Building the persistence layer with JPA2 and Bean Validation

What will you learn here?

You have set up your project successfully. Now it is time to begin working on the TicketMonster application, and the first step is adding the persistence layer. After reading this guide, you'll understand what design and implementation choices to make. Topics covered include:

- RDBMS design using JPA entity beans
- How to validate your entities using Bean Validation
- How to populate test data
- Basic unit testing using JUnit

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. For those of you who prefer to watch and learn, the included videos show you how we performed all the steps.

TicketMonster contains 14 entities, of varying complexity. In the introduction, you have seen the basic steps for creating a couple of entities (`Event` and `Venue`) and interacting with them. In this tutorial we'll go deeper into domain model design, we'll classify the entities, and walk through designing and creating one of each group.

Your first entity

The simplest kind of entities are often those representing lookup tables. `TicketCategory` is a classic lookup table that defines the ticket types available (e.g. Adult, Child, Pensioner). A ticket category has one property - *description*.

TIP

What's in a name?

Using a consistent naming scheme for your entities can help another developer get up to speed with your code base. We've named all our lookup tables XXXCategory to allow us to easily spot them.

Let's start by creating a JavaBean to represent the ticket category:

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

```
public class TicketCategory {

    /* Declaration of fields */

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     */
    private String description;

    /* Boilerplate getters and setters */

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @Override
    public String toString() {
        return description;
    }
}
```

We're going to want to keep the ticket category in collections (for example, to present it as part of drop down in the UI), so it's important that we properly implement `equals()` and `hashCode()`. At this point, we need to define a property (or group of properties) that uniquely identifies the ticket category. We refer to these properties as the "entity's natural identity".

TIP

Defining an entity's natural identity

Using an ORM introduces additional constraints on object identity. Defining the properties that make up an entity's natural identity can be tricky, but is very important. Using the object's identity, or the synthetic identity (database generated primary key) identity can introduce unexpected bugs into your application, so you should always ensure you use a natural identity. You can read more about the issue at https://community.jboss.org/wiki/EqualsAndHashCode.

For ticket category, the choice of natural identity is easy and obvious - it must be the one property, *description* that the entity has! Having identified the natural identity, adding an `equals()` and `hashCode()` method is easy. In Eclipse, choose *Source* → *Generate hashCode() and equals()…*

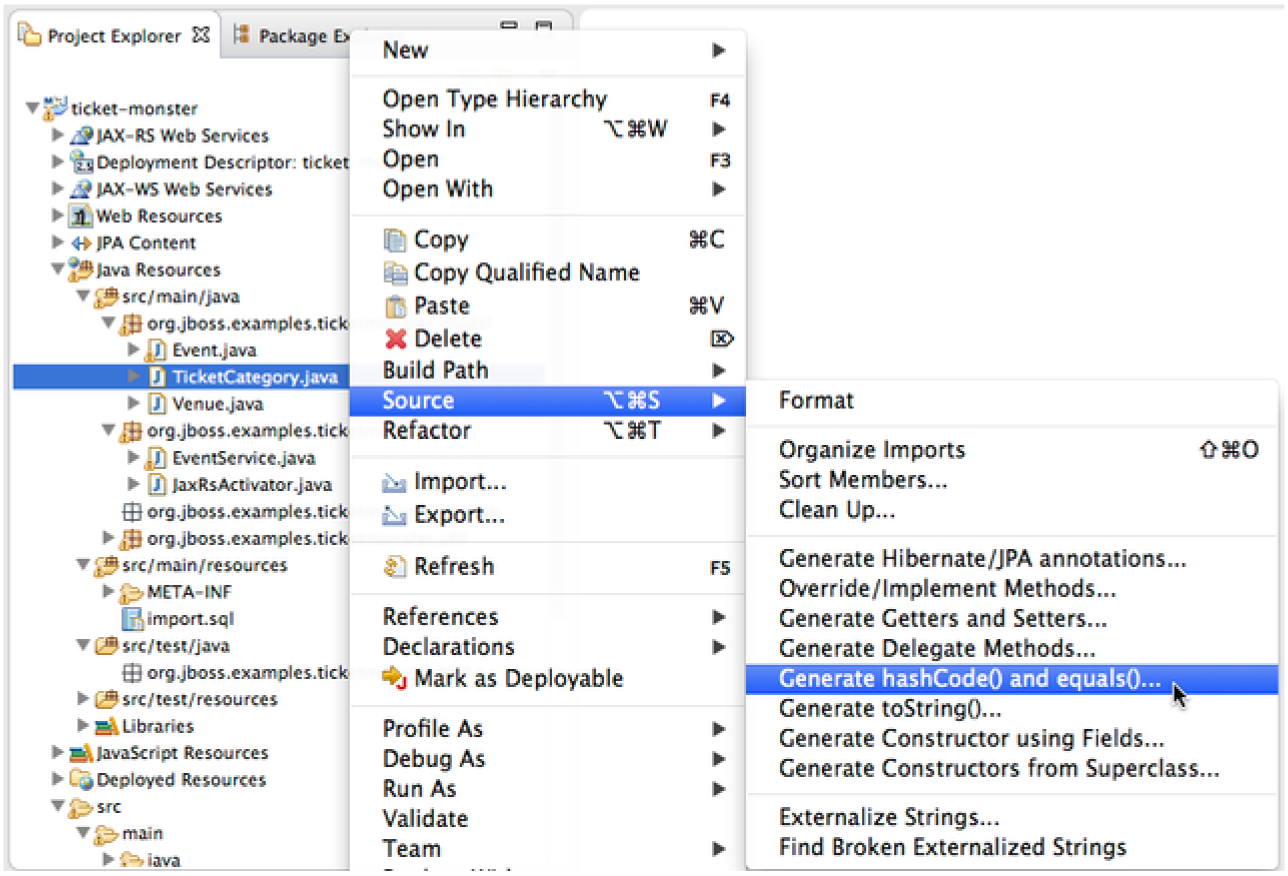


Figure 1. Generate hashCode() and equals() in Eclipse

Now, select the properties to include:

