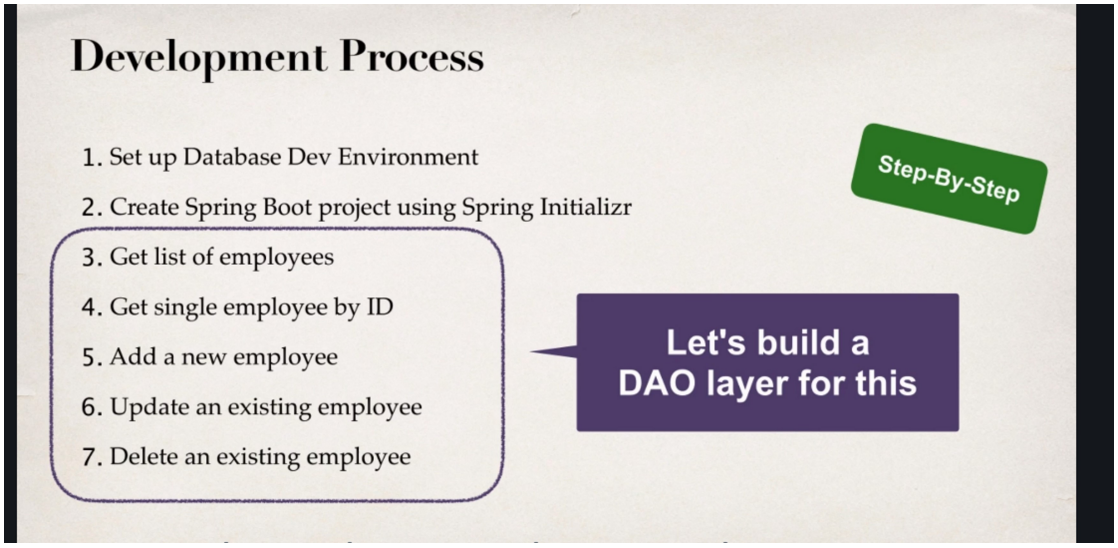
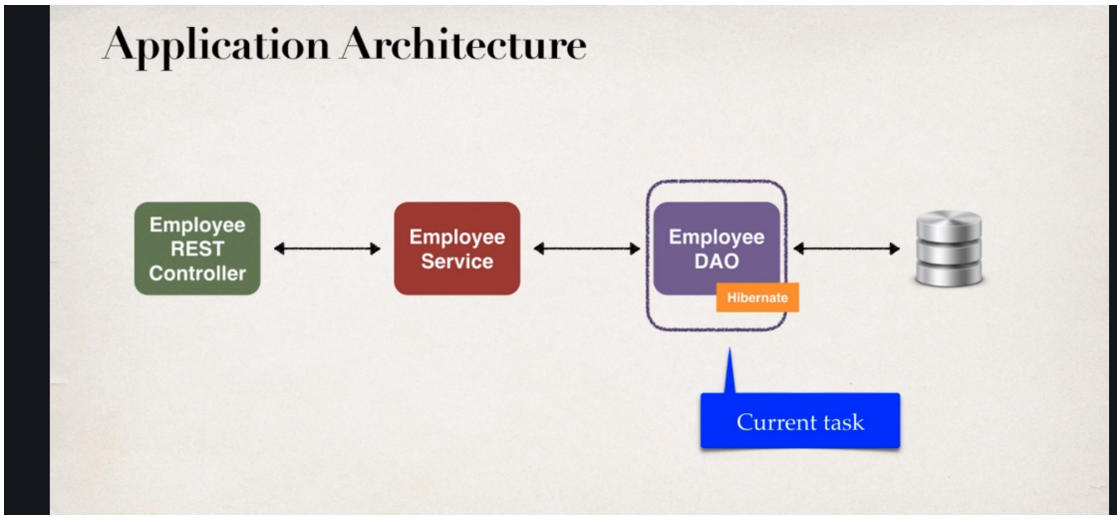
CREATE A DAO - HIBERNATE AND JPA

Tuesday, July 7, 2020 2:18 PM





Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory
- Hibernate Session Factory needs a Data Source
  - The data source defines database connection info

Traditional Spring

- We normally had to do this configuration manually

Very error-prone

Easy to make a simple mistake

XML

```
<!-- Step 1: Define Database DataSource / connection pool -->
<bean id="myDataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/web_customer_tracker?
    <property name="user" value="springstudent" />
    <property name="password" value="springstudent" />

    <!-- these are connection pool properties for C3P0 -->
    <property name="minPoolSize" value="5" />
    <property name="maxPoolSize" value="20" />
    <property name="maxIdleTime" value="30000" />
</bean>

<!-- Step 2: Setup Hibernate session factory -->
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource" />
    <property name="packagesToScan" value="com.luv2code.springdemo.entity" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>

<!-- Step 3: Setup Hibernate transaction manager -->
<bean id="myTransactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

ALL JAVA

