I'm human

We're creating unit tests for Spring Boot's service layer using JUnit 5 and Mockito. Since we're using Spring Boot 3, all necessary dependencies are included with the `spring-boot-starter-test` dependency, which imports JUnit 5 and Mockito. We only need to add this dependency to our project. Let's test classes like `EmployeeManager` and `EmployeeDao`, where the manager interacts with the database via the DAO class. To mock these interactions, we use `@Mock` for individual classes and `@InjectMocks` to create a full object graph of dependencies. Here's an example of using Mockito annotations in our JUnit test: ```java public class TestEmployeeManager { @InjectMocks EmployeeManager manager; @Mock EmployeeDao dao; //tests } ``` We use `@ExtendWith(MockitoExtension.class)` to initialize mock objects automatically. Alternatively, we can manually initialize mocks using `MockitoAnnotations.openMocks()`. For example: ```java public class ServiceTests { @InjectMocks EmployeeService service; @Mock EmployeeRepository dao; @BeforeEach public void init() { MockitoAnnotations.openMocks(this); } //tests } ``` Let's see examples of writing JUnit tests for methods like `getAllEmployees()`, `getEmployeeById(int id)`, and `createEmployee()` using mock objects created with Mockito. We'll use `Mockito.when()` methods to create test stubs and verify interactions with our mocks. ```java @Test public void testGetAllEmployees() { //mocking dao's getAllEmployees() when(dao.getAllEmployees()).thenReturn(List.of()); //calling manager's getAllEmployees() List employees = manager.getAllEmployees(); //verifying interactions with mock dao verify(dao, times(1)).getAllEmployees(); } ``` Remember to import necessary classes and static methods from Mockito and JUnit. This is just a basic example; in real-world scenarios, you'd need to adapt this to your specific use case and requirements. Given article text here import org.mockito.junit.jupiter.MockitoExtension; import com.howtodoinjava.employees.dao.EmployeeRepository; import com.howtodoinja.employees.model.Employee; @ExtendWith(MockitoExtension.class) public class ServieTests { @InjectMocks EmployeeService servive; @Mock EmployeeRepository daoo; @Test void testFindAllEmployees() { List list = new ArrayList(); Employee empOne = new Employee("John", "John"); Employee empTwo = new Employee("Alex", "kolenchiski"); Employee empThree = new Employee("Steve", "Waugh"); list.add(empOne); list.add(empTwo); list.add(empThree); when(dao.findAll()).thenReturn(list); List empList = servive.findAll(); assertEquals(3, empList.size()); verify(dao, times(1)).findAll(); } @Test void testCreateOrSaveEmployee() { Employee employee = new Employee("Lokesh", "Gupta"); servive.save(employee); verify(dao, times(1)).save(employee); } At the well-known framework for unit testing in the Java ecosystem, bundled with JUnit, uses existing predicates for making assertions. Hamcrest is commonly used with JUnit and other testing frameworks for making assertions, specifically by using a single `assertThat` statement with appropriate matchers instead of numerous assert methods. Learn more about Hamcrest at Tutorials for JavaHamcrest are available here: Another framework that is commonly used in combination with JUnit and other testing frameworks is AssertJ, which provides a rich set of assertions, truly helpful error messages, improves test code readability, and is designed to be super easy to use within your favorite IDE. Spring boot starter test dependency internally provides assertj-core dependency so we don’t have to add assertj-core dependency manually in our Spring boot project. Read more about AssertJ at For working with JSON documents, a Java DSL for reading JSON documents is available, which always refers to the "root member object" as $ regardless of whether it is an object or array. The JsonPath expressions used are similar to XPath expressions used in combination with XML documents. Read more about JsonPath at When testing Spring MVC Controllers, Spring Boot provides a `@WebMvcTest` annotation that can be used to load only the specified controller and its dependencies without loading the entire application. This results in faster test execution times compared to using other testing approaches. In an application with multiple controllers, you can even ask for only one to be instantiated by using, for example, `@WebMvcTest(HomeController.class)`. To get started with Spring Boot, you can use the "spring initialize" command to create a new project and add dependencies such as Spring Web, Spring Data JPA, Lombok, and JUnit. The project can then be generated as a zip file, extracted, and imported into IntelliJ IDEA. When adding dependencies to your Spring Boot project, make sure to include the following: org.springframework.boot spring-boot-starter-data-jpa org.springframework.boot spring-boot-starter-web org.projectlombok lombok true org.springframework.boot spring-boot-starter-test test Next, you can create an Employee JPA entity with the following content: ```java import lombok.*; import javax.persistence.*; @Entity @Table(name = "employees") public class Employee { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private long id; @Column(name = "first_name", nullable = false) private String firstName; @Column(name = "last_name", nullable = false) private String lastName; @Column(nullable = false) private String email; } ``` Note that we are using Lombok annotations to reduce boilerplate code. The `@Entity` annotation is used to mark the class as a persistent Java class, and the `@Table` annotation is used to provide details about the table that this entity will be mapped to. The `@Id` annotation defines the primary key, and the `@GeneratedValue` annotation defines the primary key generation strategy. We've designated the primary key as an Auto Increment field. The @Column annotation is used to define column properties that will be mapped to the annotated field. This allows us to specify attributes such as name, length, nullable, and updateable. Next, we create the EmployeeRepository interface by extending JpaRepository:```java import net.javaguides.springboot.model.Employee; import org.springframework.data.jpa.repository.JpaRepository; public interface EmployeeRepository extends JpaRepository { } ``` We also define an EmployeeService interface with CRUD