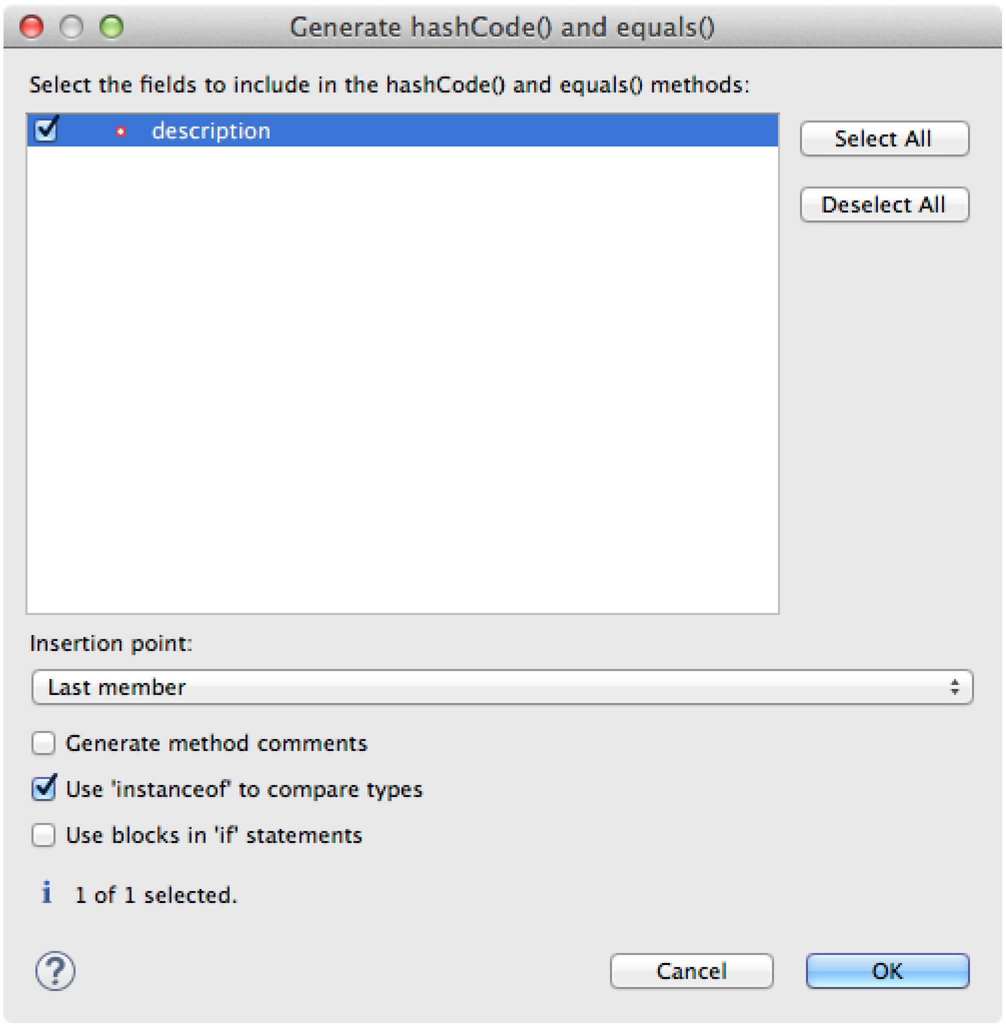

Generate hashCode() and equals()
Select the fields to include in the hashCode() and equals() methods:
☑ description
Select All
Deselect All
Insertion point:
Last member
☐ Generate method comments
☑ Use 'instanceof' to compare types
☐ Use blocks in 'if' statements
1 of 1 selected.
Cancel
OK

Figure 2. Generate hashCode() and equals() in Eclipse

Now that we have a JavaBean, let's proceed to make it an entity. First, add the `@Entity` annotation to the class:

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

```
@Entity
public class TicketCategory {

   ...

}
```

And, add the synthetic id:

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

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    ...

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    ...

}
```

As we decided that our natural identifier was the `description`, we should introduce a unique constraint on the property:

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

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    ...

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be
unique.
     * </p>
     *
     */
    @Column(unique = true)
    private String description;

    ...

}
```

It’s very important that any data you place in the database is of the highest quality - this data is probably one of your organisations most valuable assets! To ensure that bad data doesn’t get saved to the database by mistake, we’ll use Bean Validation to enforce constraints on our properties.

NOTE

What is Bean Validation?

Bean Validation (JSR 303) is a Java EE specification which:

- provides a unified way of declaring and defining constraints on an object model.
- defines a runtime engine to validate objects

Bean Validation includes integration with other Java EE specifications, such as JPA. Bean Validation constraints are automatically applied before data is persisted to the database, as a last line of defence against bad data.

The *description* of the ticket category should not be empty for two reasons. Firstly, an empty ticket category description is no use to a person trying to book a ticket - it doesn’t convey any information. Secondly, as the description forms the natural identity, we need to make sure the property is always populated.

Let’s add the Bean Validation constraint `@NotEmpty`:

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

```
@Entity
public class TicketCategory {

    /* Declaration of fields */

    ...

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be
unique.
     * </p>
     *
     * <p>
     * The description must not be null and must be one or more characters, the Bean
Validation constraint <code>@NotEmpty</code>
     * enforces this.
     * </p>
     *
     */
    @Column(unique = true)
    @NotEmpty
    private String description;
    ...
}
```

And that is our first entity! Here is the complete entity:

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

```
/**
 * <p>
 * A lookup table containing the various ticket categories. E.g. Adult, Child,
Pensioner, etc.
 * </p>
 */
@Entity
public class TicketCategory {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The description of the of ticket category.
     * </p>
     *
     * <p>
     * The description forms the natural id of the ticket category, and so must be
unique.
     * </p>
     *
     * <p>
     * The description must not be null and must be one or more characters, the Bean
Validation constraint <code>@NotEmpty</code>
     * enforces this.
     * </p>
     *
     */
    @Column(unique = true)
    @NotEmpty
    private String description;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    /* toString(), equals() and hashCode() for TicketCategory, using the natural
identity of the object */

    @Override
    public String toString() {
        return description;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result
                + ((description == null) ? 0 : description.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof TicketCategory))
            return false;
        TicketCategory other = (TicketCategory) obj;
        if (description == null) {
            if (other.description != null)
                return false;
        } else if (!description.equals(other.description))
            return false;
        return true;
    }
}
```

TicketMonster contains another lookup tables, `EventCategory`. It's pretty much identical to

TicketCategory, so we leave it as an exercise to the reader to investigate, and understand. If you are building the application whilst following this tutorial, copy the source over from the TicketMonster example.

Database design & relationships

First, let's understand the the entity design.

An Event may occur at any number of venues, on various days and at various times. The intersection between an event and a venue is a Show, and each show can have a Performance which is associated with a date and time.

Venues are a separate grouping of entities, which, as mentioned, intersect with events via shows. Each venue consists of groupings of seats, each known as a Section.

Every section, in every show is associated with a ticket category via the TicketPrice entity.

Users must be able to book tickets for performances. A Booking is associated with a performance, and contains a collection of tickets.

Finally, both events and venues can have "media items", such as images or videos attached.

