Building The Business Services With JAX-RS

What Will You Learn Here?

We've just defined the domain model of the application and created its persistence layer. Now we need to define the services that implement the business logic of the application and expose them to the front-end. After reading this, you'll understand how to design the business layer and what choices to make while developing it. Topics covered include:

- Encapsulating business logic in services and integrating with the persistence tier
- Using CDI for integrating individual services
- Integration testing using Arquillian
- Exposing RESTful services via JAX-RS

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

Business Services And Their Relationships

TicketMonster's business logic is implemented by a number of classes, with different responsibilities:

- managing media items
- allocating tickets
- handling information on ticket availability
- remote access through a RESTful interface

The services are consumed by various other layers of the application:

- the media management and ticket allocation services encapsulate complex functionality, which in turn is exposed externally by RESTful services that wrap them
- RESTful services are mainly used by the HTML5 view layer
- the ticket availability service is used by the HTML5 and JavaScript based monitor

TIP

Where to draw the line?

A business service is an encapsulated, reusable logical component that groups together a number of well-defined cohesive business operations. Business services perform business operations, and may coordinate infrastructure services such as persistence units, or even other business services as well. The boundaries drawn between them should take into account whether the newly created services represent ,

> potentially reusable components.

As you can see, some of the services are intended to be consumed within the business layer of the application, while others provide an external interface as JAX-RS services. We will start by implementing the former, and we'll finish up with the latter. During this process, you will discover how CDI, EJB and JAX-RS make it easy to define and wire together our services.

Preparations

Adding Jackson Core

The first step for setting up our service architecture is to add Jackson Core as a dependency in the project. Adding Jackson Core as a provided dependency will enable you to use the Jackson annotations in the project. This is necessary to obtain a certain degree of control over the content of the JSON responses. We can bring in the same version of Jackson Core as the one used in RESTEasy, by adding `org.jboss.resteasy:resteasy-jackson-provider` and `org.jboss.resteasy:resteasy-jaxrs` as provided-scope dependencies, through the `org.jboss.bom.eap:jboss-javaee-6.0-with-resteasy` BOM. The versions of these dependencies would depend on the version of the JBoss BOMs we use in our project. Using the same version of the JBoss BOM as the one we will deploy to production, will ensure that we use the right dependencies during compilation and build.

pom.xml

```
<project ...>
    ...

    <dependencyManagement>
        ...

        <dependency>
            <groupId>org.jboss.bom.eap</groupId>
            <artifactId>jboss-javaee-6.0-with-resteasy</artifactId>
            <version>${version.jboss.bom.eap}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencyManagement>

    <dependencies>

        ...

        <!-- RESTEasy dependencies that bring in Jackson Core and RESTEasy APIs+SPIs,
which we use for
            fine tuning the content of the JSON responses -->
        <dependency>
            <groupId>org.jboss.resteasy</groupId>
            <artifactId>resteasy-jackson-provider</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.jboss.resteasy</groupId>
            <artifactId>resteasy-jaxrs</artifactId>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

NOTE

Why do you need the Jackson annotations?

JAX-RS does not specify mediatype-agnostic annotations for certain use cases. You will encounter atleast one of them in the project. The object graph contains cyclic/bi-directional relationships among entities like `Venue`, `Section`, `Show`, `Performance` and `TicketPrice`. JSON representations for these objects will need tweaking to avoid stack oVerflow errors and the like, at runtime.

JBoss Enterprise Application 6 uses Jackson to perform serialization and dserialization of objects, thus requiring use of Jackson annotations to modify this behavior. `@JsonIgnoreProperties` from Jackson will be used to suppress serialization and deserialization of one of the fields involved in the cycle.

Verifying the versions of the JBoss BOMs

The next step is to verify if we're using the right version of the JBoss BOMs in the project. Using the right versions of the BOMs ensures that you work against a known set of tested dependencies. Verify that the property `version.jboss.bom.eap` contains the value `6.3.2.GA` or higher:

pom.xml

```
<project ...>
    ...
    <properties>
        ...
        <version.jboss.bom.eap>6.3.2.GA</version.jboss.bom.eap>
        ...
    </properties>
    ...
</project>
```

Enabling CDI

The next step is to enable CDI in the deployment by creating a `beans.xml` file in the `WEB-INF` folder of the web application.

src/main/webapp/WEB-INF/beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                           http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

NOTE

If you used the Maven archetype

If you used the Maven archetype to create the project, this file will exist already in the project - it is added automatically.

You may wonder why the file is empty! Whilst `beans.xml` can specify various deployment-time configuration (e.g. activation of interceptors, decorators or alternatives), it can also act as a marker file, telling the container to enable CDI for the deployment (which it doesn't do, unless `beans.xml` is present).

TIP

Contexts and Dependency Injection (CDI)

As it's name suggests, CDI is the contexts and dependency injection standard for Java EE. By enabling CDI in your application, deployed classes become managed components and their lifecycle and wiring becomes the responsibility of the Java EE server.

In this way, we can reduce coupling between components, which is a requirement o a well-designed architecture. Now, we can focus on implementing the responsibilities of the components and describing their dependencies in a declarative fashion. The runtime will do the rest for you: instantiating and wiring them together, as well as disposing of them as needed.

Adding utility classes

Next, we add some helper classes providing low-level utilities for the application. We won't get in their implementation details here, but you can study their source code for details.

Copy the following classes from the original example to `src/main/java/org/jboss/examples/ticketmonster/util`:

- `Base64`
- `CircularBuffer`
- `ForwardingMap`
- `MultivaluedHashMap`
- `Reflections`
- `Resources`

Internal Services

We begin the service implementation by implementing some helper services.

The Media Manager

First, let's add support for managing media items, such as images. The persistence layer simply stores URLs, referencing media items stored by online services. The URL look like http://dl.dropbox.com/u/65660684/640px-Roy_Thomson_Hall_Toronto.jpg.

Now, we could use the URLs in our application, and retrieve these media items from the provider. However, we would prefer to cache these media items in order to improve application performance and increase resilience to external failures - this will allow us to run the application successfully even if the provider is down. The `MediaManager` is a good illustration of a business service; it performs the retrieval and caching of media objects, encapsulating the operation from the rest of the application.

We begin by creating `MediaManager`:

src/main/java/org/jboss/examples/ticketmonster/service/MediaManager.java

```
/**
 * <p>
 * The media manager is responsible for taking a media item, and returning either the
URL
 * of the cached version (if the application cannot load the item from the URL), or
the
 * original URL.
 * </p>
 *
 * <p>
 * The media manager also transparently caches the media items on first load.
 * </p>
 *
 * <p>
 * The computed URLs are cached for the duration of a request. This provides a good
balance
 * between consuming heap space, and computational time.
 * </p>
 *
 */
public class MediaManager {

    /**
     * Locate the tmp directory for the machine
     */
    private static final File tmpDir;

    static {
        String dataDir = System.getenv("OPENSHIFT_DATA_DIR");
        String parentDir = dataDir != null ? dataDir :
System.getProperty("java.io.tmpdir");
        tmpDir = new File(parentDir, "org.jboss.examples.ticket-monster");
        if (tmpDir.exists()) {
            if (tmpDir.isFile())
                throw new IllegalStateException(tmpDir.getAbsolutePath() + " already
exists, and is a file. Remove it.");
        } else {
            tmpDir.mkdir();
        }
    }

    /**
     * A request scoped cache of computed URLs of media items.
     */
    private final Map<MediaItem, MediaPath> cache;

    public MediaManager() {

        this.cache = new HashMap<MediaItem, MediaPath>();
    }

    /**
     * Load a cached file by name
     *
     * @param fileName
     * @return
     */
    public File getCachedFile(String fileName) {
        return new File(tmpDir, fileName);
    }

    /**
     * Obtain the URL of the media item. If the URL h has already been computed in
this
     * request, it will be looked up in the request scoped cache, otherwise it will be
     * computed, and placed in the request scoped cache.
     */
    public MediaPath getPath(MediaItem mediaItem) {
        if (cache.containsKey(mediaItem)) {
            return cache.get(mediaItem);
        } else {
            MediaPath mediaPath = createPath(mediaItem);
            cache.put(mediaItem, mediaPath);
            return mediaPath;
        }
    }

