Integrating Spring MVC and JPA

Student Workbook

Chapter 1 - Spring MVC/JPA Integration

Objectives:

- Export content from your JPA project into your Spring MVC project.
- Create a DAO that uses JPA for persistence.

JPA Refresher

- The Java Persistence API (JPA) is an Object/Relational Mapping (ORM) standard for integrating database data into Java objects.
    - JPA is a specification which is implemented by providers such as Hibernate or Eclipse Link.
- You create annotated POJO classes for each database table you wish to model in Java.
    - At a minimum add the @Entity annotation to the class itself and the @Id annotation to the field that matches the corresponding primary key column.
    - Optionally use the @Table and @Column annotations to resolve any discrepancies between your names and those used in the table.
    - Add additional annotations such as @OneToMany and @JoinColumn to model relationships.
- Build an XML configuration file to identify connection details such as URL, username, and password.
    - This file must be named *persistence.xml* and saved in the *META-INF* directory.
- Use an instance of an EntityManager to find, persist, and remove instances of your Entity.

entities/Job.java

```
…
@Entity
@Table(name = "JOBS")
public class Job {
     // fields
     @Id
     @GeneratedValue(strategy = GenerationType.IDENTITY)
     private int id;
     private String name;
     @Column(name = "minimum_salary")
     private Long minimumSalary;
     @Column(name = "maximum_salary")
     private Long maximumSalary;

     @OneToMany(mappedBy = "job")
     private Collection<Employee> employees;
     …
}
```

client/Demo.java

```
…
public class Demo {
     public static void main(String[] args) {
          EntityManagerFactory emf =
           Persistence.createEntityManagerFactory("CompanyDB");
          EntityManager em = emf.createEntityManager();

          Job job = em.find(Job.class, 1);
          System.out.println(job.getName());
          …
}
```

META-INF/persistence.xml

```
…
<persistence …>
     <persistence-unit name="CompanyDB">
          <provider>org.hibernate.ejb.HibernatePersistence
          </provider>
          <class>entities.Job</class>
          <class>entities.Department</class>
          …
</persistence>
```

Incorporating JPA into your Dynamic Web Project

- In order to use your JPA entities within your Eclipse dynamic web project, you can either copy your entities into the new project or configure Eclipse to point to the existing project.

- To put your JPA code in the same project as your web application code:

    1. Copy your entities into your dynamic web project's source folder.

    2. Copy your persistence.xml file into a newly created *WebContent/WEB-INF/classes/META-INF* folder.

- To point your dynamic web project at an existing JPA project:

    1. Add the project to the Java Build Path.

    2. Add the project as a JAR file to the Deployment Assembly so that it is included in the *WEB-INF/libs* folder when you export a WAR file.

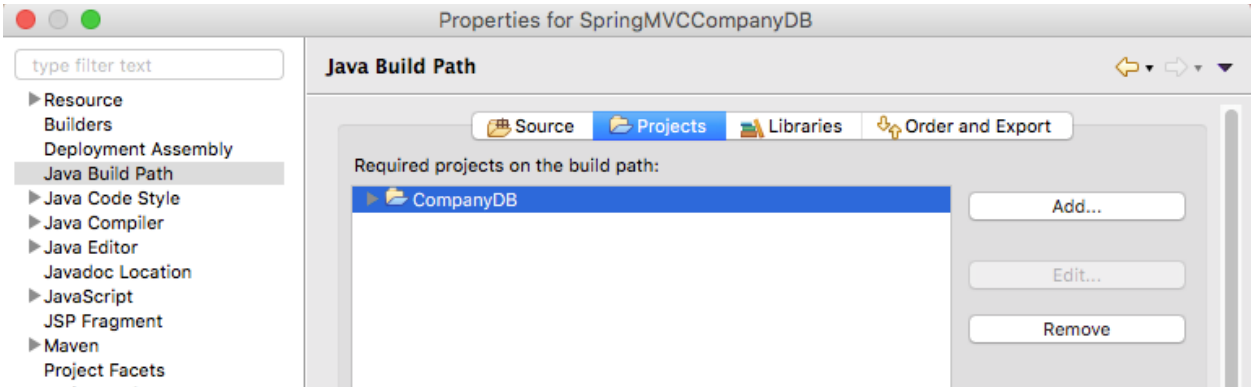
- Whichever option you choose, you will also need to add additional dependencies to your *pom.xml* file to ask Maven to download the appropriate jar files for Hibernate, Spring's ORM support, and MySQL:

```
<dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>5.1.0.Final</version>
</dependency>
<dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-orm</artifactId>
      <version>4.2.5.RELEASE</version>
</dependency>
<dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.38</version>
</dependency>
```