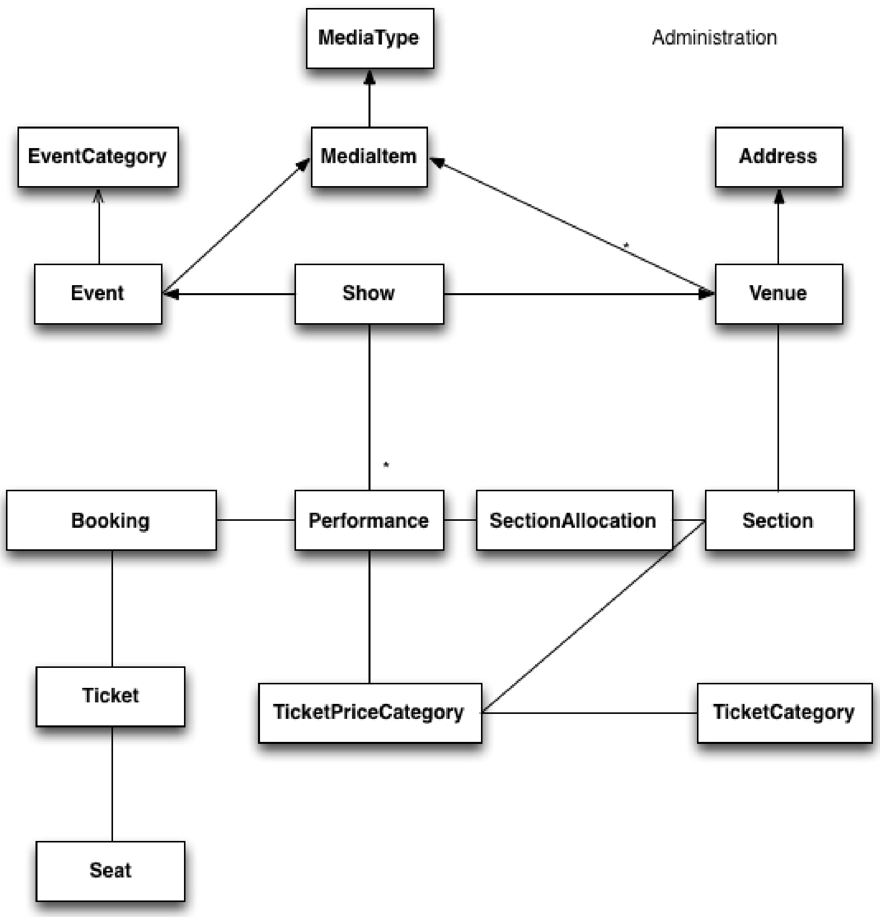


Figure 3. Entity-Relationship Diagram

Media items

Storing large binary objects, such as images or videos in the database isn’t advisable (as it can lead to performance issues), and playback of videos can also be tricky, as it depends on browser capabilities. For TicketMonster, we decided to make use of existing services to host images and videos, such as YouTube or Flickr. All we store in the database is the URL the application should use to access the media item, and the type of the media item (note that the URL forms a media items natural identifier). We need to know the type of the media item in order to render the media correctly in the view layer.

In order for a view layer to correctly render the media item (e.g. display an image, embed a media player), it’s likely that special code has had to have been added. For this reason we represent the types of media that TicketMonster understands as a closed set, unmodifiable at runtime. An enum is perfect for this!

Luckily, JPA has native support for enums, all we need to do is add the `@Enumerated` annotation:

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

```
    ...

    /**
     * <p>
     * The type of the media, required to render the media item correctly.
     * </p>
     *
     * <p>
     * The media type is a <em>closed set</em> - as each different type of media
requires support coded into the view layers, it
     * cannot be expanded upon without rebuilding the application. It is therefore
represented by an enumeration. We instruct
     * JPA to store the enum value using it's String representation, so that we can
later reorder the enum members, without
     * changing the data. Of course, this does mean we can't change the names of media
items once the app is put into
     * production.
     * </p>
     */
    @Enumerated(STRING)
    private MediaType mediaType;

    ...
```

TIP

@Enumerated(STRING) or @Enumerated(ORDINAL)?

JPA can store an enum value using it’s ordinal (position in the list of declared enums) or it’s STRING (the name it is given). If you choose to store an ordinal, you musn’t alter the order of the list. If you choose to store the name, you musn’t change the enum name. The choice is yours!

The rest of MediaItem shouldn't present a challenge to you. If you are building the application whilst following this tutorial, copy both MediaItem and MediaType from the TicketMonster project.

Events

In the section Your first entity, we saw how to build simple entities with properties, identify and apply constraints using Bean Validation, identify the natural id and add a synthetic id. From now on we'll assume you know how to build simple entities - for each new entity that we build, we will start with it's basic structure and properties filled in.

So, here we modify the Event class (from where we left at the end of the introduction). The below listing also includes some comments reflecting the explanations above. We will remove a few fields - version, major and picture, update the annotations on the id field, and update the toString, equals and hashCode methods to use the natural key of the object):

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

```
@Entity
public class Event {

    /* Declaration of fields */

    /**
     * The synthetic ID of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between
events.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the name must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 5 characters and no
more than 50 characters. This allows for
     * better formatting consistency in the view layer.</li>
     * </ol>
     */
    @Column(unique = true)
    @NotNull
    @Size(min = 5, max = 50, message = "An event's name must contain between 5 and 50
characters")
    private String name;

    /**
     * <p>
     * A description of the event.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the description must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 20 characters and no
more than 1000 characters. This allows for
     * better formatting consistency in the view layer, and also ensures that event
organisers provide at least some description
     * - a classic example of a business constraint.</li>
     * </ol>
     */
    @NotNull
    @Size(min = 20, max = 1000, message = "An event's description must contain between
20 and 1000 characters")
    private String description;


    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }

    /* toString(), equals() and hashCode() for Event, using the natural identity of
the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Event event = (Event) o;

        if (name != null ? !name.equals(event.name) : event.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

First, let's add a media item to `Event`. As multiple events (or venues) could share the same media item, we'll model the relationship as *many-to-one* - many events can reference the same media item.

TIP

Relationships supported by JPA

JPA can model four types of relationship between entities - one-to-one, one-to-many, many-to-one and many-to-many. A relationship may be bi-directional (both sides of the relationship know about each other) or uni-directional (only one side knows about the relationship).

Many database models are hierarchical (parent-child), as is TicketMonster's. As a result, you'll probably find you mostly use one-to-many and many-to-one relationships, which allow building parent-child models.

Creating a many-to-one relationship is very easy in JPA. Just add the `@ManyToOne` annotation to the field. JPA will take care of the rest. Here's the property for `Event`:

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

```
    ...

