Java EE Fundamentals

Course handout

Introduction

The principles of developing web applications didn’t change much since the dawn of providing dynamic content in Internet. There is usually a central (sometimes distributed) storage of user and application data. Then there is some business logic that works with that data. And finally there is the application view, which is used by the end user to communicate with the data.

These three tiers are not always set like that. In some solutions the business logic is developed directly in the database. While in other simpler ones the view part manipulates directly the data without going through an intermediate layer. The views themselves walked a long path from ugly representations of the underlying storage to appealing pieces of art that look much like desktop applications.

But the principles remain the same: there is data that should be processed and presented to the user. Throughout the years there were many approaches on how to put those principles to work. But quite a few of those approaches stood out. One of them is Java Enterprise Edition (or Java EE). It is a composition of a number of standards neatly weaved together to represent one whole.

This course will take you though the most prominent of these standard technologies and show you how you can use them to easily build a working web application. It is going to focus on the latest version of Java EE - Java EE 7.

Sample application

To showcase the usage of the different Java EE technologies, we'll develop a sample web application: *Online Magazine*. Its final version is supposed to be used by various personas:

- *Owner* of the magazine for managing the various parts of their business: content, advertisers, subscribers, contests, etc.
- *Authors* for submitting new articles and reviewing their current content
- *Users* for browsing the content and participating in contests

The initial version of the project is almost empty - it contains a pom.xml and part of the data model. In each installment of this training we'll add a few new features that cover the respective part of the material.

So, let's start with what we have. Initially in our project we've taken care about the project setup and about part of our domain model.

We have a very minimal pom.xml:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.vidasoft</groupId>
    <artifactId>magazine-manager</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <build>
        <finalName>magman</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
            <version>5.2.5.Final</version>
        </dependency>
        <dependency>
            <groupId>org.apache.derby</groupId>
            <artifactId>derby</artifactId>
            <version>10.13.1.1</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

At the end our domain model will be much richer than what it is now. But so far we have the users hierarchy, the content and the advertisers.

The user hierarchy includes the domain classes for the subscriber, the author and the manager. They all inherit from an abstract base class - `User`. The latter has all the necessary attributes of a website user - user name, password, first name, last name and email address. Most of the subclasses add to that:

- `Subscriber` adds things like street address and when the subscription expires
- `Author` adds whether it is regular (i.e. on the payroll) and what is their current salary. For regulars this is the monthly salary, while for contractors this is the per article royalty
- `Manager` does not add any attributes to the base class

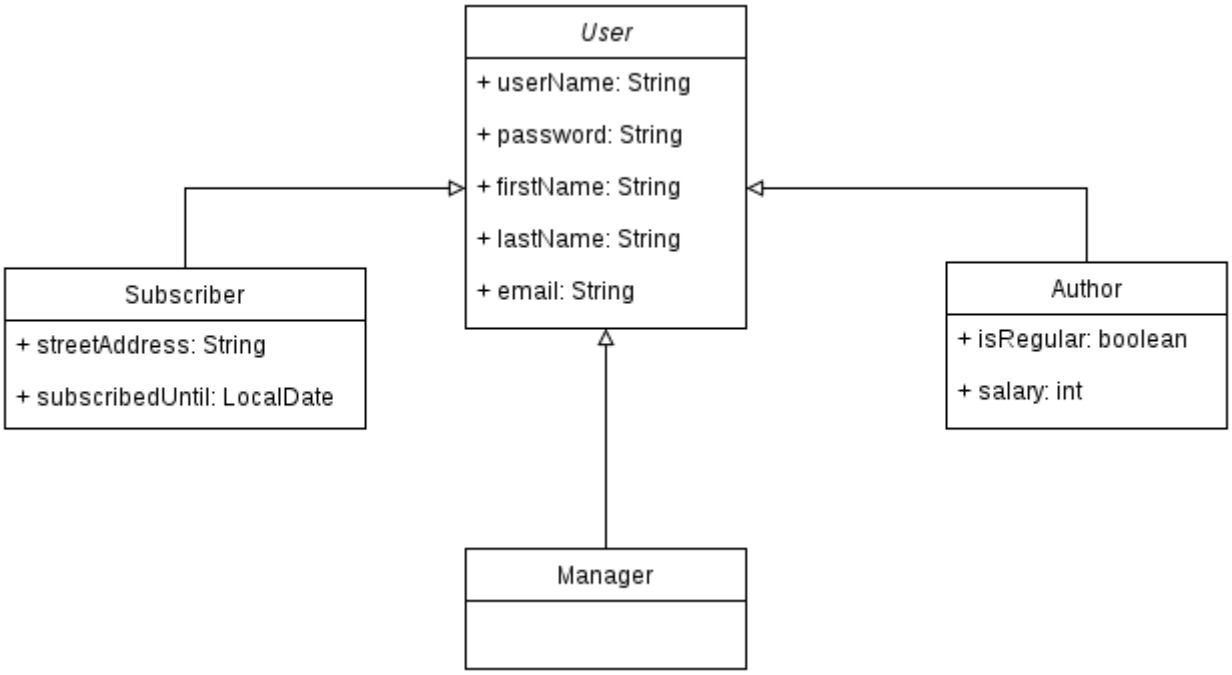

Figure 1. The Users data model

The domain objects for the content comprise `Article` and `Comment`. Articles are written by authors, have title, content, publish date and a list of comments. The comments maintain information about their author, who can be any registered user, the creation time and of course the content.

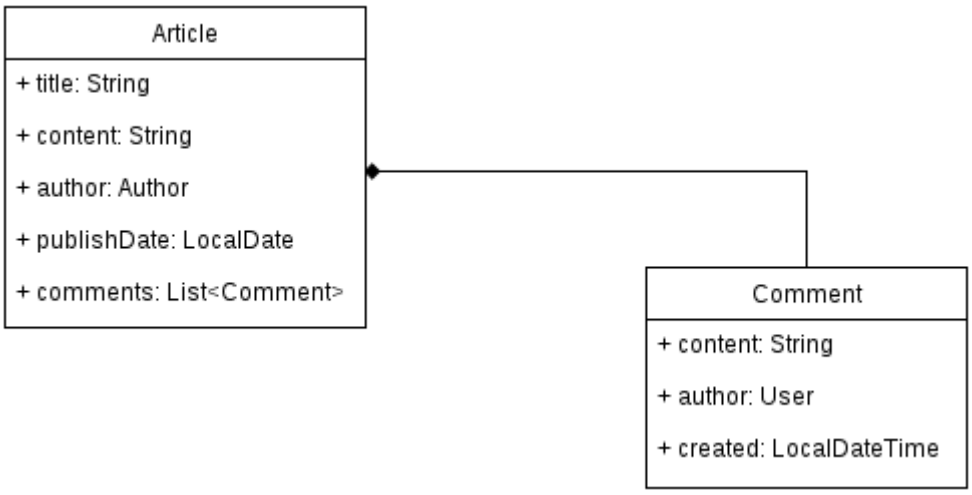

Figure 2. The content domain objects

Last but not least come the advertisers. The `Advertiser` type has data about their name, the website, the email of the contact person, the logo (in binary format) and the sponsorship package. The different packages are defined in the `SponsorPackage` enumeration. It defines three such levels: `PLATINUM`, `GOLD` and `SILVER` and for each of them maintains its price.

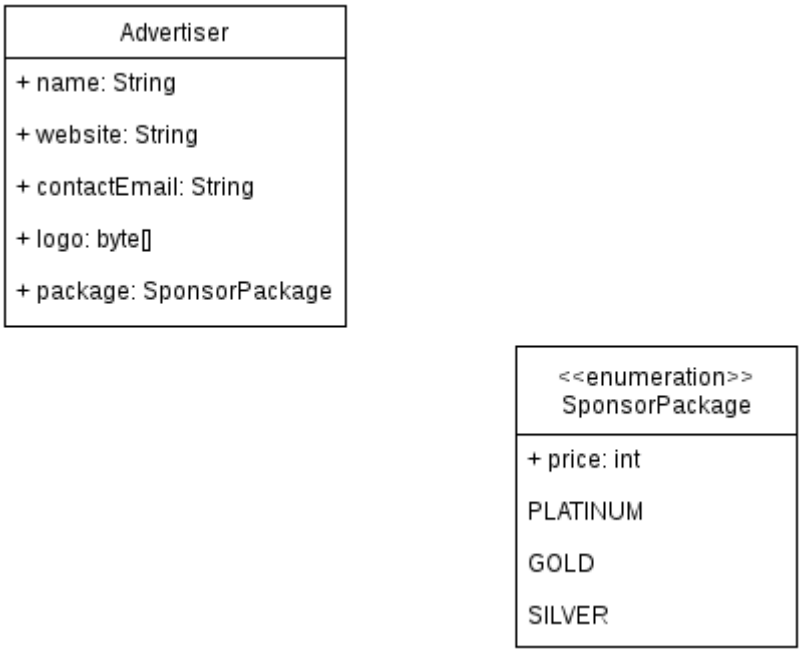

Figure 3. Advertisers

Java Persistence API (JPA)

We'll start our journey into building the magazine manager web application with setting up its data model. The standard way to do that and to then persist the modelled data is with the Java Persistence API (JPA).

JPA Terminology

Entity - A non-final plain old Java object (POJO) with a non-argument constructor. It represents a database table with its member variables representing the table columns.