methods:```java import net.javaguides.springboot.model.Employee; import java.util.List; import java.util.Optional; public interface EmployeeService { Employee saveEmployee(Employee employee); List getAllEmployees(); Optional getEmployeeById(long id); Employee updateEmployee(Employee updatedEmployee); void deleteEmployee(long id); } ``` The EmployeeServiceImpl class implements the EmployeeService interface:```java import net.javaguides.springboot.exception.ResourceNotFoundException; import net.javaguides.springboot.model.Employee; import net/javaguides/springboot/repository/EmployeeRepository; import net/javaguides/springboot/service/EmployeeService; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service; public class EmployeeServiceImpl implements EmployeeService { private EmployeeRepository employeeRepository; public EmployeeServiceImpl(EmployeeRepository employeeRepository) { this.employeeRepository = employeeRepository; } // ... implementation details ... } ``` Finally, we create a REST API controller for CRUD operations:```java import net/javaguides/springboot/model/Employee; import net/javaguides/springboot/service/EmployeeService; @RestController @RequestMapping("/api/employees") public class EmployeeController { private EmployeeService employeeService; public EmployeeController(EmployeeService employeeService) { this.employeeService = employeeService; } @PostMapping public ResponseEntity createEmployee(Employee employee){ return ResponseEntity.status(HttpStatus.CREATED).body(employeeService.saveEmployee(employee)); } @GetMapping public List getAllEmployees(){ return employeeService.getAllEmployees(); } @GetMapping("/{id}") public ResponseEntity getEmployeeById(@PathVariable("id") long employeeId){ return employeeService.getEmployeeById(employeeId) .map(ResponseEntity::ok) .orElseGet(() -> ResponseEntity.notFound().build()); } @PutMapping("/{id}") public ResponseEntity updateEmployee(@PathVariable("id") long employeeId, Employee employee){ return employeeService.getEmployeeById(employeeId) .map(employeeToSave -> { return ResponseEntity.ok(employeeService.updateEmployee(employeeToSave)); }) .orElseGet(() -> ResponseEntity.notFound().build()); } @DeleteMapping("/{id}") public ResponseEntity deleteEmployee(@PathVariable("id") long employeeId){ employeeService.deleteEmployee(employeeId); return ResponseEntity.status(HttpStatus.NO_CONTENT).build(); } } ``` Given article text here ```java (@PathVariable("id")) long employeeId public ResponseEntity deleteEmployee(("id") long employeeId){ employeeService.deleteEmployee(employeeId); return new ResponseEntity("Employee deleted successfully!.", HttpStatus.OK); } (() -> { savedEmployee.setFirstName(employee.getFirstName()); savedEmployee.setLastName(employee.getLastName()); savedEmployee.setEmail(employee.getEmail()); Employee updatedEmployee = employeeService.updateEmployee(savedEmployee); return new ResponseEntity(updatedEmployee, HttpStatus.OK) }).orElseGet() -> ResponseEntity.notFound().build()); } ``` The code is testing a REST API using JUnit, covering scenarios such as retrieving an employee by ID, updating an employee, and handling invalid employee IDs. In the first test, `givenEmployeeId_whenGetEmployeeById_thenReturnEmployeeObject`, it sets up a precondition where an employee with ID 1L exists, then performs a GET request to retrieve this employee and verifies that the response status is OK, printing the response and checking that the JSON path for the employee's first name, last name, and email matches the expected values. The second test, `givenInvalidEmployeeId_whenGetEmployeeById_thenReturnEmpty`, tests the scenario where the employee ID is invalid, expecting a NOT FOUND status in response. For updating an employee, the test `givenUpdatedEmployee_whenUpdateEmployee_thenReturnUpdateEmployeeObject` checks that when an updated employee object is sent via PUT request, the API returns this updated object with the correct details. In the negative update scenario, `givenUpdatedEmployee_whenUpdateEmployee_thenReturn404`, it tests what happens when trying to update a non-existent employee, anticipating a 404 status code in response. These tests utilize Mockito to mock the behavior of the `employeeService`, allowing for isolation and control over the test environment by specifying what the service should return under different conditions. The `mockMvc` is used to perform the HTTP requests to the API endpoints being tested, such as `/api/employees/{id}` for GET and PUT operations, with the response expectations set using `andExpect` methods for status codes and JSON paths. Given article text here **Test Case for Delete Employee REST API** In the provided test cases, we can see a scenario where an employee is deleted from the database using the deleteEmployee method of the EmployeeService interface. The test begins by invoking the getArgument(0) method on the invocation object to retrieve the first argument (i.e., the employee ID). Next, it uses MockMvc to perform a PUT request on the /api/employees/{id} endpoint with the updated employee data. The response from this request is then verified using ResultActions. However, in the second test case, we are testing the deleteEmployee method of the EmployeeService interface. We use MockMvc to perform a DELETE request on the /api/employees/{id} endpoint with the specified employee ID. Here's an example of how you can write such a test: ```java @Test public void givenEmployeeId_whenDeleteEmployee_thenReturn200() throws Exception{ // given - precondition or setup long employeeId = 1L; willDoNothing().given(employeeService).deleteEmployee(employeeId); // when - action or the behaviour that we are going test ResultActions response = mockMvc.perform(delete("/api/employees/{id}", employeeId)); // then - verify the output response.andExpect(status().isOk()) .andDo(print()); } ``` In this example, we use MockMvc to perform a DELETE request on the /api/employees/{id} endpoint with the specified employee ID. The response from this request is then verified using ResultActions, and it should return an OK status code (200).

Junit 4 and mockito example. Mockito and junit. When to use mockito and when to use junit. Mockito examples. Junit when example. Junit test with mockito and spring boot example. Junit mockito testing in java. Junit 5 and mockito example. Mockito junit tutorial.