TDD with JUnit

Test-Driven Development (TDD)

- TDD Life Cycle
- TDD vs DLP (Debug Later Programming)
- Why TDD is matter
- Unit Testing with F.I.R.S.T
- Code and Test Coverage
- Structure of good unit testing (GUT)

Unit Testing with JUnit

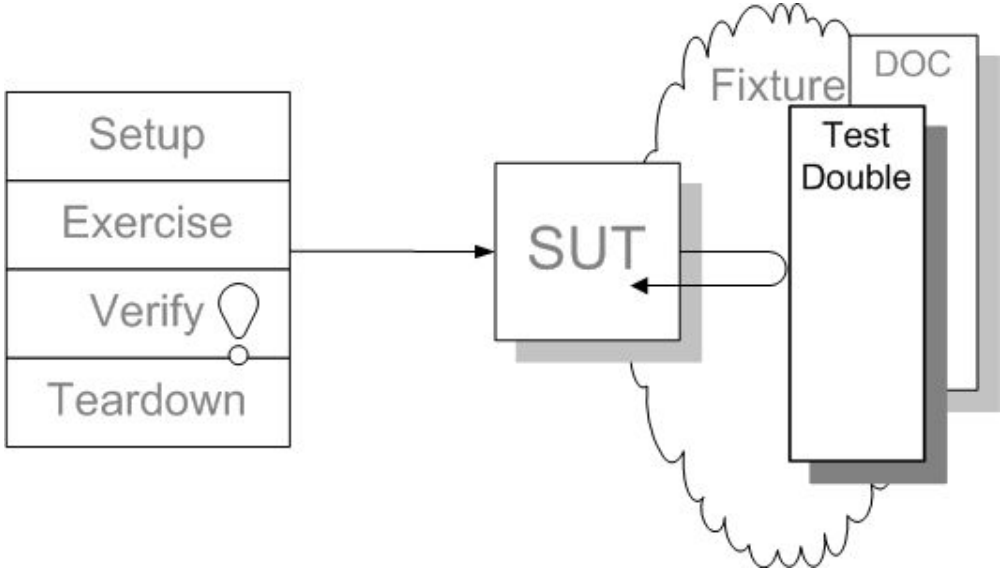
- JUnit lifeCycle
- Assertion
- Data-Driven Test with JUnit
- JUnit features
- Timeout
- Conditional
- **Category**
- **Suite**
- Running testing

Test Double

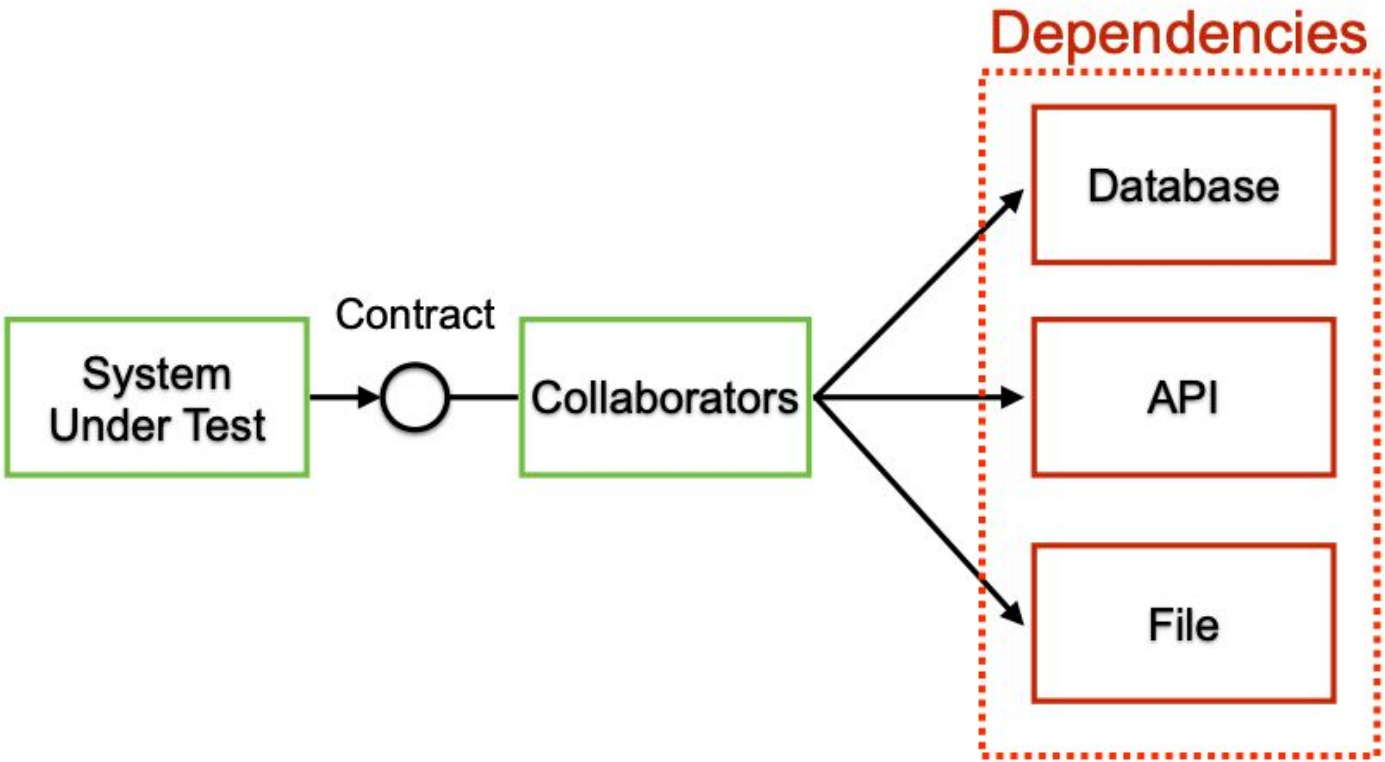
- **Dummy**
- **Stub**
- **Spy**
- **Mock**
- **Fake**

Test Double

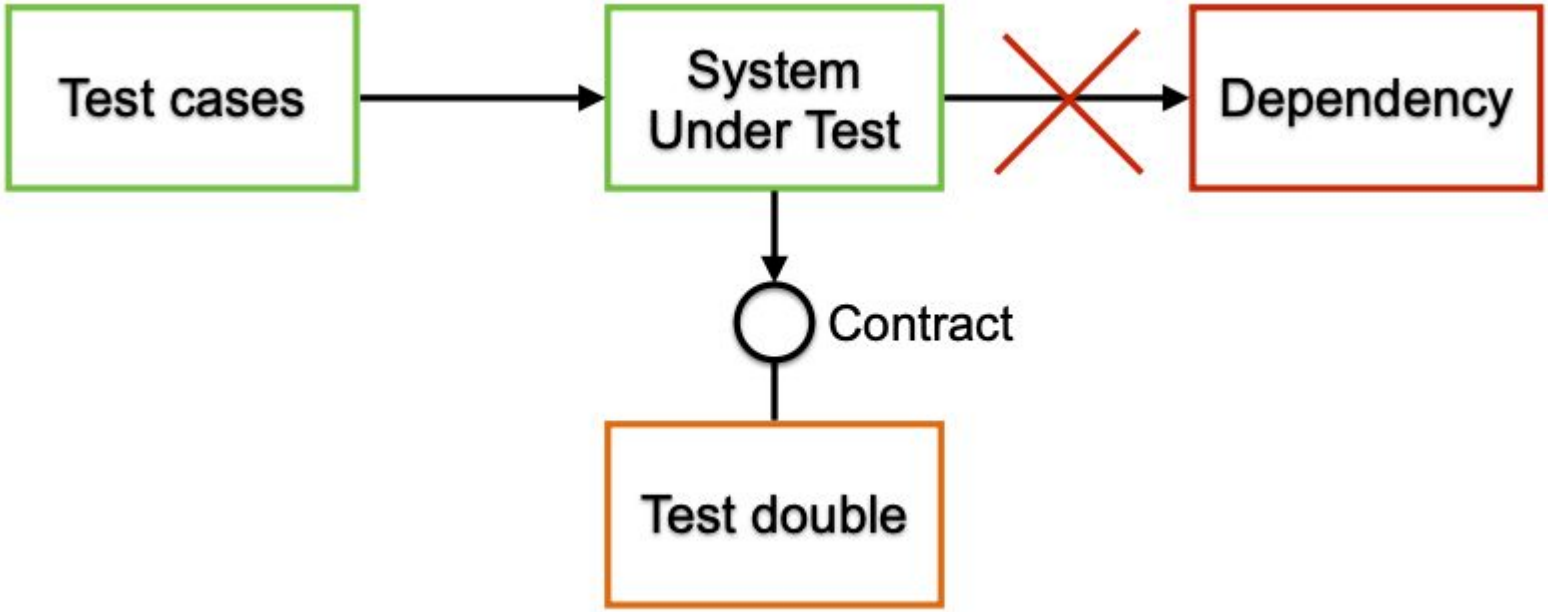
How can we verify logic independently ?
How can we avoid Slow tests ?



