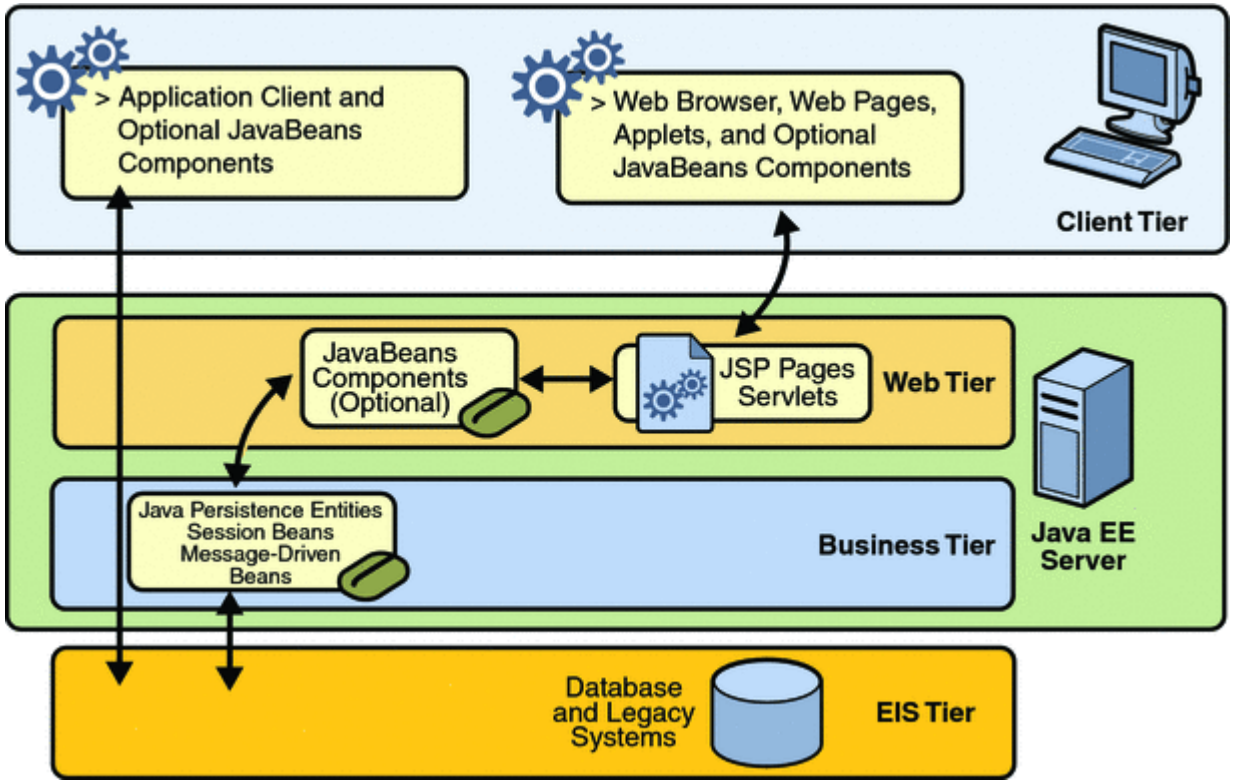
Presents

Java JPA

The ORM Problem

- The original Java J2EE looked at the problem of connection to existing corporate data centers



The ORM Problem

- The problem was that the java data objects had to be mapped to the underlying relational database
  - ✓ This was originally handled by JDBC code
  - ✓ Tried to create a layer of abstraction between the actual database and the Java code
  - ✓ But the Java code had to execute SQL statements and interpret the result
- The resulting code was often brittle and tightly coupled to the database
  - ✓ Changes to the underlying database could break a lot of Java code

Example from Oracle Docs

```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
      int total = rs.getInt("TOTAL");
      System.out.println(coffeeName + ", " + supplierID + ", " + price +
                         ", " + sales + ", " + total);
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}
```

LearnQuest

J2EE Entity Beans

- The alternative to directly accessing the database from a POJO was implemented in J2EE as “Entity Beans”

```
import javax.ejb.*;
import java.rmi.*;

public interface EmployeeLocalHome extends EJBLocalHome
{

  public EmployeeLocal create(Integer empNo) throws CreateException;

  // Find an existing employee
  public EmployeeLocal findByPrimaryKey (Integer empNo) throws FinderException;

  //Find all employees
  public Collection findAll() throws FinderException;

  //Calculate the Salaries of all employees
  public float calcSalary() throws Exception;
}
```

