Create DAO in Spring Boot

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

Step-By-Step

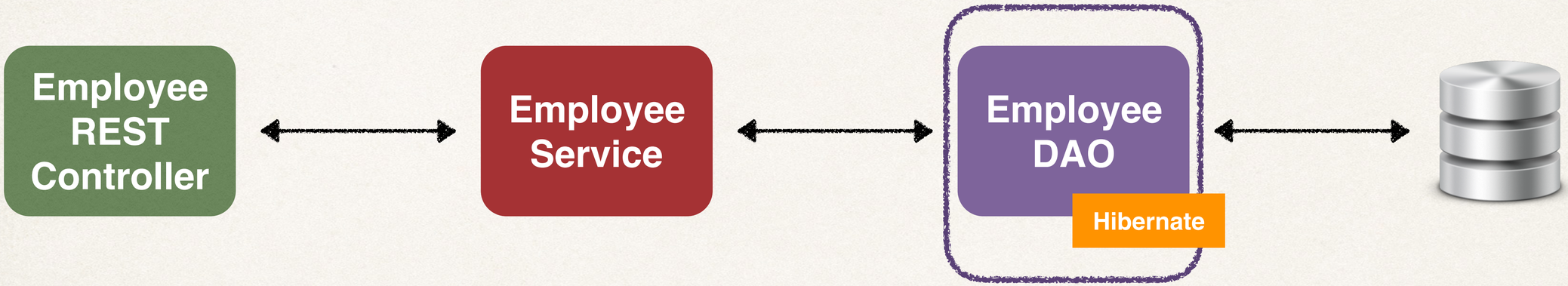
Development Process

1. Set up Database Dev Environment
2. Create Spring Boot project using Spring Initializr
3. Get list of employees
4. Get single employee by ID
5. Add a new employee
6. Update an existing employee
7. Delete an existing employee