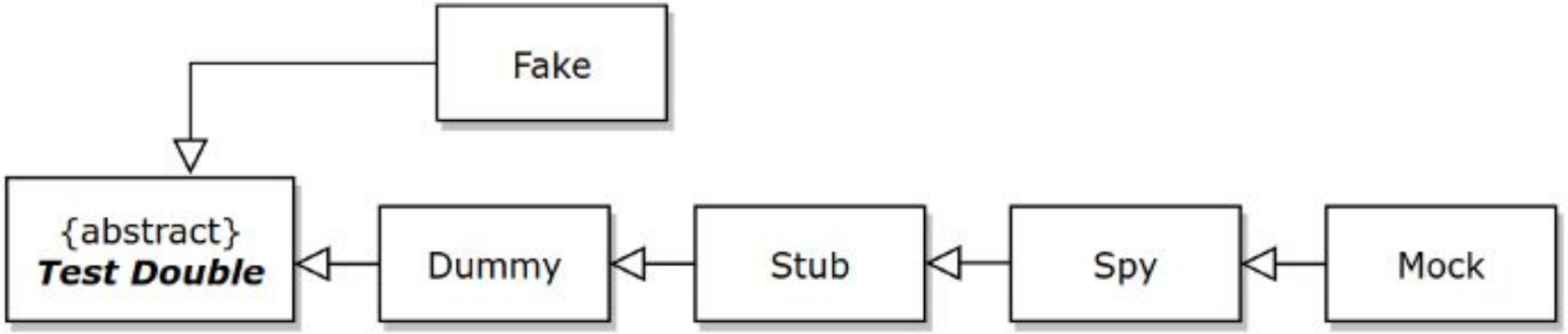
Production Code



With Test double



The five types of Test Doubles



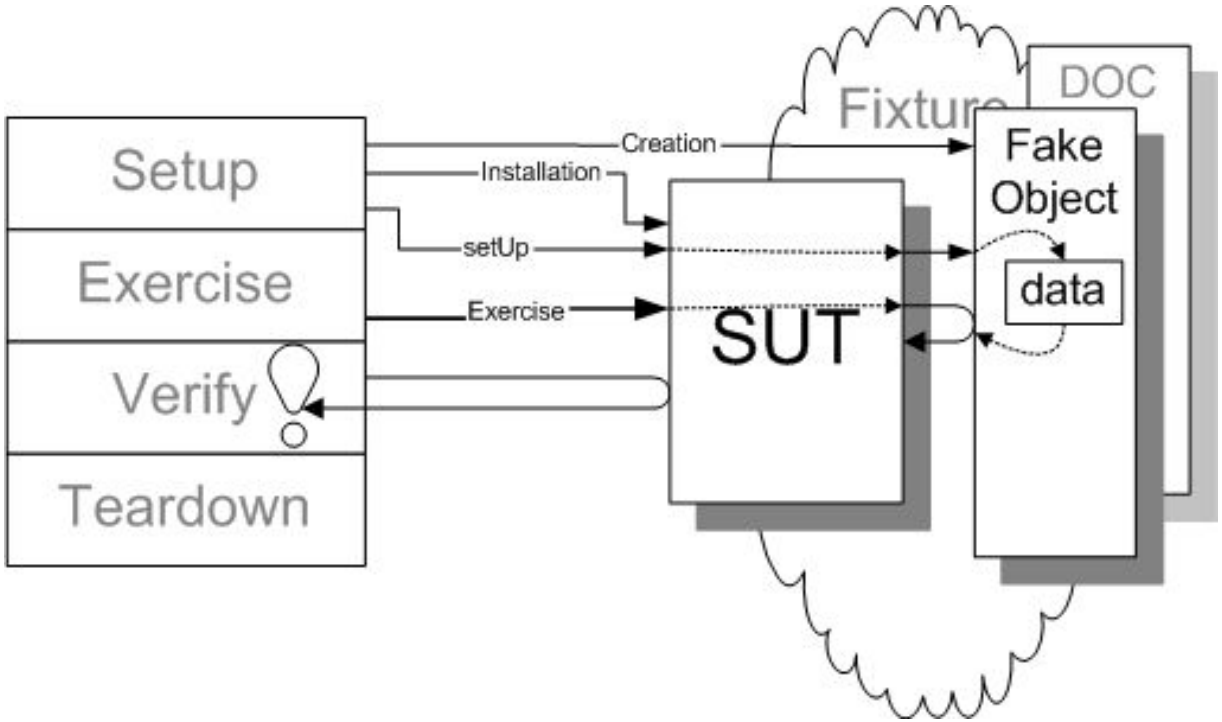
A Test Double is an object that can stand-in for a real object in a test, similar to how a stunt double stands in for an actor in a movie.

https://martinfowler.com/bliki/TestDouble.html

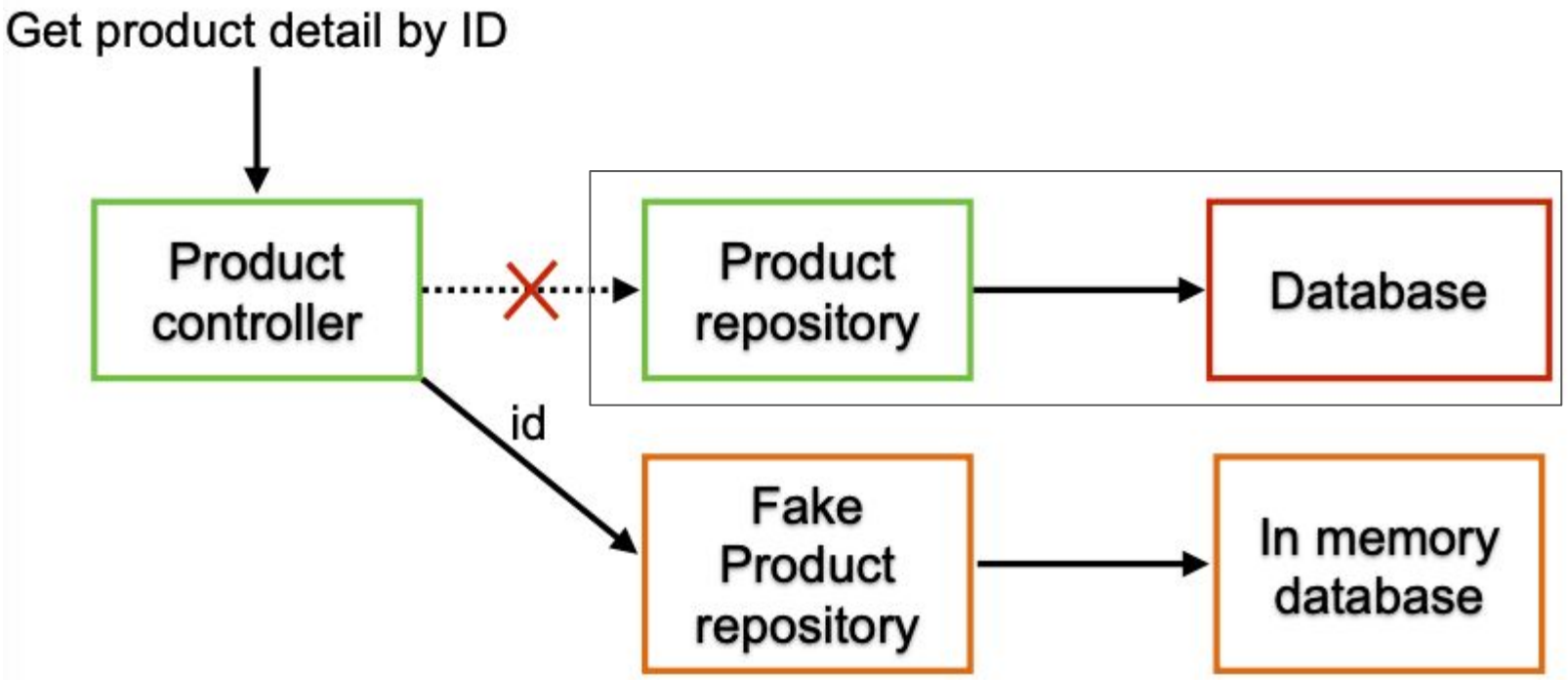
Fake

Working with implementation but take some shortcut
Not suitable for production
Use with read and write operations

