Junit & Mockito

✓ What is JUnit Test?

JUnit is a **Java testing framework** used to write and run **unit tests** — small, focused tests that verify the behavior of individual pieces of code (like methods or classes).

A **unit test** checks that a **single unit** (usually a method) of your code works as expected.

For example:

- If a method is supposed to return "Hello" — the unit test checks if it really does that.
- If a method throws an exception for invalid input — the test verifies that too.� Write **automated tests** for Java code.
- Check code correctness without manually testing.
- Improve code quality.

✓ Why Unit Testing is Important in Software Development

1. **Finds bugs early**
→ It helps catch mistakes before the app goes live.

2. **Makes code easy to change**
→ You can update code and be sure nothing breaks.

3. **Saves time**
→ No need to test everything manually each time.

4. **Keeps old features working**
→ Tests make sure new changes don't break existing code.

5. **Improves code quality**
→ Forces you to write cleaner and more organized code.

6. **Builds confidence**
→ You know your code works as expected.

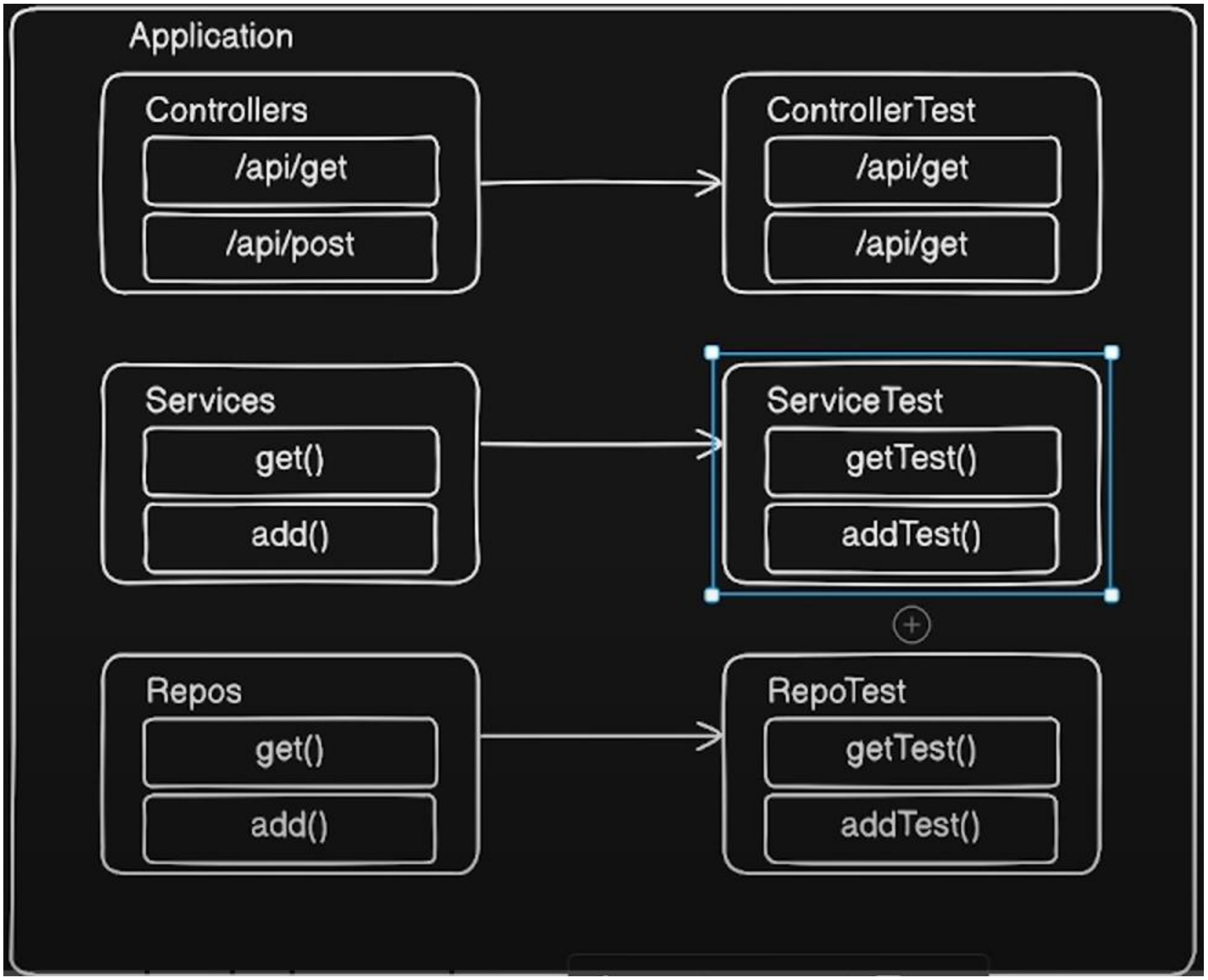
✓ Maven Dependency

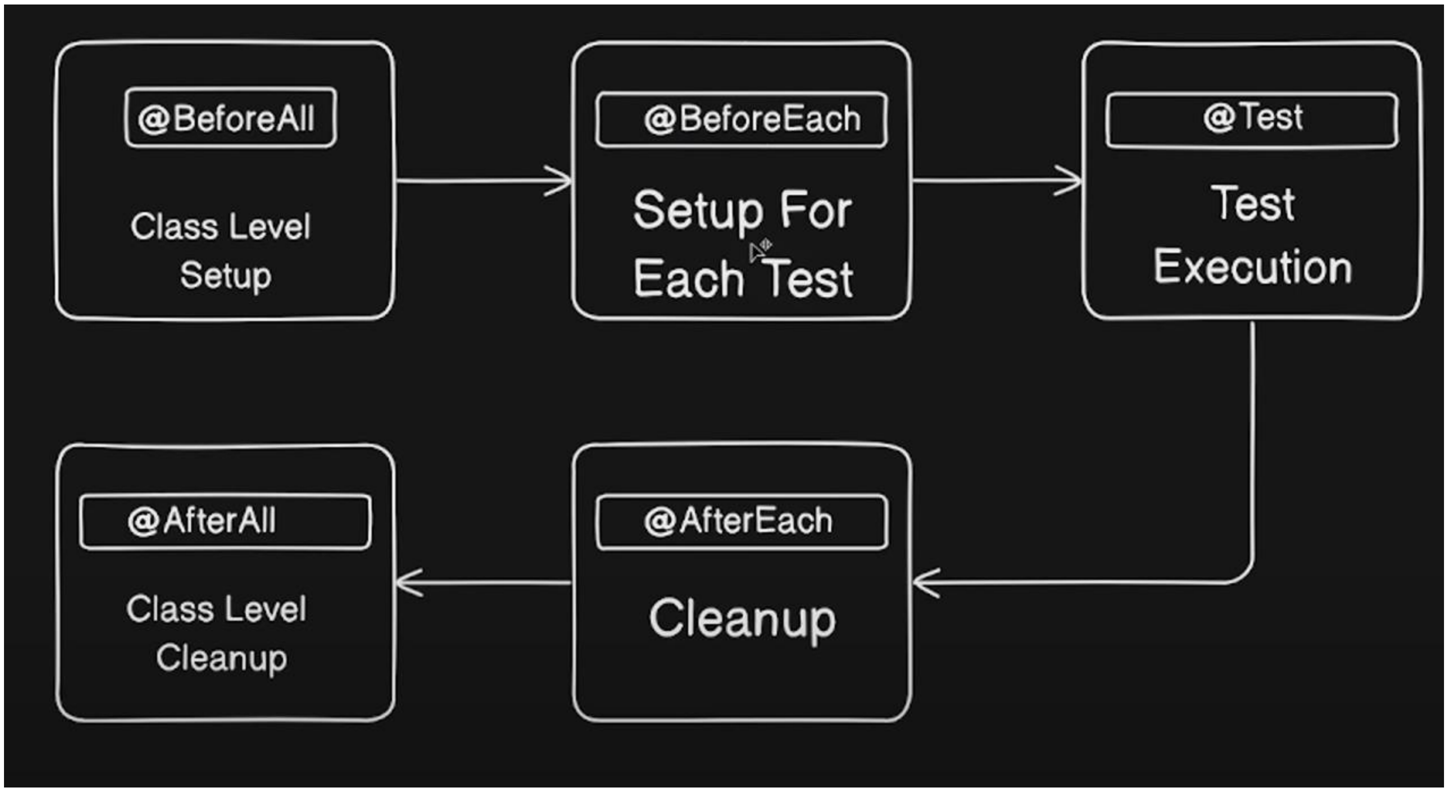
Add this to your pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

This includes:

- **JUnit 5** (Jupiter)
- **Mockito** (for mocking)
- **AssertJ** (fluent assertions)
- **Spring Test** (Spring-specific testing utilities)
- **Hamcrest** (for matchers)





✅ JUnit Annotations

1. `@BeforeAll`

- **What**: Runs once **before all test methods** in the class.
- **Why**: Used to set up expensive resources (e.g., database, server).
- **When**: Runs **once**, before everything. Must be `static`.
- **Example**:

```java
@BeforeAll
static void setupAll() {
    System.out.println("Executed once before all tests.");
}
```

2. `@BeforeEach`

- **What**: Runs **before each test method**.
- **Why**: Initialize or reset variables before every test.
- **When**: Runs **before every** `@Test` method.
- **Example**:

```java
@BeforeEach
void setup() {
    System.out.println("Executed before each test.");
}
```

3. `@Test`

- **What**: Marks a method as a **test case**.
- **Why**: To define a method that JUnit will execute as a test.
- **Example**:

```java
@Test
void testAddition() {
    assertEquals(4, 2 + 2);
}
```

4. `@AfterEach`

- **What**: Runs **after each test method**.
- **Why**: Used for cleanup after each test (e.g., closing resources).
- **Example**:

```java
@AfterEach
void tearDown() {