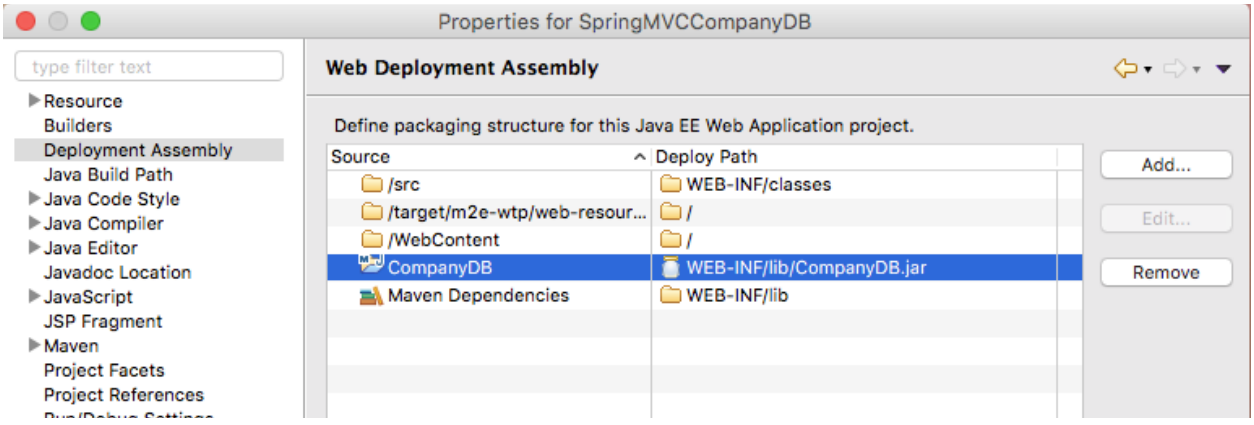
To point your dynamic web project at your JPA project:

1. Go to Project | Properties in Eclipse.

2. Choose Java Build Path in the left hand selection list.
3. Click the Add… button.
   a. Check the box next to the project you wish to add.
   b. Click OK.
4. Click Apply.



5. Choose Deployment Assembly in the left hand selection list.
6. Click the Add… button.
   a. Choose Project.
   b. Click Next.
   c. Choose the project you intend to add to your WAR file.
   d. Click Finish.
7. Click Apply.



Data Access Object

- Create a data access object (DAO) to manage access to your entities.
- Mark the whole class as **@Transactional** to have Spring automatically start a transaction for each DAO method and commit the transaction at the end of the method.

```
@Transactional
public class MyJpaDAO implements MyDAO { … }
```

  - An unchecked (runtime) exception will trigger a rollback.