E.g. In-memory database, Fake API server

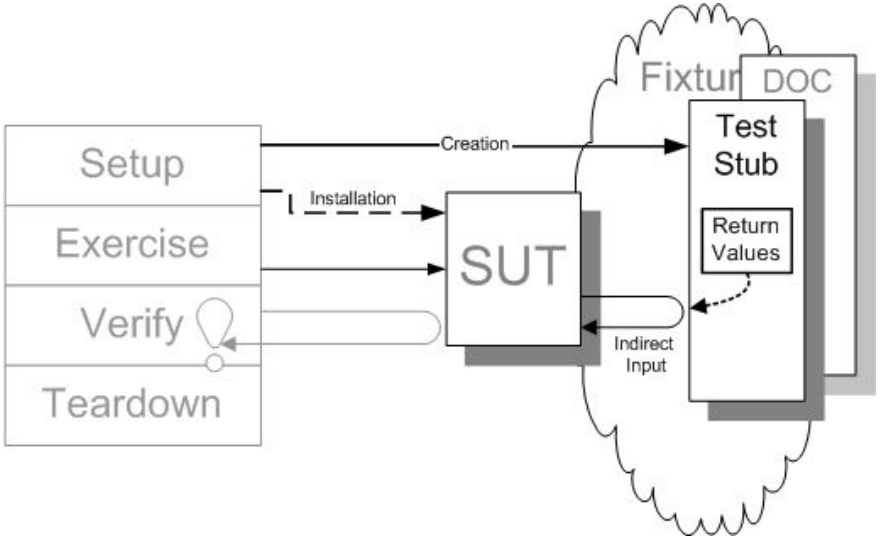

Setup
Exercise
Verify
Teardown
Creation
Installation
setUp
Exercise
Fixture
DOC
Fake
Object
data
SUT

Fake

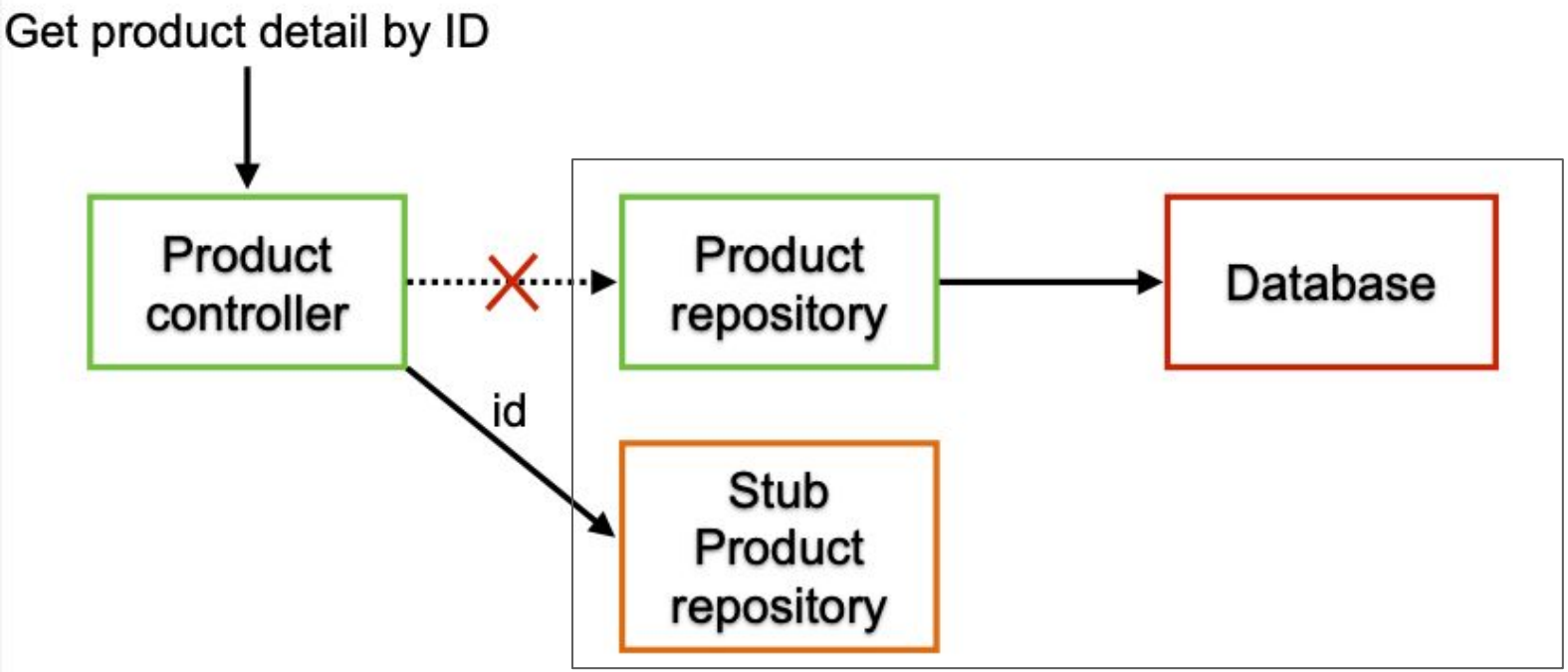

Get product detail by ID
Product controller
Product repository
Database
id
Fake Product repository
In memory database

Stub

Provide answers to calls made during the test
A double with hardcode return values



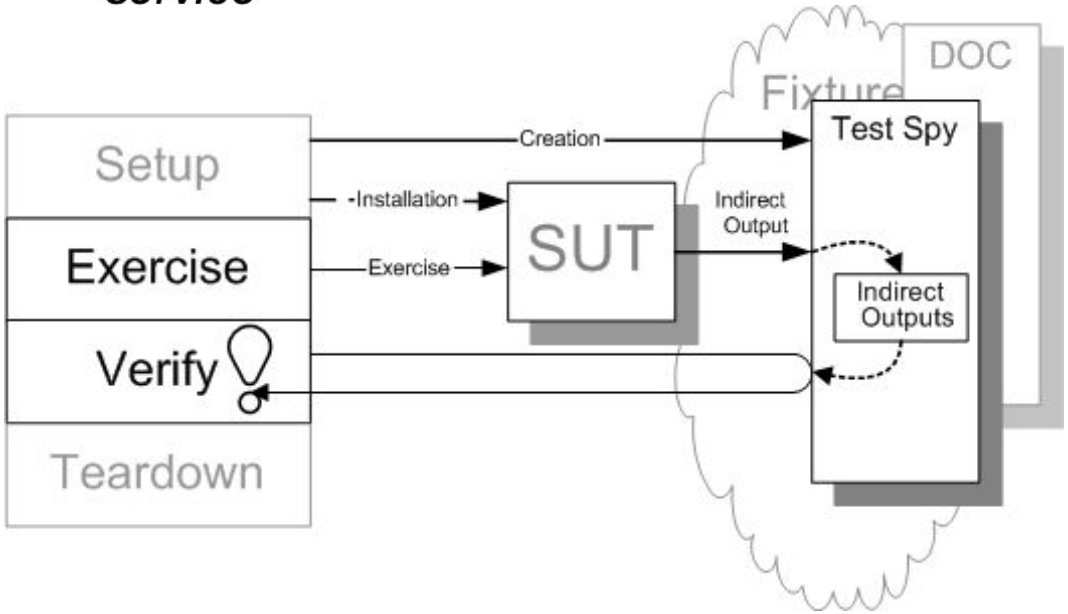
Stub


Get product detail by ID
Product controller
id
Product repository
Database
Stub Product repository