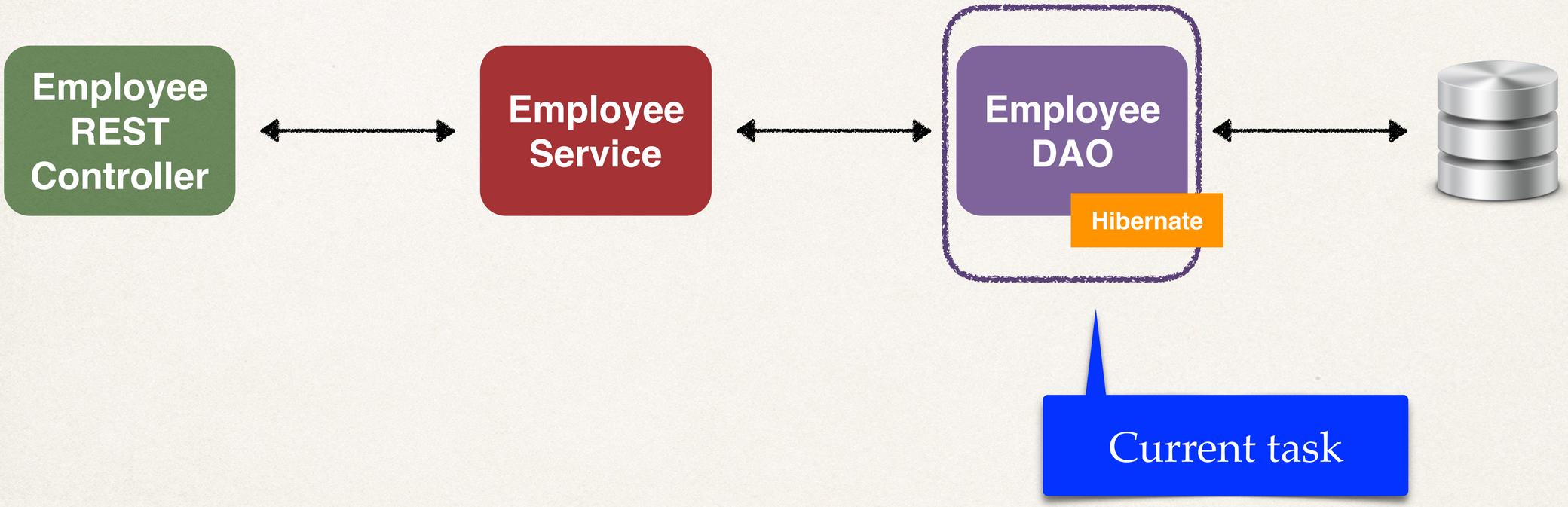
Step-By-Step

Let's build a DAO layer for this

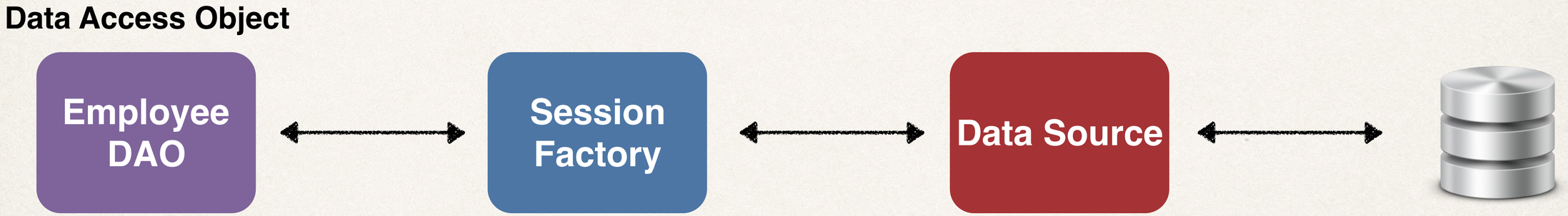
Application Architecture



Application Architecture

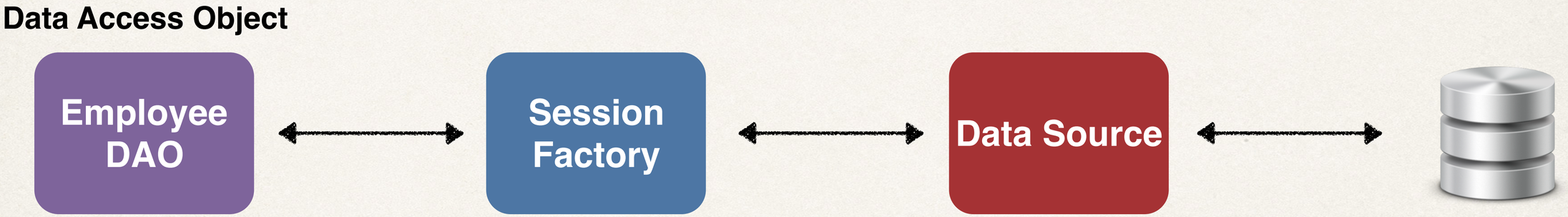


Hibernate Session Factory



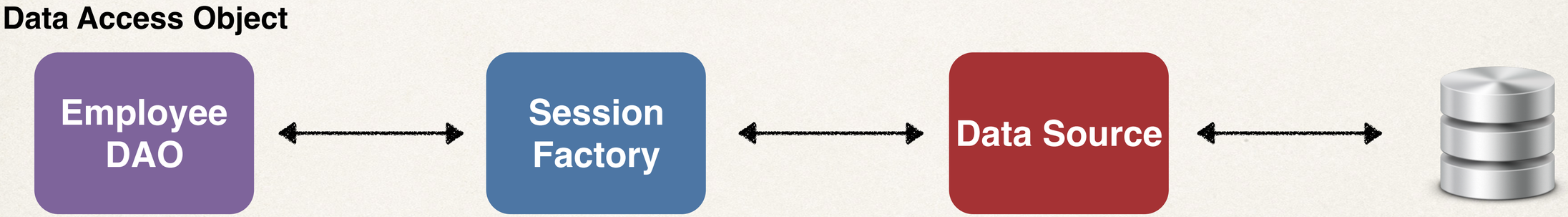
Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory



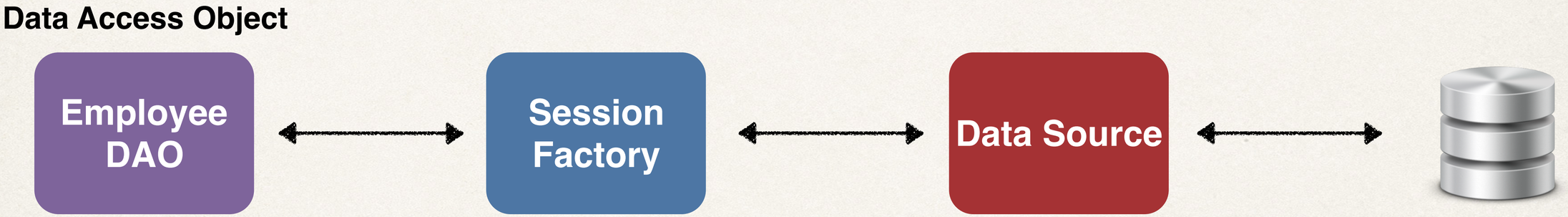
Hibernate Session Factory

- In the past, our DAO used a Hibernate Session Factory
- Hibernate Session Factory needs a Data Source



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

Traditional Spring

- We normally had to do this configuration manually

Traditional Spring

- We normally had to do this configuration manually

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>
```

Traditional Spring

- We normally had to do this configuration manually

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 factorys
    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

```
<!-- 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 factorys
    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

```
<!-- 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" />
```

```
</bean>
```

ALL JAVA

```
@Bean
public DataSource myDataSource() {
    ComboPooledDataSource myDataSource = new ComboPooledDataSource();
    try {
```

```
}
```

There should be an easier solution

Spring Boot to the Rescue

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

Spring Boot to the Rescue

- Spring Boot will automatically configure your data source for you

- Based on entries from Maven pom file

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 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`**

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

application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/employee_directory
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

Additional Data Source Properties

- Properties are available to configure connection pool etc

Additional Data Source Properties

- Properties are available to configure connection pool etc

List of Data Source Properties

www.luv2code.com/spring-boot-props

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

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:

Auto Data Source Configuration

- Based on configs, Spring Boot will automatically create the beans:
  - **`DataSource, EntityManager,`** ...

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

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?

What is JPA?

- Java Persistence API (JPA)

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification
  - Defines a set of interfaces
  - Requires an implementation to be usable

What is JPA?

- **J**ava **P**ersistence **A**PI (JPA)
  - Standard API for Object-to-Relational-Mapping (ORM)

- Only a specification

www.luv2code.com/jpa-spec

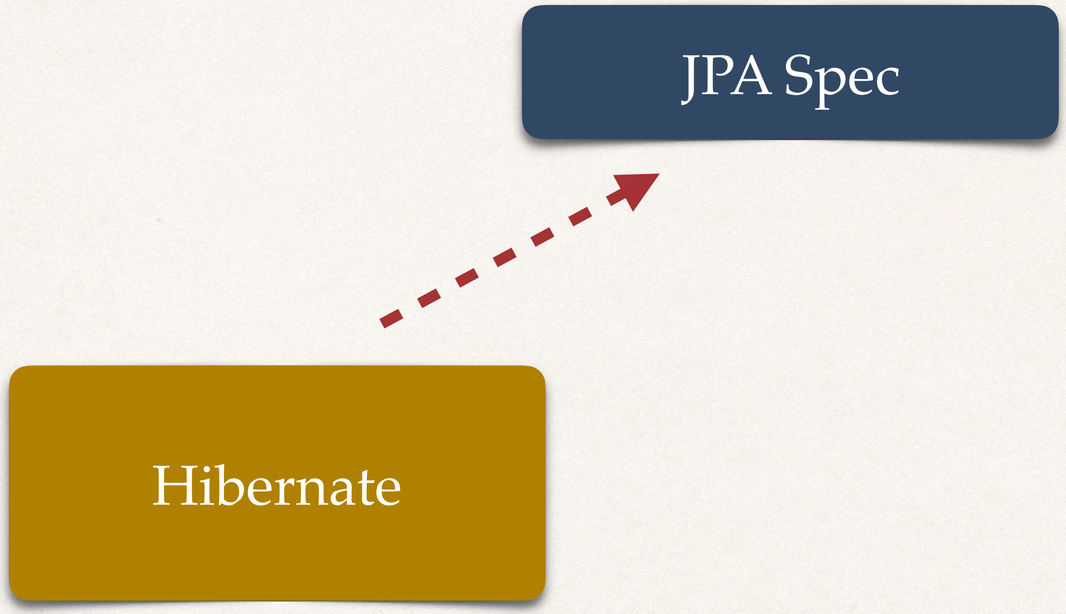
  - Defines a set of interfaces
  - Requires an implementation to be usable