    /**
     * Compute the URL to a media item. If the media item is not cacheable, then, as
```

```
long
     * as the resource can be loaded, the original URL is returned. If the resource is
not
     * available, then a placeholder image replaces it. If the media item is cachable,
it
     * is first cached in the tmp directory, and then path to load it is returned.
     */
    private MediaPath createPath(MediaItem mediaItem) {
        if(mediaItem == null) {
            return
createCachedMedia(Reflections.getResource("not_available.jpg").toExternalForm(),
IMAGE);
        } else if (!mediaItem.getMediaType().isCacheable()) {
            if (checkResourceAvailable(mediaItem)) {
                return new MediaPath(mediaItem.getUrl(), false,
mediaItem.getMediaType());
            } else {
                return
createCachedMedia(Reflections.getResource("not_available.jpg").toExternalForm(),
IMAGE);
            }
        } else {
            return createCachedMedia(mediaItem);
        }
    }

    /**
     * Check if a media item can be loaded from it's URL, using the JDK URLConnection
classes.
     */
    private boolean checkResourceAvailable(MediaItem mediaItem) {
        URL url = null;
        try {
            url = new URL(mediaItem.getUrl());
        } catch (MalformedURLException e) {
        }

        if (url != null) {
            try {
                URLConnection connection = url.openConnection();
                if (connection instanceof HttpURLConnection) {
                    return ((HttpURLConnection) connection).getResponseCode() ==
HttpURLConnection.HTTP_OK;
                } else {
                    return connection.getContentLength() > 0;
                }
            } catch (IOException e) {
            }
        }
        return false;
    }
    /**
     * The cached file name is a base64 encoded version of the URL. This means we
don't need to maintain a database of cached
     * files.
     */
    private String getCachedFileName(String url) {
        return Base64.encodeToString(url.getBytes(), false);
    }

    /**
     * Check to see if the file is already cached.
     */
    private boolean alreadyCached(String cachedFileName) {
        File cache = getCachedFile(cachedFileName);
        if (cache.exists()) {
            if (cache.isDirectory()) {
                throw new IllegalStateException(cache.getAbsolutePath() + " already
exists, and is a directory. Remove it.");
            }
            return true;
        } else {
            return false;
        }
    }

    /**
     * To cache a media item we first load it from the net, then write it to disk.
     */
    private MediaPath createCachedMedia(String url, MediaType mediaType) {
        String cachedFileName = getCachedFileName(url);
        if (!alreadyCached(cachedFileName)) {
            URL _url = null;
            try {
                _url = new URL(url);
            } catch (MalformedURLException e) {
                throw new IllegalStateException("Error reading URL " + url);
            }

            try {
                InputStream is = null;
                OutputStream os = null;
                try {
                    is = new BufferedInputStream(_url.openStream());
                    os = new
BufferedOutputStream(getCachedOutputStream(cachedFileName));
                    while (true) {
                        int data = is.read();
                        if (data == -1)
                            break;
                        os.write(data);
                    }
                } finally {
                    if (is != null)
                        is.close();
                    if (os != null)
                        os.close();
                }
            } catch (IOException e) {
                throw new IllegalStateException("Error caching " +
mediaType.getDescription(), e);
            }
        }
        return new MediaPath(cachedFileName, true, mediaType);
    }

    private MediaPath createCachedMedia(MediaItem mediaItem) {
        return createCachedMedia(mediaItem.getUrl(), mediaItem.getMediaType());
    }

    private OutputStream getCachedOutputStream(String fileName) {
        try {
            return new FileOutputStream(getCachedFile(fileName));
        } catch (FileNotFoundException e) {
            throw new IllegalStateException("Error creating cached file", e);
        }
    }

}
```

The service delegates to a number of internal methods that do the heavy lifting, but exposes a simple API, to the external observer it simply converts the `MediaItem` entities into `MediaPath` data structures, that can be used by the application to load the binary data of the media item. The service will retrieve and cache the data locally in the filesystem, if possible (e.g. streamed videos aren't cacheable!).

src/main/java/org/jboss/examples/ticketmonster/service/MediaPath.java

```
public class MediaPath {

    private final String url;
    private final boolean cached;
    private final MediaType mediaType;

    public MediaPath(String url, boolean cached, MediaType mediaType) {
        this.url = url;
        this.cached = cached;
        this.mediaType = mediaType;
    }

    public String getUrl() {
        return url;
    }

    public boolean isCached() {
        return cached;
    }

    public MediaType getMediaType() {
        return mediaType;
    }

}
```

The service can be injected by type into the components that depend on it.

We should also control the lifecycle of this service. The `MediaManager` stores request-specific state, so should be scoped to the web request, the CDI `@RequestScoped` is perfect.

src/main/java/org/jboss/examples/ticketmonster/service/MediaManager.java

```
   ...
@RequestScoped
public class MediaManager {
   ...
}
```

The Seat Allocation Service

The seat allocation service finds free seats at booking time, in a given section of the venue. It is a good example of how a service can coordinate infrastructure services (using the injected persistence unit to get access to the `SeatAllocation` instance) and domain objects (by invoking the `allocateSeats` method on a concrete allocation instance).

Isolating this functionality in a service class makes it possible to write simpler, self-explanatory code in the layers above and opens the possibility of replacing this code at a later date with a more advanced implementation (for example one using an in-memory cache).

src/main/java/org/jboss/examples/ticketmonster/service/SeatAllocationService.java

```
@SuppressWarnings("serial")
public class SeatAllocationService implements Serializable {

    @Inject
    EntityManager entityManager;

    public AllocatedSeats allocateSeats(Section section, Performance performance, int
seatCount, boolean contiguous) {
        SectionAllocation sectionAllocation =
retrieveSectionAllocationExclusively(section, performance);
```

```
        List<Seat> seats = sectionAllocation.allocateSeats(seatCount, contiguous);
        return new AllocatedSeats(sectionAllocation, seats);
    }

    public void deallocateSeats(Section section, Performance performance, List<Seat>
seats) {
        SectionAllocation sectionAllocation =
retrieveSectionAllocationExclusively(section, performance);
        for (Seat seat : seats) {
            if (!seat.getSection().equals(section)) {
                throw new SeatAllocationException("All seats must be in the same
section!");
            }
            sectionAllocation.deallocate(seat);
        }
    }

    private SectionAllocation retrieveSectionAllocationExclusively(Section section,
Performance performance) {
        SectionAllocation sectionAllocationStatus = null;
        try {
            sectionAllocationStatus = (SectionAllocation) entityManager.createQuery(
                "select s from SectionAllocation s where " +
                    "s.performance.id = :performanceId and " +
                    "s.section.id = :sectionId")
                .setParameter("performanceId", performance.getId())
                .setParameter("sectionId", section.getId())
                .getSingleResult();
        } catch (NoResultException noSectionEx) {
            // Create the SectionAllocation since it doesn't exist
            sectionAllocationStatus = new SectionAllocation(performance, section);
            entityManager.persist(sectionAllocationStatus);
            entityManager.flush();
        }
        entityManager.lock(sectionAllocationStatus, LockModeType.PESSIMISTIC_WRITE);
        return sectionAllocationStatus;
    }
}
```

Next, we define the `AllocatedSeats` class that we use for storing seat reservations for a booking, before they are made persistent.

src/main/java/org/jboss/examples/ticketmonster/service/AllocatedSeats.java

```
public class AllocatedSeats {

    private final SectionAllocation sectionAllocation;

    private final List<Seat> seats;

    public AllocatedSeats(SectionAllocation sectionAllocation, List<Seat> seats) {
        this.sectionAllocation = sectionAllocation;
        this.seats = seats;
    }

    public SectionAllocation getSectionAllocation() {
        return sectionAllocation;
    }

    public List<Seat> getSeats() {
        return seats;
    }

    public void markOccupied() {
        sectionAllocation.markOccupied(seats);
    }
}
```

JAX-RS Services

The majority of services in the application are JAX-RS web services. They are critical part of the design, as they next service is used for provide communication with the HTML5 view layer. The JAX-RS services range from simple CRUD to processing bookings and media items.

To pass data across the wire we use JSON as the data marshalling format, as it is less verbose and easier to process than XML by the JavaScript client-side framework.

Initializing JAX-RS

We shall ensure that the required dependencies are present in the project POM, to setup JAX-RS in the project:

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>org.jboss.spec.javax.ws.rs</groupId>
            <artifactId>jboss-jaxrs-api_1.1_spec</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.jboss.spec.javax.servlet</groupId>
            <artifactId>jboss-servlet-api_3.0_spec</artifactId>
            <scope>provided</scope>
        </dependency>
    </dependencies>
    ...
</project>
```

Some of these may already be present in the project POM, and should not be added again.

