Spring Boot JPA - Quick Guide

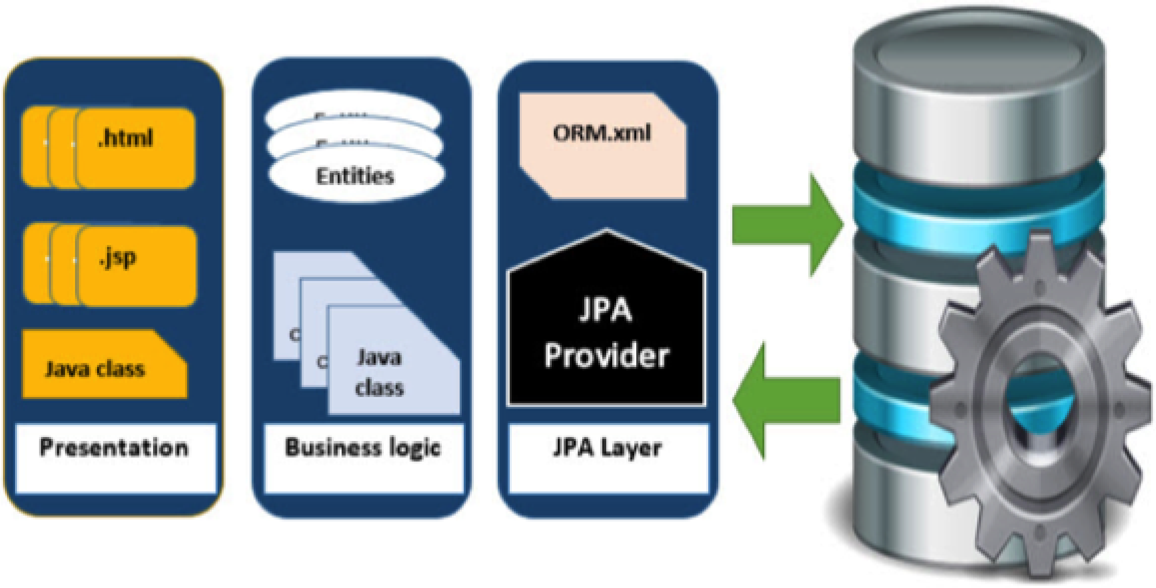
Spring Boot JPA - Overview

What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the ‘JPA Provider’ framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



JPA History

Earlier versions of EJB, defined persistence layer combined with business logic layer using javax.ejb.EntityBean Interface.

- While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
- JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
- JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338

JPA Providers

JPA is an open source API, therefore various enterprise vendors such as Oracle, Redhat, Eclipse, etc. provide new products by adding the JPA persistence flavor in them. Some of these products include −

Hibernate, Eclipselink, Toplink, Spring Data JPA, etc.

Spring Boot JPA - Environment Setup

This chapter will guide you on how to prepare a development environment to start your work with Spring Boot Framework. It will also teach you how to set up JDK, Eclipse on your machine before you set up Spring Boot Framework −

Step 1 - Setup Java Development Kit (JDK)

Java SE is available for download for free. To download click here , please download a version compatible with your operating system.

Follow the instructions to download Java, and run the **.exe** to install Java on your machine. Once you have installed Java on your machine, you would need to set environment variables to point to correct installation directories.

Setting Up the Path for Windows 2000/XP

Assuming you have installed Java in c:\Program Files\java\jdk directory −

- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now, edit the 'Path' variable and add the path to the Java executable directory at the end of it. For example, if the path is currently set to **C:\Windows\System32**, then edit it the following way

  C:\Windows\System32;c:\Program Files\java\jdk\bin.

Setting Up the Path for Windows 95/98/ME

Assuming you have installed Java in c:\Program Files\java\jdk directory −

- Edit the 'C:\autoexec.bat' file and add the following line at the end −

  SET PATH=%PATH%;C:\Program Files\java\jdk\bin

Setting Up the Path for Linux, UNIX, Solaris, FreeBSD

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

For example, if you use bash as your shell, then you would add the following line at the end of your **.bashrc** −

export PATH=/path/to/java:$PATH'

Alternatively, if you use an Integrated Development Environment (IDE) like Borland JBuilder, Eclipse, IntelliJ IDEA, or Sun ONE Studio, you will have to compile and run a simple program to confirm that the IDE knows where you have installed Java. Otherwise, you will have to carry out a proper setup as given in the document of the IDE.

Step 2 - Setup Eclipse IDE

All the examples in this tutorial have been written using Eclipse IDE. So we would suggest you should have the latest version of Eclipse installed on your machine.

To install Eclipse IDE, download the latest Eclipse binaries from www.eclipse.org/downloads/ . Once you download the installation, unpack the binary distribution into a convenient location. For example, in C:\eclipse on Windows, or /usr/local/eclipse on Linux/Unix and finally set PATH variable appropriately.

Eclipse can be started by executing the following commands on Windows machine, or you can simply double-click on eclipse.exe

```
%C:\eclipse\eclipse.exe
```

Eclipse can be started by executing the following commands on Unix (Solaris, Linux, etc.) machine −

```
$/usr/local/eclipse/eclipse
```

After a successful startup, if everything is fine then it should display the following result −

Eclipse Home page

Step 3 - Setup m2eclipse

M2Eclipse is eclipse plugin which is very useful integration for Apache Maven into the Eclipse IDE. We are using maven in this tutorial to build spring boot project and examples are run within eclipse using m2eclipse.

Install the latest M2Eclipse release by using the Install New Software dialog in Eclipse IDE,and point it to this p2 repository −

https://download.eclipse.org/technology/m2e/releases/latest/

Step 3 - Setup Spring Boot Project

Now if everything is fine, then you can proceed to set up your Spring Boot. Following are the simple steps to download and install the Spring Boot Project on your machine.