    System.out.println("Executed after each test.");
}
```

5. `@AfterAll`

- **What**: Runs once **after all test methods.**
- **Why**: Cleanup heavy resources once testing is done.
- **When**: Runs once. Must be `static`.
- **Example**:

```java
@AfterAll
static void tearDownAll() {
    System.out.println("Executed once after all tests.");
}
```

✅ Mockito Methods

1. `any()`

- **What**: A matcher used to allow any value of a specific type.
- **Why**: When you don't care about the exact input in a mock call.
- **Example**:

```java
when(userService.findUserById(any(Long.class))).thenReturn(mockUser);
```

2. `verify()`

- **What**: Verifies that a specific method was called.
- **Why**: To assert that your mocked method was invoked.
- **Example**:

```java
verify(userService).saveUser(any(User.class));
```

3. `times(n)`

- **What**: Specifies **how many times** a method should have been called.
- **Why**: Combine with `verify()` to control call frequency.
- **Example**:

```java
verify(userService, times(2)).deleteUser(anyLong());
```

✅ Other Common Mockito Matchers

Matcher	Purpose	Example
`eq(value)`	Matches exact value	`eq("Hello")`
`anyString()`	Matches any `String`	`when(service.greet(anyString()))`
`anyInt()`	Matches any `int`	`verify(service).process(anyInt())`
`isNull()`	Matches `null`	`when(repo.find(isNull())).thenReturn(null)`
`notNull()`	Matches non-null values	`assertNotNull(result)` (JUnit assertion)

✅ When to Use Which?

Task	JUnit / Mockito Tool
Setup common objects once	`@BeforeAll`
Reset data before each test	`@BeforeEach`
Write actual test	`@Test`
Clean up after each test	`@AfterEach`
Clean up heavy resources at end	`@AfterAll`
Mock method return based on input	`when(...).thenReturn(...)`
Check if a method was called	`verify(...)`
Match flexible method parameters	`any(), anyInt(), anyString()` etc.
Verify method call count	`verify(..., times(n))`

✅ JUnit + Mockito Setup in Spring Boot Testing

✅ Complete List of JUnit + Mockito + Spring Test Annotations

Annotation	Category	Description
`@Test`	JUnit	Marks a method as a test case.
`@BeforeEach`	JUnit	Runs before each test method. Used for test setup.
`@AfterEach`	JUnit	Runs after each test method. Used for cleanup.
`@BeforeAll`	JUnit	Runs once before all tests. Must be `static`.
`@AfterAll`	JUnit	Runs once after all tests. Must be `static`.
`@DisplayName("...")`	JUnit	Sets a readable name for the test.
`@Disabled`	JUnit	Temporarily disables a test method or class.
`@RepeatedTest(n)`	JUnit	Repeats a test method `n` times.
`@Timeout(n)`	JUnit	Fails a test if it runs longer than `n` seconds.