To activate JAX-RS we add the class below, which instructs the container to look for JAX-RS annotated classes and install them as endpoints. This class should exist already in your project, as it is generated by the archetype, so make sure that it is there and it contains the code below:

src/main/java/org/jboss/examples/ticketmonster/rest/JaxRsActivator.java

```
@ApplicationPath("/rest")
public class JaxRsActivator extends Application {
   /* class body intentionally left blank */
}
```

All the JAX-RS services are mapped relative to the `/rest` path, as defined by the `@ApplicationPath` annotation.

A Base Service For Read Operations

Most JAX-RS services must provide both a (filtered) list of entities or individual entity (e.g. events, venues and bookings). Instead of duplicating the implementation into each individual service we create a base service class and wire the helper objects in.

src/main/java/org/jboss/examples/ticketmonster/rest/BaseEntityService.java

```
/**
 * <p>
 *   A number of RESTful services implement GET operations on a particular type of
entity. For
 *   observing the DRY principle, the generic operations are implemented in the
<code>BaseEntityService</code>
 *   class, and the other services can inherit from here.
 * </p>
 *
 * <p>
 *    Subclasses will declare a base path using the JAX-RS {@link Path} annotation,
for example:
 * </p>
 *
 * <pre>
 * <code>
 * &#064;Path("/widgets")
 * public class WidgetService extends BaseEntityService<Widget> {
 * ...
 * }
 * </code>
 * </pre>
 *
 * <p>
 *   will support the following methods:
 * </p>
 *
 * <pre>
 * <code>
 *   GET /widgets
 *   GET /widgets/:id
 *   GET /widgets/count
 * </code>
 * </pre>
 *
 *  <p>
 *     Subclasses may specify various criteria for filtering entities when retrieving
a list of them, by supporting
 *     custom query parameters. Pagination is supported by default through the query
parameters <code>first</code>
 *     and <code>maxResults</code>.
 * </p>
 *
 * <p>
 *     The class is abstract because it is not intended to be used directly, but
subclassed by actual JAX-RS
 *     endpoints.
 * </p>
 *
 */
public abstract class BaseEntityService<T> {

    @Inject
    private EntityManager entityManager;

    private Class<T> entityClass;

    public BaseEntityService() {}

    public BaseEntityService(Class<T> entityClass) {
        this.entityClass = entityClass;
    }

    public EntityManager getEntityManager() {
        return entityManager;
    }

}
```

Now we add a method to retrieve all entities of a given type:

src/main/java/org/jboss/examples/ticketmonster/rest/BaseEntityService.java

```
public abstract class BaseEntityService<T> {

    ...

    /**
     * <p>
     *   A method for retrieving all entities of a given type. Supports the query
parameters
     *  <code>first</code>
     *   and <code>maxResults</code> for pagination.
     * </p>
     *
     *  @param uriInfo application and request context information (see {@see UriInfo}
class
     *  information for more details)
     *  @return
     */
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<T> getAll(@Context UriInfo uriInfo) {
        return getAll(uriInfo.getQueryParameters());
    }

    public List<T> getAll(MultivaluedMap<String, String> queryParameters) {
        final CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        final CriteriaQuery<T> criteriaQuery =
criteriaBuilder.createQuery(entityClass);
        Root<T> root = criteriaQuery.from(entityClass);
        Predicate[] predicates = extractPredicates(queryParameters, criteriaBuilder,
root);
        criteriaQuery.select(criteriaQuery.getSelection()).where(predicates);
        criteriaQuery.orderBy(criteriaBuilder.asc(root.get("id")));
        TypedQuery<T> query = entityManager.createQuery(criteriaQuery);
        if (queryParameters.containsKey("first")) {
            Integer firstRecord = Integer.parseInt(queryParameters.getFirst("first"))-
1;
            query.setFirstResult(firstRecord);
        }
        if (queryParameters.containsKey("maxResults")) {
            Integer maxResults =
Integer.parseInt(queryParameters.getFirst("maxResults"));
            query.setMaxResults(maxResults);
        }
        return query.getResultList();
    }

    /**
```

```
     * <p>
     *     Subclasses may choose to expand the set of supported query parameters (for
adding more filtering
     *     criteria) by overriding this method.
     * </p>
     * @param queryParameters - the HTTP query parameters received by the endpoint
     * @param criteriaBuilder - @{link CriteriaBuilder} used by the invoker
     * @param root  @{link Root} used by the invoker
     * @return a list of {@link Predicate}s that will added as query parameters
     */
    protected Predicate[] extractPredicates(MultivaluedMap<String, String>
queryParameters,
                                            CriteriaBuilder criteriaBuilder, Root<T>
root) {
        return new Predicate[]{};
    }

}
```

The newly added method `getAll` is annotated with `@GET` which instructs JAX-RS to call it when a `GET` HTTP requests on the JAX-RS' endpoint base URL '/rest/<entityRoot>' is performed. But remember, this is not a true JAX-RS endpoint. It is an abstract class and it is not mapped to a path. The classes that extend it are JAX-RS endpoints, and will have to be mapped to a path, and are able to process requests.

The `@Produces` annotation defines that the response sent back by the server is in JSON format. The JAX-RS implementation will automatically convert the result returned by the method (a list of entities) into JSON format.

As well as configuring the marshaling strategy, the annotation affects content negotiation and method resolution. If the client requests JSON content specifically, this method will be invoked.

NOTE Even though it is not shown in this example, you may have multiple methods that handle a specific URL and HTTP method, whilst consuming and producing different types of content (JSON, HTML, XML or others).

Subclasses can also override the `extractPredicates` method and add own support for additional query parameters to `GET /rest/<entityRoot>` which can act as filter criteria.

The `getAll` method supports retrieving a range of entities, which is especially useful when we need to handle very large sets of data, and use pagination. In those cases, we need to support counting entities as well, so we add a method that retrieves the entity count:

src/main/java/org/jboss/examples/ticketmonster/rest/BaseEntityService.java

```
public abstract class BaseEntityService<T> {


    ...

    /**
```

```
     * <p>
     *   A method for counting all entities of a given type
     * </p>
     *
     * @param uriInfo application and request context information (see {@see UriInfo}
class information for more details)
     * @return
     */
    @GET
    @Path("/count")
    @Produces(MediaType.APPLICATION_JSON)
    public Map<String, Long> getCount(@Context UriInfo uriInfo) {
        CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        CriteriaQuery<Long> criteriaQuery = criteriaBuilder.createQuery(Long.class);
        Root<T> root = criteriaQuery.from(entityClass);
        criteriaQuery.select(criteriaBuilder.count(root));
        Predicate[] predicates = extractPredicates(uriInfo.getQueryParameters(),
criteriaBuilder, root);
        criteriaQuery.where(predicates);
        Map<String, Long> result = new HashMap<String, Long>();
        result.put("count",
entityManager.createQuery(criteriaQuery).getSingleResult());
        return result;
    }

}
```

We use the `@Path` annotation to map the new method to a sub-path of '/rest/<entityRoot>. Now all the JAX-RS endpoints that subclass `BaseEntityService` will be able to get entity counts from '/rest/<entityRoot>/count'. Just like `getAll`, this method also delegates to `extractPredicates`, so any customizations done there by subclasses

Next, we add a method for retrieving individual entities.

src/main/java/org/jboss/examples/ticketmonster/rest/BaseEntityService.java

```
   ...
public abstract class BaseEntityService<T> {

    ...

    /**
     * <p>
     *     A method for retrieving individual entity instances.
     * </p>
     * @param id entity id
     * @return
     */
    @GET
    @Path("/{id:[0-9][0-9]*}")
    @Produces(MediaType.APPLICATION_JSON)
    public T getSingleInstance(@PathParam("id") Long id) {
        final CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();
        final CriteriaQuery<T> criteriaQuery =
criteriaBuilder.createQuery(entityClass);
        Root<T> root = criteriaQuery.from(entityClass);
        Predicate condition = criteriaBuilder.equal(root.get("id"), id);

criteriaQuery.select(criteriaBuilder.createQuery(entityClass).getSelection()).where(co
ndition);
        return entityManager.createQuery(criteriaQuery).getSingleResult();
    }
}
```

This method is similar to `getAll` and `getCount`, and we use the `@Path` annotation to map it to a sub-path of '/rest/<entityRoot>'. The annotation attribute identifies the expected format of the URL (here, the last segment has to be a number) and binds a portion of the URL to a variable (here named `id`). The `@PathParam` annotation allows the value of the variable to be passed as a method argument. Data conversion is performed automatically.