- Go to spring initializer link to create a spring boot project, https://start.spring.io/ .
- Select project as **Maven Project**.
- Select language as **Java**.
- Select Spring Boot version as **2.5.3**.
- Set Project Metadata - Group as **com.tutorialspoint**, Artifact as **springboot-h2**, name as **springboot-h2**, Description as **Demo project for Spring Boot and H2 Database** and package name as **com.tutorialspoint.springboot-h2**.
- Select packaging as **Jar**.
- Select java as **11**.
- Add dependencies as **Spring Web, Spring Data JPA, H2 Database and Spring Boot DevTools**.

Now click on GENERATE Button to generate the project structure.

Spring Initializer

Once the maven based spring boot project is downloaded, then import the maven project into eclipse and rest eclipse will handle. It will download the maven dependencies and build the project to make it ready for further development.

Step 4 - POSTMAN for REST APIs Testing

POSTMAN is a useful tool to test REST Based APIs. To install POSTMAN, download the latest POSTMAN binaries from www.postman.com/downloads/ . Once you download the installable, follow the instructions to install and use it.

Spring Boot JPA - Architecture

Java Persistence API is a source to store business entities as relational entities. It shows how to define a PLAIN OLD JAVA OBJECT (POJO) as an entity and how to manage entities with relations.

Class Level Architecture

The following image shows the class level architecture of JPA. It shows the core classes and interfaces of JPA.

JPA Class Level Architecture

The following table describes each of the units shown in the above architecture.

Sr.No	Units & Description
1	**EntityManagerFactory** This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
2	**EntityManager** It is an Interface, it manages the persistence operations on objects. It works like factory for Query instance.
3	**Entity** Entities are the persistence objects, stores as records in the database.
4	**EntityTransaction** It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class.
5	**Persistence** This class contain static methods to obtain EntityManagerFactory instance.
6	**Query** This interface is implemented by each JPA vendor to obtain relational objects that meet the criteria.

The above classes and interfaces are used for storing entities into a database as a record. They help programmers by reducing their efforts to write codes for storing data into a database so that they can concentrate on more important activities such as writing codes for mapping the classes with database tables.

JPA Class Relationships

In the above architecture, the relations between the classes and interfaces belong to the javax.persistence package. The following diagram shows the relationship between them.

JPA Class Relationships

- The relationship between EntityManagerFactory and EntityManager is **one-to-many**. It is a factory class to EntityManager instances.
- The relationship between EntityManager and EntityTransaction is **one-to-one**. For each EntityManager operation, there is an EntityTransaction instance.
- The relationship between EntityManager and Query is **one-to-many**. Many number of queries can execute using one EntityManager instance.
- The relationship between EntityManager and Entity is **one-to-many**. One EntityManager instance can manage multiple Entities.

Spring Boot JPA vs Hibernate

JPA

JPA is a specification which specifies how to access, manage and persist information/data between java objects and relational databases. It provides a standard approach for ORM, Object Relational Mapping.

Hibernate

Hibernate is an implementation of JPA. It provides a lightweight framework and is one of the most popular ORM tool used.

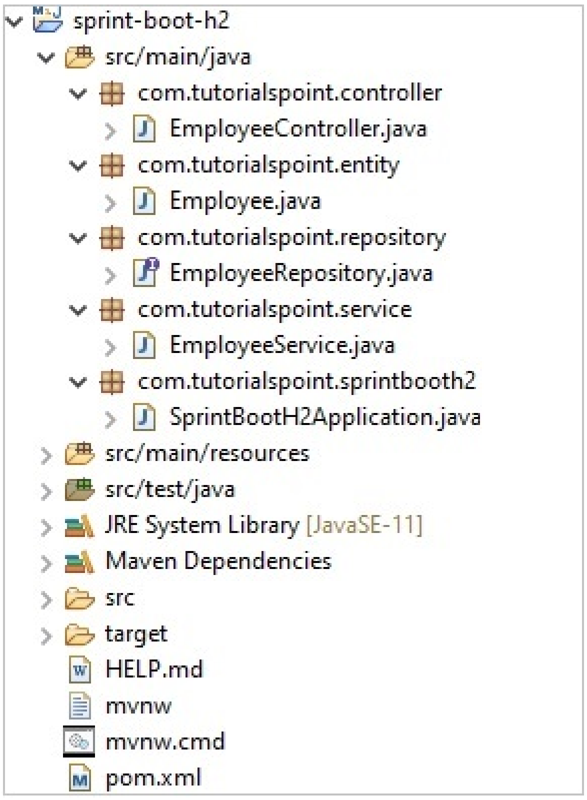
JPA Vs Hibernate

Following table summerises the differences between JPA and Hibernate.

Category	JPA	Hibernate
Type	JPA is a specification and defines the way to manage relational database data using java objects.	Hibernate is an implementation of JPA. It is an ORM tool to persist java objects into the relational databases.
Package	JPA uses javax.persistence package.	Hibernate uses org.hibernate package.
Factory	JPA uses EntityManagerFactory interface to get the entity manager to persist objects.	Hibernate uses SessionFactory interface to create session object which is then used to persist objects.
CRUD Operations	JPA uses EntityManager interface to create/read/delete operation and maintains the persistence context.	Hibernate uses Session interface to create/read/delete operation and maintains the persistence context.
Language	JPA uses JPQL (Java Persistence Query Language) as Object Oriented Query language for database operations.	Hibernate uses HQL (Hibernate Query Language) as Object Oriented Query language for database operations.