    /**
     * <p>
     * A media item, such as an image, which can be used to entice a browser to book a
ticket.
     * </p>
     *
     * <p>
     * Media items can be shared between events, so this is modeled as a
<code>@ManyToOne</code> relationship.
     * </p>
     *
     * <p>
     * Adding a media item is optional, and the view layer will adapt if none is
provided.
     * </p>
     *
     */
    @ManyToOne
    private MediaItem mediaItem;

    ...

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem picture) {
        this.mediaItem = picture;
    }

    ...
```

There is no need for a media item to know who references it (in fact, this would be a poor design, as it would reduce the reusability of `MediaItem`), so we can leave this as a uni-directional relationship.

An event will also have a category. Once again, many events can belong to the same event category, and there is no need for an event category to know what events are in it. To add this relationship, we add the `eventCategory` property, and annotate it with `@ManyToOne`, just as we did for `MediaItem`:

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

```
    ...

    /**
     * <p>
     * The category of the event
     * </p>
     *
     * <p>
     * Event categories are used to ease searching of available of events, and hence
this is modeled as a relationship
     * </p>
     *
     * <p>
     * The Bean Validation constraint <code>@NotNull</code> indicates that the event
category must be specified.
     */
    @ManyToOne
    @NotNull
    private EventCategory category;

    ...

    public EventCategory getCategory() {
        return category;
    }

    public void setCategory(EventCategory category) {
        this.category = category;
    }

    ...
```

And that’s Event created. Here is the full source:

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

```
/**
 * <p>
 * Represents an event, which may have multiple performances with different dates and
venues.
 * </p>
 *
 * <p>
 * Event's principal members are it's relationship to {@link EventCategory} -
specifying the type of event it is - and
 * {@link MediaItem} - providing the ability to add media (such as a picture) to the
event for display. It also contains
 * meta-data about the event, such as it's name and a description.
 * </p>
 *
 */
@Entity
public class Event {

    /* Declaration of fields */
    /**
     * The synthetic ID of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between
events.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the name must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 5 characters and no
more than 50 characters. This allows for
     * better formatting consistency in the view layer.</li>
     * </ol>
     */
    @Column(unique = true)
    @NotNull
    @Size(min = 5, max = 50, message = "An event's name must contain between 5 and 50
characters")
    private String name;

    /**
     * <p>
     * A description of the event.
     * </p>
     *
     * <p>
     * Two constraints are applied using Bean Validation
     * </p>
     *
     * <ol>
     * <li><code>@NotNull</code> &mdash; the description must not be null.</li>
     * <li><code>@Size</code> &mdash; the name must be at least 20 characters and no
more than 1000 characters. This allows for
     * better formatting consistency in the view layer, and also ensures that event
organisers provide at least some description
     * - a classic example of a business constraint.</li>
     * </ol>
     */
    @NotNull
    @Size(min = 20, max = 1000, message = "An event's name must contain between 20 and
1000 characters")
    private String description;

    /**
     * <p>
     * A media item, such as an image, which can be used to entice a browser to book a
ticket.
     * </p>
     *
     * <p>
     * Media items can be shared between events, so this is modeled as a
<code>@ManyToOne</code> relationship.
     * </p>
     *
     * <p>
     * Adding a media item is optional, and the view layer will adapt if none is
provided.
     * </p>
     *
     */
    @ManyToOne
    private MediaItem mediaItem;

    /**
     * <p>
     * The category of the event
     * </p>
     *
     * <p>
     * Event categories are used to ease searching of available of events, and hence
this is modeled as a relationship
     * </p>
     *
     * <p>
     * The Bean Validation constraint <code>@NotNull</code> indicates that the event
category must be specified.
     */
    @ManyToOne
    @NotNull
    private EventCategory category;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem picture) {
        this.mediaItem = picture;
    }

    public EventCategory getCategory() {
        return category;
    }

    public void setCategory(EventCategory category) {
        this.category = category;
    }

    /* toString(), equals() and hashCode() for Event, using the natural identity of
the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Event event = (Event) o;

        if (name != null ? !name.equals(event.name) : event.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        return name != null ? name.hashCode() : 0;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

Venue

Now, let’s build out the entities to represent the venue.

We start by adding an entity to represent the venue. A venue needs to have a name, a description, a capacity, an address, an associated media item and a set of sections in which people can sit. If you completed the introduction chapter, you should already have some of these properties set, so we will update the `Venue` class to look like in the definition below.

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

```
/**
 * <p>
 * Represents a single venue
 * </p>
 *
 */
@Entity
public class Venue {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The name of the event.
     * </p>
     *
     * <p>
     * The name of the event forms it's natural identity and cannot be shared between
events.
```

```
     * </p>
     *
     * <p>
     * The name must not be null and must be one or more characters, the Bean
Validation
     * constraint <code>@NotEmpty</code> enforces this.
     * </p>
     */
    @Column(unique = true)
    @NotEmpty
    private String name;

    /**
     * The address of the venue
     */
    @Embedded
    private Address address = new Address();

    /**
     * A description of the venue
     */
    private String description;

    /**
     * <p>
     * A set of sections in the venue
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
     * Collection members are fetched eagerly, so that they can be accessed even after
the
     * entity has become detached. This relationship is bi-directional (a section
knows which
     * venue it is part of), and the <code>mappedBy</code> attribute establishes this.
We
     * cascade all persistence operations to the set of performances, so, for example
if a venue
     * is removed, then all of it's sections will also be removed.
     * </p>
     */
    @OneToMany(cascade = ALL, fetch = EAGER, mappedBy = "venue")
    private Set<Section> sections = new HashSet<Section>();

    /**
     * The capacity of the venue
     */
    private int capacity;

    /**
     * An optional media item to entice punters to the venue. The
```

```
<code>@ManyToOne</code> establishes the relationship.
     */
    @ManyToOne
    private MediaItem mediaItem;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    public MediaItem getMediaItem() {
        return mediaItem;
    }

    public void setMediaItem(MediaItem description) {
        this.mediaItem = description;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public Set<Section> getSections() {
        return sections;
    }
    public void setSections(Set<Section> sections) {
        this.sections = sections;
    }

    public int getCapacity() {
        return capacity;
    }

    public void setCapacity(int capacity) {
        this.capacity = capacity;
    }

    /* toString(), equals() and hashCode() for Venue, using the natural identity of
the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Venue venue = (Venue) o;

        if (address != null ? !address.equals(venue.address) : venue.address != null)
            return false;
        if (name != null ? !name.equals(venue.name) : venue.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (address != null ? address.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

In creating this entity, we've followed all the design and implementation decisions previously discussed, with one new concept. Rather than add the properties for street, city, postal code etc. to this object, we've extracted them into the `Address` object, and included it in the `Venue` object using composition. This would allow us to reuse the Address object in other places (such as a customer's address).

A RDBMS doesn't have a similar concept to composition, so we need to choose whether to represent the address as a separate entity, and create a relationship between the venue and the address, or whether to map the properties from `Address` to the table for the owning entity, in this case `Venue`. It doesn't make much sense for an address to be a full entity - we're not going to want to run queries against the address in isolation, nor do we want to be able to delete or update an address in isolation - in essence, the address doesn't have a standalone identity outside of the object into which it is composed.

To *embed* the `Address` into `Venue` we add the `@Embeddable` annotation to the `Address` class. However, unlike a full entity, there is no need to add an identifier. Here's the source for `Address`:

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