Persistence context, **EntityManager** - Both terms mean one and the same thing. It is the interface that you use to load and store entities from and to the database. It acts also as a first level cache for your application.

Persistence unit, **EntityManagerFactory** - The factory which you use to obtain the entity manager. There you configure the way to connect to the database - a server datasource or direct JDBC properties.

JPQL - Object-oriented query language, based on SQL. It works on objects and their fields, rather than on tables, columns and relationships.

As we haven't introduced any of the web tier technologies yet, we'll resort to the Java SE usage of JPA. This means that we as applicaton developers will have to initialize the `EntityManagerFactory` with the JDBC properties to connect to the database. Then we'll need to manually create `EntityManager` instances, begin and commit transactions. This will change in the next parts of this series when we hit the application server. But today all the code that we write will execute in a main class and run as Java SE application.

You might have noticed the two dependencies in the pom.xml. They will also disappear in the next parts of the course. But we need them today to provide JPA API and implementation as well as database management system and the Java driver to that. Here is a short description of both of them:

- Hibernate carries both the `javax.persistence` APIs along with its own implementation
- We'll use Derby as in-memory database in this installment of our series. The jar contains the JDBC driver as well as embedded, in-memory DBMS

Modelling the data

Enabling JPA

The purpose of our first task is to make sure that the classes in the `com.vidasoft.magman.model` package are properly annotated so that the persistence provider is able to map them to the underlying database. In order to at all bootstrap JPA, we need to create the descriptor file persistence.xml. It should end up in the WEB-INF/classes/META-INF directory in the war file or MATA-INF in a jar. For a Maven project, this means that persistence.xml should be created in the

src/main/resources/META-INF folder.

Here is the content of the file (explanation of the various parts follow after that):

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
    <persistence-unit name="magazine-manager" transaction-type="RESOURCE_LOCAL">
        <properties>
            <!-- JDBC properties -->
            <property name="javax.persistence.jdbc.url" value=
"jdbc:derby:memory:library;create=true"/>
            <property name="javax.persistence.jdbc.user" value="APP"/>
            <property name="javax.persistence.jdbc.password" value="APP"/>
            <property name="javax.persistence.jdbc.driver" value=
"org.apache.derby.jdbc.EmbeddedDriver"/>
            <property name="javax.persistence.schema-generation.database.action"
value="create"/>
            <!-- Hibernate properties -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

The root element contains information that will help the persistence provider determine which version of JPA will be in use.

Then comes the persistence-unit element. It is used to define common settings for the entity classes, that are managed by the persistence provider. There are two types of persistence units - `RESOURCE_LOCAL` and `JTA`. This defines who is taking care of creating the persistence context and managing the transactions. We already said that our program will run as a Java SE application and not in the container in the first installment of this course. That is why the chosen persistence unit type is `RESOURCE_LOCAL`. Once we start running our app on a server in the next installment, we will change that to `JTA`.

Version 2.1 of the JPA greatly reduced the ammount of configurations that the user has to put in persistence.xml. Most of it is optional, especially with the `JTA` transaction type. As we are running in resource local mode, we need to tell our persistence provider how to connect to the database. For that we specify standard JDBC parameters like URL, user, password and JDBC driver. In this installment we are going to use Derby in-memory database, hence the URL is like that.

We also want to let our persistence provider generate the DB schema for us based on the entities configuration each time the application starts. That is why we set `create` to the `javax.persistence.schema-generation.database.action` property.

All these above properties are defined in the spec and have to be supported by all the JPA providers. However, there is some amount of configuration that is not standard, but is still very useful. For example, while developing and debugging our application, we might want to see the SQL that is generated by the persistence provider. Thus we will be able to fine tune and optimize our model and our queries. For that purpose we added these Hibernate specific entries:

```
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
```

If you still want to monitor the activity in the productive database, you need to look into one of the DB auditing solutions, like Hibernate Envers for example.

Creating our first entities

Onve we've set up JPA's persistence.xml, it is time to tell it which are our entities. In order to turn a simple class into a JPA entity, it needs two things:

- `@Entity` class-level annotation
- The fields that is chosen as primary key must be annotated with `@Id`

We'll start with the user entity hierarchy. It comprises the `User` abstract base class and its concrete realizations: `Subscriber`, `Author` and `Manager`.

Even though they can't be directly instantiated, abstract classes may be declared entities. If an abstract class is queried, the query will span all it's subclasses. For example if we want to check whether a user exists and we don't care whether it is manager, author or just subscriber, we can look up directly the `User` entity.

So, let's start by turning `com.vidasoft.magman.model.User` into JPA entity:

```
@Entity
public abstract class User {

   // ...

}
```

We also need to assign a field that will work as primary key. Even though it might be tempting to use `userName` for that purpose, it is almost always a better idea to have surrogate primary keys. That is why we will create one - a field of type `Long` that we'll call `id` and annotate it with `@Id`.

Another important concern about surrogate primary keys is who is responsible for their generation. Again, a common practice is to leave that to the persistence provider. You can do that by introducing another annotation on the `id` field: `@GeneratedValue`.

You can further instruct the persistence provider on the strategy it should use for generating the IDs: database sequence, database identity column or database identity table. By choosing `AUTO` we

let the persistence provider decide the best strategy. Hibernate will usually pick database sequence when available.

Another interesting aspect is what is the actual database table name that the entities map to. By convention it is the same as the class name. That means that the `User` entity would map to `User` table. However, for some databases (like Derby) that is not a valid identifier. So we will need to override the default by picking a table name of our own, for example `Users`. This is done with the `@Table` annotation and its `name` attribute, placed on the class level.

Having that sorted out, our `User` entity is good to go:

```
@Entity
@Table(name = "Users")
public abstract class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    // Other fields and constructors

    public Long getId() {
        return id;
    }

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

    // Other methods

}
```

Now the question is what happens to the User subclasses. We said that the `User` class is abstract and cannot be instantiated. So all its attributes (`userName`, `password`, `firstName`, etc.) including `id` have no meaning in the context of a `User`. They are all inherited and there is no real point in individual subclasses to override them. So we will not add `id` field to `Manager`, `Author` and `Subscriber`. The only thing we need to do is annotate them with `@Entity`:

```
@Entity
public class Manager extends User {

   // no id attribute, it is inheritted

}

@Entity
public class Author extends User {

   // no id attribute, it is inheritted

}

@Entity
public class Subscriber extends User {

   // no id attribute, it is inheritted

}
```

How will that be represented in the database? Well, there are different strategies in JPA for entities extending from another entity. The default one, which we'll also pick here, is that the data of all the four entities is located in one single table. That is the table corresponding to the parent entity (in our case `Users`). There is a separate column for each attribute both of the parent and child entities. If the respective column doesn't make sense for a certain entity, e.g. `salary` for `Subscriber`, it will be left empty for the corresponding record. There is also a discriminator column (called by default DTYPE for Hibernate), which contains information about the entity to which the corresponding record belongs.

So, for example, if we insert a `Manager` record and we rely on the conventions, `isRegular`, `salary`, `streetAddress` and `subscribedUntil` will be empty, while DTYPE will be set to `Manager`.

This strategy is called *Single Table per Class*. There are two more strategies in the spec:

- *Table per Concrete Class* - there is a separate table for each concrete class in the hierarchy. All the fields, including the ones from the parent class, are maintained in this table
- *Joined Subclass* - the base class is represented by a single table and each class in the hierarchy has its own table, which holds just the fields specific to that entity. Loading the entity almost always requires at least one database JOIN operation.

The selected strategy is controlled by the `@Inheritance` annotation placed on the root of the hierarchy. It can be skipped for the *Single Table per Class* strategy.

With the most complex entity structure in our domain out of the way, we can go on now and model the rest of the entities.

Turn the rest of the POJOs from the data model into JPA entities by adding the `id` field and using the proper `javax.persistence` annotations.

Managing entity relationships

So far we've only looked at the entities living in isolation. But this is not a common practice for the enterprise applications. The *Magazine Manager* is not an exception to that rule either.

Here the Article entity is in relation to the Author entity in the sense that each article has an author and an author can write multiple articles. For the sake of simplicity we assume that an article can only have one author. Additionally, articles can have zero or more comments.

In a relational world, this is modelled with the help of foreign keys that point to the other table's primary keys, as well as with intermediate tables. However, in the object oriented world the same concept is represented with object composition - `Article` has `Author` and `List<Comment>` member variables. JPA tackles this mismatch in the concepts (often referred to as *Object-relational impedance mismatch*) by introducing a few annotations.

In our case, we'll annotate the `author` and `comments` fields in the `Article` entity with `@ManyToOne` and `@OneToMany` respectively:

```
@Entity
public class Article {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String title;
    private String content;
    private LocalDate publishDate;

    @ManyToOne
    private Author author;
    @OneToMany
    private List<Comment> comments = new ArrayList<>();

    // constructors and methods

}
```

Use the appropriate annotation for the `author` member variable of the `Comment` entity.

Additional concerns

The persistence provider knows exactly what to do with most of the entity attribute types. String member variables are mapped to VARCHAR(255) columns, int variables - to INTEGER columns, long

variables - to BIGINT, etc. However, it needs additional configuration for some of the Java types. The `Advertiser` entity has two such fields - the `SponsorPackage` enum and the `byte[]` for the company logo.