Spring Boot JPA - Application Setup

As in previous chapter Environment Setup , we've imported the generated spring boot project in eclipse. Now let's create the following structure in **src/main/java** folder.



```
sprint-boot-h2
  src/main/java
    com.tutorialspoint.controller
      EmployeeController.java
    com.tutorialspoint.entity
      Employee.java
    com.tutorialspoint.repository
      EmployeeRepository.java
    com.tutorialspoint.service
      EmployeeService.java
    com.tutorialspoint.sprintbooth2
      SprintBootH2Application.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- **com.tutorialspoint.controller.EmployeeController** − A REST Based Controller to implement REST based APIs.

- **com.tutorialspoint.entity.Employee** − An entity class representing the corresponding table in database.
- **com.tutorialspoint.repository.EmployeeRepository** − A Repository Interface to implement the CRUD operations on the database.
- **com.tutorialspoint.service.EmployeeService** − A Service Class to implement the business opearations over repository functions.
- **com.tutorialspoint.springbooth2.SprintBootH2Application** − A Spring Boot Application class.

SprintBootH2Application class is already present. We need to create the above packages and relevant classes and interface as shown below −

Entity - Entity.java

Following is the default code of Employee. It represents a Employee table with id, name, age and email columns.

```
package com.tutorialspoint.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table
public class Employee {
   @Id
   @Column
   private int id;

   @Column
   private String name;

   @Column
   private int age;

   @Column
   private String email;

   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
}
```

Repository - EmployeeRepository.java

Following is the default code of Repository to implement CRUD operations on above entity, Employee.

```
package com.tutorialspoint.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
}
```

Service - EmployeeService.java

Following is the default code of Service to implement operations over repository functions.

```
package com.tutorialspoint.service;

import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.repository.EmployeeRepository;

@Service
public class EmployeeService {
   @Autowired
   EmployeeRepository repository;

   public Employee getEmployeeById(int id) {
      return repository.findById(id).get();
   }
   public List<Employee> getAllEmployees(){
      List<Employee> employees = new ArrayList<Employee>();
      repository.findAll().forEach(employee -> employees.add(employee));
      return employees;
   }
   public void saveOrUpdate(Employee employee) {
      repository.save(employee);
   }
   public void deleteEmployeeById(int id) {
      repository.deleteById(id);
   }
}
```

Controller - EmployeeController.java

Following is the default code of Controller to implement REST APIs.

```
package com.tutorialspoint.controller;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.service.EmployeeService;
```

```
@RestController
@RequestMapping(path = "/emp")
public class EmployeeController {
   @Autowired
   EmployeeService employeeService;

   @GetMapping("/employees")
   public List<Employee> getAllEmployees(){
      return employeeService.getAllEmployees();
   }
   @GetMapping("/employee/{id}")
   public Employee getEmployee(@PathVariable("id") int id) {
      return employeeService.getEmployeeById(id);
   }
   @DeleteMapping("/employee/{id}")
   public void deleteEmployee(@PathVariable("id") int id) {
      employeeService.deleteEmployeeById(id);
   }
   @PostMapping("/employee")
   public void addEmployee(@RequestBody Employee employee) {
      employeeService.saveOrUpdate(employee);
   }
   @PutMapping("/employee")
   public void updateEmployee(@RequestBody Employee employee) {
      employeeService.saveOrUpdate(employee);
   }
}
```

Application - SprintBootH2Application.java

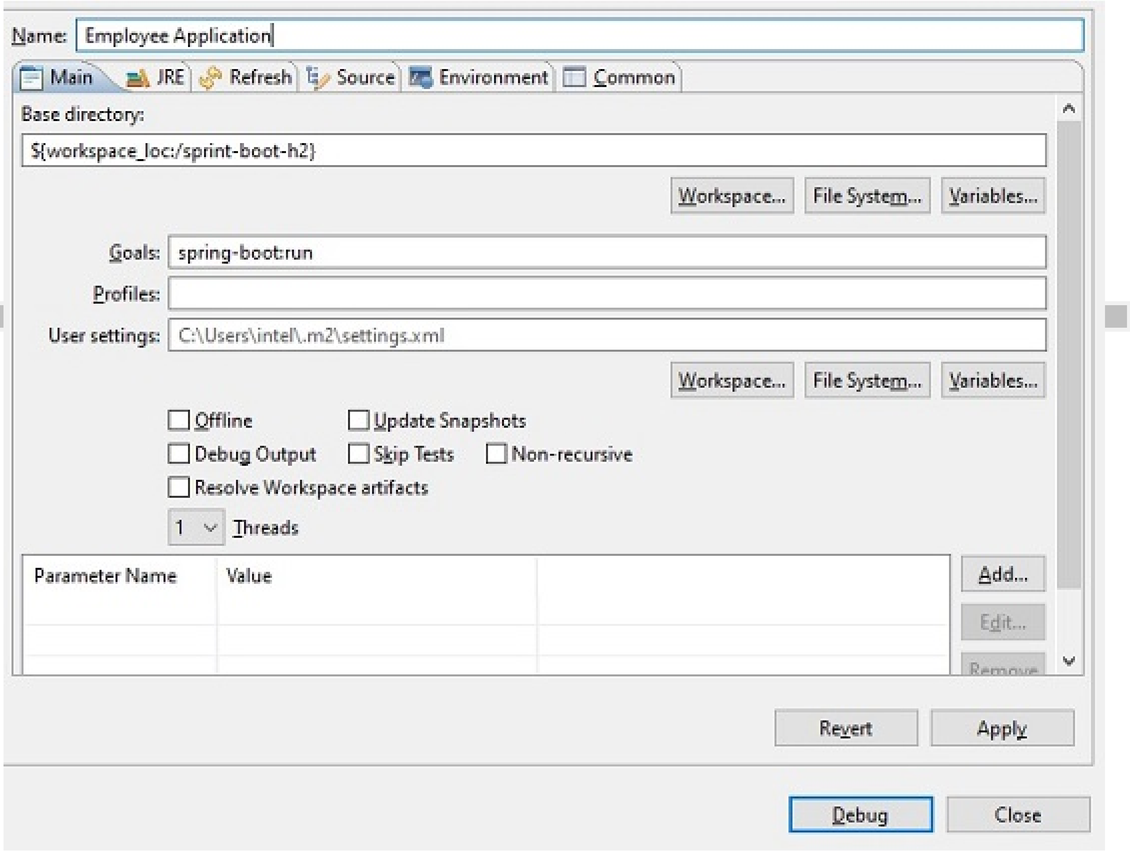
Following is the updated code of Application to use above classes.