Annotation	Category	Description
`@ExtendWith(MockitoExtension.class)`	Mockito + JUnit	Enables Mockito in JUnit 5. Required for using `@Mock`, `@InjectMocks`, etc.
`@Mock`	Mockito	Creates a mock instance of a class or interface.
`@InjectMocks`	Mockito	Injects mock dependencies into the class under test.
`@Spy`	Mockito	Wraps a real object and allows partial mocking.
`@Captor`	Mockito	Captures arguments passed to a mocked method.
`@MockBean`	Spring Boot	Creates a mock Spring bean for injection into the application context (used in controller tests).

Annotation	Category	Description
`@WebMvcTest(Class.class)`	Spring Boot (Web Layer)	Sets up a Spring test context focused only on the web layer (controllers).
`@DataJpaTest`	Spring Boot (JPA Layer)	Configures in-memory DB & tests only JPA repositories.
`@SpringBootTest`	Spring Boot (Full Integration)	Loads the full application context for integration testing.
`@AutoConfigureMockMvc`	Spring Boot	Enables and injects `MockMvc` in full SpringBootTest.
`@RestClientTest`	Spring Boot	Tests REST client beans like `RestTemplate` or `WebClient`.
`@TestConfiguration`	Spring Boot	Define test-specific Spring beans.
`@TestPropertySource`	Spring Boot	Overrides properties for testing environment.

Component	Category	Description
`MockMvc`	Spring Boot	Used to perform HTTP requests and test controllers without starting a server.
`ObjectMapper`	Jackson	Used to convert between Java objects and JSON.

✅ Example Quick Summary for Each Group:

🧪 JUnit Core:

- `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, `@AfterAll`, `@DisplayName`, `@Disabled`

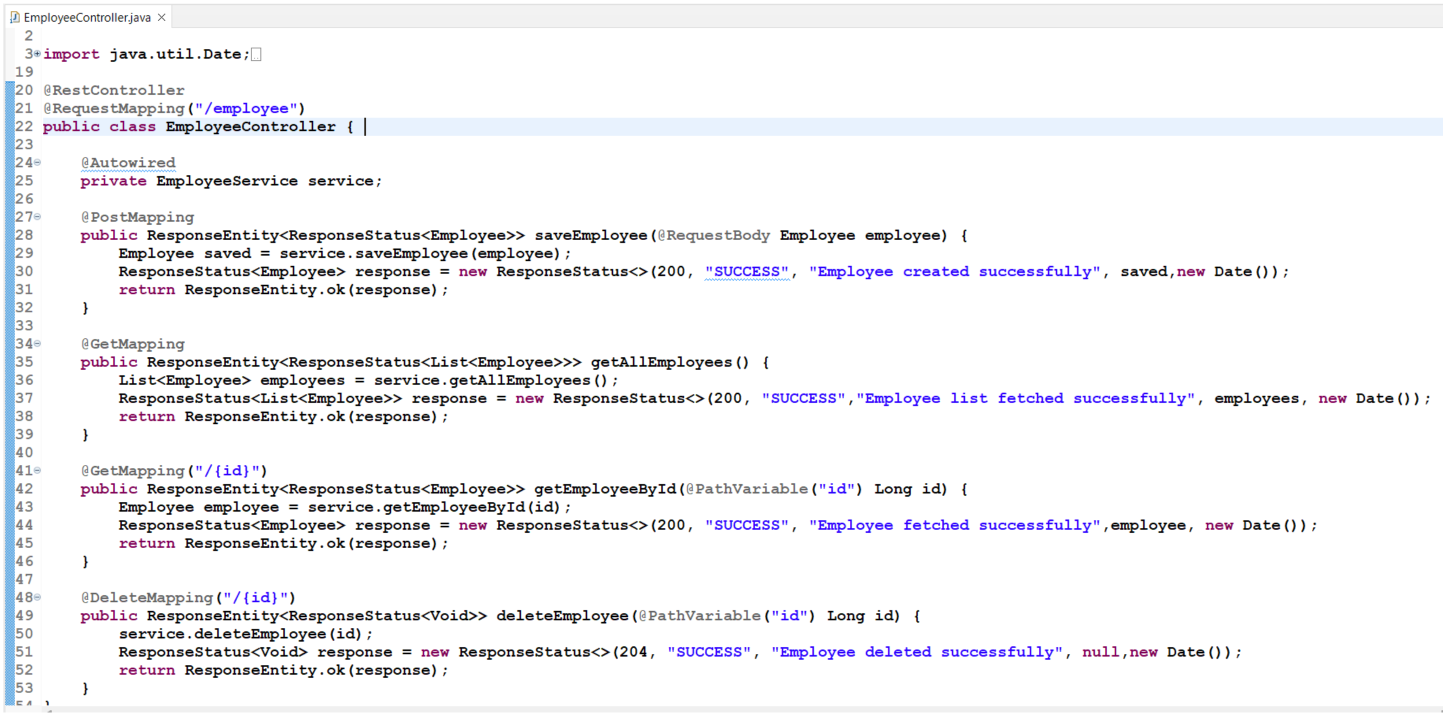
🧪 Mockito:

- `@ExtendWith(MockitoExtension.class)`, `@Mock`, `@InjectMocks`, `@Spy`, `@Captor`

🧪 Spring Boot Testing:

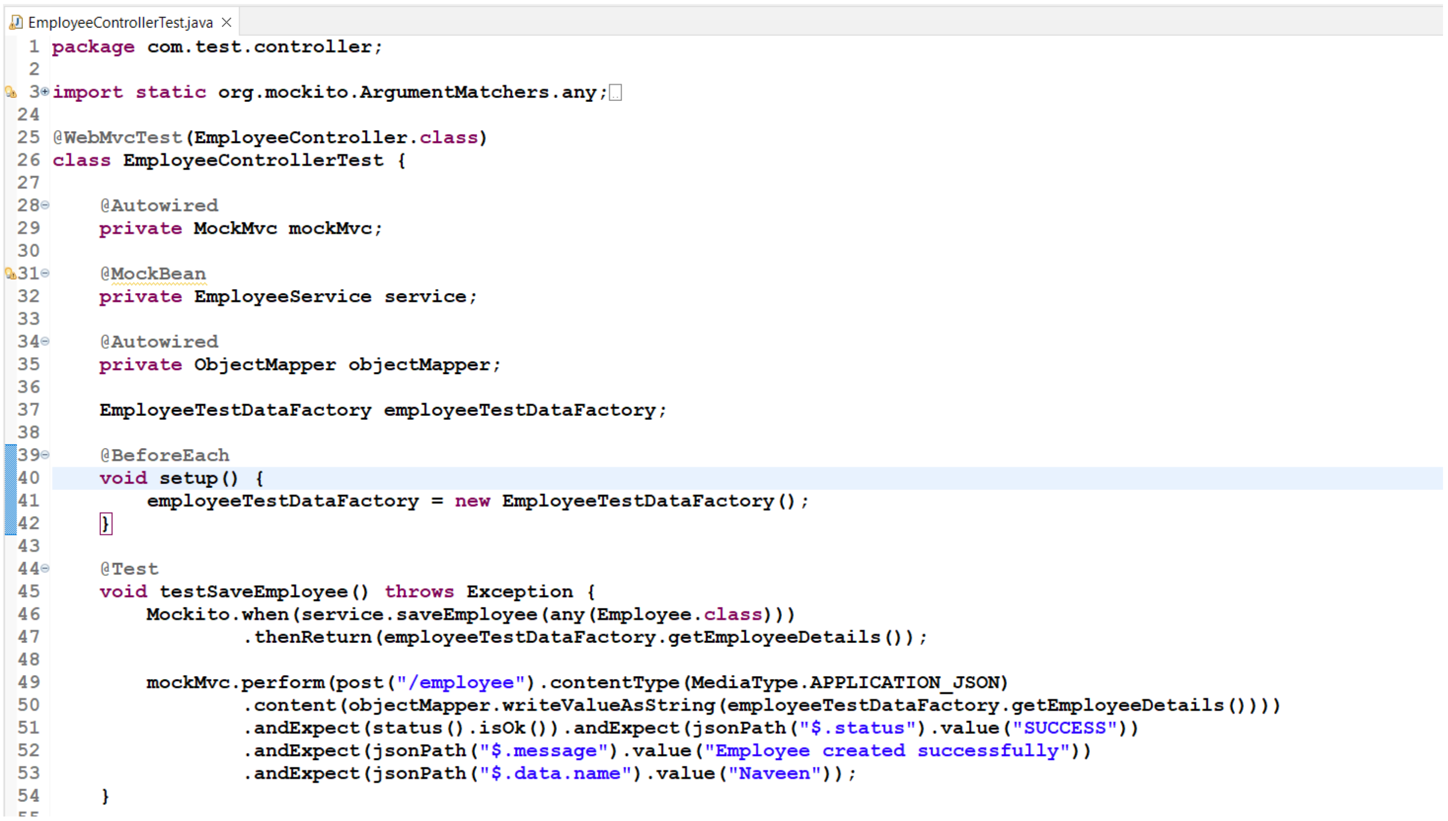
- `@WebMvcTest`, `@MockBean`, `@SpringBootTest`, `@DataJpaTest`, `@AutoConfigureMockMvc`, `@TestConfiguration`, `@TestPropertySource`

Controller Layer:

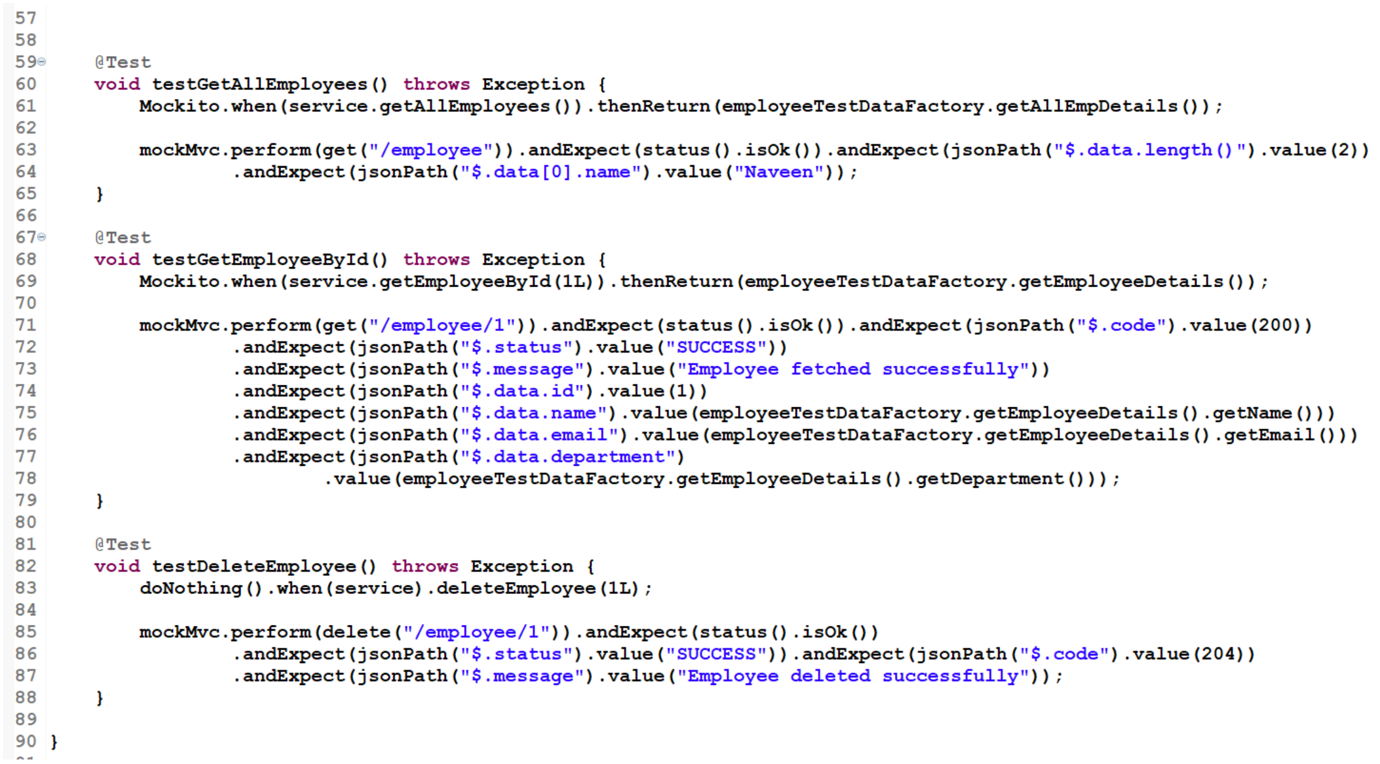


```java

import java.util.Date;

@RestController
@RequestMapping("/employee")
public class EmployeeController {

    @Autowired
    private EmployeeService service;

    @PostMapping
    public ResponseEntity<ResponseStatus<Employee>> saveEmployee(@RequestBody Employee employee) {
        Employee saved = service.saveEmployee(employee);
        ResponseStatus<Employee> response = new ResponseStatus<>(200, "SUCCESS", "Employee created successfully", saved,new Date());
        return ResponseEntity.ok(response);
    }

    @GetMapping
    public ResponseEntity<ResponseStatus<List<Employee>>> getAllEmployees() {
        List<Employee> employees = service.getAllEmployees();
        ResponseStatus<List<Employee>> response = new ResponseStatus<>(200, "SUCCESS","Employee list fetched successfully", employees, new Date());
        return ResponseEntity.ok(response);
    }

    @GetMapping("/{id}")
    public ResponseEntity<ResponseStatus<Employee>> getEmployeeById(@PathVariable("id") Long id) {
        Employee employee = service.getEmployeeById(id);
        ResponseStatus<Employee> response = new ResponseStatus<>(200, "SUCCESS", "Employee fetched successfully",employee, new Date());
        return ResponseEntity.ok(response);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<ResponseStatus<Void>> deleteEmployee(@PathVariable("id") Long id) {
        service.deleteEmployee(id);
        ResponseStatus<Void> response = new ResponseStatus<>(204, "SUCCESS", "Employee deleted successfully", null,new Date());
        return ResponseEntity.ok(response);
    }
```