```
/**
 * <p>
 * A reusable representation of an address.
 * </p>
 *
 * <p>
 * Addresses are used in many places in an application, so to observe the DRY
principle, we model Address as an embeddable
 * entity. An embeddable entity appears as a child in the object model, but no
relationship is established in the RDBMS..
 * </p>
 */
@Embeddable
public class Address {

    /* Declaration of fields */
    private String street;
    private String city;
    private String country;

    /* Declaration of boilerplate getters and setters */

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }
    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

    /* toString(), equals() and hashCode() for Address, using the natural identity of
the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Address address = (Address) o;

        if (city != null ? !city.equals(address.city) : address.city != null)
            return false;
        if (country != null ? !country.equals(address.country) : address.country !=
null)
            return false;
        if (street != null ? !street.equals(address.street) : address.street != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = street != null ? street.hashCode() : 0;
        result = 31 * result + (city != null ? city.hashCode() : 0);
        result = 31 * result + (country != null ? country.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return street + ", " + city + ", " + country;
    }
}
```

Sections

A venue consists of a number of seating sections. Each seating section has a name, a description, the number of rows in the section, and the number of seats in a row. It's natural identifier is the name of section combined with the venue (a venue can't have two sections with the same name). `Section` doesn't introduce any new concepts, so go ahead and copy the source from the below listing:

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

```
@SuppressWarnings("serial")
@Entity
@Table(uniqueConstraints=@UniqueConstraint(columnNames={"name", "venue_id"}))
public class Section implements Serializable {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Long id;

    /**
     * <p>
     * The short name of the section, may be a code such as A12, G7, etc.
     * </p>
     *
     * <p>
     * The
     * <code>@NotEmpty<code> Bean Validation constraint means that the section name
must be at least 1 character.
     * </p>
     */
    @NotEmpty
    private String name;

    /**
     * <p>
     * The description of the section, such as 'Rear Balcony', etc.
     * </p>
     *
     * <p>
     * The
     * <code>@NotEmpty<code> Bean Validation constraint means that the section
description must be at least 1 character.
     * </p>
     */
    @NotEmpty
    private String description;

    /**
     * <p>
     * The venue to which this section belongs. The <code>@ManyToOne<code> JPA mapping
establishes this relationship.
```

```
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the venue must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Venue venue;

    /**
     * The number of rows that make up the section.
     */
    private int numberOfRows;

    /**
     * The number of seats in a row.
     */
    private int rowCapacity;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getNumberOfRows() {
        return numberOfRows;
    }
    public void setNumberOfRows(int numberOfRows) {
        this.numberOfRows = numberOfRows;
    }

    public int getRowCapacity() {
        return rowCapacity;
    }

    public void setRowCapacity(int rowCapacity) {
        this.rowCapacity = rowCapacity;
    }

    public int getCapacity() {
        return this.rowCapacity * this.numberOfRows;
    }

    public Venue getVenue() {
        return venue;
    }

    public void setVenue(Venue venue) {
        this.venue = venue;
    }

    /* toString(), equals() and hashCode() for Section, using the natural identity of
the object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Section section = (Section) o;

        if (venue != null ? !venue.equals(section.venue) : section.venue != null)
            return false;
        if (name != null ? !name.equals(section.name) : section.name != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + (venue != null ? venue.hashCode() : 0);
        return result;
    }
    @Override
    public String toString() {
        return name;
    }

}
```

Shows

A show is an event at a venue. It consists of a set of performances of the show. A show also contains the list of ticket prices available.

Let's start building Show. Here's is our starting point:

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

```
/**
 * <p>
 * A show is an instance of an event taking place at a particular venue. A show can
have multiple performances.
 * </p>
 */
@Entity
public class Show {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA
mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Event event;

    /**
```

```
     * <p>
     * The venue where this show takes place. The <code>@ManyToOne<code> JPA mapping
establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the venue must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Venue venue;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Event getEvent() {
        return event;
    }

    public void setEvent(Event event) {
        this.event = event;
    }

    public Venue getVenue() {
        return venue;
    }

    public void setVenue(Venue venue) {
        this.venue = venue;
    }

    /* toString(), equals() and hashCode() for Show, using the natural identity of the
object */
    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Show show = (Show) o;
        if (event != null ? !event.equals(show.event) : show.event != null)
            return false;
        if (venue != null ? !venue.equals(show.venue) : show.venue != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = event != null ? event.hashCode() : 0;
        result = 31 * result + (venue != null ? venue.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return event + " at " + venue;
    }
}
```

If you've been paying attention, you'll notice that there is a problem here. We've identified that the natural identity of this entity is formed of two properties - the *event* and the *venue*, and we've correctly coded the `equals()` and `hashCode()` methods (or had them generated for us!). However, we haven't told JPA that these two properties, in combination, must be unique. As there are two properties involved, we can no longer use the `@Column` annotation (which operates on a single property/table column), but now must use the class level `@Table` annotation (which operates on the whole entity/table). Change the class definition to read:

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

```
...

@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "event_id", "venue_id"
}))
public class Show {

    ...
}
```

You'll notice that JPA requires us to use the column names, rather than property names here. The column names used in the `@UniqueConstraint` annotation are those generated by default for properties called `event` and `venue`.

Additionally, `Show` is a reserved word in certain databases, most notable MySQL. We'll specify a different table name as a result, so that Hibernate will generate correct DDL statements:

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

```
...

@Entity
@Table(name="Appearance", uniqueConstraints = @UniqueConstraint(columnNames = {
"event_id", "venue_id" }))
public class Show {

    ...
}
```

Now, let’s add the set of performances to the event. Unlike previous relationships we’ve seen, the relationship between a show and it’s performances is bi-directional. We chose to model this as a bi-directional relationship in order to improve the generated database schema (otherwise you end with complicated mapping tables which makes updates to collections hard). Let’s add the set of performances:

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

```
    ...