```
package com.tutorialspoint.sprintbooth2;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.domain.EntityScan;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.data.jpa.repository.config.EnableJpaRepositorie

@ComponentScan({"com.tutorialspoint.controller","com.tutorialspoint.service
@EntityScan("com.tutorialspoint.entity")
@EnableJpaRepositories("com.tutorialspoint.repository")
@SpringBootApplication
```

```
public class SprintBootH2Application {
   public static void main(String[] args) {
      SpringApplication.run(SprintBootH2Application.class, args);
   }
}
```

Run/Debug Configuration

Create following **maven configuration** in eclipse to run the springboot application with goal **spring-boot:run**. This configuration will help to run the REST APIs and we can test them using POSTMAN.



Name: Employee Application

Base directory:
${workspace_loc:/sprint-boot-h2}

Goals: spring-boot:run

Profiles:

User settings: C:\Users\intel\.m2\settings.xml

□ Offline □ Update Snapshots
□ Debug Output □ Skip Tests □ Non-recursive
□ Resolve Workspace artifacts

1 Threads

Parameter Name	Value

Run the application

In eclipse, run the **Employee Application** configuration. Eclipse console will show the similar output.

```
[INFO] Scanning for projects...
...
2021-07-24 20:51:14.823  INFO 9760 --- [restartedMain] c.t.s.SprintBootH2Applic
Started SprintBootH2Application in 7.353 seconds (JVM running for 8.397)
```

Once server is up and running, Use Postman to make a POST request to add a record first.

Set the following parameters in POSTMAN.

- HTTP Method - **POST**
- URL - **http://localhost:8080/emp/employee**
- BODY - **An employee JSON**

```
{
   "id": "1",
   "age": "35",
   "name": "Julie",
   "email": "julie@gmail.com"
}
```

Click on Send Button and check the response status to be OK. Now make a GET Request to get all records.

Set the following parameters in POSTMAN.

- HTTP Method - **GET**
- URL - **http://localhost:8080/emp/employees**

Click the send button and verify the response.

```
[{
   "id": "1",
   "age": "35",
   "name": "Julie",
   "email": "julie@gmail.com"
}]
```

Spring Boot JPA - Unit Test Repository

To test a Repository, we need the following annotation and classes −

- **@ExtendWith(SpringExtension.class)** − Mark the class to run as test case using SpringExtension class.
- **@SpringBootTest(classes = SprintBootH2Application.class)** − Configure the Spring Boot application.
- **@Transactional** − To mark repository to do CRUD Operation capable.
- **@Autowired private EmployeeRepository employeeRepository** − EmployeeRepository object to be tested.

Example

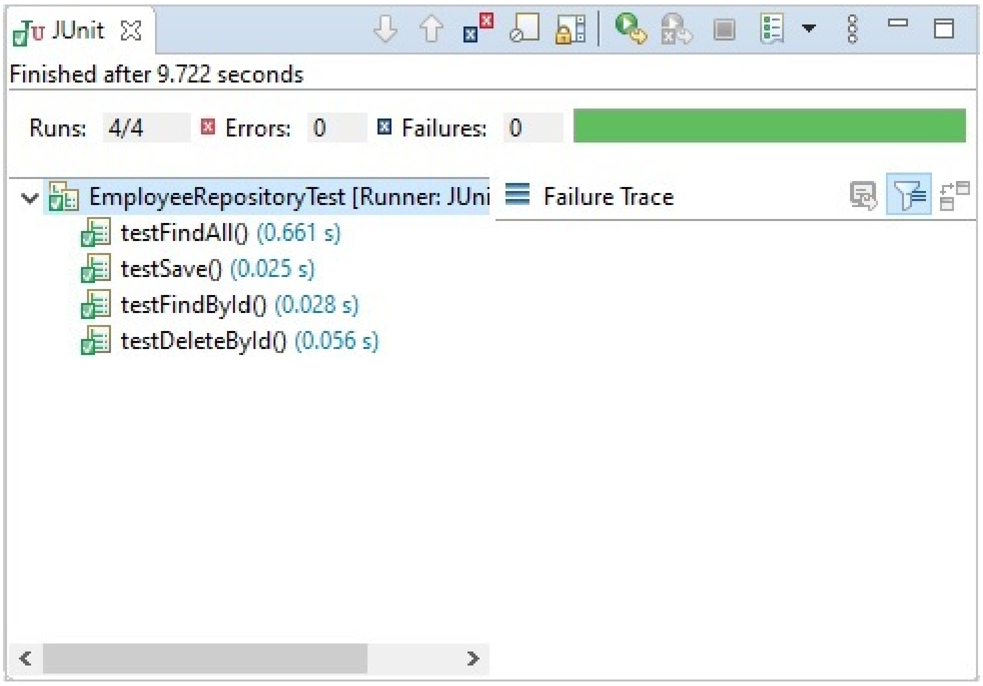
Following is the complete code of EmployeeRepositoryTest.