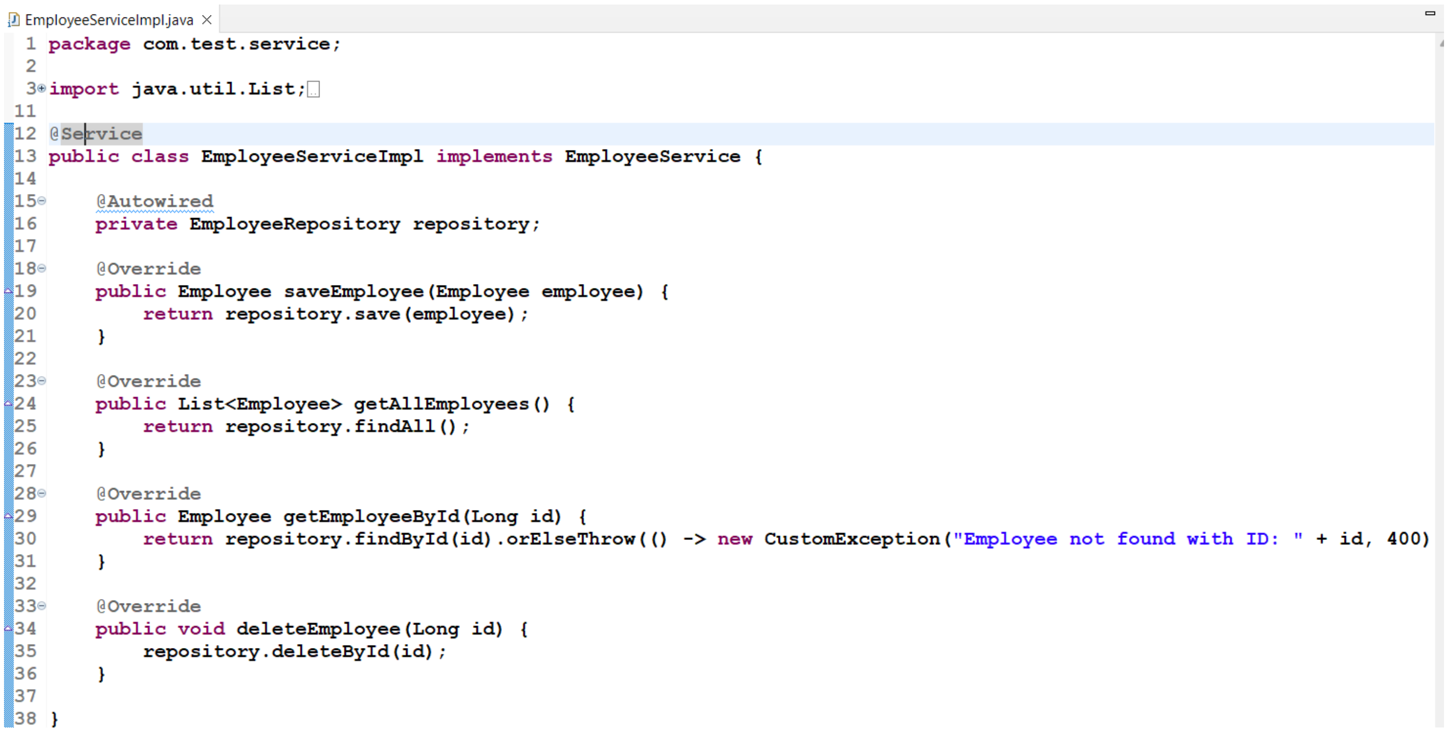
Controller Layer Test:



```java
package com.test.controller;

import static org.mockito.ArgumentMatchers.any;

@WebMvcTest(EmployeeController.class)
class EmployeeControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private EmployeeService service;

    @Autowired
    private ObjectMapper objectMapper;

    EmployeeTestDataFactory employeeTestDataFactory;

    @BeforeEach
    void setup() {
        employeeTestDataFactory = new EmployeeTestDataFactory();
    }

    @Test
    void testSaveEmployee() throws Exception {
        Mockito.when(service.saveEmployee(any(Employee.class)))
                .thenReturn(employeeTestDataFactory.getEmployeeDetails());

        mockMvc.perform(post("/employee").contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(employeeTestDataFactory.getEmployeeDetails())))
                .andExpect(status().isOk()).andExpect(jsonPath("$.status").value("SUCCESS"))
                .andExpect(jsonPath("$.message").value("Employee created successfully"))
                .andExpect(jsonPath("$.data.name").value("Naveen"));
    }
```