JPA - Vendor Implementations

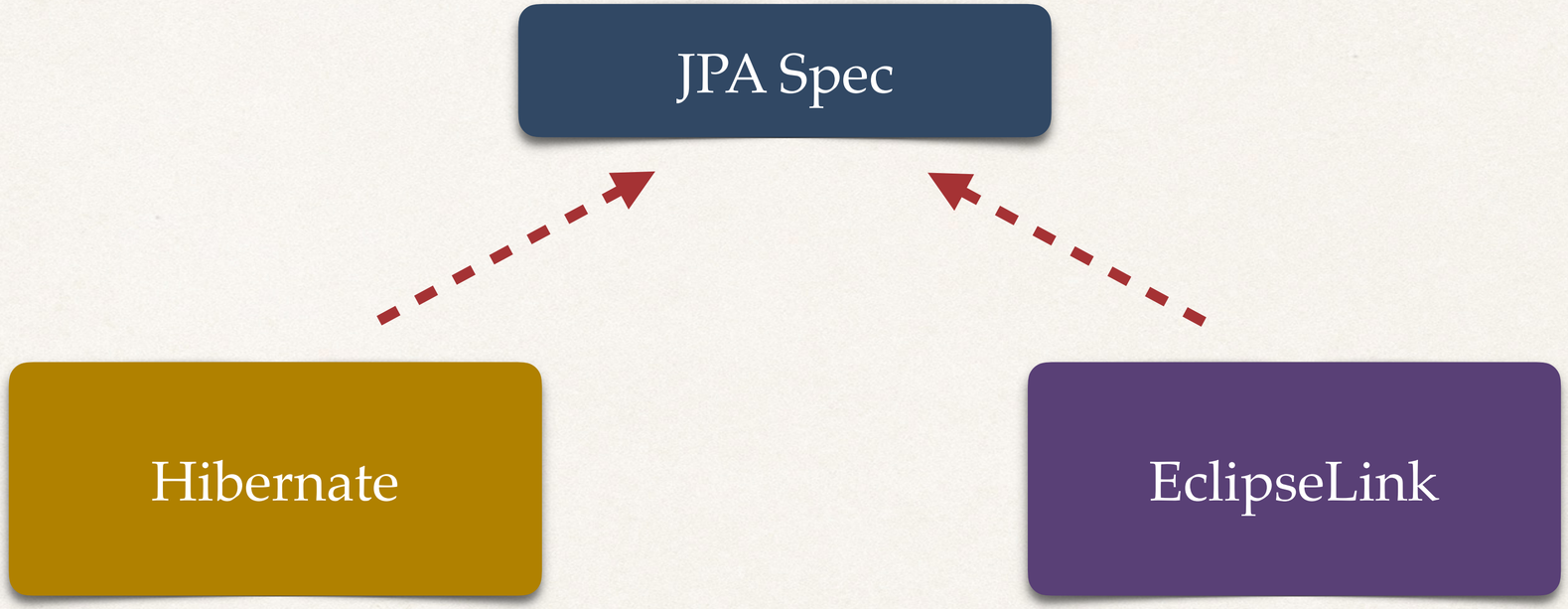
JPA - Vendor Implementations

JPA Spec

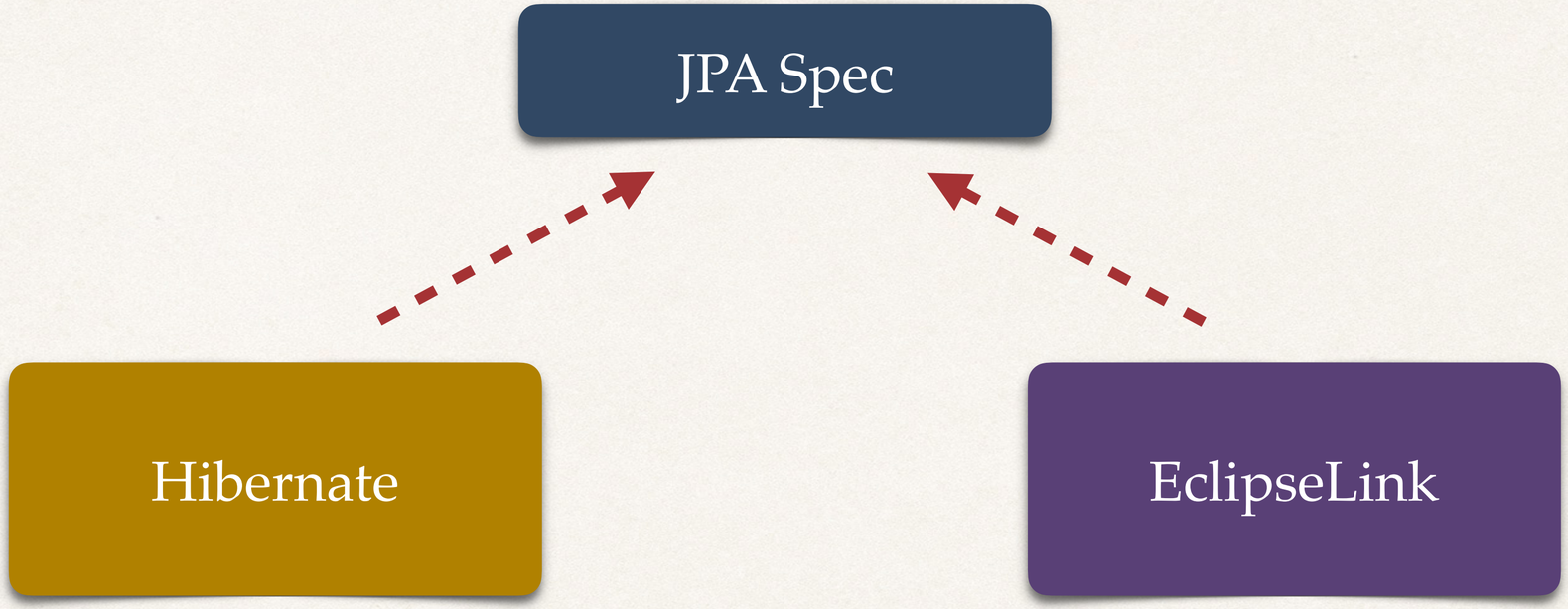
JPA - Vendor Implementations



JPA - Vendor Implementations



JPA - Vendor Implementations



www.luv2code.com/jpa-vendors

What are Benefits of JPA

What are Benefits of JPA

- By having a standard API, you are not locked to vendor's implementation

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)