```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.sprintbooth2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
   @Autowired
   private EmployeeRepository employeeRepository;

   @Test
   public void testFindById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), result.getId());
   }
   @Test
   public void testFindAll() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 1);
   }
   @Test
   public void testSave() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee found = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), found.getId());
   }
   @Test
   public void testDeleteById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      employeeRepository.deleteById(employee.getId());
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 0);
   }
   private Employee getEmployee() {
      Employee employee = new Employee();
      employee.setId(1);
      employee.setName("Mahesh");
      employee.setAge(30);
      employee.setEmail("mahesh@test.com");
      return employee;
   }
}
```

Run the test cases

Output

Right Click on the file in eclipse and select **Run a JUnit Test** and verify the result.



```
Finished after 9.722 seconds
Runs: 4/4   Errors: 0   Failures: 0
EmployeeRepositoryTest [Runner: JUni
   testFindAll() (0.661 s)
   testSave() (0.025 s)
   testFindById() (0.028 s)
   testDeleteById() (0.056 s)
```

Spring Boot JPA - Repository methods

Let's now analyze the methods available in repository interface which we've created.

Repository - EmployeeRepository.java

Following is the default code of Repository to implement CRUD operations on above entity, Employee.

```
package com.tutorialspoint.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
}
```

Now this repository contains following methods by default.

Sr.No	Method & Description
1	**count(): long** returns the number of entities available.
2	**delete(Employee entity): void** deletes an entity.
3	**deleteAll():void** deletes all the entities.
4	**deleteAll(Iterable< extends Employee > entities):void** deletes the entities passed as argument.
5	**deleteAll(Iterable< extends Integer > ids):void** deletes the entities identified using their ids passed as argument.
6	**existsById(Integer id):boolean** checks if an entity exists using its id.
7	**findAll():Iterable< Employee >** returns all the entities.
8	**findAllByIds(Iterable< Integer > ids):Iterable< Employee >** returns all the entities identified using ids passed as argument.
9	**findById(Integer id):Optional< Employee >** returns an entity identified using id.
10	**save(Employee entity): Employee** saves an entity and return the updated one.
11	**saveAll(Iterable< Employee> entities): Iterable< Employee>**

	saves all entities passed and return the updated entities.

Spring Boot JPA - Custom methods

We've checked the methods available by default in Repository in JPA Methods chapter. Now let's add a method and test it.

Repository - EmployeeRepository.java

Example

Add a method to find an employee by its name.

```
package com.tutorialspoint.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
   public List<Employee> findByName(String name);
   public List<Employee> findByAge(int age);
}
```

Now Spring JPA will create the implementation of above methods automatically as we've following the property based nomenclature. Let's test the methods added by adding their test cases in test file. Last two methods of below file tests the custom methods added.

Following is the complete code of EmployeeRepositoryTest.

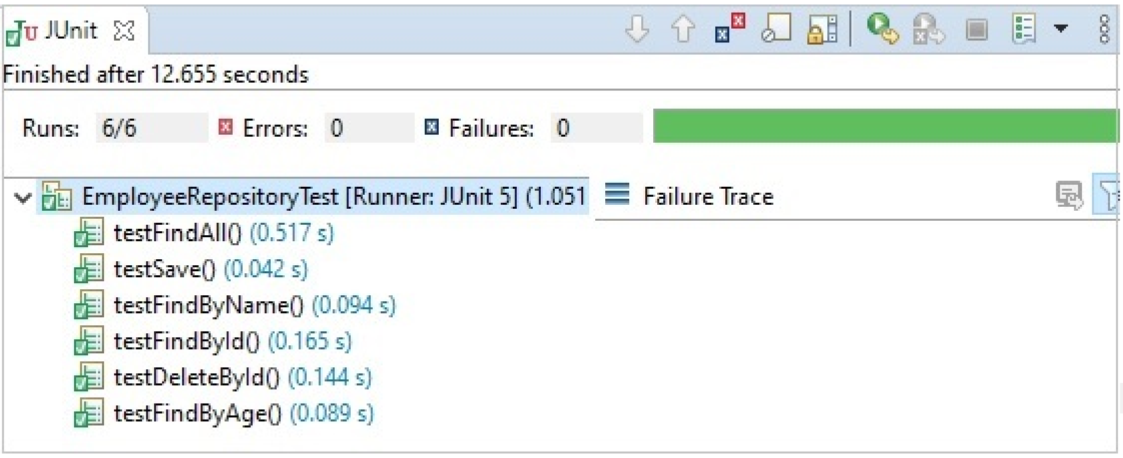
```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.sprintbooth2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
   @Autowired
   private EmployeeRepository employeeRepository;
   @Test
   public void testFindById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findById(employee.getId()).get(
      assertEquals(employee.getId(), result.getId());
   }
   @Test
   public void testFindAll() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 1);
   }
   @Test
   public void testSave() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee found = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), found.getId());
   }
   @Test
   public void testDeleteById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      employeeRepository.deleteById(employee.getId());
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 0);
   }
   private Employee getEmployee() {
      Employee employee = new Employee();
      employee.setId(1);
      employee.setName("Mahesh");
      employee.setAge(30);
      employee.setEmail("mahesh@test.com");
      return employee;
```

```
    }
    @Test
    public void testFindByName() {
       Employee employee = getEmployee();
       employeeRepository.save(employee);
       List<Employee> result = new ArrayList<>();
       employeeRepository.findByName(employee.getName()).forEach(e -> result
       assertEquals(result.size(), 1);
    }
    @Test
    public void testFindByAge() {
       Employee employee = getEmployee();
       employeeRepository.save(employee);
       List<Employee> result = new ArrayList<>();
       employeeRepository.findByAge(employee.getAge()).forEach(e -> result.a
       assertEquals(result.size(), 1);
    }
 }
```

Run the test cases

Output

Right Click on the file in eclipse and select **Run a JUnit Test** and verify the result.