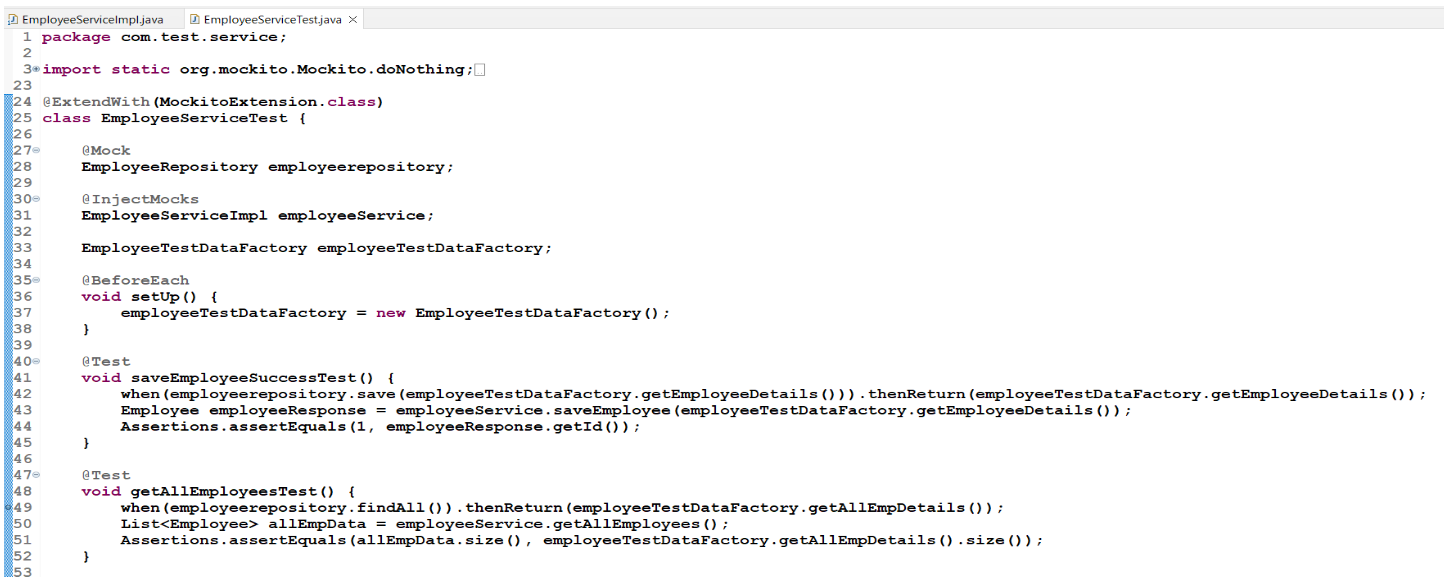

```java


    @Test
    void testGetAllEmployees() throws Exception {
        Mockito.when(service.getAllEmployees()).thenReturn(employeeTestDataFactory.getAllEmpDetails());

        mockMvc.perform(get("/employee")).andExpect(status().isOk()).andExpect(jsonPath("$.data.length()").value(2))
                .andExpect(jsonPath("$.data[0].name").value("Naveen"));
    }

    @Test
    void testGetEmployeeById() throws Exception {
        Mockito.when(service.getEmployeeById(1L)).thenReturn(employeeTestDataFactory.getEmployeeDetails());

        mockMvc.perform(get("/employee/1")).andExpect(status().isOk()).andExpect(jsonPath("$.code").value(200))
                .andExpect(jsonPath("$.status").value("SUCCESS"))
                .andExpect(jsonPath("$.message").value("Employee fetched successfully"))
                .andExpect(jsonPath("$.data.id").value(1))
                .andExpect(jsonPath("$.data.name").value(employeeTestDataFactory.getEmployeeDetails().getName()))
                .andExpect(jsonPath("$.data.email").value(employeeTestDataFactory.getEmployeeDetails().getEmail()))
                .andExpect(jsonPath("$.data.department")
                        .value(employeeTestDataFactory.getEmployeeDetails().getDepartment()));
    }

    @Test
    void testDeleteEmployee() throws Exception {
        doNothing().when(service).deleteEmployee(1L);

        mockMvc.perform(delete("/employee/1")).andExpect(status().isOk())
                .andExpect(jsonPath("$.status").value("SUCCESS")).andExpect(jsonPath("$.code").value(204))
                .andExpect(jsonPath("$.message").value("Employee deleted successfully"));
    }

}
```

Service Layer:

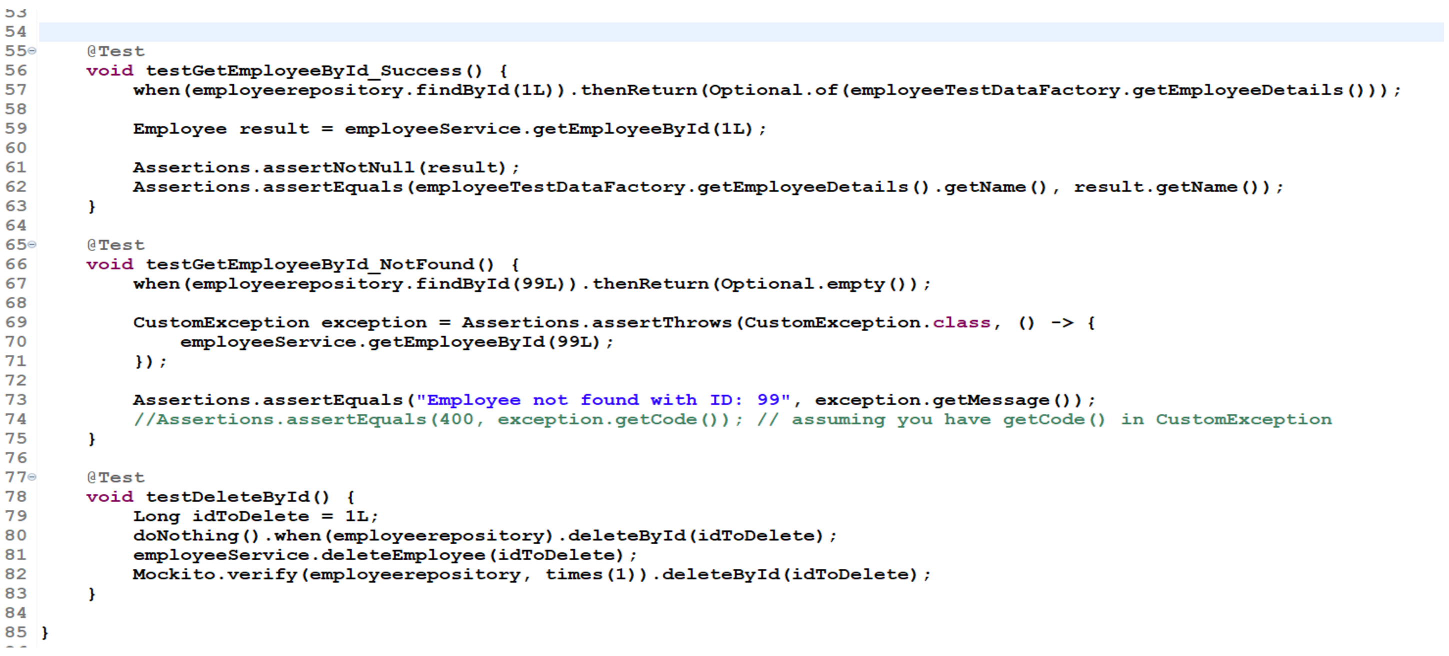

EmployeeServiceImpl.java

```java
package com.test.service;

import java.util.List;

@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository repository;

    @Override
    public Employee saveEmployee(Employee employee) {
        return repository.save(employee);
    }

    @Override
    public List<Employee> getAllEmployees() {
        return repository.findAll();
    }

    @Override
    public Employee getEmployeeById(Long id) {
        return repository.findById(id).orElseThrow(() -> new CustomException("Employee not found with ID: " + id, 400)
    }

    @Override
    public void deleteEmployee(Long id) {
        repository.deleteById(id);
    }

}
```

Service Layer Test:

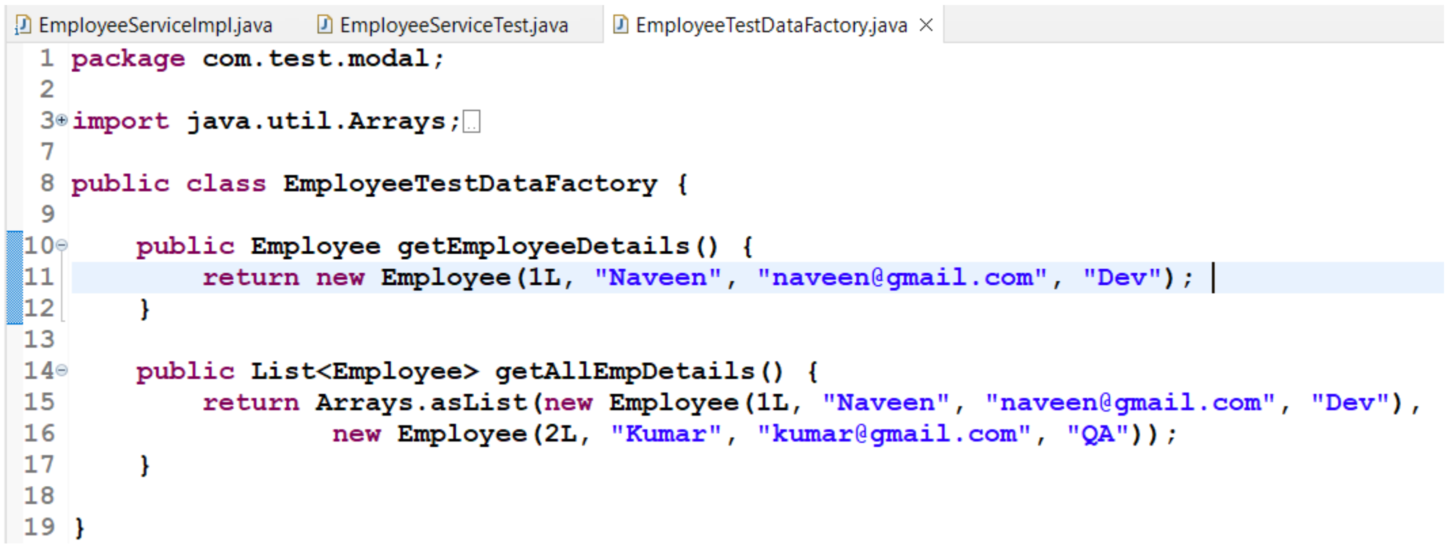


```java
package com.test.service;

import static org.mockito.Mockito.doNothing;
@ExtendWith(MockitoExtension.class)
class EmployeeServiceTest {

    @Mock
    EmployeeRepository employeerepository;

    @InjectMocks
    EmployeeServiceImpl employeeService;

    EmployeeTestDataFactory employeeTestDataFactory;

    @BeforeEach
    void setUp() {
        employeeTestDataFactory = new EmployeeTestDataFactory();
    }

    @Test
    void saveEmployeeSuccessTest() {
        when(employeerepository.save(employeeTestDataFactory.getEmployeeDetails())).thenReturn(employeeTestDataFactory.getEmployeeDetails());
        Employee employeeResponse = employeeService.saveEmployee(employeeTestDataFactory.getEmployeeDetails());
        Assertions.assertEquals(1, employeeResponse.getId());
    }

    @Test
    void getAllEmployeesTest() {
        when(employeerepository.findAll()).thenReturn(employeeTestDataFactory.getAllEmpDetails());
        List<Employee> allEmpData = employeeService.getAllEmployees();
        Assertions.assertEquals(allEmpData.size(), employeeTestDataFactory.getAllEmpDetails().size());
    }

```



```java


    @Test
    void testGetEmployeeById_Success() {
        when(employeerepository.findById(1L)).thenReturn(Optional.of(employeeTestDataFactory.getEmployeeDetails()));

        Employee result = employeeService.getEmployeeById(1L);

        Assertions.assertNotNull(result);
        Assertions.assertEquals(employeeTestDataFactory.getEmployeeDetails().getName(), result.getName());
    }

    @Test
    void testGetEmployeeById_NotFound() {
        when(employeerepository.findById(99L)).thenReturn(Optional.empty());

        CustomException exception = Assertions.assertThrows(CustomException.class, () -> {
            employeeService.getEmployeeById(99L);
        });

        Assertions.assertEquals("Employee not found with ID: 99", exception.getMessage());
        //Assertions.assertEquals(400, exception.getCode()); // assuming you have getCode() in CustomException
    }

    @Test
    void testDeleteById() {
        Long idToDelete = 1L;
        doNothing().when(employeerepository).deleteById(idToDelete);
        employeeService.deleteEmployee(idToDelete);
        Mockito.verify(employeerepository, times(1)).deleteById(idToDelete);
    }

}
```