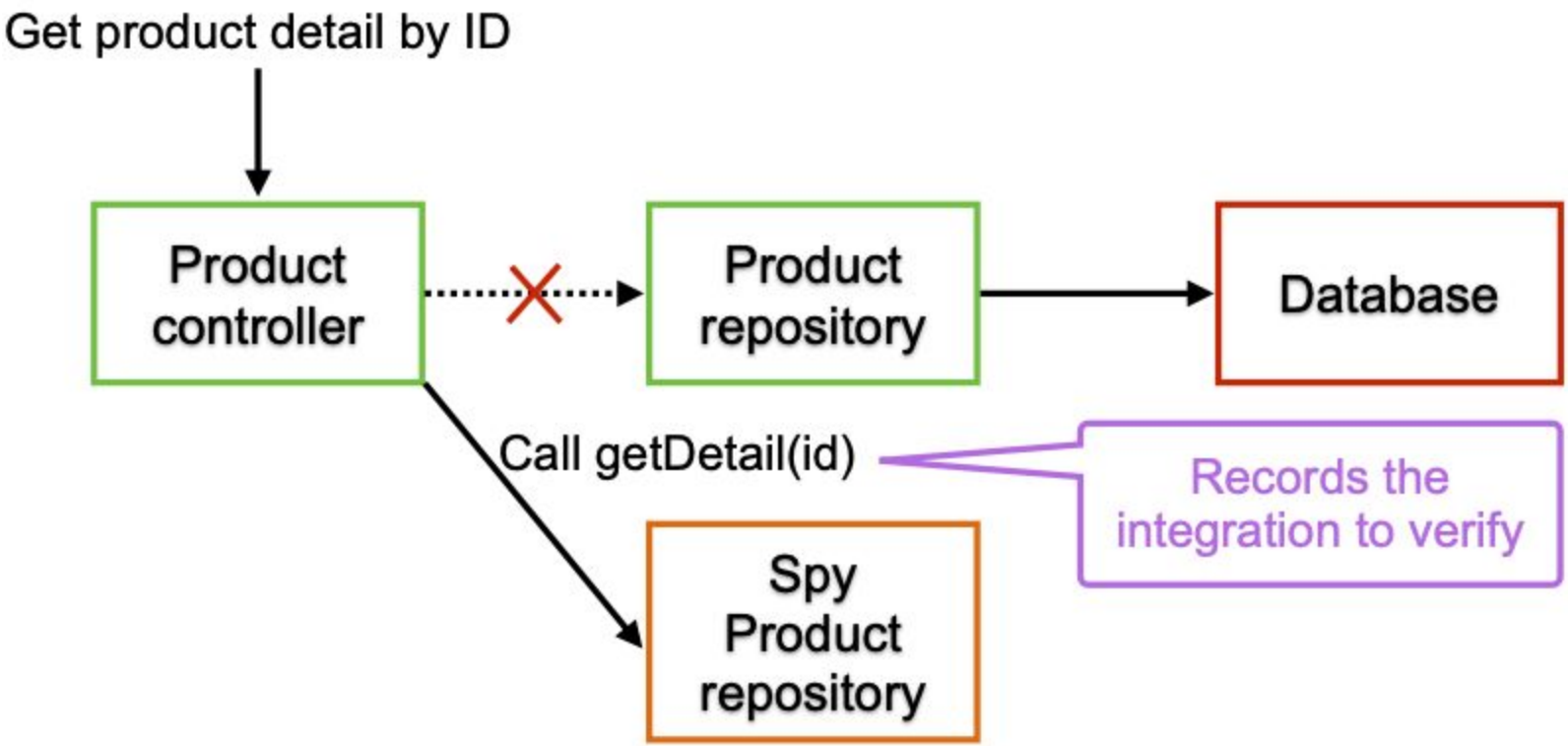
Spy

Like stub
Record some information based on how its called

E.g. how many message it was sent via email service

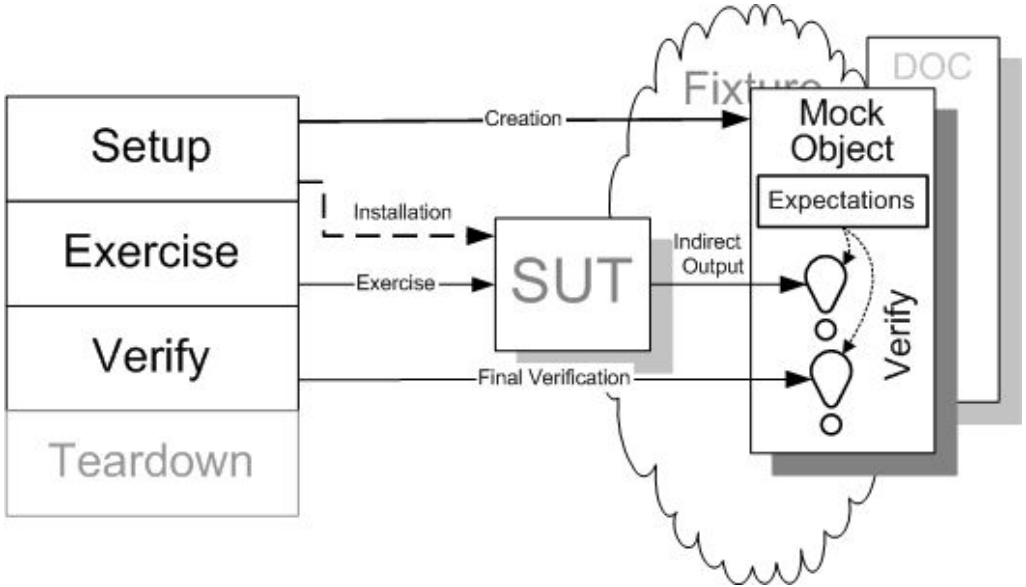


Spy

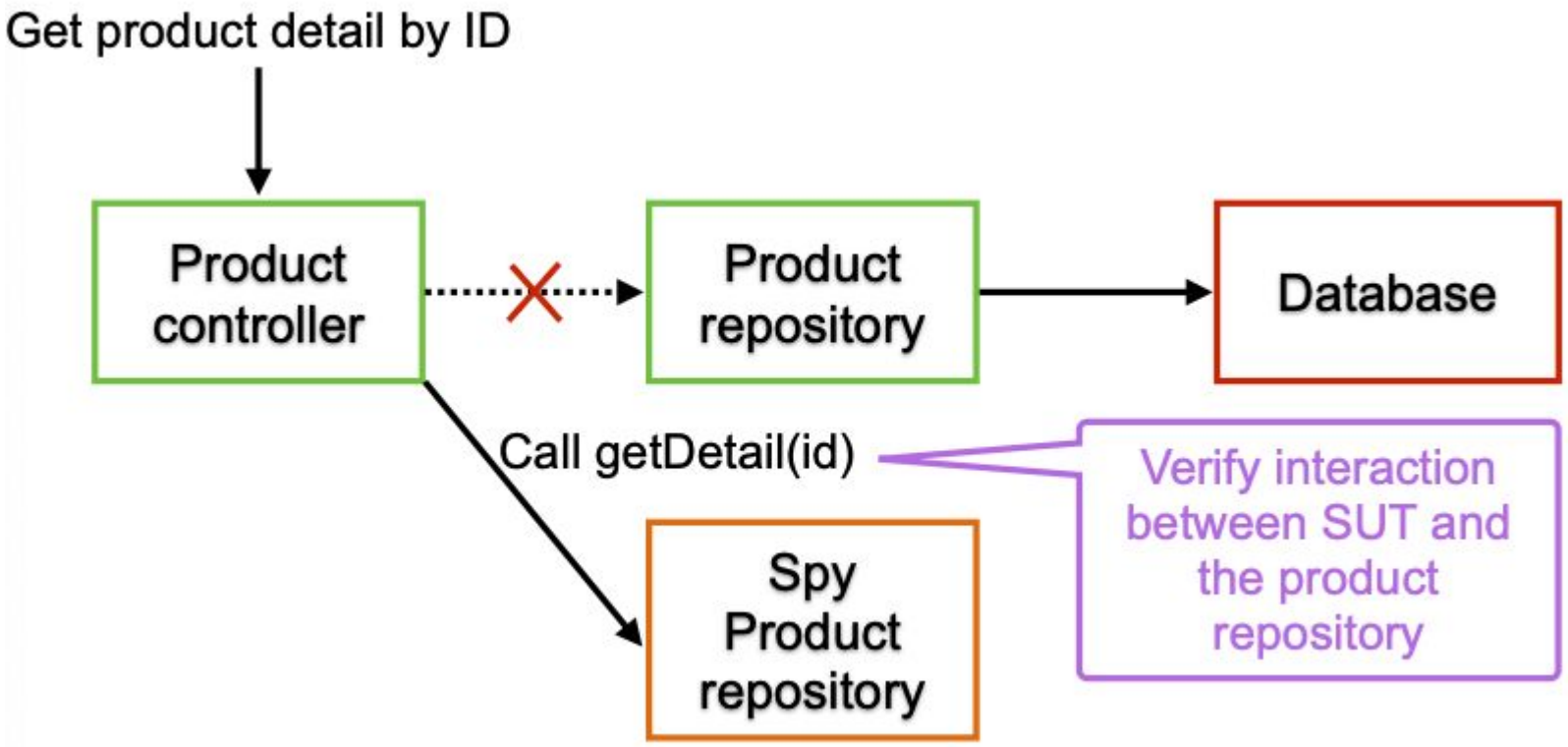

Get product detail by ID
Product controller
Product repository
Database
Call getDetail(id)
Records the integration to verify
Spy Product repository