```
Finished after 12.655 seconds
Runs: 6/6   Errors: 0   Failures: 0
EmployeeRepositoryTest [Runner: JUnit 5] (1.051
   testFindAll() (0.517 s)
   testSave() (0.042 s)
   testFindByName() (0.094 s)
   testFindById() (0.165 s)
   testDeleteById() (0.144 s)
   testFindByAge() (0.089 s)
```

Spring Boot JPA - Named Queries

Some time case arises, where we need a custom query to fulfil one test case. We can use @NamedQuery annotation to specify a named query within an entity class and then declare that method in repository. Following is an example.

We've added custom methods in Repository in JPA Custom Methods chapter. Now let's add another method using @NamedQuery and test it.

Entity - Entity.java

Following is the default code of Employee. It represents a Employee table with id, name, age and email columns.

```
package com.tutorialspoint.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table
@NamedQuery(name = "Employee.findByEmail",
query = "select e from Employee e where e.email = ?1")
public class Employee {
   @Id
   @Column
   private int id;

   @Column
   private String name;

   @Column
   private int age;

   @Column
   private String email;

   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
}
```

Repository - EmployeeRepository.java

Add a method to find an employee by its name and age.

```
package com.tutorialspoint.repository;

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
   public List<Employee> findByName(String name);
   public List<Employee> findByAge(int age);
   public Employee findByEmail(String email);
}
```

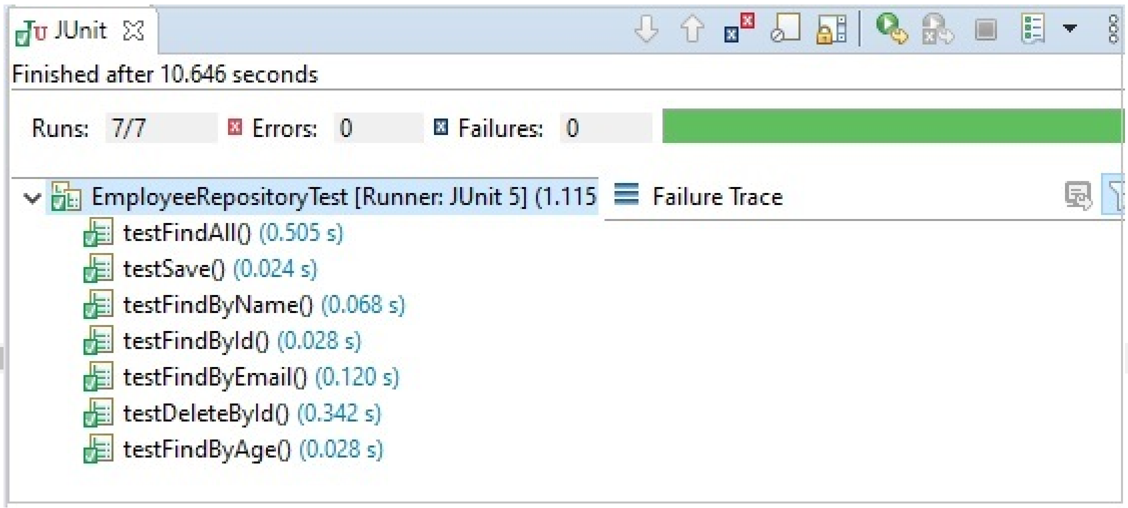
Now Spring JPA will create the implementation of above methods automatically using the query provided in named query. Let's test the methods added by adding their test cases in test file. Last two methods of below file tests the named query method added.

Following is the complete code of EmployeeRepositoryTest.

```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.sprintbooth2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
   @Autowired
   private EmployeeRepository employeeRepository;

   @Test
   public void testFindById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), result.getId());
   }
   @Test
   public void testFindAll() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 1);
   }
   @Test
   public void testSave() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee found = employeeRepository.findById(employee.getId()).get();
      assertEquals(employee.getId(), found.getId());
   }
   @Test
   public void testDeleteById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      employeeRepository.deleteById(employee.getId());
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 0);
   }
   private Employee getEmployee() {
      Employee employee = new Employee();
      employee.setId(1);
      employee.setName("Mahesh");
      employee.setAge(30);
      employee.setEmail("mahesh@test.com");
      return employee;
   }
   @Test
   public void testFindByName() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByName(employee.getName()).forEach(e -> resul
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByAge() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByAge(employee.getAge()).forEach(e -> result.
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByEmail() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findByEmail(employee.getEmail()
      assertNotNull(result);
   }
}
```

Run the test cases

Right Click on the file in eclipse and select **Run a JUnit Test** and verify the result.



Finished after 10.646 seconds

Runs: 7/7 Errors: 0 Failures: 0

```
EmployeeRepositoryTest [Runner: JUnit 5] (1.115
   testFindAll() (0.505 s)
   testSave() (0.024 s)
   testFindByName() (0.068 s)
   testFindById() (0.028 s)
   testFindByEmail() (0.120 s)
   testDeleteById() (0.342 s)
   testFindByAge() (0.028 s)
```

Spring Boot JPA - Custom Query

Some time case arises, where we need a custom query to fulfil one test case. We can use @Query annotation to specify a query within a repository. Following is an example. In this example, we are using JPQL, Java Persistence Query Language.

We've added name query custom methods in Repository in JPA Named Query chapter. Now let's add another method using @Query and test it.

Repository - EmployeeRepository.java

Add a method to get list of employees order by their names.