- Inject an **EntityManager** into a field using the **@PersistenceContext** annotation.

```
@PersistenceContext
private EntityManager em;
```

  - You can now use this field in your DAO methods to call **find()**, **persist()**, etc.

dao/CompanyDBDAO.java

```
…
public interface CompanyDBDAO {
     public Employee getEmployee(int id);
     public void updateEmployee(int id, Employee emp);
}
```

dao/CompanyDBJPADAO.java

```
…
@Transactional
public class CompanyDBJPADAO implements CompanyDBDAO {
     @PersistenceContext
     private EntityManager em;

     @Override
     public Employee getEmployee(int id) {
          Employee emp = em.find(Employee.class, id);
          em.detach(emp);
          return emp;
     }
     @Override
     public void updateEmployee(int id, Employee emp) {
          Employee managedEmp = em.find(Employee.class, id);
          managedEmp.setFirstName(emp.getFirstName());
          managedEmp.setLastName(emp.getLastName());
     }
}
```

Controller

- Inject your DAO in your controller so you can access it from within your **@RequestMapping** methods.

```
@Controller
public class MyController {
    @Autowired
    private MyDAO myDAO;
    …
}
```

controllers/CompanyDBController.java

```
@Controller
public class CompanyDBController {
     @Autowired
     private CompanyDBDAO companyDAO;

     @RequestMapping(path="GetEmployee.do",
                    method=RequestMethod.GET)
     public ModelAndView getEmployee() {
          return new ModelAndView("getEmployee.jsp");
     }

     @RequestMapping(path="GetEmployee.do",
                    method=RequestMethod.POST)
     public ModelAndView getEmployee(int id) {
          Employee emp = employeeDAO.getEmployee(id);
          return new ModelAndView("employee.jsp", "emp", emp);
     }

     @RequestMapping(path="EditEmployee.do",
                    method=RequestMethod.GET)
     public ModelAndView editEmployee(int id) {
          Employee emp = companyDAO.getEmployee(id);
          return new ModelAndView("editEmployee.jsp",
               "emp", emp);

     }

     @RequestMapping(path="EditEmployee.do",
                    method=RequestMethod.POST)
     public ModelAndView editEmployee(int id, Employee emp) {
          companyDAO.updateEmployee(id, emp);
          return new ModelAndView("employee.jsp",
               "emp", companyDAO.getEmployee(id));
     }
}
```

Spring Configuration File

- Like before, use the **context:component-scan** element to identify the package in which your controllers reside.

- Declare your data access object (DAO) as a bean so it can use Spring annotations and be injected into the controller.

```
<bean id="dao" class="dao.CompanyJPADAO" />
```

- Declare an **EntityManagerFactory** so that it can be injected into the DAO.

```
 <bean id="myEntityManagerFactory" class=
     "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
     <property name="persistenceUnitName" value="CompanyDB" />
</bean>
```

- Declare a **TransactionManager** so that Spring can automatically handle transactions in our DAO.

  - The string referenced by the entityManagerFactory should match the persistence unit name in your *persistence.xml* file.

```
<bean id="myTxnManager" class=
   "org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory"
              ref="myEntityManagerFactory" />
</bean>
```

- Use the **tx:annotation-driven** element to enable the **@Transactional** annotation.

  - Set the **transaction-manager** attribute to the name of the **TransactionManager** you set above.

```
<tx:annotation-driven transaction-manager="myTxnManager"/>
```

  - You will also need to include the **tx** namespace in your config file to support this element.

WebContent/WEB-INF/CompanyDB-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:mvc="http://www.springframework.org/schema/mvc"
   xmlns:context="http://www.springframework.org/schema/context"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-
beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-
context-4.2.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-4.2.xsd">

     <context:component-scan base-package="controllers" />

     <bean id="dao" class="dao.CompanyJPADAO" />

     <bean id="myEntityManagerFactory"
          class="org.springframework.orm.jpa.
                    LocalContainerEntityManagerFactoryBean">
          <property name="persistenceUnitName"
                    value="CompanyDB" />
     </bean>

     <bean id="myTransactionManager" class=
            "org.springframework.orm.jpa.JpaTransactionManager">
          <property name="entityManagerFactory"
                    ref="myEntityManagerFactory" />
     </bean>

     <tx:annotation-driven
          transaction-manager="myTransactionManager" />
</beans>
```