The Java enumerations are handled with a special column in the database table. That column will either contain the ordinal number of the enum value or its name. If you think that your enum is going to change in future versions of your application, which may lead to change in ordinal numbers of existing values, then you should pick the second enum type. Put the `@Enumerated` constant on such member variables, where if needed specify the `EnumType` (it is EnumType.NUMBER by default).

Similar special handling applies for the `byte[]` fields. There we want to tell the persistence provider to map them to a database binary large object type (or BLOB). You can do that by using the @Lob annotation for the field that you want to be treated that way.

So here is how our `Advertiser` entity looks like:

```
@Entity
public class Advertiser {

    // member variables

    @Enumerated(EnumType.STRING)
    private SponsorPackage sponsorPackage;

    @Lob
    private byte[] logo;

    // constructors and methods
}
```

As mentioned earlier on in this section, the default column length for a String variable is 255. However, this is not enough for article and comment content. We want to override that default and for that we will use the `@Column` annotation placed on the field we want to customize.

Let's say tat we want to set the column length for the article's content to 20000 symbols. For that purpose we just need to do the following:

```
@Entity
public class Article {

    // member variables

    @Column(length = 20000)
    private String content;

    // constructors and methods

}
```

Make comment's column length 10000 symbols.

Converting the data

As already mentioned, JPA persistence providers can map the Java primitive types and the most used reference types. This also includes `java.util.Date`, `java.sql.Date` and `java.sql.Timestamp`. Java 8, however, comes with far better support for dates and times via the types in the `java.time` package. Unfortunately, Java EE 7 targets Java SE 7, so there is no built in support for classes like `LocalDate` and `LocalDateTime`. But still we want to use these types to represent publish times of articles and comments as well as the expiration date of the subscriptions.

A useful feature that will help us in this direction is JPA's attribute converters. This is a simple interface with two methods, that you need to implement in order to tell the persistence provider how to convert a custom type to one of the supported ones. In our case we want to convert `java.time.LocalDate` to `java.sql.Date` and `java.time.LocalDateTime` to `java.sql.Timestamp`. So we want to implement `AttributeConverter` with the following specialization: `<LocalDateTime, Timestamp>`. We also have to annotate our class with `@Converter`.

So let's create this class in a new package `com.vidasoft.magman.model.converter`:

```
@Converter(autoApply = true)
public class LocalDateTimeConverter implements AttributeConverter<LocalDateTime,
Timestamp> {

    @Override
    public Timestamp convertToDatabaseColumn(LocalDateTime localDateTime) {
        return localDateTime == null ? null : Timestamp.valueOf(localDateTime);
    }

    @Override
    public LocalDateTime convertToEntityAttribute(Timestamp timestamp) {
        return timestamp == null ? null : timestamp.toLocalDateTime();
    }
}
```

The `autoApply=true` attribute of the annotation tells JPA that this converter applies for all the `LocalDateTime` member variables in the project. If you skip that or set it to `false`, you need to annotate the fields, where you want to apply this converter, with `@Convert(converter = LocalDateTimeConverter.class)`.

Create a similar converter for all the `java.time.LocalDate` member variables in the project. They should be converted to `java.sql.Date`.

Accessing the data

Modelling the data and mapping it to the relational database is just one aspect of the Java Persistence API. The other one is about working with that data - querying and updating the database. That happens via the `EntityManager` interface. In a managed environment (like application server), where the persistence unit type is JTA, you can get directly the `EntityManager`. But in our firs part of the course we as application developers are responsible for its creation and disposal, as well as for handling transactions.

Now we are going to build a few data access objects (DAOs). As we are going to follow the separation of concerns principle and the dependency injection practices, we are not going to create the `EntityManager` here. Our classes will simply receive it as a parameter to their constructor.

As we have three big domain areas in our application so far - the user hierarchy, the content (articles and comments) and the advertisers, we are going to create three packages under `com.vidasoft.magman` to put the business logic inside. We'll start with the `AuthorDAO`. Let's create this class in the `com.vidasoft.magman.user` package. Let's also add `EntityManager` member variable and a constructor that initializes it:

```
public class AuthorDAO {

    private EntityManager entityManager;

    public AuthorDAO(EntityManager entityManager) {
        this.entityManager = entityManager;
    }
}
```

Next we'll add a method that receives `Author` parameter and persists it in the database via the entity manager. This method will generally get an ID-less author - remember, IDs are automatically assigned by the persistence provider once an entity is persisted. So it is a good practice once persisted, to return the same object, countint on the fact that JPA will set the generated primary key to it after successful insert. Here is the code for storing a new author entity in the DB:

```
public Author addAuthor(Author author) {
    entityManager.persist(author);
    return author;
}
```

Go on and create `ArticleDAO` in `com.vidasoft.magman.content` package and implement the `addArticle()` method

Now that we learned how we can persist data, it is time to see how we can query it. There are two ways to do that in JPA:

- By using `entityManager.find()`. This is used if you want to find an entity by its ID
- By using JPQL query. This is used for more sophisticated searches

The first approach is pretty straightforward. Let's see it in action with the `findArticleById` method:

```
public class ArticleDAO {

    // ...

    public Optional<Article> findArticleById(Long id) {
        return Optional.ofNullable(entityManager.find(Article.class, id));
    }
}
```

The `entityManager.find()` method takes two parameters - the type that you are looking up and the primary key. As it may return null if record with such ID doesn't exist, it is a good practice to wrap the result in `java.util.Optional`.

In the case of retrieving all the articles, we'll need to build our first JPQL query:

```
public List<Article> getAllArticles() {
    TypedQuery<Article> articleQuery = entityManager.createQuery(
        "SELECT article FROM Article article ORDER BY article.publishDate",
        Article.class);
    return articleQuery.getResultList();
}
```

The JPQL language is very similar to SQL. The subtle difference is that it works with entities and not with tables. So for example, if we decide to query the `User` entity, which we mapped to `Users` table, we will refer to it as its entity name (`User`).

After setting up the query we should call either its `getResultList()` method if we expect 0 or more records returned from the DB or `getSingleResult()` if we expect exactly one instance.

If you call `getSingleResult()` and the returned records are 0 or more than 1, then the respective runtime exception will be thrown.

Next we'll see how we can parameterize a query, i.e. make use of its `WHERE` clause. Now we want to return all the articles that were written by a certain author. Our method is going to receive an `Author` parameter and is going to look for their articles by executing the proper JPQL statement.

Getting an author object doesn't always mean that you get an author entity. This is because the objects have different states when JPA is concerned. The below figure shows them as well as the methods which you call on `EntityManager` to go from one state to the next:

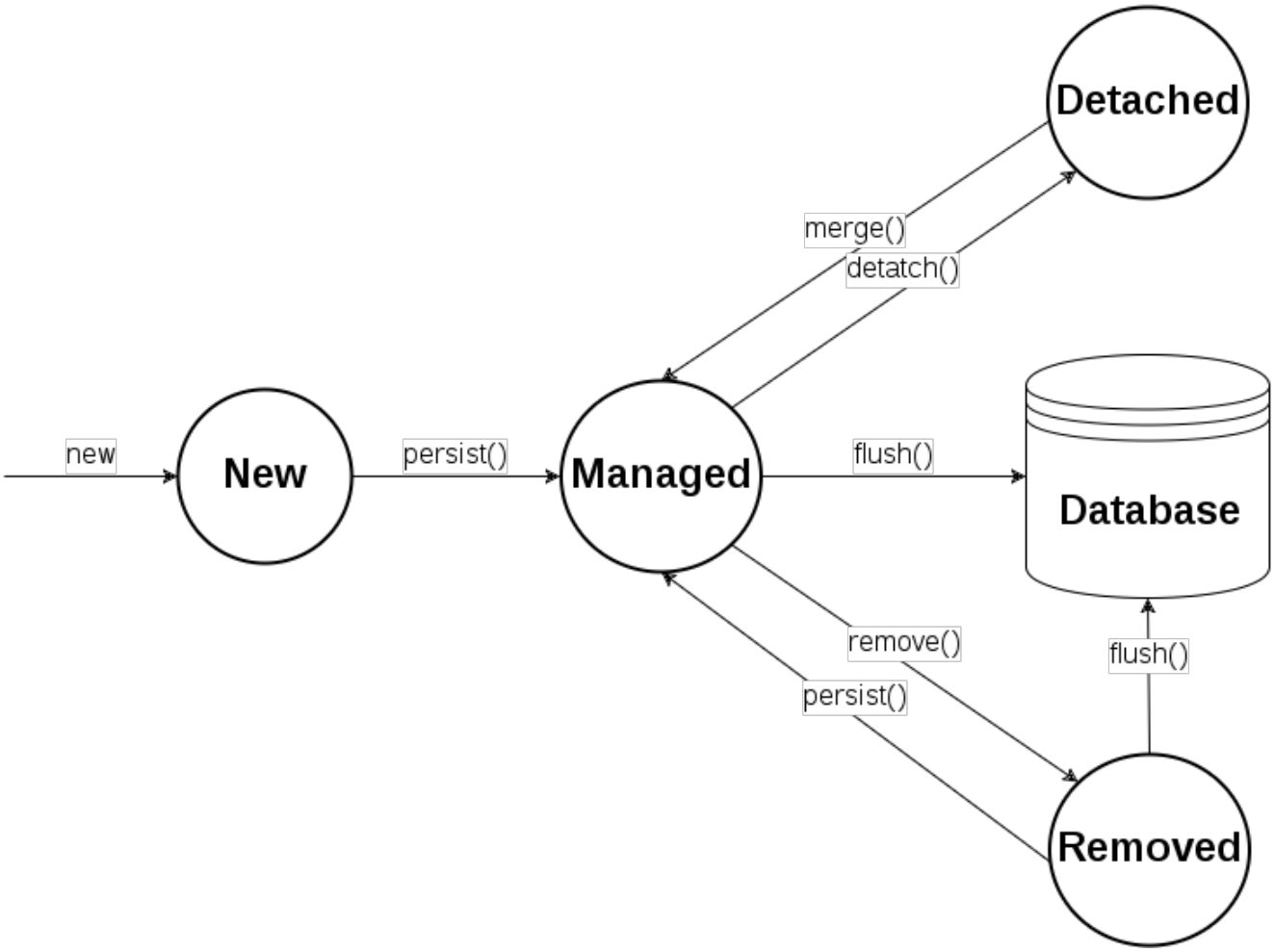

Figure 4. Entity state diagram

Getting back to the task at hand. We need to make sure that the author that we receive is in managed state. We do that by first merging it in the persistence context. After that we can use it as a parameter in our query:

```
public class ArticleDAO {

    private EntityManager entityManager;

    public ArticleDAO(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    // Other methods

    public List<Article> findArticlesByAuthor(Author author) {
        entityManager.merge(author);
        TypedQuery<Article> query = entityManager.createQuery(
            "SELECT article FROM Article article WHERE article.author = :author",
            Article.class);
        query.setParameter("author", author);
        return query.getResultList();
    }
}
```

You noticed how we passed the merged author as parameter - by specifying a placeholder starting with : in the JPQL and then providing the real value for that placeholder.

Add the final DAO method for this installment. Create a new class `com.vidasoft.magman.content.CommentDAO`. Add there a method `Comment addCommentToArticle(Long articleId, Comment newComment)`. The method should first lookup the article by its ID. If it exists, the new comment should be persisted and then added to the retrieved article.

Running the code

So far we've developed a lot of functionality without ever trying to run it and see whether it works. But finally we are good to write our main class that will exercise the data access object that we implemented. Let's create `com.vidasoft.magman.MagazineManager` and add a main method to it.

The first thing it should do is create the `EntityManagerFactory` (our persistence unit), which we will use to provide us with entity managers:

```
public class MagazineManager {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("magazine-
manager");
    }
}
```

You noticed how the parameter of the static factory method matches the persistence unit name

from the persistence.xml.

Now we are going to add a few authors. As this operation involves writing to the database, it should be executed inside a transaction. Our persistence unit type is `RESOURCE_LOCAL`, so it is us as application developers that have to take care of starting and completing transactions. Here is the code to do that:

```
        /* Add a couple of authors */
        EntityManager entityManager = emf.createEntityManager();
        AuthorDAO authorDAO = new AuthorDAO(entityManager);
        entityManager.getTransaction().begin();
        Author johnSmith = authorDAO.addAuthor(new Author("john", "smith", "John",
                "Smith", "john.smit@vida-soft.com", true, 2000));
        Author paulaChester = authorDAO.addAuthor(new Author("paula", "chester",
                "Paula", "Chester", "paula.chester@vida-soft.com", false, 750));
        entityManager.getTransaction().commit();
        entityManager.close();
```

You noticed that at the end, after the transaction is committed, we closed the entity manager. This is not really necessary in a resource local persistence unit, where the same entity manager can run multiple transaction. But we do it in order to simulate the way you work with the persistence context in a managed environment, where you get a fresh one on each request. The caveat here is that you have to do `emf.createEntityManager()` again before the next time you use it.

Add three articles - two by the first user and the third one by the second one. Assign comments to one of the articles.

Don't forget the order:

1. Create entity manager
2. Begin transaction
3. Do the work
4. Commit the transaction
5. Close the entity manager

The test data is in the database now. It is time to query it. Let's start by finding all the articles and printing them in the console:

```
        /* List all articles */
        entityManager = emf.createEntityManager();
        articleDAO = new ArticleDAO(entityManager);
        System.out.println("All articles:");
        articleDAO.getAllArticles().forEach(System.out::println);
        entityManager.close();
```

Your turn now.

1. Find the article to which you added to comments by its ID. Print it in the console, then get its comments and print them as well
2. Display all the articles from the author that has two of them.

Performance improvements

So far we've been modelling our domain and running our queries in a quite idiomatic and even naive way. We've seen how JPA can be our friend. But is this optimal from performance point of view? Let's check the following few concerns.

Optimistic locking

A lot of times more than one concurrent transaction attempts to update a certain database table. In order to avoid collisions, a lock can be acquired by one of the transactions. This way nobody else is allowed to modify the table.

The problem with this approach is that in most of the cases the concurrent transactions work on different data sets (even though they are in one and the same table). So holding a lock seems like overkill.

Here comes optimistic locking. JPA supports it since version 2.0 (Java EE 6). You just add an attribute of type `int`, `short`, `long` or `java.sql.Timestamp` to your class and annotate it with `@Version`. Then each time when the persistence provider updates a certain entity (i.e. record in the database), it will compare the version to the one that was read in the beginning. If it was changed by some other transaction, the update operation will result in exception. Otherwise the entity will be updated and the version attribute will be incremented.

The advantage of this approach is obvious - updating an entity does not need to necessarily involve locking. The disadvantage is that if a collision occurs, it is the job of the application developer to repeat the update.

So let's see how we enable optimistic locking for the classes in the user hierarchy. We simply need to add the `version` member variable to the `User` class and annotate it:

```
@Entity
@Table(name = "Users")
public abstract class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Version
    private int version;

    // ...

    public int getVersion() {
        return version;
    }

    public void setVersion(int version) {
        this.version = version;
    }

    // ...
}
```

Enable optimistic locking for the rest of the entities. You may skip `Manager`, `Author` and `Subscriber`, because they all inherit from `User`.

Number of executed queries

There was a reason why we added the Hibernate specific properties to the properties.xml. Thus after we run the program we can inspect its output and see how many SQL statements were issued.

So what can we notice.

The `getAllArticles()` method does not fetch automatically the authors of the articles. That is why when we try to print an article's author, i.e. call `article.getAuthor().toString()`, the persistence provider issues a new select statement to the database to fetch the corresponding author. You might wonder why, as Many-to-one and One-to-one relations are said to be eagerly loaded (as opposed to One-to-many and Many-to-many). Well, the guarantee for eager loading applies only if you query entities with `EntityManager::find`. When it comes to executing JPQL, the entity manager will not fetch automatically the depending entity.

In order to overcome that and have our author loaded, we need to tweak a little bit our query:

```
public List<Article> getAllArticles() {
    TypedQuery<Article> articleQuery = entityManager.createQuery(
        "SELECT article FROM Article article LEFT JOIN FETCH article.author author
ORDER BY article.publishDate",
        Article.class);
    return articleQuery.getResultList();
}
```

Run again the main class and look at the output. You will notice that now there is only one query.

We have the same lazy loading issue with getting our article by ID and trying to print its content along with the comments inside. Because One-to-many relationships are always loaded lazily, if we want to get an article (even with `EntityManager::find`) and then call `article.getComment(0).toString()`, we end up having a query to the database for each comment. How can we avoid this problem, known in the industry as the *N+1 query problem*?

A naive solution would be to declare our `@OneToMany` relationship to be fetched eagerly:

```
public class Article {

    // ...

    @OneToMany(fetch = FetchType.EAGER)
    private List<Comment> comments = new ArrayList<>();

    // ...
}
```

This will surely help us here, but is not a good approach. Because each time we will bloat our memory with the comment entities, even when we don't want them. For example when we get the list of all articles, we are mostly interested in their titles and not in their comments.

So there is a reason why it was define that the default fetch type for One-to-many relations is `LAZY`. So what can we do here? Well, right after retrieving the owning entity in the `getArticleById()` method, we can call the `size()` method on its comments attribute:

```
    public Optional<Article> findArticleById(Long id) {
        Optional<Article> article = Optional.ofNullable(entityManager.find(Article
.class, id));
        article.ifPresent(a -> a.getComments().size());
        return article;
    }
```

This will make the persistence provider fetch the whole collection from the database in one shot. Which makes 2 queries in total. Not the most elegant solution, but at least it solved the N+1 query problem.

Named queries

So far we've defined our JPQL queries on the use site. However, JPA provides another approach which has many advantages. You can place the JPQL code on any of the entities and give it a name. This turns your simple query into named one.

Let's see how this will work with the `getAllArticles()` query. We'll first place it on top of the `Article` entity class.

```
@Entity
@NamedQueries({
        @NamedQuery(name = "getAllArticles",
                query = "SELECT article FROM Article article LEFT JOIN FETCH
article.author author ORDER BY article.publishDate")
})
public class Article {
    // ...
}
```

Then in the `ArticleDAO` we'll replace its declaration with its name. The other thing that we need to change in the DAO, is the entity manager method that we call. When dealing with named queries it is `createNamedQuery()` and not `createQuery()`:

```
    public List<Article> getAllArticles() {
        TypedQuery<Article> articleQuery = entityManager.createNamedQuery(
"getAllArticles", Article.class);
        return articleQuery.getResultList();
    }
```

One of the advantages of this approach is that JPQL syntax of all our named queries is checked by the container upon application startup. While in the normal query case, this happens at runtime. Another advantage is that the query is loaded and compiled only once.