```
@Bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
        myDataSource.setDriverClass("com.mysql.jdbc.Driver");
    }
    catch (PropertyVetoException exc) {
        throw new RuntimeException(exc);
    }

    myDataSource.setJdbcUrl(env.getProperty("jdbc.url"));
    myDataSource.setUser(env.getProperty("jdbc.user"));
    myDataSource.setPassword(env.getProperty("jdbc.password"));
    myDataSource.setInitialPoolSize(getIntProperty("connection.pool.initialPoolSize"));
    myDataSource.setMinPoolSize(getIntProperty("connection.pool.minPoolSize"));
    myDataSource.setMaxPoolSize(getIntProperty("connection.pool.maxPoolSize"));
    myDataSource.setMaxIdleTime(getIntProperty("connection.pool.maxIdleTime"));

    return myDataSource;
}
@Bean
public LocalSessionFactoryBean sessionFactory(){

    // create session factory
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();

    // set the properties
    sessionFactory.setDataSource(myDataSource());
    sessionFactory.setPackagesToScan(env.getProperty("hibernate.packagesToScan"));
    sessionFactory.setHibernateProperties(getHibernateProperties());

    return sessionFactory;
```


Traditional Spring

- We normally had to do this configuration manually

Very error-prone

Easy to make a simple mistake

XML

ALL JAVA

There should be an easier solution

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you
- Based on entries from Maven pom file
  - JDBC Driver: **`mysql-connector-java`**
  - Spring Data (ORM): **`spring-boot-starter-data-jpa`**
- DB connection info from **`application.properties`**

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory?
                                                    useSSL=false&serverTimezone=UTC

spring.datasource.username=springstudent
spring.datasource.password=springstudent
```

No need to give JDBC driver class name
Spring Boot will automatically detect it based on URL

Additional Data Source Properties

- Properties are available to configure connection pool etc

List of Data Source Properties

www.luv2code.com/spring-boot-props

`spring.datasource.*`

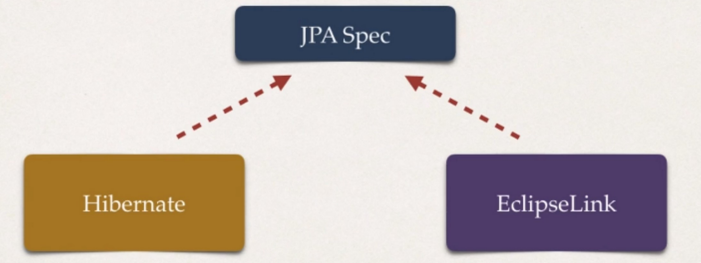
Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** ...
- You can then inject these into your app, for example your DAO
- **`EntityManager`** is from Java Persistence API (JPA)

What is JPA?

- Java Persistence API (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)
- Only a specification **www.luv2code.com/jpa-spec**
  - Defines a set of interfaces
  - Requires an implementation to be usable

JPA - Vendor Implementations

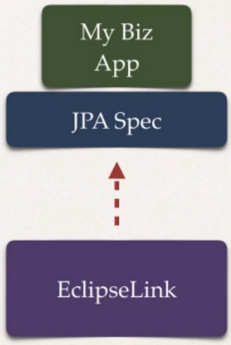


www.luv2code.com/jpa-vendors

What are Benefits of JPA

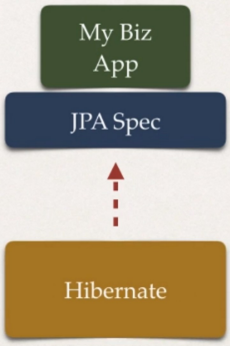
- By having a standard API, you are not locked to vendor's implementation
- Maintain portable, flexible code by coding to JPA spec (interfaces)
- Can theoretically switch vendor implementations
  - For example, if Vendor ABC stops supporting their product
  - You could switch to Vendor XYZ without vendor lock in

JPA - Vendor Implementations



SWAP WITH A DIFFERENT IMPLEMENTATION

JPA - Vendor Implementations



Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA
- **`EntityManager`** is similar to Hibernate **`SessionFactory`**
- **`EntityManager`** can serve as a wrapper for a Hibernate **`Session`** object
- We can inject the **`EntityManager`** into our DAO

Various DAO Techniques

- Version 1: Use EntityManager but leverage native Hibernate API
- Version 2: Use EntityManager and standard JPA API
- Version 3: Spring Data JPA

DAO Interface

```
public interface EmployeeDAO {

 public List<Employee> findAll();

}
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

 private EntityManager entityManager;

 @Autowired
 public EmployeeDAOHibernateImpl(EntityManager theEntityManager) {
  entityManager = theEntityManager;
 }

 ...
}
```

Optional

Automatically created by Spring Boot

Constructor injection

DAO Impl

```
@Override
@Transactional
public List<Employee> findAll() {

  // get the current hibernate session
  Session currentSession = entityManager.unwrap(Session.class);

  // create a query
  Query<Employee> theQuery =
     currentSession.createQuery("from Employee", Employee.class);

  // execute query and get result list
  List<Employee> employees = theQuery.getResultList();

  // return the results
  return employees;
}
```

Get current Hibernate session

Using native Hibernate API

Development Process

Step-By-Step

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation
5. Create REST controller to use DAO