J2EE Entity Beans Problem

- The underlying database representation was no longer required in the Java code
  - ✓ Instead, it was moved into XML configuration files
  - ✓ These became very difficult to work with

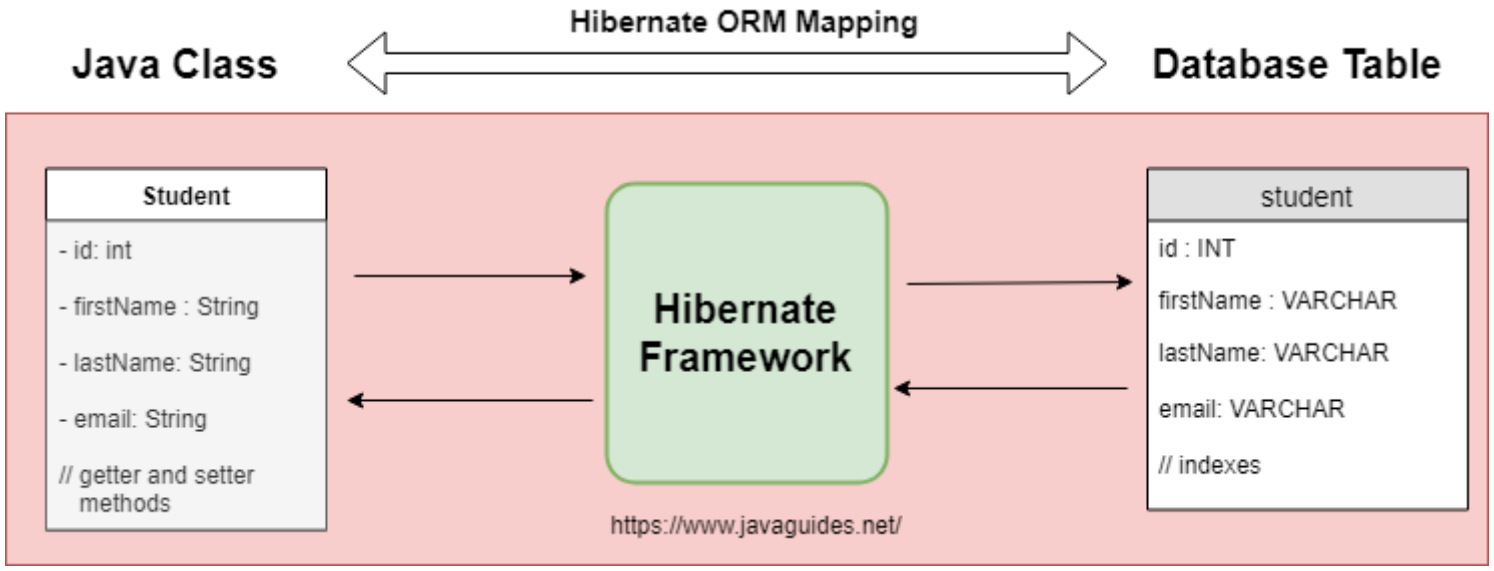
```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

The JPA Standard

- By the time EJB 3.0 came around the JPA specification had been released
  - ✓ Like the rest of the EE specifications, it defined an interface
  - ✓ The interface standardizes how Java interacts with persistent data
  - ✓ Utilizes the concept of an “entity”
  - ✓ Abstracts out the general concept of a query to be independent of the underlying database

The JPA Standard

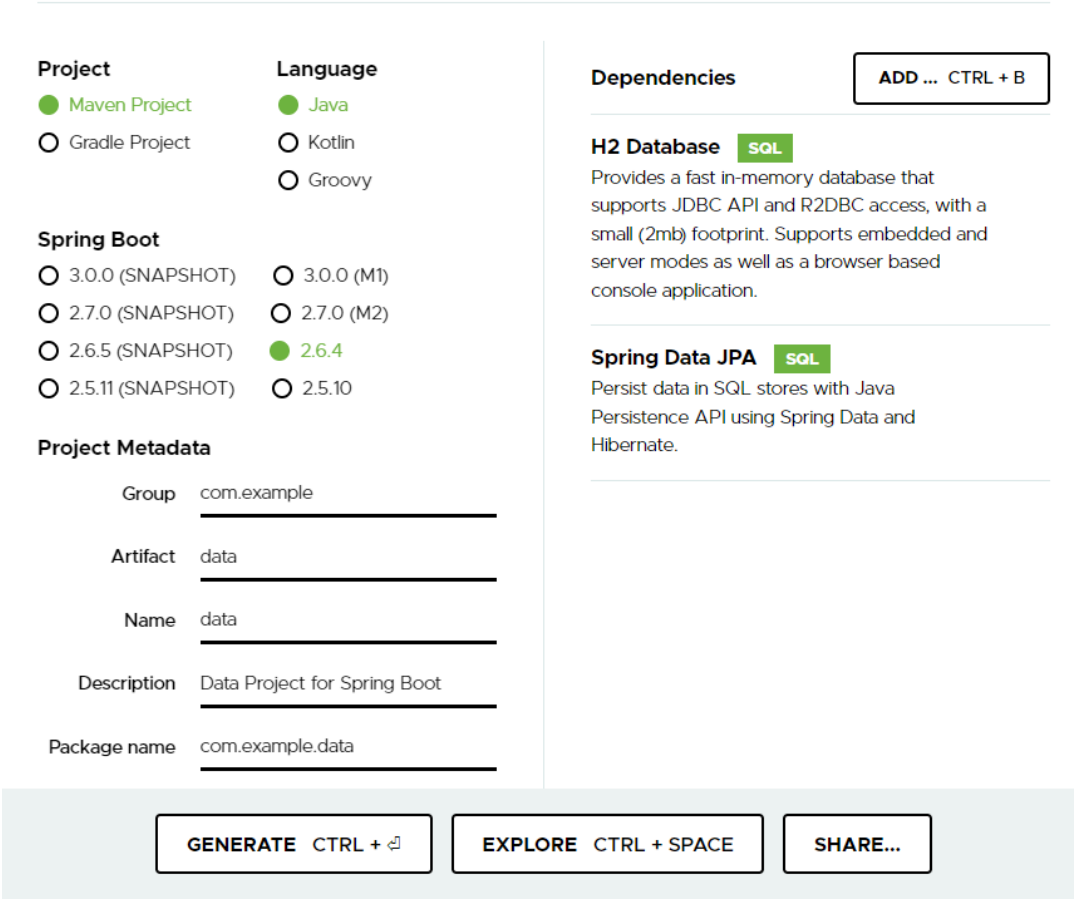
- Like other specifications, JPA defines an interface
  - ✓ This is implemented in various ORM products
  - ✓ Hibernate is a popular implementation



Spring Boot JPA

- The Spring Data project provides an implementation of the JPA interface
  - ✓ In our case, our lab will use the starter that implements the JPA using Hibernate

Spring Boot Project



LearnQuest

Defining an Entity

- A Java class annotated with @Entity is used to indicate a representation of a table
  - ✓ The annotations on id indicate it is the primary key and is to be autogenerated

```
@Entity
public class Customer {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private Long id;
	private String firstName;
	private String lastName;
}
```

Constructor

- We can define whatever constructors we need
  - ✓ But we need to provide a protected constructor of no arguments for Spring to use

```
protected Customer() {}

public Customer(String firstName, String lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

@Override
public String toString() {
  return String.format(
      "Customer[id=%d, firstName='%s', lastName='%s']",
      id, firstName, lastName);
}
```

The Repository

- The Spring JPA provides a number of standard query methods
  - ✓ In an interface called CrudRepository<Entitytype,keytype>
  - ✓ We extend this to add more customized methods
- This is an Interface
  - ✓ Spring writes the implementation of these methods
  - ✓ We do not have to code any of them
  - ✓ The implementations are created when the code is run.

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

      List<Customer> findByLastName(String lastName);
      Customer findById(long id);
    }
```

Using the Repository

- The repository methods can be called without further work
  - ✓ The methods we defined and the methods in the CRUD repository are both available
- This is demonstrated in the lab

Questions