Convert the other JPQL query into named query

Frontend with Servlets

In the previous installment of this course we introduced the Java Persistence API. We showed you how you can use the JPA annotations and the Entity Manager to bridge the mismatch between the object-oriented data model of the Java programs and the model of relational databases. However, our application was mostly Java SE: it was not deployed on any application server and the output went into the console rather than in the web browser.

Today we are going to take the first step towards making our application hit the web. We will walk you through one of the three possible ways to write web apps in Java EE 7 - using the *Java Servlet* technology.

The servlets are fundamental for almost all modern Java application stacks. Not just Java EE, but also for example Spring MVC. Even in Java EE you would ultimately use more high level technologies for your frontend like JavaServer Faces or JAX-RS plus any of the JavaScript libraries out there. However, as they all use Java Servlets for the low level tasks, it is really important to understand how the latter work.

Initial state of the project

The features that we are going to add today (and this applies for each next installment) will build on top of those that we did the last time. In order to do a smoother transition and so that you focus on the tasks at hand, we did some small adjustments to the project. Go on and checkout the `servletbegin` branch of our showcase. Before that you may need to fetch the latest version from the repository and commit or stash locally your work on JPA.

The `pom.xml` file last time contained compile time dependency to Hibernate Core for the JPA API and implementation as well as runtime dependency to Derby in-memory database. This time the dependency list is much more simplified:

```
    <dependencies>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
```

One of the greatest features of Java EE is that the application server comes with implementations of all the specifications that it supports. So your deliverable, i.e. your WAR file, should just contain your code and you don't need to bother and find which are the technologies you use, what are all the artifacts that implement them, etc. The only thing you need for compiling your code are the Java EE APIs. But as they come on the server as well, you just need them in the *provided* dependency scope.

As our application will be running on the application server (WildFly 10 in our case), we are also

taking advantage of that. One of the things that is guaranteed by the JPA spec when it comes to managed environments is that each application server should create a default data source. And if the application developer did not specify any other means to connecting to the database, the application's persistence unit will be bound to that data source.

For the time being we will not configure our application to connect to a real persistence store (like MySQL or PostgreSQL). We are rather going to rely on the H2 in-memory DB that is provided as default data source in WildFly. So now our persistence.xml looks much shorter:

```
    <persistence-unit name="magazine-manager" transaction-type="RESOURCE_LOCAL">
        <properties>
            <!-- JDBC properties -->
            <property name="javax.persistence.schema-generation.database.action"
value="create"/>
            <!-- Hibernate properties -->
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
        </properties>
    </persistence-unit>
```

Notice that the transaction type is still `RESOURCE_LOCAL`. This means that regardless that we are running inside a managed environment, we will still be responsible for managing the transactions.

The `MagazineManager.java` main class is no longer needed, as we are going to display the results of our queries in the browser. However, we still need to import the initial data set with authors and articles in the database. That is why we moved just that part of our old main method into a new class - `com.vidasoft.magman.admin.DataLoader`.

The goal of this installment is to showcase the Java Servlet technology. It can't be done without proper web pages, though. The technology that we will be using for rendering the HTML views is JavaServer Pages.

You can (ab)use JSP in quite a lot of ways, including writing all your business logic in Java inside scriptlet tags. However, we've decided to use it as template technology.

This course does not include JSP tutorial. That is why we will not touch the pages, but will use them as they are.

Here is a short overview of the content of the webapp directory:

- The `css` and `js` directories contain the free SB Admin 2 bootstrap theme, which we are using for styling our pages.
- The `img` folder contains images for the web app
- The `jsp` directory holds the three pages which we are going to show today: for displaying the articles, the login screen and the one used for submitting new articles.
- The `index.html` in the root simply redirects to `jsp/viewArticles.jsp`

Where's the web.xml?

The developers with experience in previous versions of Java EE (and J2EE) might ask where is the `web.xml` file in this project. To remind you, this is the file that described all the web resources in an enterprise application - servlets, web filters, constraints, security roles, etc.

As of Java EE 6 the presence of this file in the `WEB-INF` folder of the WAR is no longer mandatory. Most of the stuff that you used to configure there is now configurable with annotations in the Java code.

There are still some things that can be only configured in `web.xml`. These are customizations that describe the application as a whole, such as the HTTP session timeout, the error page returned upon a certain HTTP code, custom welcome files, etc. Also filter chains, i.e. which filter is invoked after which, can only be configured in `web.xml`.

If you need to specify any of these or override other configurations, the location of `web.xml` in a Maven project is under the `src/main/webapp/WEB-INF` folder.

Show the list of articles

The first task that we are going to implement is showing the list of all the articles in the database. If you quickly take a look at the index.html, you will notice that it forwards automatically to the `ArticleServlet` URI. Let's try and run our application. We'll get the default 404 page of the WildFly server:

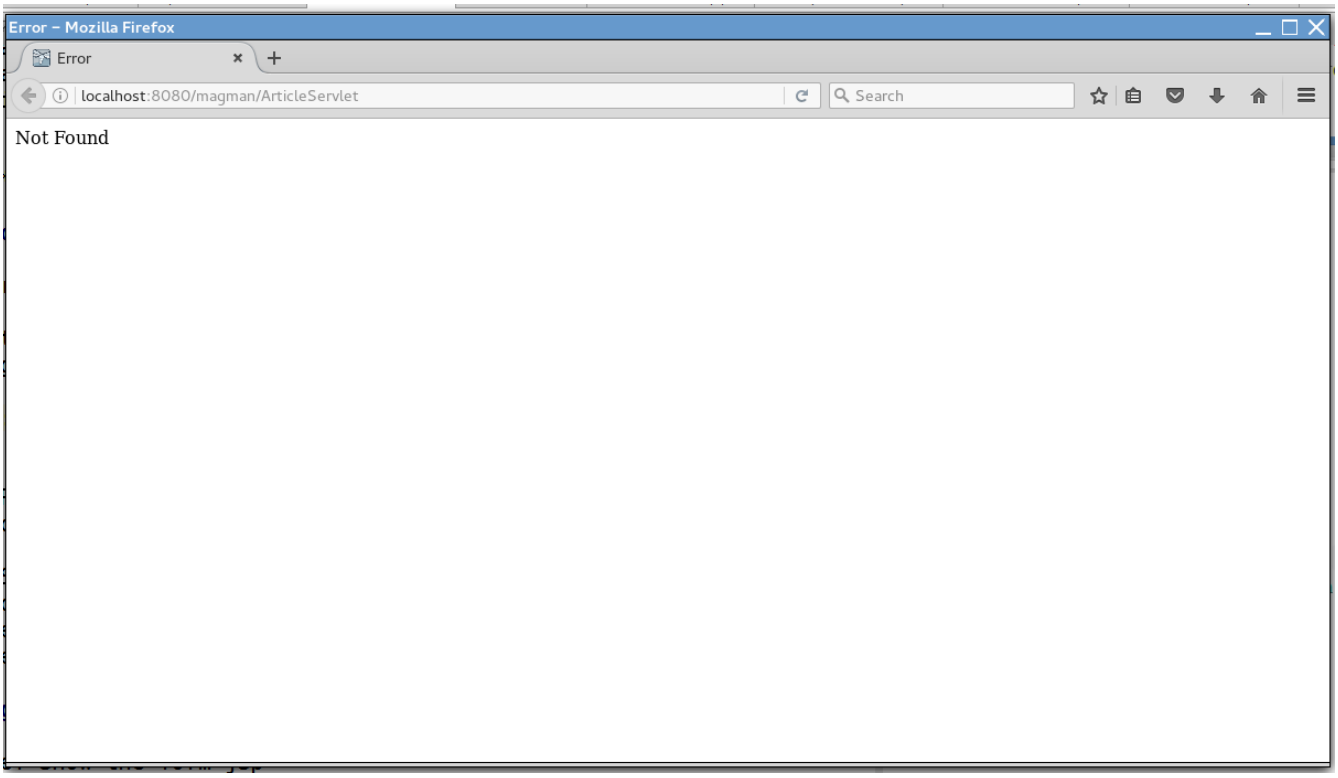

Figure 5. The page was not implemented

What we need to do is write a component that will be responsible for handling requests to the `ArticleServlet` URI. As its name implies, we need to implement an HTTP servlet. This means adding

a new class, that extends the `javax.servlet.http.HttpServlet` abstract class and annotate it with the `@WebServlet` annotation. The annotation has one very important attribute: `urlPatterns`. There you may specify all the URIs, requests to which will be handled by this Servlet. In our case this is just `ArticleServlet`. As the spec suggests we need to prefix the URL pattern with a `/`. So let's add this class to the `com.vidasoft.magman.content` package:

```
@WebServlet(urlPatterns = "/ArticleServlet")
public class ArticleServlet extends HttpServlet {

}
```

After the web container identifies which servlet will handle the request, it will call one of its methods, based on the request method. For example for GET requests, the method is `doGet()`, while for POST requests - `doPost()`. These methods come from the parent class of our servlet and receive as parameters objects wrapping the HTTP request and response.