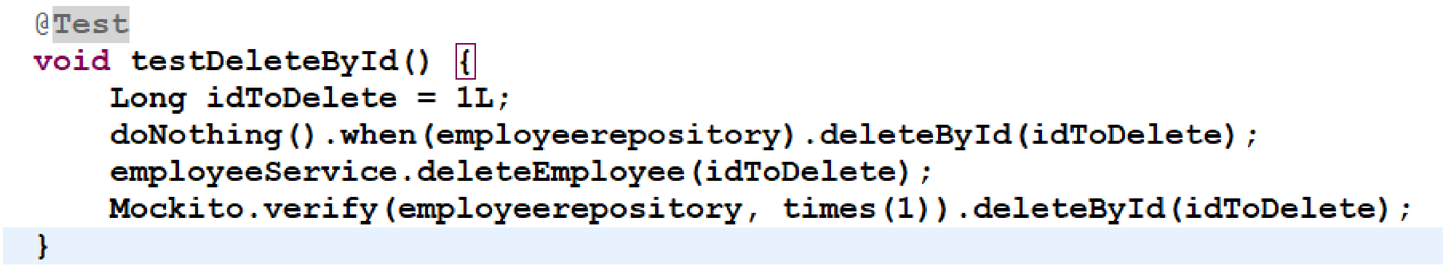
Test Data:



```java
package com.test.modal;

import java.util.Arrays;

public class EmployeeTestDataFactory {

    public Employee getEmployeeDetails() {
        return new Employee(1L, "Naveen", "naveen@gmail.com", "Dev");
    }

    public List<Employee> getAllEmpDetails() {
        return Arrays.asList(new Employee(1L, "Naveen", "naveen@gmail.com", "Dev"),
                new Employee(2L, "Kumar", "kumar@gmail.com", "QA"));
    }

}
```

How to test void methods?

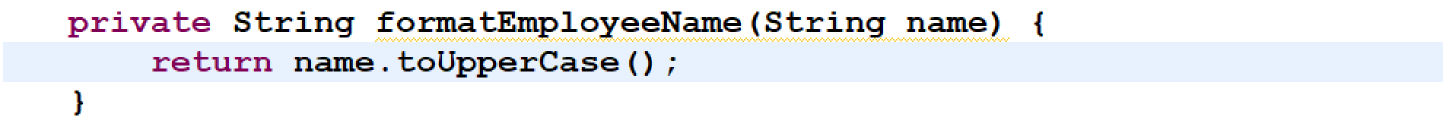
- Void methods don’t return a value.
- So, you check **what they do**
- Use Mockito’s verify() to confirm if other methods were called (e.g., deleting from a repository).
- *For void methods, test their **side effects** or **interactions**..*



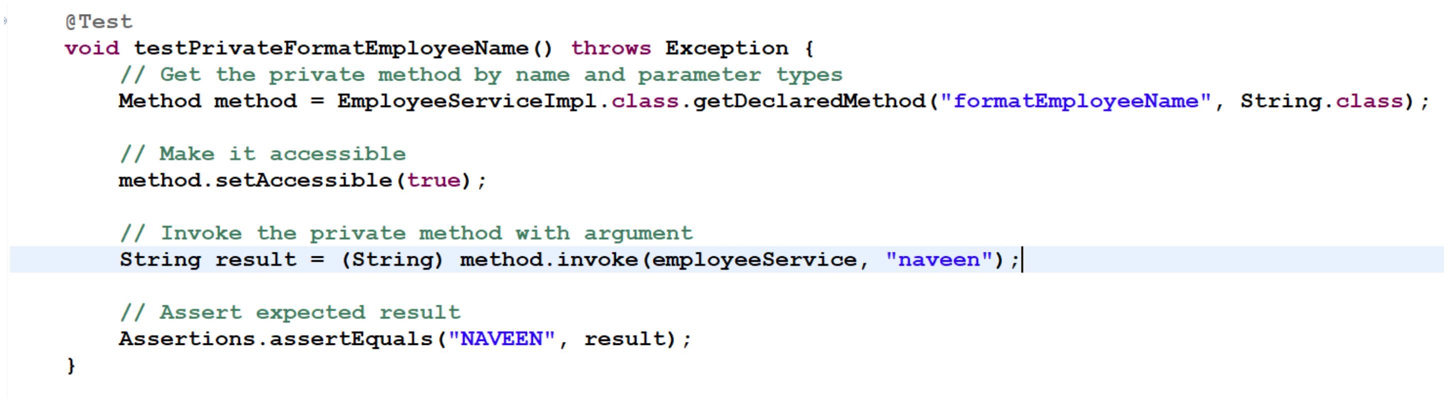
```
@Test
void testDeleteById() {
    Long idToDelete = 1L;
    doNothing().when(employeerepository).deleteById(idToDelete);
    employeeService.deleteEmployee(idToDelete);
    Mockito.verify(employeerepository, times(1)).deleteById(idToDelete);
}
```

How to test a private method using Reflection API?

- **Private methods are not accessible directly** from your test class.
- You can use Java’s **Reflection API** to access and call private methods.
- Steps:
  1.Use **getDeclaredMethod()** to get the private method by name and parameter.
  2.Use **setAccessible(true)** to allow access to the private method.
  3.Use **invoke()** to call the private method with any arguments.
  4.If the method returns something, capture and assert the result.
- This way you can test private methods, but it’s **better to test through public methods** when possible.



```
private String formatEmployeeName(String name) {
    return name.toUpperCase();
}
```



```
@Test
void testPrivateFormatEmployeeName() throws Exception {
    // Get the private method by name and parameter types
    Method method = EmployeeServiceImpl.class.getDeclaredMethod("formatEmployeeName", String.class);

    // Make it accessible
    method.setAccessible(true);

    // Invoke the private method with argument
    String result = (String) method.invoke(employeeService, "naveen");

    // Assert expected result
    Assertions.assertEquals("NAVEEN", result);
}
```