    /**
     * <p>
     * The set of performances of this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
Collection members
     * are fetched eagerly, so that they can be accessed even after the entity has
become detached.
     * This relationship is bi-directional (a performance knows which show it is part
of), and the <code>mappedBy</code>
     * attribute establishes this.
     * </p>
     *
     */
    @OneToMany(fetch=EAGER, mappedBy = "show", cascade = ALL)
    @OrderBy("date")
    private Set<Performance> performances = new HashSet<Performance>();

    ...

    public Set<Performance> getPerformances() {
        return performances;
    }

    public void setPerformances(Set<Performance> performances) {
        this.performances = performances;
    }

    ...
```

As the relationship is bi-directional, we specify the `mappedBy` attribute on the `@OneToMany` annotation, which informs JPA to create a bi-directional relationship. The value of the attribute is name of property which forms the other side of the relationship - in this case, not unsuprisingly `show`!

As `Show` is the owner of `Performance` (and without a show, a performance cannot exist), we add the `cascade = ALL` attribute to the `@OneToMany` annotation. As a result, any persistence operation that occurs on a show, will be propagated to it's performances. For example, if a show is removed, any associated performances will be removed as well.

When retrieving a show, we will also retrieve its associated performances by adding the `fetch = EAGER` attribute to the `@OneToMany` annotation. This is a design decision which required careful consideration. In general, you should favour the default lazy initialization of collections: their content should be accessible on demand. However, in this case we intend to marshal the contents of the collection and pass it across the wire in the JAX-RS layer, after the entity has become detached, and cannot initialize its members on demand.

We'll also need to add the set of ticket prices available for this show. Once more, this is a bi-directional relationship, owned by the show. It looks just like the set of performances:

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

```
    ...

    /**
     * <p>
     * The set of ticket prices available for this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
     * This relationship is bi-directional (a ticket price category knows which show
 it is part of), and the <code>mappedBy</code>
     * attribute establishes this. We cascade all persistence operations to the set of
 performances, so, for example if a show
     * is removed, then all of it's ticket price categories are also removed.
     * </p>
     */
    @OneToMany(mappedBy = "show", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    private Set<TicketPrice> ticketPrices = new HashSet<TicketPrice>();

    ...

    public Set<TicketPrice> getTicketPrices() {
        return ticketPrices;
    }
    public void setTicketPrices(Set<TicketPrice> ticketPrices) {
        this.ticketPrices = ticketPrices;
    }

    ...
```

Here’s the full source for Show:

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

```
/**
 * <p>
 * A show is an instance of an event taking place at a particular venue. A show can
have multiple performances.
 * </p>
 *
 * <p>
 * A show contains a set of performances, and a set of ticket prices for each section
of the venue for this show.
 * </p>
 *
 * <p>
 * The event and venue form the natural id of this entity, and therefore must be
unique. JPA requires us to use the class level
 * <code>@Table</code> constraint.
 * </p>
 *
 */
/*
 * We suppress the warning about not specifying a serialVersionUID, as we are still
developing this app, and want the JVM to
 * generate the serialVersionUID for us. When we put this app into production, we'll
generate and embed the serialVersionUID
 */
@SuppressWarnings("serial")
@Entity
@Table(name="Appearance", uniqueConstraints = @UniqueConstraint(columnNames = {
"event_id", "venue_id" }))
public class Show implements Serializable {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA
mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Event event;

    /**
     * <p>
     * The event of which this show is an instance. The <code>@ManyToOne<code> JPA
mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the event must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Venue venue;

    /**
     * <p>
     * The set of performances of this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship. TODO
Explain EAGER fetch.
     * This relationship is bi-directional (a performance knows which show it is part
of), and the <code>mappedBy</code>
     * attribute establishes this. We cascade all persistence operations to the set of
performances, so, for example if a show
     * is removed, then all of it's performances will also be removed.
     * </p>
     *
     * <p>
     * Normally a collection is loaded from the database in the order of the rows, but
here we want to make sure that
     * performances are ordered by date - we let the RDBMS do the heavy lifting. The
     * <code>@OrderBy<code> annotation instructs JPA to do this.
     * </p>
     */
    @OneToMany(fetch = EAGER, mappedBy = "show", cascade = ALL)
    @OrderBy("date")
    private Set<Performance> performances = new HashSet<Performance>();

    /**
     * <p>
     * The set of ticket prices available for this show.
     * </p>
     *
     * <p>
     * The <code>@OneToMany<code> JPA mapping establishes this relationship.
     * This relationship is bi-directional (a ticket price category knows which show
it is part of), and the <code>mappedBy</code>
     * attribute establishes this. We cascade all persistence operations to the set of
performances, so, for example if a show
     * is removed, then all of it's ticket price categories are also removed.
     * </p>
     */
    @OneToMany(mappedBy = "show", cascade = ALL, fetch = EAGER)
    private Set<TicketPrice> ticketPrices = new HashSet<TicketPrice>();

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Event getEvent() {
        return event;
    }

    public void setEvent(Event event) {
        this.event = event;
    }

    public Venue getVenue() {
        return venue;
    }

    public void setVenue(Venue venue) {
        this.venue = venue;
    }

    public Set<Performance> getPerformances() {
        return performances;
    }

    public void setPerformances(Set<Performance> performances) {
        this.performances = performances;
    }

    public Set<TicketPrice> getTicketPrices() {
        return ticketPrices;
    }

    public void setTicketPrices(Set<TicketPrice> ticketPrices) {
        this.ticketPrices = ticketPrices;
    }

    /* toString(), equals() and hashCode() for Show, using the natural identity of the
object */
    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Show show = (Show) o;

        if (event != null ? !event.equals(show.event) : show.event != null)
            return false;
        if (venue != null ? !venue.equals(show.venue) : show.venue != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = event != null ? event.hashCode() : 0;
        result = 31 * result + (venue != null ? venue.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return event + " at " + venue;
    }
}
```

TicketPrices

The `Show` entity references two classes - `TicketPrice` and `Performance`, that are not yet created. Let's first create the `TicketPrice` class which represents the price for a ticket in a particular `Section` at a `Show` for a specific `TicketCategory`. It does not introduce any new concepts, so go ahead and copy the source from the below listing:

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

```
/**
 * <p>
 * Contains price categories - each category represents the price for a ticket in a
particular section at a particular venue for
 * a particular event, for a particular ticket category.
 * </p>
 *
 * <p>
 * The section, show and ticket category form the natural id of this entity, and
therefore must be unique. JPA requires us to use the class level
 * <code>@Table</code> constraint
 * </p>
 *
 */
/*
 * We suppress the warning about not specifying a serialVersionUID, as we are still
developing this app, and want the JVM to
 * generate the serialVersionUID for us. When we put this app into production, we'll
generate and embed the serialVersionUID
 */
@SuppressWarnings("serial")
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "section_id", "show_id",
"ticketcategory_id" }))
public class TicketPrice implements Serializable {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
     * <p>
     * The show to which this ticket price category belongs. The
<code>@ManyToOne<code> JPA mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the show must
be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Show show;
    /**
     * <p>
     * The section to which this ticket price category belongs. The
<code>@ManyToOne<code> JPA mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the section
must be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Section section;