```
package com.tutorialspoint.repository;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
   public List<Employee> findByName(String name);
   public List<Employee> findByAge(int age);
   public Employee findByEmail(String email);

   @Query(value = "SELECT e FROM Employee e ORDER BY name")
   public List<Employee> findAllSortedByName();
}
```

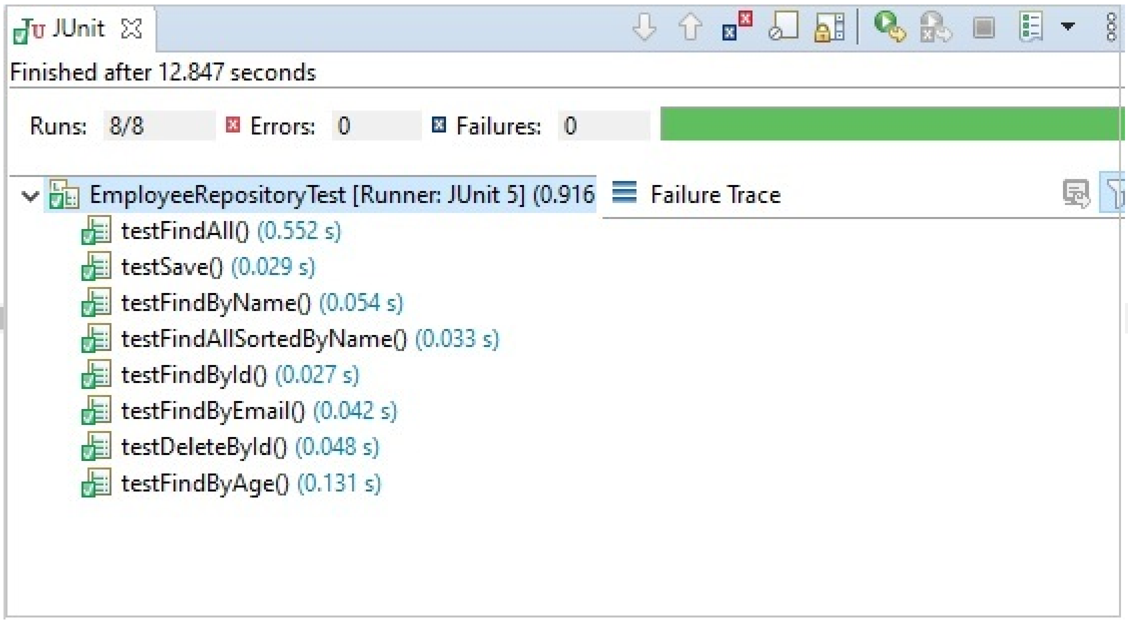
Let's test the methods added by adding their test cases in test file. Last two methods of below file tests the custom query method added.

Following is the complete code of EmployeeRepositoryTest.

```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.sprintbooth2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
   @Autowired
   private EmployeeRepository employeeRepository;
   @Test
   public void testFindById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findById(employee.getId()).get(
      assertEquals(employee.getId(), result.getId());
   }
   @Test
   public void testFindAll() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 1);
   }
   @Test
   public void testSave() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee found = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), found.getId());
   }
   @Test
   public void testDeleteById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      employeeRepository.deleteById(employee.getId());
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 0);
   }
   private Employee getEmployee() {
      Employee employee = new Employee();
      employee.setId(1);
      employee.setName("Mahesh");
      employee.setAge(30);
      employee.setEmail("mahesh@test.com");
      return employee;
   }
   @Test
   public void testFindByName() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByName(employee.getName()).forEach(e -> result
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByAge() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByAge(employee.getAge()).forEach(e -> result.a
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByEmail() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findByEmail(employee.getEmail()
      assertNotNull(result);
   }
   @Test
   public void testFindAllSortedByName() {
      Employee employee = getEmployee();
      Employee employee1 = new Employee();
      employee1.setId(2);
      employee1.setName("Aarav");
      employee1.setAge(20);
      employee1.setEmail("aarav@test.com");
      employeeRepository.save(employee);
      employeeRepository.save(employee1);
      List<Employee> result = employeeRepository.findAllSortedByName();
      assertEquals(employee1.getName(), result.get(0).getName());
   }
}
```

Run the test cases

Right Click on the file in eclipse and select **Run a JUnit Test** and verify the result.



```
EmployeeRepositoryTest [Runner: JUnit 5] (0.916
    testFindAll() (0.552 s)
    testSave() (0.029 s)
    testFindByName() (0.054 s)
    testFindAllSortedByName() (0.033 s)
    testFindById() (0.027 s)
    testFindByEmail() (0.042 s)
    testDeleteById() (0.048 s)
    testFindByAge() (0.131 s)
```

Spring Boot JPA - Native Query

Some time case arises, where we need a custom native query to fulfil one test case. We can use @Query annotation to specify a query within a repository. Following is an example. In this example, we are using native query, and set an attribute **nativeQuery=true** in Query annotation to mark the query as native.

We've added custom methods in Repository in JPA Custom Query chapter. Now let's add another method using native query and test it.

Repository - EmployeeRepository.java

Add a method to get list of employees order by their names.