In our case we want to implement `doGet()`. Inside we want to instantiate the `ArticleDAO`, call its `getAllArticles()` method and somehow transfer the returned result to the browser.

Instantiating the DAO class involves obtaining an instance of `EntityManager` and passing it to the constructor. In the previous installment we used the `Persistence` class to create `EntityManagerFactory` and then used that to create `EntityManager`. This time we'll let the application server do the initialization for us and will directly inject the `EntityManagerFactory` for our persistence unit.

We can actually inject both `EntityManagerFactory` and `EntityManager` in a Servlet. However, as one instance of a Servlet class can be accessed by multiple threads, it is a better idea to inject the factory, as the entity manager is not thread safe.

This injection can easily be done with the help of the `@PersistenceUnit` annotation like so:

```
@WebServlet(urlPatterns = "/ArticleServlet")
public class ArticleServlet extends HttpServlet {

    @PersistenceUnit(unitName = "magazine-manager")
    private EntityManagerFactory emf;

    @Override
    protected void doGet(HttpServletRequest req,  HttpServletResponse resp)
                    throws ServletException, IOException {
    }
}
```

Now let's grab the available articles the way we did it in the `MagazineManager.java` main class the last time:

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException {
        EntityManager entityManager = emf.createEntityManager();
        ArticleDAO articleDAO = new ArticleDAO(entityManager);
        List<Article> articles = articleDAO.getAllArticles();
        entityManager.close();
    }
```

Now that we got all the articles, it is time to render the HTML that will display them in the browser. In the first version of the Servlet specification, developers used the `HttpServletResponse` to get hold of a `PrintWriter` object. Where they "printed" manually the HTML. Something like that:

```
resp.getWriter().print("<html><head><title>Web Page</title>...");
```

However, this is quite inconvenient, especially when the web pages are implemented by the design team that usually doesn't contain Java developers.

That is why we will forward processing of the request to a JSP file by using the `RequestDispatcher`. The JSP that we'll use here is located in the `webapp/jsp` folder and as you might guess, it is called `viewArticles.jsp`. It displays an HTML table, where it iterates through all the objects that it receives in the `articles` variable. For each such article it displays in a separate column the title and the author names.

Which means that before we forward the request to the JSP, we need to make sure that we attach there the `articles` list that we obtained from `ArticleDAO`. After we do that, we will get the `RequestDispatcher` to the view file and forward to it:

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                throws ServletException, IOException {
        EntityManager entityManager = emf.createEntityManager();
        ArticleDAO articleDAO = new ArticleDAO(entityManager);
        List<Article> articles = articleDAO.getAllArticles();
        entityManager.close();
        req.setAttribute("articles", articles);
        req.getRequestDispatcher("jsp/viewArticles.jsp")
           .forward(req, resp);
    }
```

Now we should be able to get the list of articles page when we load our web app:

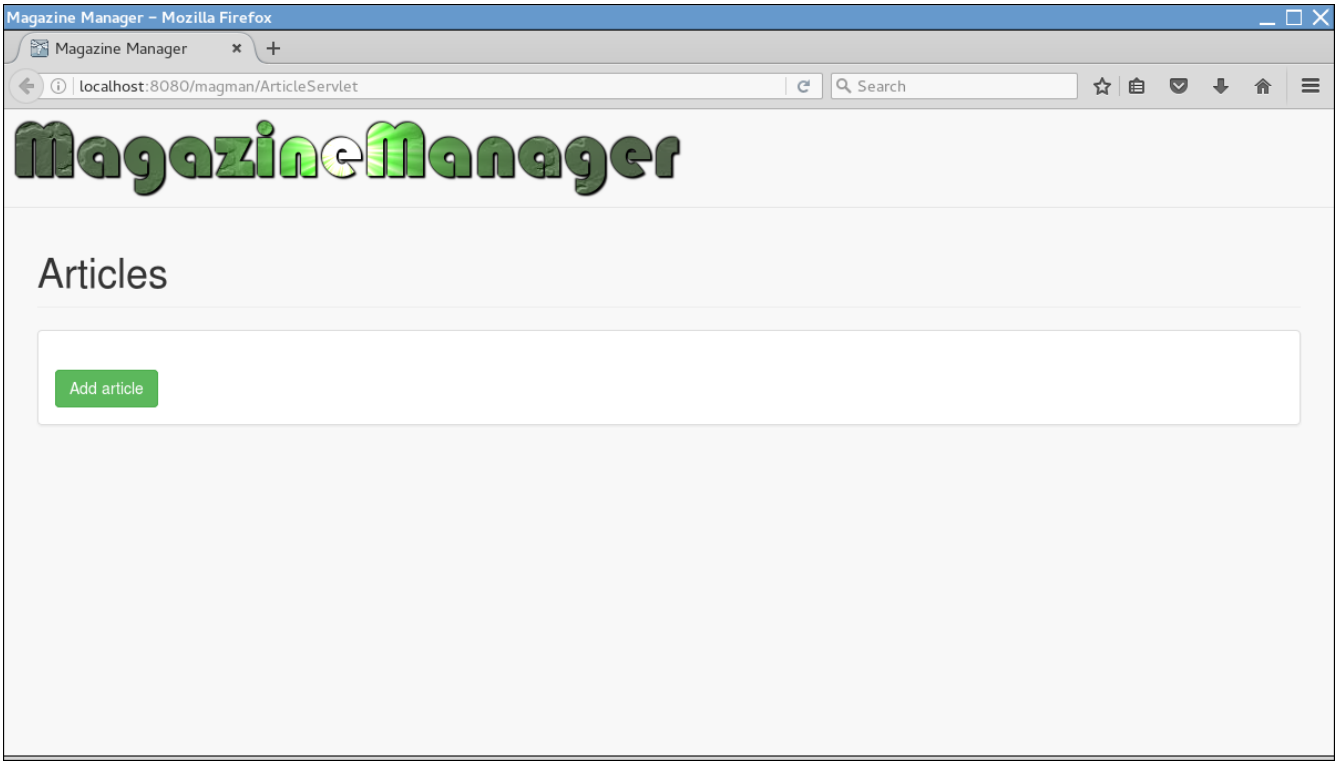

Figure 6. Empty list of articles

But wait, where are the articles?

Remember the `DataLoader` class where the initialization code from the last installment was move to? Well, nobody called its `load()` method to do the import. We can easily introduce a call to it in the `doGet()` method of our `ArticleServlet`. But this means that a bunch of test data will be inserted in the DB each time someone accesses our web page.

That is why we will plug in another point of the Servlet's lifecycle - the `init()` method. We will override that as well and there we will instantiate the `DataLoader` class and call its `load()` method:

```
@Override
public void init() throws ServletException {
    DataLoader dataLoader = new DataLoader(emf);
    dataLoader.load();
}
```

Now upon the first request to our Servlet, the container will instantiate it, inject all the dependencies and call its `init()` method. After that `doGet()` will find the test data and we should be able to see on the article page all the articles that we inserted in the database:

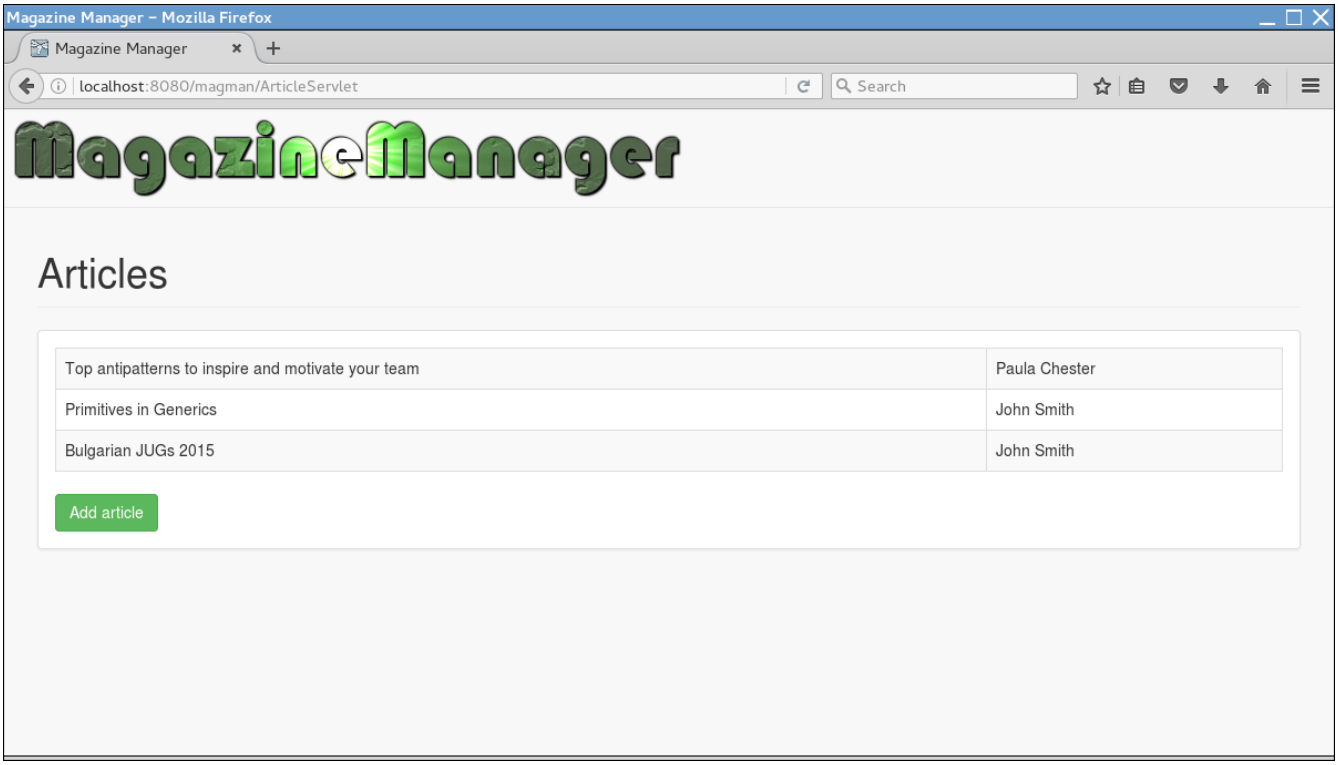

Figure 7. The list with all articles

Allow access to logged users only

Showing the login screen

The next feature that we are going to develop for the magazine manager is introduce user login. We want to allow only logged in users to access the content of our website (and the list of articles in particular).

