based front-end that contains HTML based Angular templates along with AngularJS factories, services and controllers.

We now have a database-driven CRUD UI for all the entities used in TicketMonster!

Test the CRUD UI

Let's test our UI on our local JBoss AS instance. As usual, we'll build and deploy using Maven:

```
mvn clean package jboss-as:deploy
```

Make some changes to the UI

Let's add support for images to the Admin UI. `Events` and `Venues` have `MediaItem`s associated with them, but they're only displayed as URLs. Let's display the corresponding images in the AngularJS views, by adding the required bindings:

src/main/webapp/admin/views/Event/detail.html

```
        ...
        <div id="mediaItemControls" class="controls">
        <select id="mediaItem" name="mediaItem" ng-model="mediaItemSelection" ng-
options="m.text for m in mediaItemSelectionList"  >
            <option value="">Choose a Media Item</option>
        </select>
        <br/>
        <img class="img-polaroid span4" ng-hide="!mediaItemSelection.text" ng-
src="{{mediaItemSelection.text}}" />
        </div>
        ...
```

src/main/webapp/admin/views/Venue/detail.html

```
        ...
        <div id="mediaItemControls" class="controls">
        <select id="mediaItem" name="mediaItem" ng-model="mediaItemSelection" ng-
options="m.text for m in mediaItemSelectionList"  >
            <option value="">Choose a Media Item</option>
        </select>
        <br/>
        <img class="img-polaroid span4" ng-hide="!mediaItemSelection.text" ng-
src="{{mediaItemSelection.text}}" />
        </div>
        ...
```

Now that the bindings are set, we'll modify the underlying controllers to provide the URL of the MediaItem when the `{{mediaItemSelection.text}}` expression is evaluated:

src/main/webapp/admin/scripts/scripts/controllers/editEventController.js

```
...
            MediaItemResource.queryAll(function(items) {
                $scope.mediaItemSelectionList = $.map(items, function(item) {
                    ...
                    var labelObject = {
                        value : item.id,
                        text : item.url
                    };
                    ...
                });
            });
...
```

src/main/webapp/admin/scripts/scripts/controllers/editVenueController.js

```
...
            MediaItemResource.queryAll(function(items) {
                $scope.mediaItemSelectionList = $.map(items, function(item) {
                    ...
                    var labelObject = {
                        value : item.id,
                        text : item.url
                    };
                    ...
                });
            });
...
```

The admin site will now display the corresponding image if a media item is associated with the venue or event.

TIP

The location of the MediaItem is present in the `text` property of the `mediaItemSelection` object. The parameter to the `ngSrc` directive is set to this value. This ensures that the browser fetches the image present at this location. The expression `src={{mediaItemSelection.text}}` should be avoided since the browser would attempt to fetch the URL with the literal text `{{hash}}` before AngularJS replaces the expression with the actual URL.

Let's also modify the UI to make it more user-friendly. Shows and Performances are displayed in a non-intuitive manner at the moment. Shows are displayed as their object identities, while performances are displayed as date-time values. This makes it difficult to identify them in the views. Let's modify the UI to display more semantically useful values.

These values will be computed at the server-side, since these are already available in the `toString()`

implementations of these classes. This would be accomplished by adding a read-only property `displayTitle` to the `Show` and `Performance` REST resource representations:

src/main/java/org/jboss/examples/ticketmonster/rest/dto/ShowDTO.java

```
   ...
   private Set<NestedPerformanceDTO> performances = new
HashSet<NestedPerformanceDTO>();
   private NestedVenueDTO venue;
   private String displayTitle;

   public ShowDTO()
         ...
         }
         this.venue = new NestedVenueDTO(entity.getVenue());
         this.displayTitle = entity.toString();
      }
   }
   ...
   public String getDisplayTitle()
   {
      return this.displayTitle;
   }
}
```

src/main/java/org/jboss/examples/ticketmonster/rest/dto/PerformanceDTO.java

```
   ...
   private NestedShowDTO show;
   private Date date;
   private String displayTitle;

   public PerformanceDTO()
         ...
         this.show = new NestedShowDTO(entity.getShow());
         this.date = entity.getDate();
         this.displayTitle = entity.toString();
      }
   }
   ...
   public String getDisplayTitle()
   {
      return this.displayTitle;
   }
}
```