Mock object

Pre-programmed with expectations with spec
Mock object can throw an exception if receive a call that don't expect



Mock object


Get product detail by ID
Product controller
Product repository
Database
Call getDetail(id)
Spy Product repository
Verify interaction between SUT and the product repository

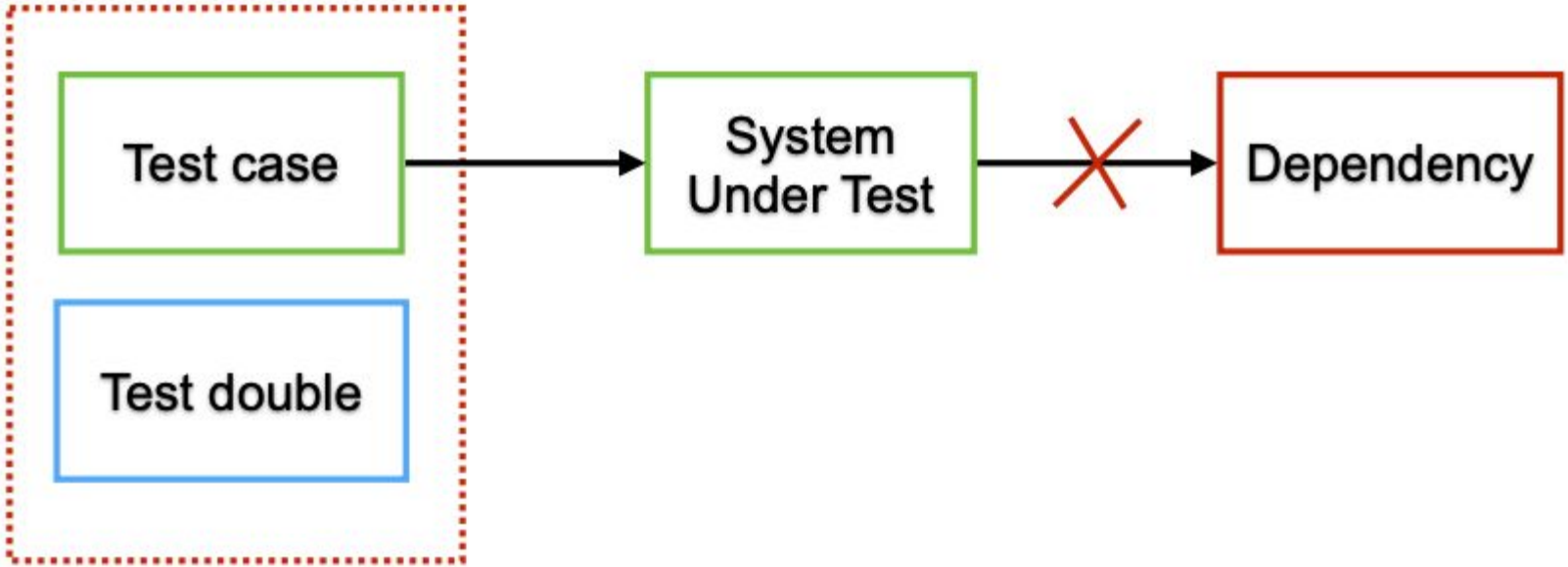
Dummy object

Passed around but never actually used
Used to fill parameter lists

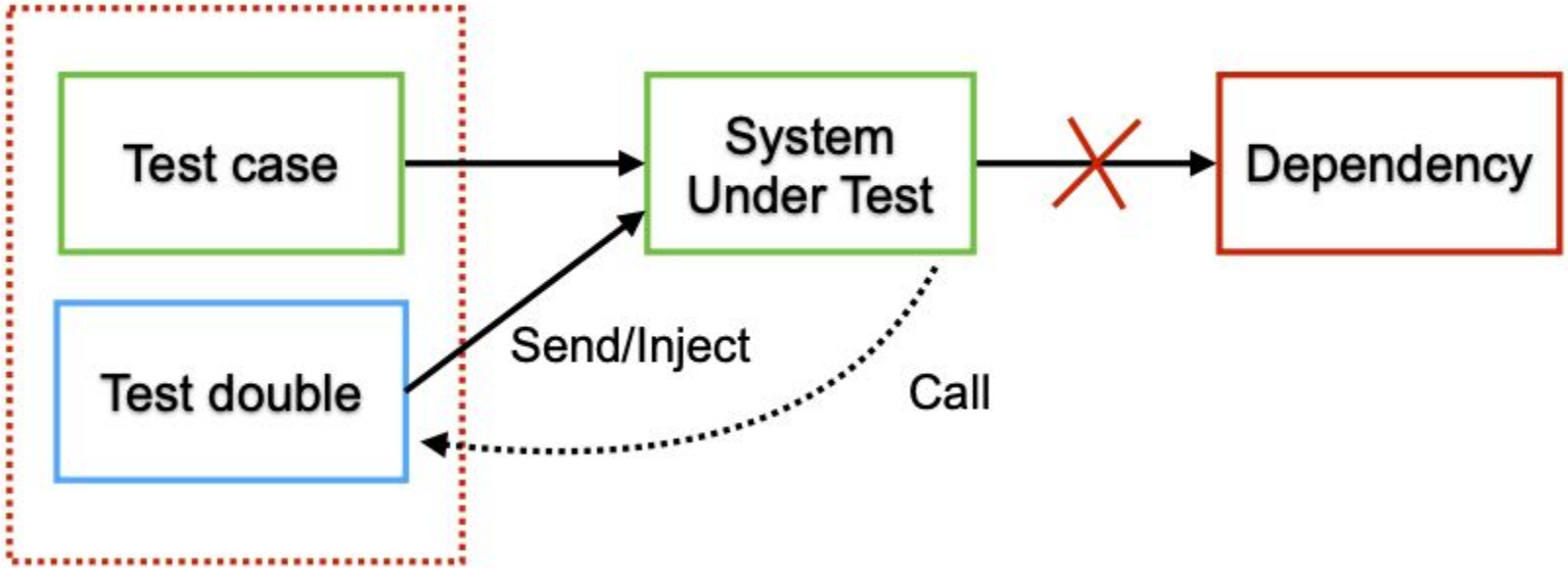
Dummy: These are objects that are passed around but never actually used. Usually, they are just used to fill parameter lists. They don't have any kind of test implementation.

Test double ?

Create test double



Send/inject test double to SUT



Workshop

Test random number = 5 ?

Workshop

Test random number must called 1 time ?