First, let's see how we can show the login screen. Let's create `LoginServlet` in the `com.vidasoft.magman.user` package. And let's make sure that its `doGet()` method simply forwards to the `jsp/login.jsp` file.

Implement `com.vidasoft.magman.user.LoginServlet` and make it forward to `jsp/login.jsp` when the `LoginServlet` URI is requested in the browser address bar.

If you do that correctly, when you access http://localhost:8080/magman/LoginServlet from your browser, you should be able to see this login screen:

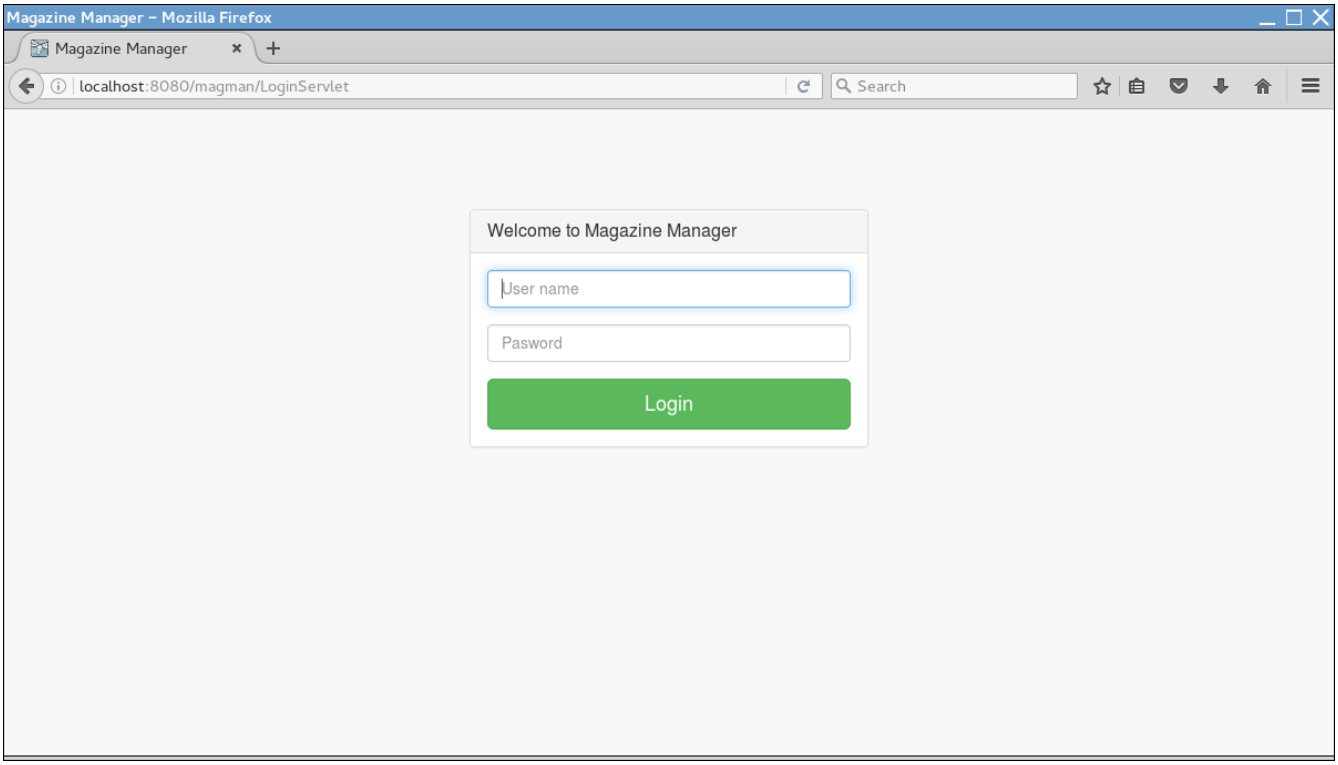

Figure 8. The login screen

Checking for user correctness

Taking a closer look at the login.jsp, you will find that it contains an HTML form. Its `action` and `method` parameters indicate that upon submitting, the form will send a POST request to the `LoginServlet` that we just created. Looking further down at the `name` attributes of form's input fields, we can deduce that the body of the request will contain two parameters: `userName` and `password`.

This quick analysis suggests that the next step should be implementing a `doPost()` method in the `LoginServlet`. It should extract the values entered by the user in the form and delegate to the backend finding whether they match an existing user with the specified password. The user input extraction can easily be done by calling the `getParameter()` method of the `HttpServerRequest` parameter of the `doPost()` method:

```
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                    throws ServletException, IOException {
        String userName = req.getParameter("userName");
        String password = req.getParameter("password");
    }
```

In order to check whether the entered parameters match, we need the respective functionality in the backend. For that, we need to implement a new DAO, let's call it `UserDAO`, that creates a query for a `User` entity with a specified `userName` and `password`.

Find a user by user name and password.

1. Create a new class `com.vidasoft.magman.user.UserDAO` resembling the other DAO classes that we created so far.
2. Add a `findUserByUserNameAndPassword` method that takes `userName` and `password` parameters. Consider returning `Optional<User>` from the method.
3. Create a query for a User and pass the user name and password as parameters to the query.
4. Look for the most appropriate method of the `TypedQuery` class to return a single result and look at it javadoc what happens if no result is found.
5. Return appropriate results from the DAO method on both occasions.

Now that we have the backend implementation, we can go back to `LoginServlet`. Let's look up the user and password entered in the login screen:

```
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String userName = req.getParameter("userName");
        String password = req.getParameter("password");

        EntityManager entityManager = emf.createEntityManager();
        UserDAO userDAO = new UserDAO(entityManager);
        Optional<User> user = userDAO.findUserByUserNameAndPassword(userName,
password);
        entityManager.close();
    }
```

Redirecting to appropriate view

What are we going to do with the `user` object?

If it is not empty, i.e. we found a user with the entered user name and password, we need to redirect to the page that shows the list of articles. Otherwise, we need to show again the login screen, but this time with an error message, suggesting that the entered data is incorrect. We also want to make sure that upon successful login we will make the user available to subsequent requests. In other words, we will associate that user to the currently running session.

We'll start with the second task - putting the logged user in the session. If you want to make an object survive between HTTP requests, you have to put it in a special HTTP session object, which you can obtain from the request instance. In our case, this is what we do in the `doPost()` method:

```
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // Obtain the user from the database

        if (user.isPresent()) {
            req.getSession().setAttribute("loggedUser", user.get());
        }
    }
```

We obtain the `HttpSession` object from the `HttpServletRequest` and set an attribute with a certain name. Now, if we want to get the logged user in a subsequent request, we just need to obtain the session in the same fashion and call its `getAttribute()` method.

How do the sessions work?

When a browser session starts, in most of the cases the container sends a special cookie to the browser. Its standard name is `JSESSIONID`. The server generates an ID that is sent as part of the cookie, which ID is uniquely associated with the current client.

Then the client will send this cookie upon each subsequent request. Thus the container will unambiguously associate the request with a session.

The session ends whenever a certain timeout expires (configurable in `web.xml`) or when the `HttpSesson::invalidate()` method is called.

Finally, we should tell the web container to which view to redirect us. So far we've forwarded the request to a JSP using the request dispatcher. But what was essential, was that that forward to the JSP was basically completing **the same** HTTP request.

Now the request completes fully in the POST method. So we want to start a completely new one. For that we'll use `HttpServletResponse::sendRedirect()` method. We'll pass it the URI of the respective servlet: in case of successful login it is `ArticleServlet`, otherwise it should again be `LoginServlet`. In addition to that, we want to tell the login Servlet that this time it was a failed login attempt. So that it makes sure that a proper message is displayed to the user. This last concern is easily implemented by adding a parameter to the redirect URI.

Here is one way to implement it:

```
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // Obtain the user from the database

        if (user.isPresent()) {
            req.getSession().setAttribute("loggedUser", user.get());
            resp.sendRedirect("ArticleServlet");
        } else {
            resp.sendRedirect("LoginServlet?failed");
        }
    }
```

When in doubt when do you forward and when do you redirect, remember that in most of the cases `doGet()` does `request.getRequestDispatcher("someJsp").forward()`, while `doPost()` does `response.sendRedirect("someServlet")`.