    /**
     * <p>
     * The ticket category to which this ticket price category belongs. The
<code>@ManyToOne<code> JPA mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The <code>@NotNull</code> Bean Validation constraint means that the ticket
category must be specified.
     * </p>
     */
    @ManyToOne
    @NotNull
    private TicketCategory ticketCategory;

    /**
     * The price for this category of ticket.
     */
    private float price;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Show getShow() {
        return show;
    }

    public void setShow(Show show) {
        this.show = show;
    }

    public Section getSection() {
        return section;
    }

    public void setSection(Section section) {
        this.section = section;
    }

    public TicketCategory getTicketCategory() {
        return ticketCategory;
    }

    public void setTicketCategory(TicketCategory ticketCategory) {
        this.ticketCategory = ticketCategory;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    /* equals() and hashCode() for TicketPrice, using the natural identity of the
object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        TicketPrice that = (TicketPrice) o;

        if (section != null ? !section.equals(that.section) : that.section != null)
            return false;
        if (show != null ? !show.equals(that.show) : that.show != null)
            return false;
        if (ticketCategory != null ? !ticketCategory.equals(that.ticketCategory) :
that.ticketCategory != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = show != null ? show.hashCode() : 0;
        result = 31 * result + (section != null ? section.hashCode() : 0);
        result = 31 * result + (ticketCategory != null ? ticketCategory.hashCode() :
0);
        return result;
    }

    @Override
    public String toString() {
        return "$ " + price + " for " + ticketCategory + " in " + section;
    }
}
```

Performances

Finally, let's create the `Performance` class, which represents an instance of a `Show`. Performance is pretty straightforward. It contains the date and time of the performance, and the show of which it is a performance. Together, the show, and the date and time, make up the natural identity of the performance. Here's the source for `Performance`:

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

```
/**
 * <p>
 * A performance represents a single instance of a show.
 * </p>
 *
 * <p>
 * The show and date form the natural id of this entity, and therefore must be unique.
JPA requires us to use the class level
 * <code>@Table</code> constraint.
 * </p>
 *
 */
@SuppressWarnings("serial")
@Entity
@Table(uniqueConstraints = @UniqueConstraint(columnNames = { "date", "show_id" }))
public class Performance implements Serializable {

    /* Declaration of fields */

    /**
     * The synthetic id of the object.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    /**
```

```
     * <p>
     * The date and start time of the performance.
     * </p>
     *
     * <p>
     * A Java {@link Date} object represents both a date and a time, whilst an RDBMS
splits out Date, Time and Timestamp.
     * Therefore we instruct JPA to store this date as a timestamp using the
<code>@Temporal(TIMESTAMP)</code> annotation.
     * </p>
     *
     * <p>
     * The date and time of the performance is required, and the Bean Validation
constraint <code>@NotNull</code> enforces this.
     * </p>
     */
    @Temporal(TIMESTAMP)
    @NotNull
    private Date date;

    /**
     * <p>
     * The show of which this is a performance. The <code>@ManyToOne<code> JPA mapping
establishes this relationship.
     * </p>
     *
     * <p>
     * The show of which this is a performance is required, and the Bean Validation
constraint <code>@NotNull</code> enforces
     * this.
     * </p>
     */
    @ManyToOne
    @NotNull
    private Show show;

    /* Boilerplate getters and setters */

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setShow(Show show) {
        this.show = show;
    }

    public Show getShow() {
        return show;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    /* equals() and hashCode() for Performance, using the natural identity of the
object */

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        Performance that = (Performance) o;

        if (date != null ? !date.equals(that.date) : that.date != null)
            return false;
        if (show != null ? !show.equals(that.show) : that.show != null)
            return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = date != null ? date.hashCode() : 0;
        result = 31 * result + (show != null ? show.hashCode() : 0);
        return result;
    }
}
```

Of interest here is the storage of the date and time.

A Java `Date` represents "a specific instance in time, with millisecond precision" and is the recommended construct for representing date and time in the JDK. A RDBMS's *DATE* type typically has day precision only, and uses the *DATETIME* or *TIMESTAMP* types to represent an instance in time, and often only to second precision.

As the mapping between Java date and time, and database date and time isn't straightforward, JPA requires us to use the `@Temporal` annotation on any property of type `Date`, and to specify whether the `Date` should be stored as a date, a time or a timestamp (date and time).

Booking, Ticket & Seat

There aren't many new concepts to explore in `Booking`, `Ticket` and `Seat`, so if you are following along with the tutorial, you should copy in the `Booking`, `Ticket` and `Seat` classes.

Once the user has selected an event, identified the venue, and selected a performance, they have the opportunity to request a number of seats in a given section, and select the category of tickets required. Once they've chosen their seats, and entered their email address, a `Booking` is created.

A booking consists of the date the booking was created, an email address (as TicketMonster doesn't yet have fully fledged user management), a set of tickets and the associated performance. The set of tickets shows us how to create a uni-directional one-to-many relationship:

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

```
    ...

    /**
     * <p>
     * The set of tickets contained within the booking. The <code>@OneToMany<code> JPA
mapping establishes this relationship.
     * </p>
     *
     * <p>
     * The set of tickets is eagerly loaded because FIXME . All operations are
cascaded to each ticket, so for example if a
     * booking is removed, then all associated tickets will be removed.
     * </p>
     *
     * <p>
     * This relationship is uni-directional, so we need to inform JPA to create a
foreign key mapping. The foreign key mapping
     * is not visible in the {@link Ticket} entity despite being present in the
database.
     * </p>
     *
     */
    @OneToMany(fetch = EAGER, cascade = ALL)
    @JoinColumn
    @NotEmpty
    @Valid
    private Set<Ticket> tickets = new HashSet<Ticket>();

    ...
```

We add the `@JoinColumn` annotation, which sets up a foreign key in `Ticket`, but doesn't expose the booking on Ticket. This prevents the use of messy mapping tables, whilst preserving the integrity of the entity model.

A ticket embeds the seat allocated, and contains a reference to the category under which it was sold. It also contains the price at which it was sold.

SectionAllocation and SeatAllocationException

Finally, we'd like to track the seats to be allocated from a section during the course of booking tickets. We'll use the `SectionAllocation` entity to track the allocations in every section for every performance. You can copy in the `SectionAllocation` class from the project sources.

The notable member in this class is the two-dimensional array, named `allocated`. It tracks the state of the section - the first dimension represents the rows in the section, and the second represents the state of every seat in the row. A typical RDBMS would have to store such a structure as a LOB (Large Object) or a BLOB (Binary Large Object), since a n-dimensional array does not map easily to a native data type supported by the database. Thus, we denote the field as a `@Lob` using the JPA annotation:

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

```
    ...