Mockito

Using Mockito with JUnit 5

```
@ExtendWith(MockitoExtension.class)
public class DemoWithMockito {

    @Mock
    Random random;

    @Test
    public void usingMockito() {
        // Create stub
        when(random.nextInt(10))
                .thenReturn(5);
    }

}
```

Test suite

Test suite is used to bundle a few unit test cases and run them together. In JUnit, **@Suite** annotations are used to run the suite tests. This chapter takes an example having two test classes, **tag** that run together using Test Suite.

```
@IncludeTags("production")
@Suite
@SuiteDisplayName("A demo Test Suite")
public class JUnit5TestSuiteExample {

}
```

JUnit Category (JUnit 4)

We can create as many categories by implementing marker interfaces where the name of the marker interface represents the name of the category.

```
public interface UnitTest {
}
```

```
public interface IntegrationTest {
}
```

```
@Test
@Category(IntegrationTest.class)
public void testAddEmployeeUsingSimpelJdbcInsert() {
}

@Test
@Category(UnitTest.class)
public void givenNumberOfEmployeeWhenCountEmployeeThenCountMatch() {
}
```

JUnit 5 we can filter tests by tagging a subset of them under a unique **tag** name. For example, suppose we have both unit tests and integration tests implemented using JUnit 5. We can add tags on both sets of test cases:

```
@Test
@Tag("IntegrationTest")
public void testAddEmployeeUsingSimpelJdbcInsert() {
}

@Test
@Tag("UnitTest")
public void givenNumberOfEmployeeWhenCountEmployeeThenCountMatch() {
}
```

Break

10:35

Testing in Spring Boot

@SpringBootTest
@WebMVCTest
@JsonTest
@DataJpaTest
@RestClientTest

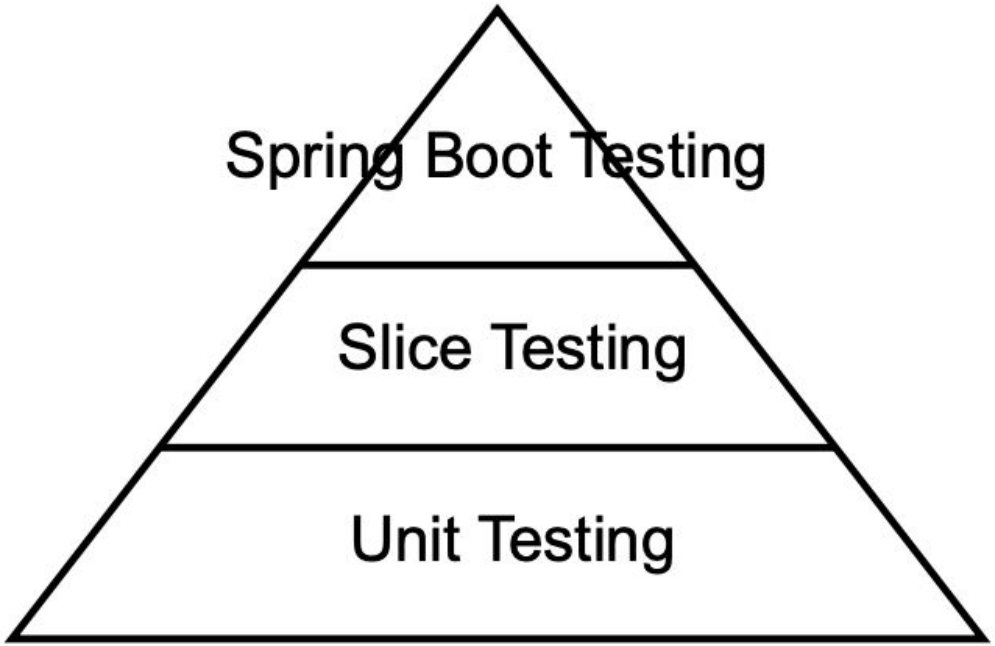
Testing in Spring Boot

@SpringBootTest

@WebMVCTest
@JsonTest
@DataJpaTest
@RestClientTest

Slice testing

Testing in Spring Boot



Controller testing

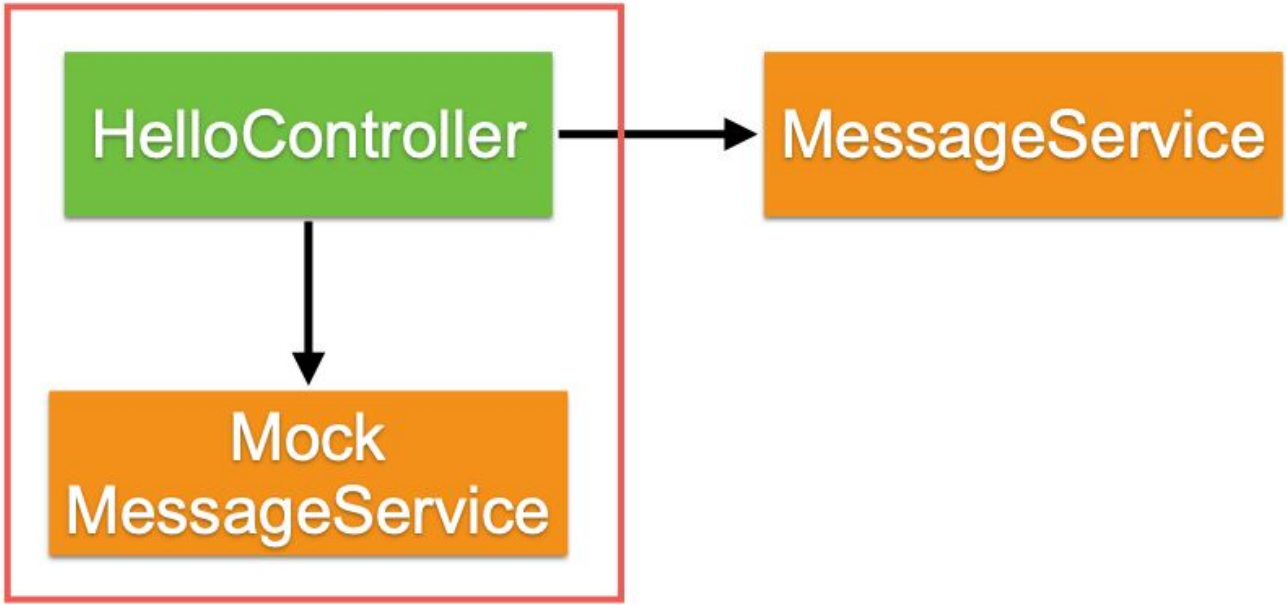
How to testing with Spring Boot ?

Controller testing

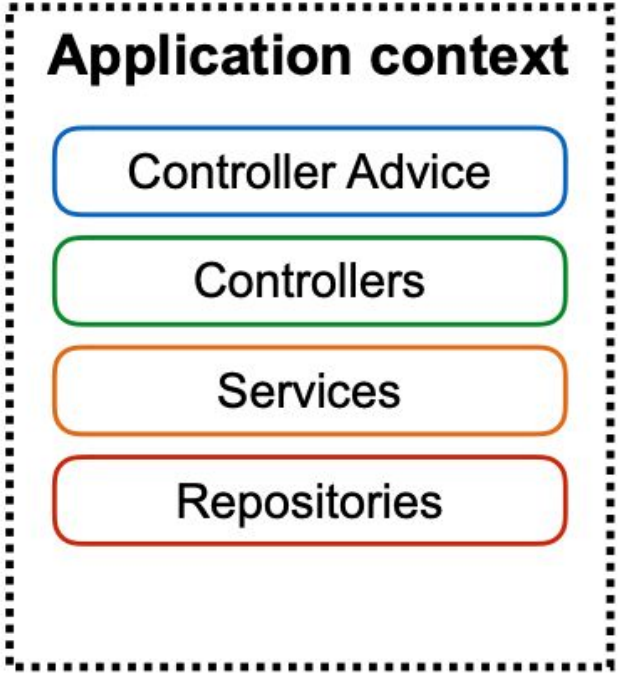
1. Spring Boot Testing
2. Slice Testing with MockMvc
3. Unit Testing

Testing controller with service

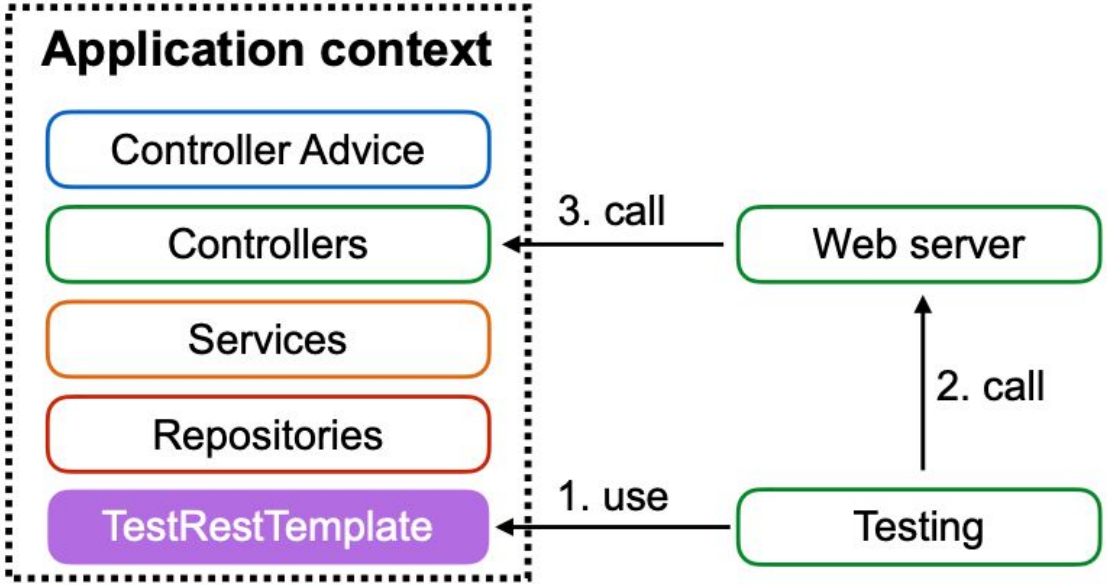
Try to mocking service with Mockito



1. SpringBootTest



1. SpringBootTest



SpringBootTest #1

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
                = testRestTemplate.getForObject("/hello/        ",
                                                Hello.class);