Now, all the JAX-RS endpoints that subclass `BaseEntityService` will get two operations for free:

`GET /rest/<entityRoot>`
: retrieves all entities of a given type

`GET /rest/<entityRoot>/<id>`
: retrieves an entity with a given id

Retrieving Venues

Adding support for retrieving venues is now extremely simple. We refactor the class we created during the introduction, and make it extend `BaseEntityService`, passing the entity type to the superclass constructor. We remove the old retrieval code, which is not needed anymore.

src/main/java/org/jboss/examples/ticketmonster/rest/VenueService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Venue}s. Inherits the actual
 *     methods from {@link BaseEntityService}.
 * </p>
 */
@Path("/venues")
/**
 * <p>
 *     This is a stateless service, so a single shared instance can be used in this
case.
 * </p>
 */
```

```
@Stateless
public class VenueService extends BaseEntityService<Venue> {

    public VenueService() {
        super(Venue.class);
    }

}
```

We add the `@Path` annotation to the class, to indicate that this is a JAX-RS resource which can serve URLs starting with `/rest/venues`.

We define this service (along with all the other JAX-RS services) as an EJB (see how simple is that in Java EE 6!) to benefit from automatic transaction enrollment. Since the service is fundamentally stateless, we take advantage of the new EJB 3.1 singleton feature.

Before we proceed, . Retrieving shows from URLs like `/rest/venues` or `/rest/venues/1` almost always results in invalid JSON responses. The root cause is the presence of a bi-directional relationship in the `Venue` entity. A `Venue` contains a 1:M relationship with `Section` s that also links back to a `Venue`. JSON serialiers like Jackson (the one used in JBoss Enterprise Application Platform) need to be instructed on how to handle such cycles in object graphs, failing which the serializer will traverse between the entities in the cycle, resulting in an infinite loop (and often an `OutOfMemoryError` or a `StackOverflowError`). We'll address this, by instructing Jackson to not serialize the `venue` field in a `Section`, through the `@JsonIgnoreProperties` annotation on the `Section` entity:

src/main/java/org/jboss/examples/ticketmonster/model/Section.java

```
...
@JsonIgnoreProperties("venue")
public class Section implements Serializable {

...

}
```

Now, we can retrieve venues from URLs like `/rest/venues` or `rest/venues/1`.

Retrieving Events

Just like `VenueService`, the `EventService` implementation we use for TicketMonster is a direct subclass of `BaseEntityService`. Refactor the existing class, remove the old retrieval code and make it extend `BaseEntityService`.

One additional functionality we will implement is querying events by category. We can use URLs like `/rest/events?category=1` to retrieve all concerts, for example (`1` is the category id of concerts). This is done by overriding the `extractPredicates` method to handle any query parameters (in this case, the `category` parameter).

src/main/java/org/jboss/examples/ticketmonster/rest/EventService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Event}s. Inherits the actual
 *     methods from {@link BaseEntityService}, but implements additional search
 *     criteria.
 * </p>
 */
@Path("/events")
/**
 * <p>
 *     This is a stateless service, we declare it as an EJB for transaction
demarcation
 * </p>
 */
@Stateless
public class EventService extends BaseEntityService<Event> {

    public EventService() {
        super(Event.class);
    }

    /**
     * <p>
     *    We override the method from parent in order to add support for additional
search
     *    criteria for events.
     * </p>
     * @param queryParameters - the HTTP query parameters received by the endpoint
     * @param criteriaBuilder - @{link CriteriaBuilder} used by the invoker
     * @param root  @{link Root} used by the invoker
     * @return
     */
    @Override
    protected Predicate[] extractPredicates(
            MultivaluedMap<String, String> queryParameters,
            CriteriaBuilder criteriaBuilder,
            Root<Event> root) {
        List<Predicate> predicates = new ArrayList<Predicate>() ;

        if (queryParameters.containsKey("category")) {
            String category = queryParameters.getFirst("category");
            predicates.add(criteriaBuilder.equal(root.get("category").get("id"),
category));
        }

        return predicates.toArray(new Predicate[]{});
    }
}
```

Retrieving Shows

The ShowService follows the same pattern and we leave the implementation as an exercise to the reader (knowing that its contents can always be copied over to the appropriate folder).

Just like the Venue entity, a Show also contains bi-directional relationships that need to be handled as a special case for the JSON serializer. A Show contains a 1:M relationship with Performance s that also links back to a Show; a Show also contains a 1:M relationship with TicketPrice s that also links back to a Show. We'll address this, by instructing Jackson to not serialize the show field in a Performance, through the @JsonIgnoreProperties annotation on the Performance entity:

src/main/java/org/jboss/examples/ticketmonster/model/Performance.java

```
...
@JsonIgnoreProperties("show")
public class Performance implements Serializable {

...

}
```

Likewise, we'll also instruct Jackson to not serialize the Show in a TicketPrice:

src/main/java/org/jboss/examples/ticketmonster/model/TicketPrice.java

```
...
@JsonIgnoreProperties("show")
public class TicketPrice implements Serializable {

...

}
```

Creating and deleting bookings

Of course, we also want to change data with our services - we want to create and delete bookings as well!

To create a booking, we add a new method, which handles POST requests to /rest/bookings. This is not a simple CRUD method, as the client does not send a booking, but a booking request. It is the responsibility of the service to process the request, reserve the seats and return the full booking details to the invoker.