    @Lob
    private long[][] allocated;

    ...
```

The rest of the class contains business logic to update the state of the `allocated` field. These methods will come in handy later, when we write the business services.

Remember to also copy the `SeatAllocationException` class, that is referenced in this class, from the project sources. This class represents an Application exception that will be recognized by the EJB container as one that should force a transaction rollback. When this exception is thrown by the business logic in the `SectionAllocation` entity, and propagated to the EJB container, it will implcitly cause the current transaction to roll back. It is to be noted that, this exception class is not a checked exception (it extends `RuntimeException`), and thus the compiler does not complain when it is uncaught in the business services that will consume the methods in the `SectionAllocation` entity.

Connecting to the database

In this example, we are using the in-memory H2 database, which is very easy to set up on JBoss AS. JBoss AS allows you deploy a datasource inside your application's `WEB-INF` directory. You can locate the source in `src/main/webapp/WEB-INF/ticket-monster-ds.xml` (which should have been created in the previous chapter):

src/main/webapp/WEB-INF/ticket-monster-ds.xml

```
<datasources xmlns="http://www.jboss.org/ironjacamar/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.org/ironjacamar/schema
http://docs.jboss.org/ironjacamar/schema/datasources_1_0.xsd">
```

```
    <!-- The datasource is bound into JNDI at this location. We reference
        this in META-INF/persistence.xml -->
    <datasource jndi-name="java:jboss/datasources/ticket-monsterDS"
        pool-name="ticket-monster" enabled="true" use-java-context="true">
        <connection-url>
            jdbc:h2:mem:ticket-monster;DB_CLOSE_ON_EXIT=FALSE;DB_CLOSE_DELAY=-1
        </connection-url>
        <driver>h2</driver>
        <security>
            <user-name>sa</user-name>
            <password>sa</password>
        </security>
    </datasource>
</datasources>
```

The datasource configures an H2 in-memory database, called *ticket-monster*, and registers a datasource in JNDI at the address:

```
java:jboss/datasources/ticket-monsterDS
```

Now we need to configure JPA to use the datasource. This is done in `src/main/resources/META-INF/persistence.xml`:

src/main/resources/persistence.xml

```
<persistence version="2.0"
   xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="primary">
      <!-- If you are running in a production environment, add a managed
         data source, this example data source is just for development and testing!
-->
      <!-- The datasource is deployed as WEB-INF/ticket-monster-ds.xml, you
         can find it in the source at src/main/webapp/WEB-INF/ticket-monster-ds.xml
-->
      <jta-data-source>java:jboss/datasources/ticket-monsterDS</jta-data-source>
      <properties>
         <!-- Properties for Hibernate -->
         <property name="hibernate.hbm2ddl.auto" value="create-drop" />
         <property name="hibernate.show_sql" value="false" />
      </properties>
   </persistence-unit>
</persistence>
```

As our application has only one datasource, and hence one persistence unit, the name given to the persistence unit doesn't really matter. We call ours `primary`, but you can change this as you like. We tell JPA about the datasource bound in JNDI.

Hibernate includes the ability to generate tables from entities, which we have configured here. We don't recommend using this outside of development. Updates to databases in production should be done in a staged manner by a database administrator.

Populating test data

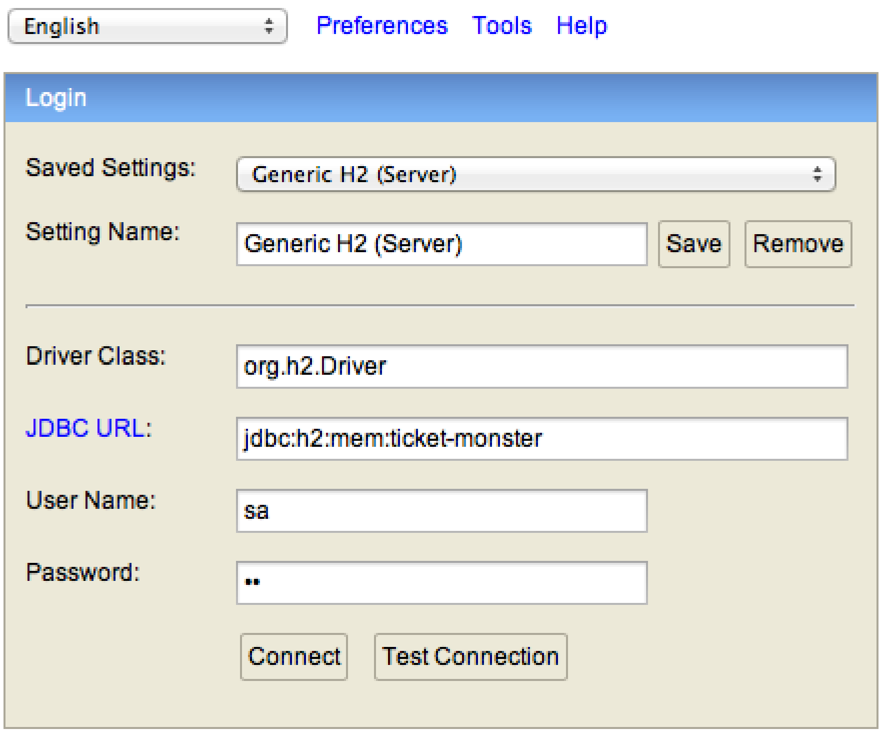
Whilst we develop our application, it's useful to be able to populate the database with test data. Luckily, Hibernate makes this easy. Just add a file called `import.sql` onto the classpath of your application (we keep it in `src/main/resources/import.sql`). In it, we just write standard sql statements suitable for the database we are using. To do this, you need to know the generated column and table names for your entities. The best way to work these out is to look at the h2console.

The h2console is included in the JBoss AS quickstarts, along with instructions on how to use it. For more information, see http://www.jboss.org/quickstarts/eap/h2-console/

TIP

Where do I look for my data?

The database URL is `jdbc:h2:mem:ticket-monster`. After you have downloaded `h2console.war` and deployed it on the server, make sure that the application is running on the server and use this value to connect to your running application's database.



Saved Settings: Generic H2 (Server)
Setting Name: Generic H2 (Server)
Driver Class: org.h2.Driver
JDBC URL: jdbc:h2:mem:ticket-monster
User Name: sa
Password: ••

Figure 4. h2console settings

You should copy over the `import.sql` file from the project sources, to populate the database with the same data, as the one used in the OpenShift-hosted TicketMonster application. The contents of this file already account for the generated table and column names.

Conclusion

You now have a working data model for your TicketMonster application, our next tutorial will show you how to create the business services layer or something like that - it seems to end abruptly.