        // Assertion :: Check result with expected result
        assertEquals("Hello,        ", actualResult.getMessage());

    }
}
```

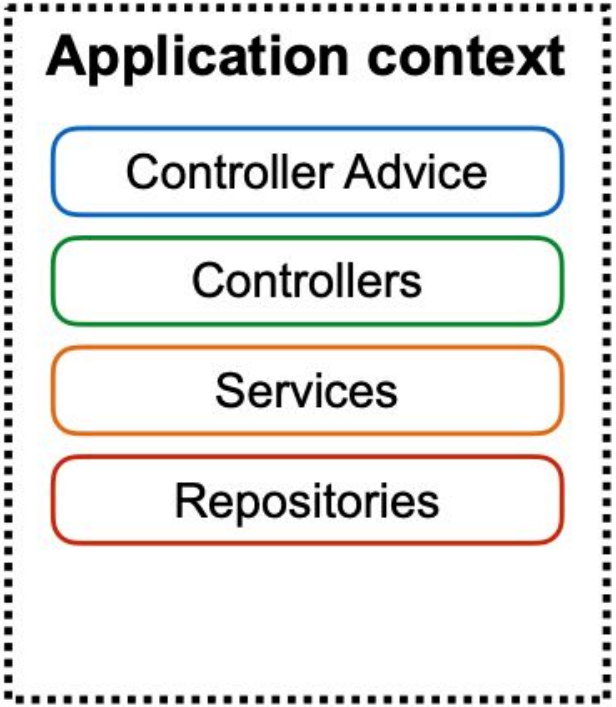
SpringBootTest #2

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
                = testRestTemplate.getForObject("/hello/       ",
                                                Hello.class);

        // Assertion :: Check result with expected result
        assertEquals("Hello,        ", actualResult.getMessage());

    }
}
```

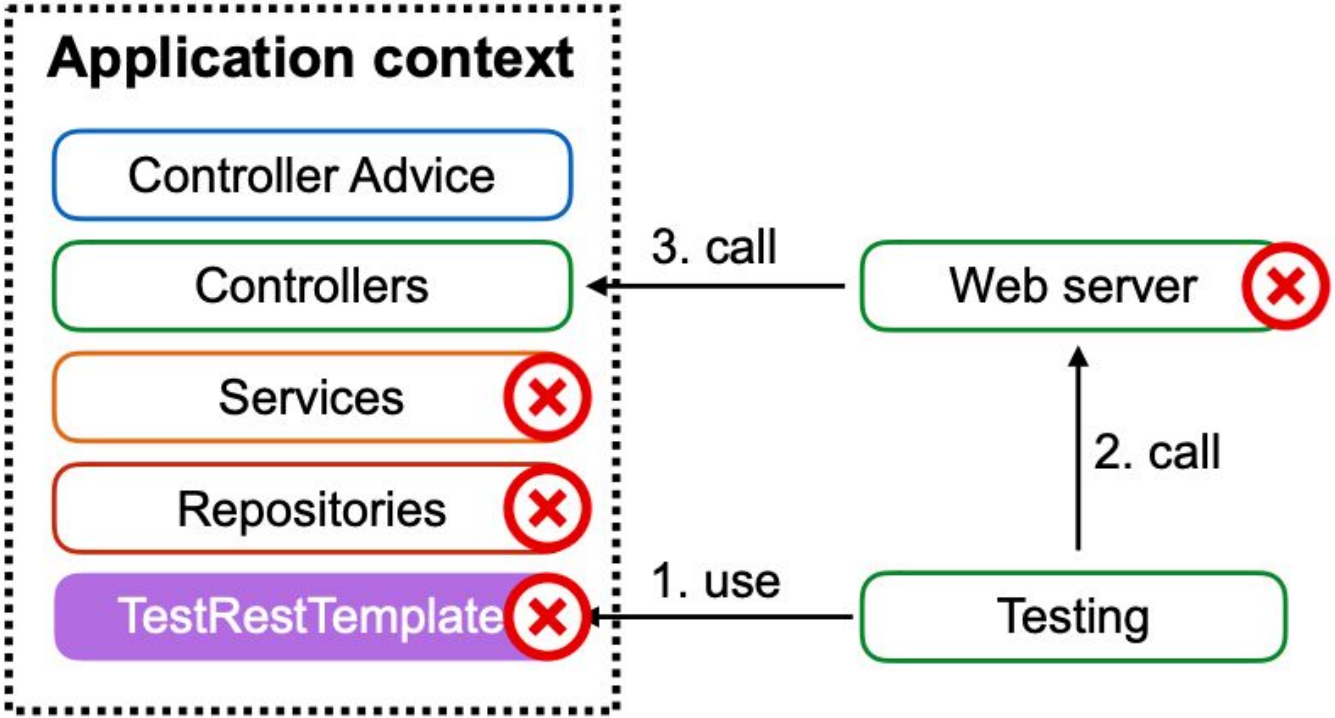
SpringBootTest #3

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment
        = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class HelloControllerTest {

    @Autowired
    private TestRestTemplate testRestTemplate;

    @Test
    public void sayHi() {
        // Action :: Call controller
        Hello actualResult
                = testRestTemplate.getForObject("/hello/        ",
                                                Hello.class);

        // Assertion :: Check result with expected result
        assertEquals("Hello,        ", actualResult.getMessage());

    }
}
```

2. Slice Testing with MockMVC



2. Slice Testing with MockMVC



MockMvcTest #1

```
@RunWith(SpringRunner.class)
@WebMvcTest(NumberController.class)
public class NumberControllerMockMvcTest {

    @MockBean
    private MyRandom stubRandom;

    @Autowired
    private MockMvc mvc;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
```

MockMvcTest #2

```
@RunWith(SpringRunner.class)
@WebMvcTest(NumberController.class)
public class NumberControllerMockMvcTest {

    @MockBean
    private MyRandom stubRandom;

    @Autowired
    private MockMvc mvc;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
            = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
```

MockMvcTest #3

Use **ObjectMapper** to convert JSON to object

```
// Call API HTTP response code = 200
String response =
        this.mvc.perform(get("/number"))
        .andExpect(status().isOk())
        .andReturn()
        .getResponse().getContentAsString();

// Convert JSON message to Object
ObjectMapper mapper = new ObjectMapper();
NumberControllerResponse actual =
                        mapper.readValue(response,
                                NumberControllerResponse.class);
```

Write controller testing ?

$mvnw clean test

3. Unit testing with Controller

Unit test

Use Test Double

In java, use Mockito library

Unit testing with Mockito #1

```
@RunWith(MockitoJUnitRunner.class)
public class NumberControllerUnitTest {

    @Mock
    private MyRandom stubRandom;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
                    = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
```

Unit testing with Mockito #2

```
@RunWith(MockitoJUnitRunner.class)
public class NumberControllerUnitTest {

    @Mock
    private MyRandom stubRandom;

    @Test
    public void success() throws Exception {
        NumberControllerResponse expected
                    = new NumberControllerResponse("5555");