And let us do the same for the nested representations:

src/main/java/org/jboss/examples/ticketmonster/rest/dto/NestedPerformanceDTO.java

```
   ...
   private Long id;
   private Date date;
   private String displayTitle;

   public NestedPerformanceDTO()
         ...
         this.id = entity.getId();
         this.date = entity.getDate();
         this.displayTitle = entity.toString();
      }
   }
   ...
   public String getDisplayTitle()
   {
      return this.displayTitle;
   }
}
```

src/main/java/org/jboss/examples/ticketmonster/rest/dto/NestedShowDTO.java

```
   ...
   private Long id;
   private String displayTitle;

   public NestedShowDTO()
      ...
      {
         this.id = entity.getId();
         this.displayTitle = entity.toString();
      }
   }
   ...
   public String getDisplayTitle()
   {
      return this.displayTitle;
   }
}
```

We shall now proceed to modify the AngularJS views to use the new properties in the resource representations:

src/main/webapp/admin/scripts/controllers/editPerformanceController.js

```
    ...
    var labelObject = {
     value : item.id,
     text : item.displayTitle
```

```
    };
    if($scope.performance.show && item.id == $scope.performance.show.id) {
    ...
```

src/main/webapp/admin/scripts/controllers/editSectionAllocationController.js

```
    ...
    var labelObject = {
     value : item.id,
     text : item.displayTitle
    };
    if($scope.sectionAllocation.performance && item.id ==
$scope.sectionAllocation.performance.id) {
    ...
```

src/main/webapp/admin/scripts/controllers/editShowController.js

```
    ...
    var labelObject = {
     value : item.id,
     text : item.displayTitle
    };
    if($scope.show.performances){
    ...
```

src/main/webapp/admin/scripts/controllers/editTicketPriceController.js

```
    ...
    var labelObject = {
     value : item.id,
     text : item.displayTitle
    };
    if($scope.ticketPrice.show && item.id == $scope.ticketPrice.show.id) {
    ...
```

src/main/webapp/admin/scripts/controllers/newPerformanceController.js

```
    ...
    $scope.showSelectionList = $.map(items, function(item) {
        return ( {
            value : item.id,
            text : item.displayTitle
        });
    });
    ...
```

src/main/webapp/admin/scripts/controllers/newSectionAllocationController.js

```
    ...
    $scope.performanceSelectionList = $.map(items, function(item) {
        return ( {
            value : item.id,
            text : item.displayTitle
        });
    });
    ...
```

src/main/webapp/admin/scripts/controllers/newShowController.js

```
    ...
    $scope.performancesSelectionList = $.map(items, function(item) {
        return ( {
            value : item.id,
            text : item.displayTitle
        });
    });
    ...
```

src/main/webapp/admin/scripts/controllers/newTicketPriceController.js

```
    ...
    $scope.showSelectionList = $.map(items, function(item) {
        return ( {
            value : item.id,
            text : item.displayTitle
        });
    });
    ...
```

src/main/webapp/admin/views/Performance/search.html

```
        <label for="show" class="control-label">Show</label>
        <div class="controls">
            <select id="show" name="show" ng-model="search.show" ng-options="s as
s.displayTitle for s in showList">
                <option value="">Choose a Show</option>
            </select>
        ...
        <tbody id="search-results-body">
            <tr ng-repeat="result in searchResults | searchFilter:searchResults |
startFrom:currentPage*pageSize | limitTo:pageSize">
                    <td><a
href="#/Performances/edit/{{result.id}}">{{result.show.displayTitle}}</a></td>
                <td><a href="#/Performances/edit/{{result.id}}">{{result.date|
date:'yyyy-MM-dd HH:mm:ss Z'}}</a></td>
```

```
            </tr>
```

src/main/webapp/admin/views/SectionAllocation/search.html

```
         <label for="performance" class="control-label">Performance</label>
         <div class="controls">
             <select id="performance" name="performance" ng-model="search.performance"
ng-options="p as p.displayTitle for p in performanceList">
                 <option value="">Choose a Performance</option>
             </select>
            ...
          <tbody id="search-results-body">
            <tr ng-repeat="result in searchResults | searchFilter:searchResults |
startFrom:currentPage*pageSize | limitTo:pageSize">
                <td><a
href="#/SectionAllocations/edit/{{result.id}}">{{result.occupiedCount}}</a></td>
                    <td><a
href="#/SectionAllocations/edit/{{result.id}}">{{result.performance.displayTitle}}</a>
</td>
                    <td><a
href="#/SectionAllocations/edit/{{result.id}}">{{result.section.name}}</a></td>
            </tr>
```