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

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

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

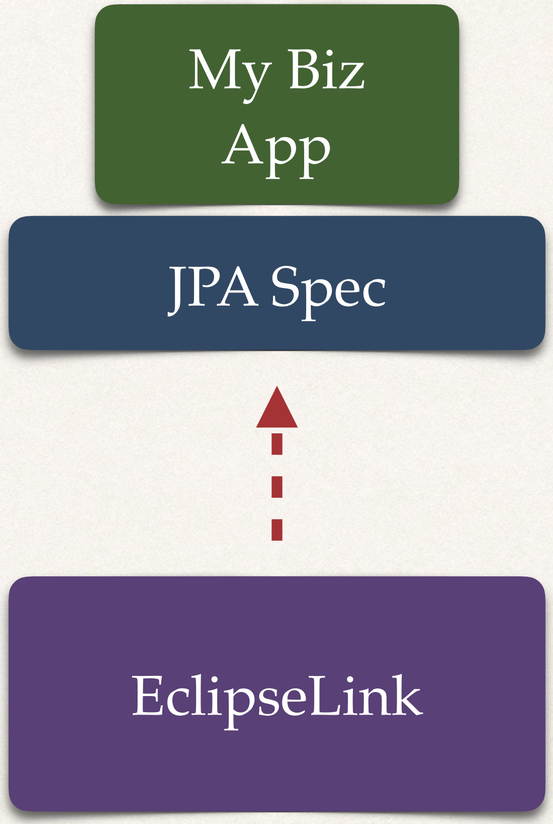
JPA - Vendor Implementations

JPA - Vendor Implementations

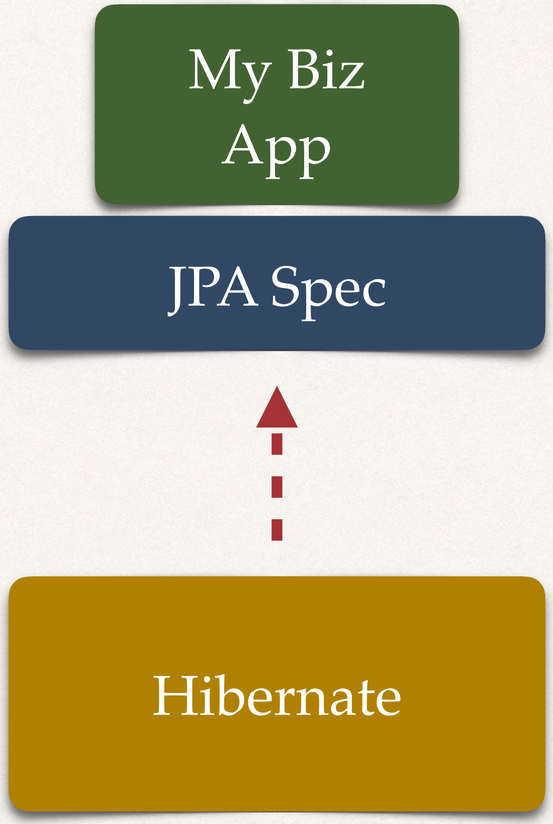
My Biz App

JPA - Vendor Implementations

JPA - Vendor Implementations



JPA - Vendor Implementations



Auto Data Source Configuration

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA

Auto Data Source Configuration

- In Spring Boot, Hibernate is default implementation of JPA

- **`EntityManager`** is similar to Hibernate **`SessionFactory`**

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

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

Show Me The Code!

Various DAO Techniques

Various DAO Techniques

- Version 1: Use EntityManager but leverage native Hibernate API

Various DAO Techniques

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

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

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

Automatically created by Spring Boot

DAO Impl

```
@Repository
public class EmployeeDAOHibernateImpl implements EmployeeDAO {

  private EntityManager entityManager;

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

 ...
}
```

Automatically created by Spring Boot

Constructor injection

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

DAO Impl

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

DAO Impl

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

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

Get current Hibernate session

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);
```

Get current Hibernate session

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

Development Process

1. Update db configs in application.properties

Development Process

1. Update db configs in application.properties
2. Create Employee entity

Development Process

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface

Development Process

1. Update db configs in application.properties
2. Create Employee entity
3. Create DAO interface
4. Create DAO implementation

Development Process

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