src/main/java/org/jboss/examples/ticketmonster/rest/BookingService.java

```
/**
 * <p>
 *     A JAX-RS endpoint for handling {@link Booking}s. Inherits the GET
 *     methods from {@link BaseEntityService}, and implements additional REST methods.
 * </p>
 */
@Path("/bookings")
/**
 * <p>
 *     This is a stateless service, we declare it as an EJB for transaction
demarcation
 * </p>
 */
@Stateless
public class BookingService extends BaseEntityService<Booking> {

    @Inject
    SeatAllocationService seatAllocationService;

    @Inject @Created
    private Event<Booking> newBookingEvent;

    public BookingService() {
        super(Booking.class);
    }

   /**
     * <p>
     *   Create a booking. Data is contained in the bookingRequest object
     * </p>
     * @param bookingRequest
     * @return
     */
    @SuppressWarnings("unchecked")
    @POST
    /**
     * <p> Data is received in JSON format. For easy handling, it will be unmarshalled
in the support
     * {@link BookingRequest} class.
     */
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createBooking(BookingRequest bookingRequest) {
        try {
            // identify the ticket price categories in this request
            Set<Long> priceCategoryIds = bookingRequest.getUniquePriceCategoryIds();

            // load the entities that make up this booking's relationships
            Performance performance = getEntityManager().find(Performance.class,
bookingRequest.getPerformance());

            // As we can have a mix of ticket types in a booking, we need to load all
of them that are relevant,
            // id
            Map<Long, TicketPrice> ticketPricesById =
loadTicketPrices(priceCategoryIds);
```

```
            // Now, start to create the booking from the posted data
            // Set the simple stuff first!
            Booking booking = new Booking();
            booking.setContactEmail(bookingRequest.getEmail());
            booking.setPerformance(performance);
            booking.setCancellationCode("abc");

            // Now, we iterate over each ticket that was requested, and organize them
by section and category
            // we want to allocate ticket requests that belong to the same section
contiguously
            Map<Section, Map<TicketCategory, TicketRequest>> ticketRequestsPerSection
                    = new TreeMap<Section, java.util.Map<TicketCategory,
TicketRequest>>(SectionComparator.instance());
            for (TicketRequest ticketRequest : bookingRequest.getTicketRequests()) {
                final TicketPrice ticketPrice =
ticketPricesById.get(ticketRequest.getTicketPrice());
                if (!ticketRequestsPerSection.containsKey(ticketPrice.getSection())) {
                    ticketRequestsPerSection
                            .put(ticketPrice.getSection(), new HashMap<TicketCategory,
TicketRequest>());
                }
                ticketRequestsPerSection.get(ticketPrice.getSection()).put(

ticketPricesById.get(ticketRequest.getTicketPrice()).getTicketCategory(),
ticketRequest);
            }

            // Now, we can allocate the tickets
            // Iterate over the sections, finding the candidate seats for allocation
            // The process will acquire a write lock for a given section and
performance
            // Use deterministic ordering of sections to prevent deadlocks
            Map<Section, AllocatedSeats> seatsPerSection =
                   new TreeMap<Section,
org.jboss.examples.ticketmonster.service.AllocatedSeats>(SectionComparator.instance())
;
            List<Section> failedSections = new ArrayList<Section>();
            for (Section section : ticketRequestsPerSection.keySet()) {
                int totalTicketsRequestedPerSection = 0;
                // Compute the total number of tickets required (a ticket category
doesn't impact the actual seat!)
                final Map<TicketCategory, TicketRequest> ticketRequestsByCategories =
ticketRequestsPerSection.get(section);
                // calculate the total quantity of tickets to be allocated in this
section
                for (TicketRequest ticketRequest :
ticketRequestsByCategories.values()) {
                    totalTicketsRequestedPerSection += ticketRequest.getQuantity();
                }
                // try to allocate seats

                AllocatedSeats allocatedSeats =
                      seatAllocationService.allocateSeats(section, performance,
totalTicketsRequestedPerSection, true);
                if (allocatedSeats.getSeats().size() ==
totalTicketsRequestedPerSection) {
                    seatsPerSection.put(section, allocatedSeats);
                } else {
                    failedSections.add(section);
                }
            }
            if (failedSections.isEmpty()) {
                for (Section section : seatsPerSection.keySet()) {
                    // allocation was successful, begin generating tickets
                    // associate each allocated seat with a ticket, assigning a price
category to it
                    final Map<TicketCategory, TicketRequest>
ticketRequestsByCategories = ticketRequestsPerSection.get(section);
                    AllocatedSeats allocatedSeats = seatsPerSection.get(section);
                    allocatedSeats.markOccupied();
                    int seatCounter = 0;
                    // Now, add a ticket for each requested ticket to the booking
                    for (TicketCategory ticketCategory :
ticketRequestsByCategories.keySet()) {
                        final TicketRequest ticketRequest =
ticketRequestsByCategories.get(ticketCategory);
                        final TicketPrice ticketPrice =
ticketPricesById.get(ticketRequest.getTicketPrice());
                        for (int i = 0; i < ticketRequest.getQuantity(); i++) {
                            Ticket ticket =
                                  new Ticket(allocatedSeats.getSeats().get(seatCounter
+ i), ticketCategory, ticketPrice.getPrice());
                            // getEntityManager().persist(ticket);
                            booking.getTickets().add(ticket);
                        }
                        seatCounter += ticketRequest.getQuantity();
                    }
                }
                // Persist the booking, including cascaded relationships
                booking.setPerformance(performance);
                booking.setCancellationCode("abc");
                getEntityManager().persist(booking);
                newBookingEvent.fire(booking);
                return
Response.ok().entity(booking).type(MediaType.APPLICATION_JSON_TYPE).build();
            } else {
                Map<String, Object> responseEntity = new HashMap<String, Object>();
                responseEntity.put("errors", Collections.singletonList("Cannot
allocate the requested number of seats!"));
                return
```

```
Response.status(Response.Status.BAD_REQUEST).entity(responseEntity).build();
            }
        } catch (ConstraintViolationException e) {
            // If validation of the data failed using Bean Validation, then send an
error
            Map<String, Object> errors = new HashMap<String, Object>();
            List<String> errorMessages = new ArrayList<String>();
            for (ConstraintViolation<?> constraintViolation :
e.getConstraintViolations()) {
                errorMessages.add(constraintViolation.getMessage());
            }
            errors.put("errors", errorMessages);
            // A WebApplicationException can wrap a response
            // Throwing the exception causes an automatic rollback
            throw new
WebApplicationException(Response.status(Response.Status.BAD_REQUEST).entity(errors).bu
ild());
        } catch (Exception e) {
            // Finally, handle unexpected exceptions
            Map<String, Object> errors = new HashMap<String, Object>();
            errors.put("errors", Collections.singletonList(e.getMessage()));
            // A WebApplicationException can wrap a response
            // Throwing the exception causes an automatic rollback
            throw new
WebApplicationException(Response.status(Response.Status.BAD_REQUEST).entity(errors).bu
ild());
        }
    }

    /**
     * Utility method for loading ticket prices
     * @param priceCategoryIds
     * @return
     */
    private Map<Long, TicketPrice> loadTicketPrices(Set<Long> priceCategoryIds) {
        List<TicketPrice> ticketPrices = (List<TicketPrice>) getEntityManager()
                .createQuery("select p from TicketPrice p where p.id in :ids")
                .setParameter("ids", priceCategoryIds).getResultList();
        // Now, map them by id
        Map<Long, TicketPrice> ticketPricesById = new HashMap<Long, TicketPrice>();
        for (TicketPrice ticketPrice : ticketPrices) {
            ticketPricesById.put(ticketPrice.getId(), ticketPrice);
        }
        return ticketPricesById;
    }
}
```

You should also copy over the `BookingRequest`, `TicketRequest` and `SectionComparator` classes referenced in these methods, from the project sources.

We won’t get into the details of the inner workings of the method - it implements a fairly complex algorithm - but we’d like to draw attention to a few particular items.

We use the `@POST` annotation to indicate that this method is executed on inbound HTTP POST requests. When implementing a set of RESTful services, it is important that the semantic of HTTP methods are observed in the mappings. Creating new resources (e.g. bookings) is typically associated with HTTP POST invocations. The `@Consumes` annotation indicates that the type of the request content is JSON and identifies the correct unmarshalling strategy, as well as content negotiation.

The `BookingService` delegates to the `SeatAllocationService` to find seats in the requested section, the required `SeatAllocationService` instance is initialized and supplied by the container as needed. The only thing that our service does is to specify the dependency in form of an injection point - the field annotated with `@Inject`.

We would like other parts of the application to be aware of the fact that a new booking has been created, therefore we use the CDI to fire an event. We do so by injecting an `Event<Booking>` instance into the service (indicating that its payload will be a booking). In order to individually identify this event as referring to event creation, we use a CDI qualifier, which we need to add:

src/main/java/org/jboss/examples/ticketmonster/util/qualifier/Created.java

```
/**
 * {@link Qualifier} to mark a Booking as new (created).
 */
@Qualifier
@Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Created {

}
```

TIP

What are qualifiers?

CDI uses a type-based resolution mechanism for injection and observers. In order to distinguish between implementations of an interface, you can use qualifiers, a type of annotations, to disambiguate. Injection points and event observers can use qualifiers to narrow down the set of candidates

We also need allow the removal of bookings, so we add a method:

src/main/java/org/jboss/examples/ticketmonster/rest/BookingService.java

```
@Singleton
public class BookingService extends BaseEntityService<Booking> {
    ...

    @Inject @Cancelled
    private Event<Booking> cancelledBookingEvent;
    ...
    /**
     * <p>
     * Delete a booking by id
     * </p>
     * @param id
     * @return
     */
    @DELETE
    @Path("/{id:[0-9][0-9]*}")
    public Response deleteBooking(@PathParam("id") Long id) {
        Booking booking = getEntityManager().find(Booking.class, id);
        if (booking == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        getEntityManager().remove(booking);
        cancelledBookingEvent.fire(booking);
        return Response.noContent().build();
    }
}
```

We use the `@DELETE` annotation to indicate that it will be executed as the result of an HTTP DELETE request (again, the use of the DELETE HTTP verb is a matter of convention).