```
package com.tutorialspoint.repository;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;
import com.tutorialspoint.entity.Employee;

@Repository
public interface EmployeeRepository extends CrudRepository<Employee, Intege
   public List<Employee> findByName(String name);
   public List<Employee> findByAge(int age);
   public Employee findByEmail(String email);

   @Query(value = "SELECT e FROM Employee e ORDER BY name")
   public List<Employee> findAllSortedByName();

   @Query(value = "SELECT * FROM Employee ORDER BY name", nativeQuery =
   public List<Employee> findAllSortedByNameUsingNative();
}
```

Let's test the methods added by adding their test cases in test file. Last two methods of below file tests the custom query method added.

Example

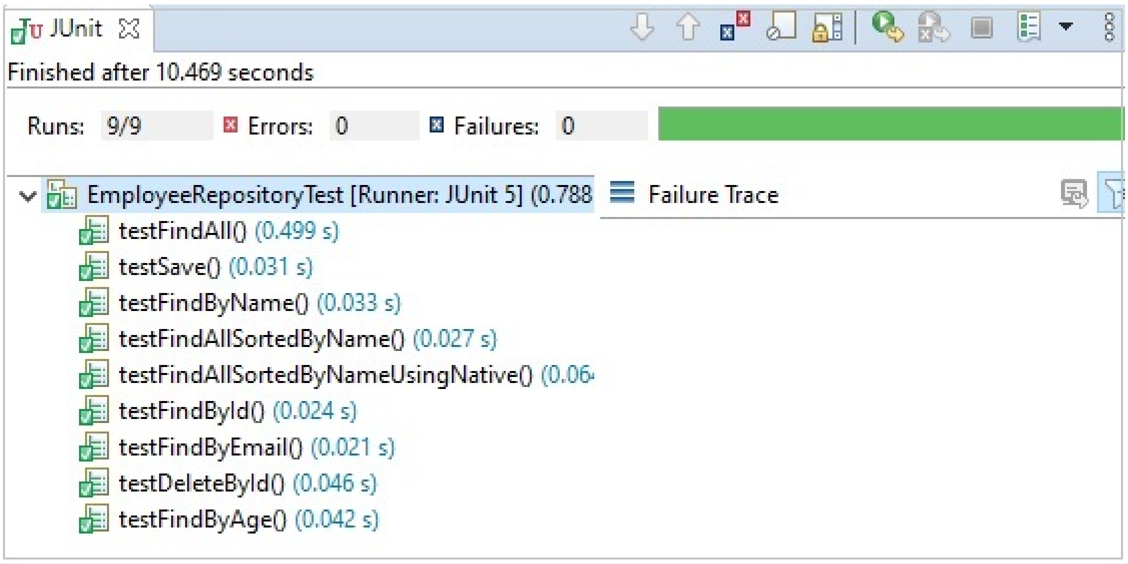
Following is the complete code of EmployeeRepositoryTest.

```
package com.tutorialspoint.repository;

import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.ArrayList;
import java.util.List;
import javax.transaction.Transactional;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import com.tutorialspoint.entity.Employee;
import com.tutorialspoint.sprintbooth2.SprintBootH2Application;

@ExtendWith(SpringExtension.class)
@Transactional
@SpringBootTest(classes = SprintBootH2Application.class)
public class EmployeeRepositoryTest {
   @Autowired
   private EmployeeRepository employeeRepository;

   @Test
   public void testFindById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findById(employee.getId()).get(
      assertEquals(employee.getId(), result.getId());
   }
   @Test
   public void testFindAll() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 1);
   }
   @Test
   public void testSave() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee found = employeeRepository.findById(employee.getId()).get()
      assertEquals(employee.getId(), found.getId());
   }
   @Test
   public void testDeleteById() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      employeeRepository.deleteById(employee.getId());
      List<Employee> result = new ArrayList<>();
      employeeRepository.findAll().forEach(e -> result.add(e));
      assertEquals(result.size(), 0);
   }
   private Employee getEmployee() {
      Employee employee = new Employee();
      employee.setId(1);
      employee.setName("Mahesh");
      employee.setAge(30);
      employee.setEmail("mahesh@test.com");
      return employee;
   }
   @Test
   public void testFindByName() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByName(employee.getName()).forEach(e -> resul
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByAge() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      List<Employee> result = new ArrayList<>();
      employeeRepository.findByAge(employee.getAge()).forEach(e -> result.
      assertEquals(result.size(), 1);
   }
   @Test
   public void testFindByEmail() {
      Employee employee = getEmployee();
      employeeRepository.save(employee);
      Employee result = employeeRepository.findByEmail(employee.getEmail
      assertNotNull(result);
   }
   @Test
   public void testFindAllSortedByName() {
      Employee employee = getEmployee();
      Employee employee1 = new Employee();
      employee1.setId(2);
      employee1.setName("Aarav");
      employee1.setAge(20);
      employee1.setEmail("aarav@test.com");
      employeeRepository.save(employee);
      employeeRepository.save(employee1);
      List<Employee> result = employeeRepository.findAllSortedByName();
      assertEquals(employee1.getName(), result.get(0).getName());
   }
   @Test
   public void testFindAllSortedByNameUsingNative() {
      Employee employee = getEmployee();
      Employee employee1 = new Employee();
      employee1.setId(2);
      employee1.setName("Aarav");
      employee1.setAge(20);
      employee1.setEmail("aarav@test.com");
      employeeRepository.save(employee);
      employeeRepository.save(employee1);
      List<Employee> result = employeeRepository.findAllSortedByNameUsingNa
      assertEquals(employee1.getName(), result.get(0).getName());
   }
}
```

Run the test cases

Output

Right Click on the file in eclipse and select **Run a JUnit Test** and verify the result.



Finished after 10.469 seconds

Runs: 9/9 Errors: 0 Failures: 0

```
EmployeeRepositoryTest [Runner: JUnit 5] (0.788
    testFindAll() (0.499 s)
    testSave() (0.031 s)
    testFindByName() (0.033 s)
    testFindAllSortedByName() (0.027 s)
    testFindAllSortedByNameUsingNative() (0.06
    testFindById() (0.024 s)
    testFindByEmail() (0.021 s)
    testDeleteById() (0.046 s)
    testFindByAge() (0.042 s)
```