The final thing we need to do when it comes to our login screen is adding support for the `failed` parameter in the `LoginServlet::doGet()` method. Remember, this was part of the redirect URI if the `user` object was empty.

In the `doGet()` method we can easily extract this URL parameter:

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String failedParam = req.getParameter("failed");
        req.getRequestDispatcher("jsp/login.jsp").forward(req, resp);
    }
```

If it exists, it will not be `null`. Now, let's go back to our `login.jsp`. If you look closely to its code, you will notice this code snippet:

```
<form method="post" action="LoginServlet" role="form">
    <fieldset>
        <c:if test="${not empty message}">
            <div class="form-group alert alert-danger">
                <c:out value="${message}"/>
            </div>
        </c:if>
        <!-- input fields -->
    </fieldset>
</form>
```

So, if we put `message` attribute in the request, the login screen will render its value. Let's do that:

```
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String failedParam = req.getParameter("failed");
        if (failedParam != null) {
            req.setAttribute("message", "Login failed");
        }
        req.getRequestDispatcher("jsp/login.jsp").forward(req, resp);
    }
```

Now if we enter a wrong user password combination, we will get bac the login screen, but this time with our nice little message:

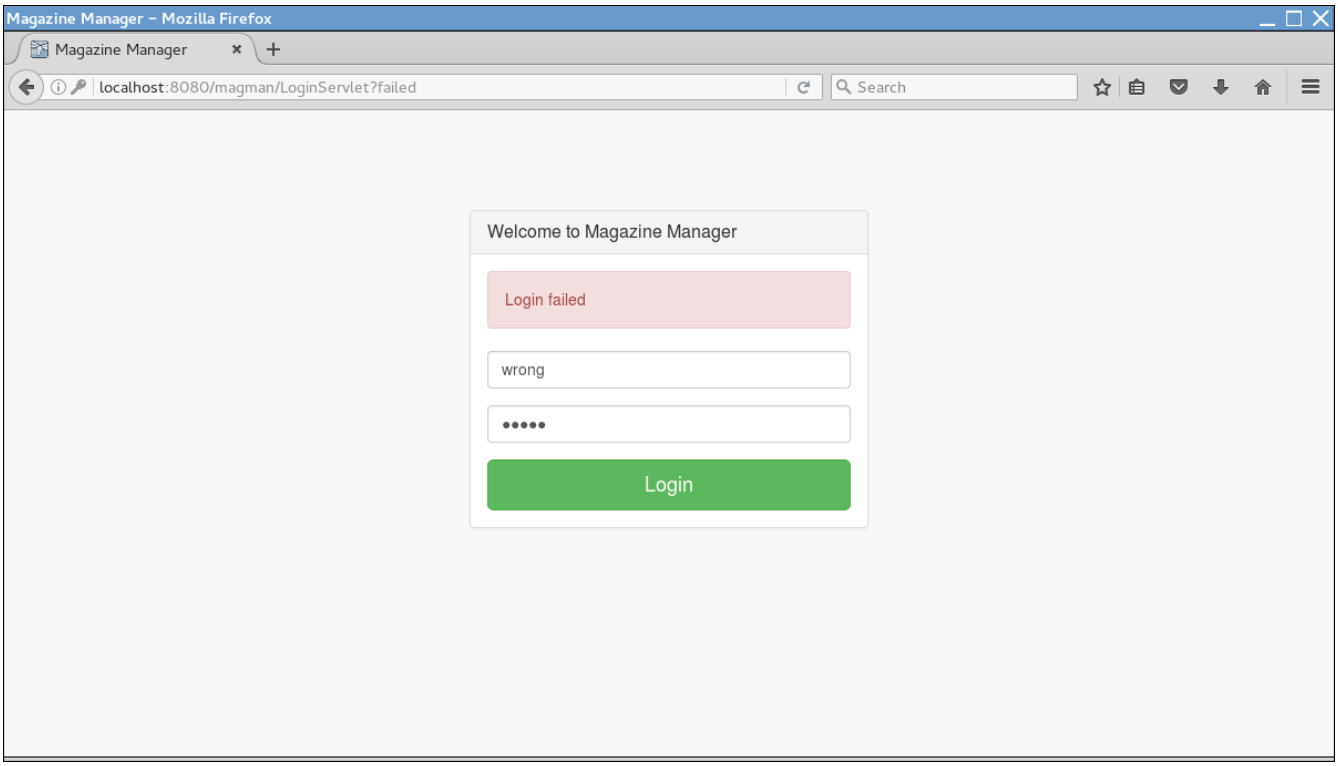

Figure 9. Login failed screen

Intercepting unauthorized requests

It might seem awkward that `index.html` initially takes you straight to the list of articles and you then manually enter the `LoginServlet` URI in the browser address bar. Usually it is the other way round: if you enter address that you are not allowed to access unless you are logged in, you are first forwarded to the login page and then if you log on successfully, the browser brings you back to the required resource.

You can implement the same behavior in a Java EE web application using web filters. What you need to do is to implement the `javax.servlet.Filter` interface and put the `@WebFilter` annotation on top of your implementation. The annotation has the same `urlPatterns` attribute as `@WebServlet`, where you specify the list of the URIs that will be intercepted by this filter.

Let's implement a filter in our app. It should follow the given rules:

- If the user is logged in, the request should pass through no matter which the requested resource is
- If the request is about a resource in the `css`, `img` or `js` directories, the request should pass through
- If the request is for `LoginServlet` it should pass through
- If the request is for all the rest of the application resources and the user was not logged in, the flow should be forwarded to the login screen

Let's start by adding an empty implementation of `javax.servlet.Filter` in the `com.vidasoft.magman.security` package:

```
@WebFilter(urlPatterns = "/*")
public class LoggedUserFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {

    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {

    }

    @Override
    public void destroy() {

    }
}
```

It will be called for all the requested resources. We can narrow down its scope by putting a different URI as a value to the `urlPatterns` attribute. For example, if we use the `/admin` prefix for all the resources available to the manager role, we can add a Servlet filter that takes care about the `/admin/*` URL pattern.

We'll leave the `init()` and `destroy()` methods empty and we'll put our implementation in the `doFilter()` method. Unlike the Servlets class hierarchy, where we have the `HttpServlet` base class adding some HTTP touches to the root `Servlet` interface, the Servlet filter hierarchy lacks that specialization. That is why we need to first cast the `request` parameter from `ServletRequest` to `HttpServletRequest`. Now we are able to get the request URI as well as we can inspect the session:

```
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        String reqURI = httpRequest.getRequestURI();
        Object loggedUser = httpRequest.getSession().getAttribute("loggedUser");
    }
```

Passing through a request means simply calling `doFilter()` on the `FilterChain` object passed as parameter to the current method. If we don't want to pass through a request, then we need to do a redirect to a certain URI (the `LoginServlet` in our case). This redirect happens in the same way as in Servlets' `doPost()` methods. We just need to first cast the `response` parameter from `ServletResponse` to `HttpServletResponse` and then call its `sendRedirect()` method with the desired URI:

```
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;

        String reqURI = httpRequest.getRequestURI();
        Object loggedUser = httpRequest.getSession().getAttribute("loggedUser");

        if (loggedUser != null || reqURI.contains("/css/") ||
            reqURI.contains("/js/") || reqURI.contains("/img/") ||
            reqURI.contains("LoginServlet")) {
            chain.doFilter(request, response);
        } else {
            ((HttpServletResponse) response).sendRedirect("LoginServlet");
        }
    }
```

We are almost done. With the current implementation in almost all the cases the `LoginServlet` will be the first one that is loaded in our application. While `ArticleServlet` will only be loaded if the login succeeds.

However, remember that the initial loading of the database with test users and articles happens in the `ArticleServlet::init` method. This means that if we leave it like that, the database will be empty when our `LoginServlet` gets loaded. And the login will never succeed.

So, in order to avoid that, let's move `ArticleServlet::init` to `LoginServlet`:

```
@WebServlet(urlPatterns = "/LoginServlet")
public class LoginServlet extends HttpServlet {

    @PersistenceUnit(unitName = "magazine-manager")
    private EntityManagerFactory emf;

    @Override
    public void init() throws ServletException {
        DataLoader dataLoader = new DataLoader(emf);
        dataLoader.load();
    }

    // doGet() and // doPost() methods
}
```

Don't forget to remove the `init()` method from the ArticleServlet. Otherwise you will get duplicate entries in the database.

We are good to go now.

Add a new article

To summarize everything that we did today, you will implement a completely new feature. Maybe you noticed the Add article button in the bottom left corner of the all articles view. If you try to click it, it wil lead you to a non-existing resource - `NewArticleServlet`.

So your final assignment for this part of the course will be to implement this missing peace of functionality.

Implement `com.vidasoft.magman.content.NewArticleServlet`.

Upon receiving GET requests it should forward to `jsp/newArticle.jsp`.

When it receives a POST request, it should read the title and content from the body. Then it should create a new Article object using them.

It should use the currently logged user as author and the current date as publish date of the article. The current date can be obtained by calling `LocalDate.now()`.

Finally it should redirect to the all articles view.

Remember that you need to start transaction and commit it around operations that perform changes to the database.