We need to notify the other components of the cancellation of the booking, so we fire an event, with a different qualifier.

src/main/java/org/jboss/examples/ticketmonster/util/qualifier/Cancelled.java

```
/**
 * {@link Qualifier} to mark a Booking as cancelled.
 */
@Qualifier
@Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.METHOD,ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Cancelled {

}
```

The other services, including the `MediaService` that handles media items follow roughly the same patterns as above, so we leave them as an exercise to the reader.

Testing the services

We’ve now finished implementing the services and there is a significant amount of functionality in the application. Before taking any step forward, you need to make sure the services work correctly: we need to test them.

Testing enterprise services be a complex task as the implementation is based on services provided by a container: dependency injection, access to infrastructure services such as persistence, transactions etc. Unit testing frameworks, whilst offering a valuable infrastructure for running tests, do not provide these capabilities.

One of the traditional approaches has been the use of mocking frameworks to simulate 'what will happen' in the runtime environment. While certainly providing a solution mocking brings its own set of problems (e.g. the additional effort required to provide a proper simulation or the risk of introducing errors in the test suite by incorrectly implemented mocks.

TIP

What to test?

A common asked question is: how much application functionality should we test? The truth is, you can never test too much. That being said, resources are always limited and tradeoffs are part of an engineer's work. Generally speaking, trivial functionality (setters/getters/toString methods) is a big concern compared to the actual business code, so you probably want to focus your efforts on the business code. Testing should include individual parts (unit testing), as well as aggregates (integration testing).

Fortunately, Arquillian provides the means to testing your application code within the container, with access to all the services and container features. In this section we will show you how to create a few Arquillian tests for your business services.

TIP

New to Arquillian?

The Arquillian project site contains several tutorials to help you get started. If you're new to Arquillian and Shrinkwrap, we recommend going through the beginner-level Arquillian guides, at the very least.

Adding ShrinkWrap Resolvers

We'll need to use an updated version of the ShrinkWrap Resolvers project, that is not provided by the existing `org.jboss.bom.eap:jboss-javaee-6.0-with-tools` BOM. Fortunately, the JBoss WFK project provides this for us. It provides us with the `shrinkwrap-resolver-depchain` module, which allows us to use ShrinkWrap resolvers in our project through a single dependency. We can bring in the required version of ShrinkWrap Resolvers, by merely using the `org.jboss.bom.wfk:jboss-javaee-6.0-with-tools` BOM instead of the pre-existing tools BOM from EAP:

pom.xml

```
<project ...>
    ...
    <properties>
        ...
        <version.jboss.bom.wfk>2.7.0-redhat-1</version.jboss.bom.wfk>
        ...
    </properties>


    <dependencyManagement>
        ...
        <dependency>
            <groupId>org.jboss.bom.wfk</groupId>
            <artifactId>jboss-javaee-6.0-with-resteasy</artifactId>
            <version>${version.jboss.bom.wfk}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencyManagement>

    ...

    <dependencies>
        ...
        <dependency>
            <groupId>org.jboss.shrinkwrap.resolver</groupId>
            <artifactId>shrinkwrap-resolver-depchain</artifactId>
            <type>pom</type>
            <scope>test</scope>
        </dependency>
    </dependencies>
    ...

</project>
```

Remember to remove the original Tools BOM with the `org.jboss.bom.eap` groupId.

A Basic Deployment Class

In order to create Arquillian tests, we need to define the deployment. The code under test, as well as its dependencies is packaged and deployed in the container.

Much of the deployment contents is common for all tests, so we create a helper class with a method that creates the base deployment with all the general content.

src/test/java/org/jboss/examples/ticketmonster/test/TicketMonsterDeployment.java

```
public class TicketMonsterDeployment {

    public static WebArchive deployment() {
        return ShrinkWrap
                .create(WebArchive.class, "test.war")
                .addPackage(Resources.class.getPackage())
                .addAsResource("META-INF/test-persistence.xml", "META-
INF/persistence.xml")
                .addAsResource("import.sql")
                .addAsWebInfResource(EmptyAsset.INSTANCE, "beans.xml")
                // Deploy our test datasource
                .addAsWebInfResource("test-ds.xml");
    }
}
```

Remember to copy over the `test-persistence.xml` file into the `src/test/resources` directory of your project.

Arquillian uses Shrinkwrap to define the contents of the deployment. At runtime, when the test executes, Arquillian employs Shrinkwrap to create a WAR file that will be deployed to a running instance of JBoss Enterprise Application Platform. The WAR file would be composed of:

- all classes from the `org.jboss.examples.ticketmonster.util` package,
- the test `persistence.xml` file that defines a JPA persistence unit against a test datasource,
- the `import.sql` file,
- an empty `beans.xml` file to activate CDI
- and, a test data source definition.

We use a separate data source for our integration tests, and we recommend the same for real applications. This would allow you to run your tests against a pristine test environment, without having to clean your development, or worse, your production environment!

Writing RESTful service tests

For testing our JAX-RS RESTful services, we need to add the corresponding application classes to the deployment. Since we need to do that for each test we create, we abide by the DRY principles and create a utility class.

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());
    }

}
```

Now, we create the first test to validate the proper retrieval of individual events.

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

```
@RunWith(Arquillian.class)
public class VenueServiceTest {

    @Deployment
    public static WebArchive deployment() {
        return RESTDeployment.deployment();
    }

    @Inject
    private VenueService venueService;

    @Test
    public void testGetVenueById() {

        // Test loading a single venue
        Venue venue = venueService.getSingleInstance(1l);
        assertNotNull(venue);
        assertEquals("Roy Thomson Hall", venue.getName());
    }

}
```

In the class above we specify the deployment, and we define the test method. The test supports CDI injection - one of the strengths of Arquillian is the ability to inject the object being tested.

Now, we test a more complicated use cases, query parameters for pagination.

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

```
...
@RunWith(Arquillian.class)
public class VenueServiceTest {

    ...

    @Test
    public void testPagination() {

        // Test pagination logic
        MultivaluedMap<String, String> queryParameters = new
MultivaluedHashMap<String, String>();

        queryParameters.add("first", "2");
        queryParameters.add("maxResults", "1");

        List<Venue> venues = venueService.getAll(queryParameters);
        assertNotNull(venues);
        assertEquals(1, venues.size());
        assertEquals("Sydney Opera House", venues.get(0).getName());
    }

}
```

We add another test method (`testPagination`), which tests the retrieval of all venues, passing the search criteria as parameters. We use a Map to simulate the passing of query parameters.

Now, we test more advanced use cases such as the creation of a new booking. We do so by adding a new test for bookings

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

```
@RunWith(Arquillian.class)
public class BookingServiceTest {

    @Deployment
    public static WebArchive deployment() {
        return RESTDeployment.deployment();
    }

    @Inject
    private BookingService bookingService;

    @Inject
    private ShowService showService;

    @Test
    @InSequence(1)
    public void testCreateBookings() {
        BookingRequest br = createBookingRequest(1l, 0, new int[]{4, 1}, new
int[]{1,1}, new int[]{3,1});
        bookingService.createBooking(br);

        BookingRequest br2 = createBookingRequest(2l, 1, new int[]{6,1}, new
int[]{8,2}, new int[]{10,2});
        bookingService.createBooking(br2);

        BookingRequest br3 = createBookingRequest(3l, 0, new int[]{4,1}, new
int[]{2,1});
        bookingService.createBooking(br3);
    }

    @Test
    @InSequence(10)
    public void testGetBookings() {
        checkBooking1();
        checkBooking2();
        checkBooking3();
    }

    private void checkBooking1() {
        Booking booking = bookingService.getSingleInstance(1l);
        assertNotNull(booking);
        assertEquals("Roy Thomson Hall",
booking.getPerformance().getShow().getVenue().getName());
        assertEquals("Rock concert of the decade",
booking.getPerformance().getShow().getEvent().getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        // Test the ticket requests created

        assertEquals(3 + 2 + 1, booking.getTickets().size());

        List<String> requiredTickets = new ArrayList<String>();
        requiredTickets.add("A @ 219.5 (Adult)");
        requiredTickets.add("A @ 219.5 (Adult)");
        requiredTickets.add("D @ 149.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");
        requiredTickets.add("C @ 179.5 (Adult)");

        checkTickets(requiredTickets, booking);
    }

    private void checkBooking2() {
        Booking booking = bookingService.getSingleInstance(2l);
        assertNotNull(booking);
        assertEquals("Sydney Opera House",
booking.getPerformance().getShow().getVenue().getName());
        assertEquals("Rock concert of the decade",
booking.getPerformance().getShow().getEvent().getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        assertEquals(3 + 2 + 1, booking.getTickets().size());