src/main/webapp/admin/views/TicketPrice/search.html

```
        <label for="show" class="control-label">Show</label>
        <div class="controls">
            <select id="show" name="show" ng-model="search.show" ng-options="s as
s.displayTitle for s in showList">
                <option value="">Choose a Show</option>
            </select>
            ...
        <tbody id="search-results-body">
            <tr ng-repeat="result in searchResults | searchFilter:searchResults |
startFrom:currentPage*pageSize | limitTo:pageSize">
                    <td><a
href="#/TicketPrices/edit/{{result.id}}">{{result.show.displayTitle}}</a></td>
                    <td><a
href="#/TicketPrices/edit/{{result.id}}">{{result.section.name}}</a></td>
                    <td><a
href="#/TicketPrices/edit/{{result.id}}">{{result.ticketCategory.description}}</a></td
>
```

Fixing the landing page of the Administration site

The generated administration site contains a landing page - `app.html` that works well as a standalone site. However, we need to fix this page to make it work with the rest of the site.

For brevity, the significant sections of the corrected page are listed below:

src/main/webapp/admin/app.html

```
<!DOCTYPE html>
<html lang="en" ng-app="ticketmonster">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Ticket-monster</title>
    <link href='http://fonts.googleapis.com/css?family=Rokkitt' rel='stylesheet'
type='text/css'/>
    <link href="styles/bootstrap.css" rel="stylesheet" media="screen">
    <link href="styles/bootstrap-theme.css" rel="stylesheet" media="screen">
    <link href="styles/main.css" rel="stylesheet" media="screen">
    <link href="styles/custom-forge.css" rel="stylesheet" media="screen">
</head>
<body>
    <div id="wrap">

      <div id="logo" class="hidden-xs"><div class="wrap"><h1>Ticket
Monster</h1></div></div>
      <div class="navbar">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle pull-left" data-
toggle="collapse" data-target="#navbar-items">
                    <span class="glyphicon glyphicon-list"> Links</span>
                </button>
                <button type="button" class="navbar-toggle" data-toggle="offcanvas">
                    TicketMonster Entities <span class="glyphicon glyphicon-th text-
right"></span>
                </button>
            </div>

            <!-- Collect the nav links, forms, and other content for toggling -->
            <div id="navbar-items" class="collapse navbar-collapse">
                <ul class="nav navbar-nav">
                    <li><a href="../index.html#about">About</a></li>
                    <li><a href="../index.html#events">Events</a></li>
                    <li><a href="../index.html#venues">Venues</a></li>
                    <li><a href="../index.html#bookings">Bookings</a></li>
                    <li><a href="../index.html#monitor">Monitor</a></li>
                    <li><a href="#">Administration</a></li>
                </ul>
            </div>
      </div>

      <div class="container">

          ...
```

```
      </div>

      ...

</body>
</html>
```

It is sufficient to copy the corrected page from the project sources. Additionally, do not forget to copy the `src/main/webapp/admin/styles/custom-forge.css` file, that we now reference it in the corrected page.

Updating the ShrinkWrap deployment for the test suite

We've added classes to the project that should be in the ShrinkWrap deployment used in the test suite. Let us update the ShrinkWrap deployment to reflect this.

src/test/java/org/jboss/examples/ticketmonster/test/rest/RESTDeployment.java

```
public class RESTDeployment {

    public static WebArchive deployment() {
        return TicketMonsterDeployment.deployment()
                .addPackage(Booking.class.getPackage())
                .addPackage(BaseEntityService.class.getPackage())
                .addPackage(MultivaluedHashMap.class.getPackage())
                .addPackage(SeatAllocationService.class.getPackage())
                .addPackage(VenueDTO.class.getPackage());
    }

}
```

We can test these changes by executing

```
mvn clean test -Parq-jbossas-managed
```

or (against an already running JBoss EAP 6.2 instance)

```
mvn clean test -Parq-jbossas-remote
```

as usual.