        List<String> requiredTickets = new ArrayList<String>();
        requiredTickets.add("S2 @ 197.75 (Adult)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S4 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S6 @ 145.0 (Child 0-14yrs)");
        requiredTickets.add("S4 @ 145.0 (Child 0-14yrs)");

        checkTickets(requiredTickets, booking);
    }

    private void checkBooking3() {
        Booking booking = bookingService.getSingleInstance(3l);
        assertNotNull(booking);
        assertEquals("Roy Thomson Hall",
booking.getPerformance().getShow().getVenue().getName());
        assertEquals("Shane's Sock Puppets",
booking.getPerformance().getShow().getEvent().getName());
        assertEquals("bob@acme.com", booking.getContactEmail());

        assertEquals(2 + 1, booking.getTickets().size());
        List<String> requiredTickets = new ArrayList<String>();
        requiredTickets.add("B @ 199.5 (Adult)");
        requiredTickets.add("D @ 149.5 (Adult)");
        requiredTickets.add("B @ 199.5 (Adult)");

        checkTickets(requiredTickets, booking);
    }

    @Test
    @InSequence(10)
    public void testPagination() {

        // Test pagination logic
        MultivaluedMap<String, String> queryParameters = new
MultivaluedHashMap<java.lang.String, java.lang.String>();

        queryParameters.add("first", "2");
        queryParameters.add("maxResults", "1");

        List<Booking> bookings = bookingService.getAll(queryParameters);
        assertNotNull(bookings);
        assertEquals(1, bookings.size());
        assertEquals("Sydney Opera House",
bookings.get(0).getPerformance().getShow().getVenue().getName());
        assertEquals("Rock concert of the decade",
bookings.get(0).getPerformance().getShow().getEvent().getName());
    }

    @Test
    @InSequence(20)
    public void testDelete() {
        bookingService.deleteBooking(2l);
        checkBooking1();
        checkBooking3();
        try {
            bookingService.getSingleInstance(2l);
        } catch (Exception e) {
            if (e.getCause() instanceof NoResultException) {
                return;
            }
        }
        fail("Expected NoResultException did not occur.");
    }

    private BookingRequest createBookingRequest(Long showId, int performanceNo,
int[]... sectionAndCategories) {
        Show show = showService.getSingleInstance(showId);

        Performance performance = new
ArrayList<Performance>(show.getPerformances()).get(performanceNo);
```

```
        BookingRequest bookingRequest = new BookingRequest(performance,
"bob@acme.com");

        List<TicketPrice> possibleTicketPrices = new
ArrayList<TicketPrice>(show.getTicketPrices());
        int i = 1;
        for (int[] sectionAndCategory : sectionAndCategories) {
            for (TicketPrice ticketPrice : possibleTicketPrices) {
                int sectionId = sectionAndCategory[0];
                int categoryId = sectionAndCategory[1];
                if(ticketPrice.getSection().getId() == sectionId &&
ticketPrice.getTicketCategory().getId() == categoryId) {
                    bookingRequest.addTicketRequest(new TicketRequest(ticketPrice,
i));
                    i++;
                    break;
                }
            }
        }

        return bookingRequest;
    }

    private void checkTickets(List<String> requiredTickets, Booking booking) {
        List<String> bookedTickets = new ArrayList<String>();
        for (Ticket t : booking.getTickets()) {
            bookedTickets.add(new
StringBuilder().append(t.getSeat().getSection()).append(" @
").append(t.getPrice()).append("
(").append(t.getTicketCategory()).append(")").toString());
        }
        System.out.println(bookedTickets);
        for (String requiredTicket : requiredTickets) {
            Assert.assertTrue("Required ticket not present: " + requiredTicket,
bookedTickets.contains(requiredTicket));
        }
    }

}
```

First we test booking creation in a test method of its own (`testCreateBookings`). Then, we test that the previously created bookings are retrieved correctly (`testGetBookings` and `testPagination`). Finally, we test that deletion takes place correctly (`testDelete`).

The other tests in the application follow roughly the same pattern and are left as an exercise to the reader. You could in fact copy over the `EventServiceTest` and `ShowServiceTest` classes from the project sources.

Running the tests

If you have followed the instructions in the introduction and used the Maven archetype to generate the project structure, you should have two profiles already defined in your application.

/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

        ...
        <profile>
            <!-- An optional Arquillian testing profile that executes tests
                in your JBoss AS instance -->
            <!-- This profile will start a new JBoss AS instance, and execute
                the test, shutting it down when done -->
            <!-- Run with: mvn clean test -Parq-jbossas-managed -->
            <id>arq-jbossas-managed</id>
            <dependencies>
                <dependency>
                    <groupId>org.jboss.as</groupId>
                    <artifactId>jboss-as-arquillian-container-managed</artifactId>
                    <scope>test</scope>
                </dependency>
            </dependencies>
        </profile>

        <profile>
            <!-- An optional Arquillian testing profile that executes tests
                in a remote JBoss AS instance -->
            <!-- Run with: mvn clean test -Parq-jbossas-remote -->
            <id>arq-jbossas-remote</id>
            <dependencies>
                <dependency>
                    <groupId>org.jboss.as</groupId>
                    <artifactId>jboss-as-arquillian-container-remote</artifactId>
                    <scope>test</scope>
                </dependency>
            </dependencies>
        </profile>

    </profiles>
</project>
```

If you haven’t used the archetype, or the profiles don’t exist, create them.

Each profile defines a different Arquillian container. In both cases the tests execute in an application server instance. In one case (`arq-jbossas-managed`) the server instance is started and stopped by the test suite, while in the other (`arq-jbossas-remote`), the test suite expects an already started server instance.

Once these profiles are defined, we can execute the tests in two ways:

- from the command-line build
- from an IDE

Executing tests from the command line

You can now execute the test suite from the command line by running the Maven build with the appropriate target and profile, as in one of the following examples.

After ensuring that the `JBOSS_HOME` environment variable is set to a valid JBoss EAP 6.2 installation directory), you can run the following command:

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

Or, after starting a JBoss EAP 6.2 instance, you can run the following command

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

These tests execute as part of the Maven build and can be easily included in an automated build and test harness.

Running Arquillian tests from within Eclipse

Running the entire test suite as part of the build is an important part of the development process - you may want to make sure that everything is working fine before releasing a new milestone, or just before committing new code. However, running the entire test suite all the time can be a productivity drain, especially when you're trying to focus on a particular problem. Also, when debugging, you don't want to leave the comfort of your IDE for running the tests.

Running Arquillian tests from JBoss Developer Studio or JBoss tools is very simple as Arquillian builds on JUnit (or TestNG).

First enable one of the two profiles in the project. In Eclipse, select the project, right-click on it to open the context menu, drill down into the *Maven* sub-menu:

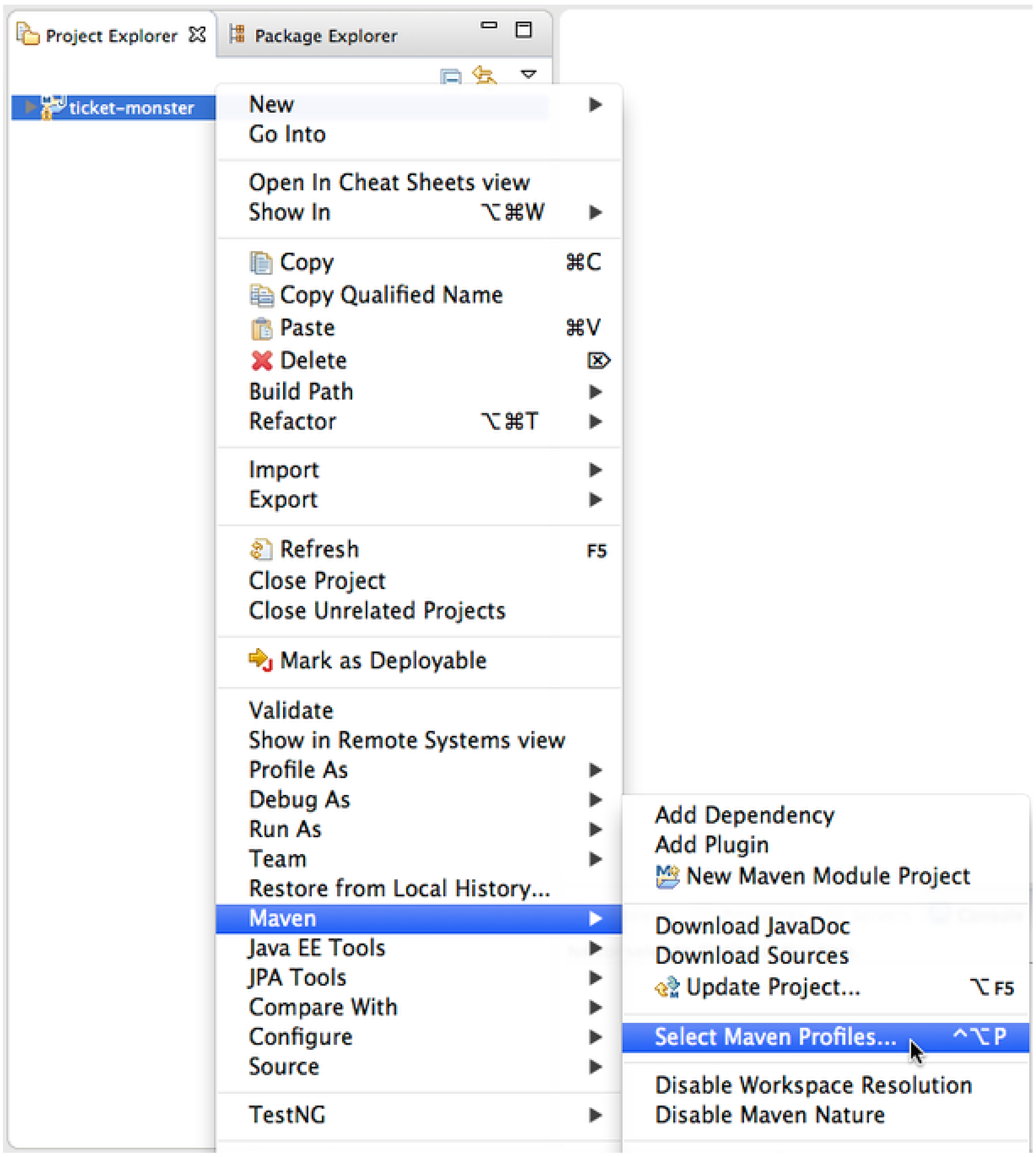
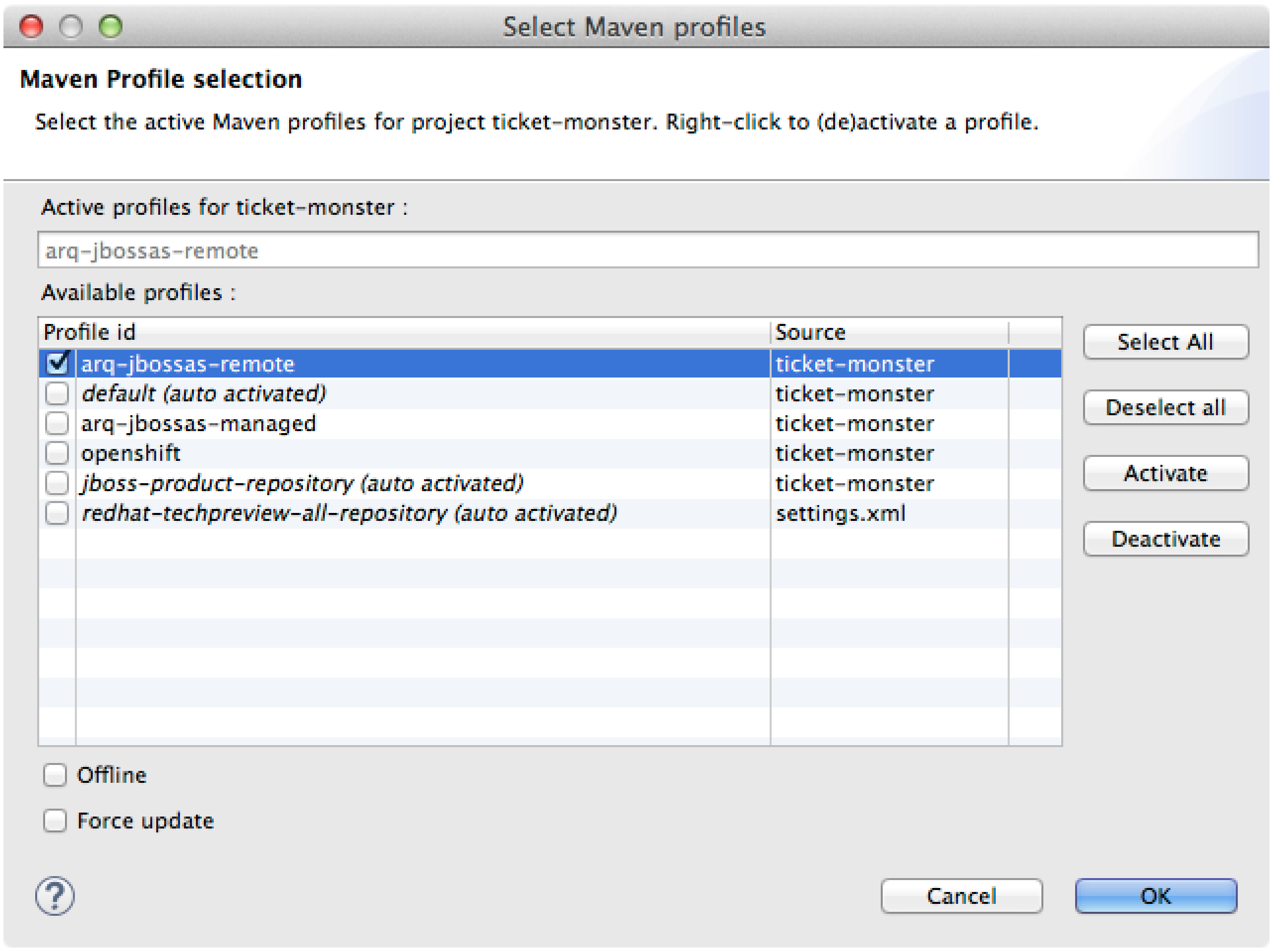


Figure 1. Select the Maven profiles for the project

Activate the profile as shown in the picture below.


Select Maven profiles

Maven Profile selection

Select the active Maven profiles for project ticket-monster. Right-click to (de)activate a profile.

Active profiles for ticket-monster :

arq-jbossas-remote

Available profiles :

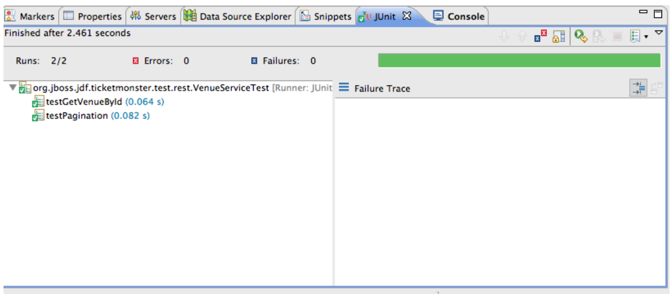
	Profile id	Source
✓	arq-jbossas-remote	ticket-monster
	default (auto activated)	ticket-monster
	arq-jbossas-managed	ticket-monster
	openshift	ticket-monster
	jboss-product-repository (auto activated)	ticket-monster
	redhat-techpreview-all-repository (auto activated)	settings.xml

Figure 2. Update Maven profiles in Eclipse

The project configuration will be updated automatically.

Now, you can click right on one of your test classes, and select **Run As** → **JUnit Test**.

The test suite will run, deploying the test classes to the application server, executing the tests and finally producing the much coveted green bar.



Finished after 2.461 seconds

Runs: 2/2 Errors: 0 Failures: 0

org.jboss.jdf.ticketmonster.test.rest.VenueServiceTest [Runner: JUnit
testGetVenueById (0.064 s)
testPagination (0.082 s)

Figure 3. Running the tests