        // Stub
        given(stubRandom.nextInt(10)).willReturn(5555);
```

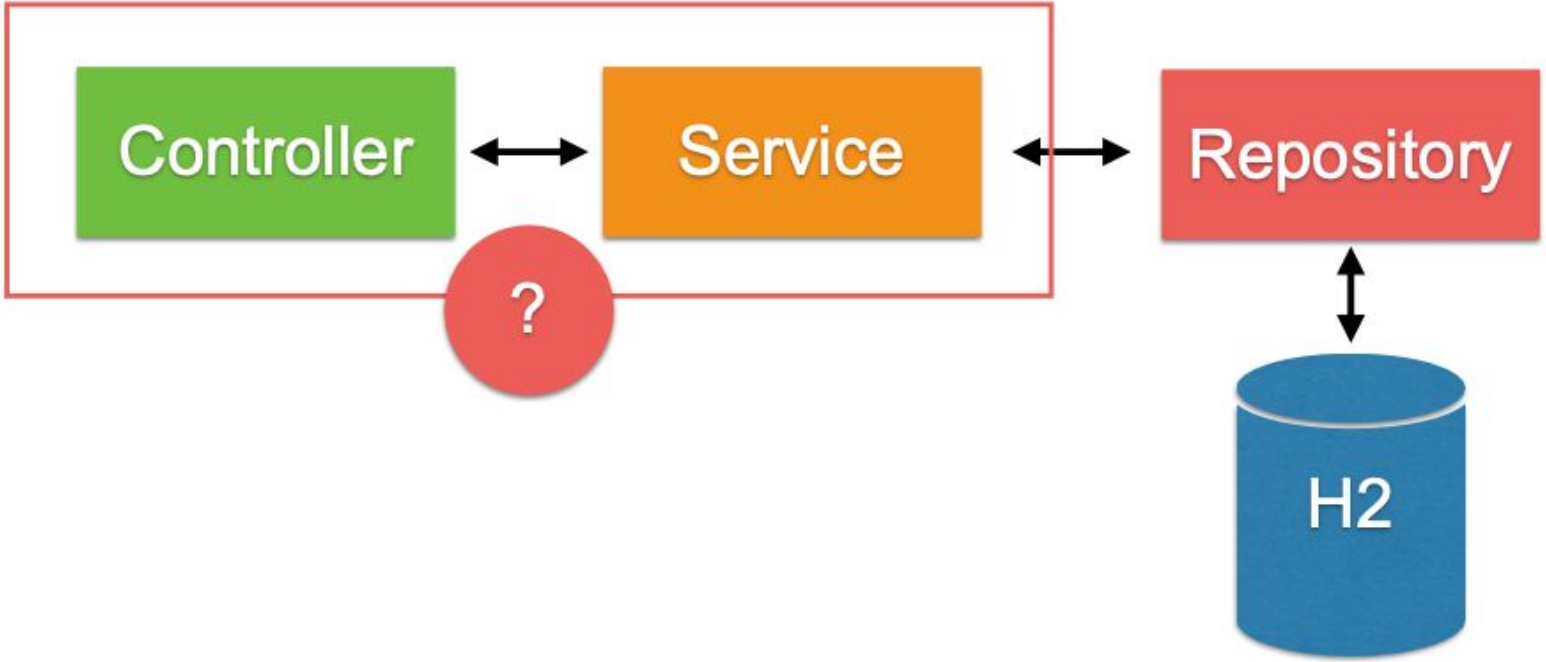
Unit testing with Mockito #3

```
@Test
public void success() throws Exception {
    NumberControllerResponse expected
                = new NumberControllerResponse("5555");

    // Stub
    given(stubRandom.nextInt(10)).willReturn(5555);

    // Call
    NumberController controller = new NumberController(stubRandom);
    NumberControllerResponse actual = controller.randomNumber();

    // Assert
    assertEquals("5555", actual.getValue());
    assertEquals(expected, actual);
}
```

How to test ?

REST API

How to test with Error/Exception ?



Testing with WebMvcTest and MockMvc

Try to check data in response

```
@Test
public void getByIdWithNotFoundAccount() throws Exception {
    // Stub
    given(userService.getAccount(2))
            .willThrow(new MyAccountNotFoundException("Not found"));

    mockMvc.perform(
        get("/account/2")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isNotFound())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
        .andExpect(jsonPath("$.message", is("Not found")));

}
```

1

Try to check data in response

```
@Test
public void getByIdWithNotFoundAccount() throws Exception {
    // Stub
    given(userService.getAccount(2))
            .willThrow(new MyAccountNotFoundException("Not found"));

    mockMvc.perform(
        get("/account/2")
                .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isNotFound())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
        .andExpect(jsonPath("$.message", is("Not found")));

}
```

2

Testing with Service

Try to check exception

```java
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private UserRepository userRepository;

    @Test
    public void user_not_found_with_exception() {
        given(userRepository.findById(1))
                .willReturn(Optional.empty());
        UserService userService = new UserService();
        userService.setRepository(userRepository);

        Assertions.assertThrows(RuntimeException.class, () -> {
            userService.getData(1);
        });
    }
```

Compile with testing

$mvnw clean test

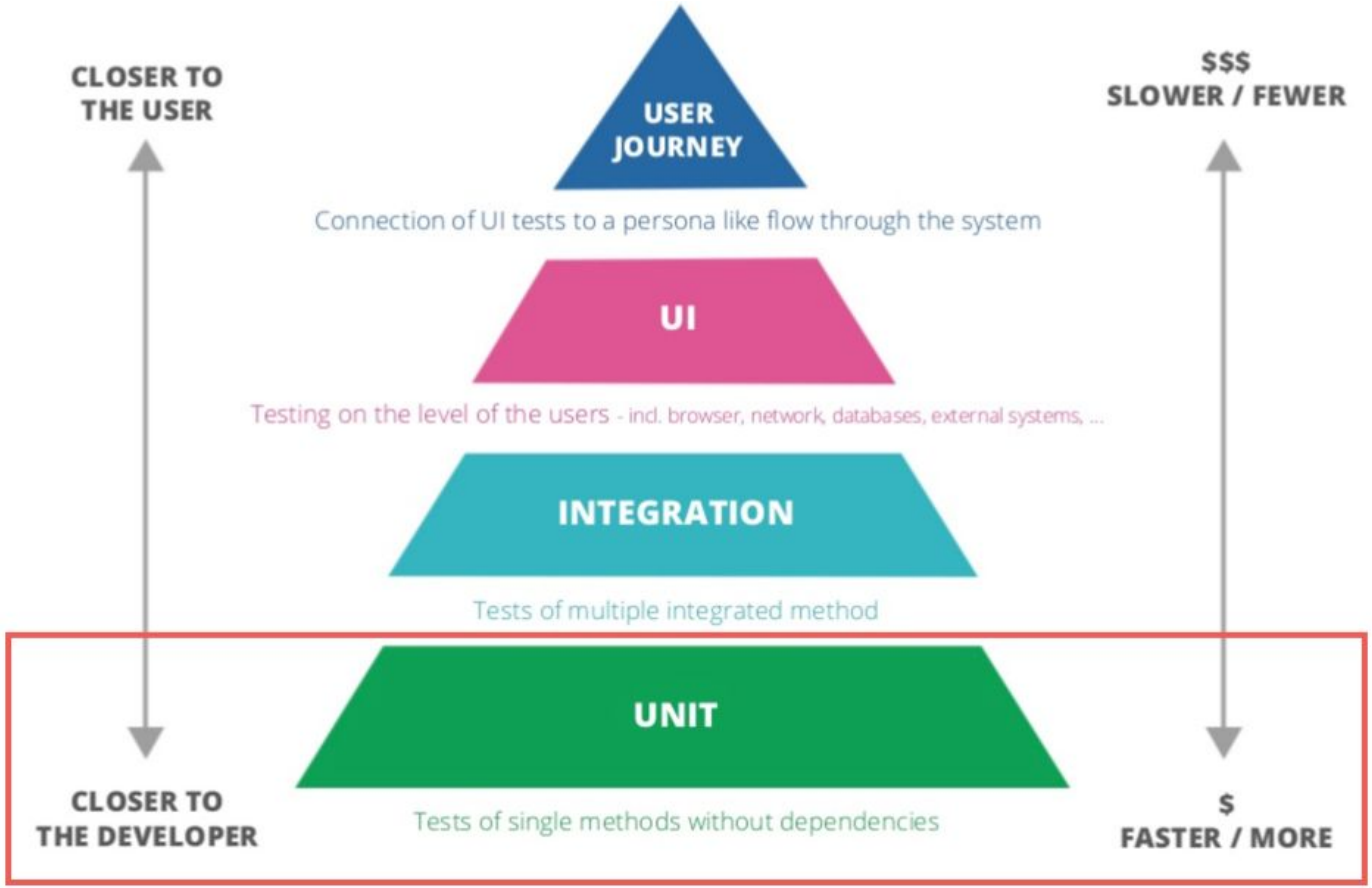
```
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
```

Run all tests !!

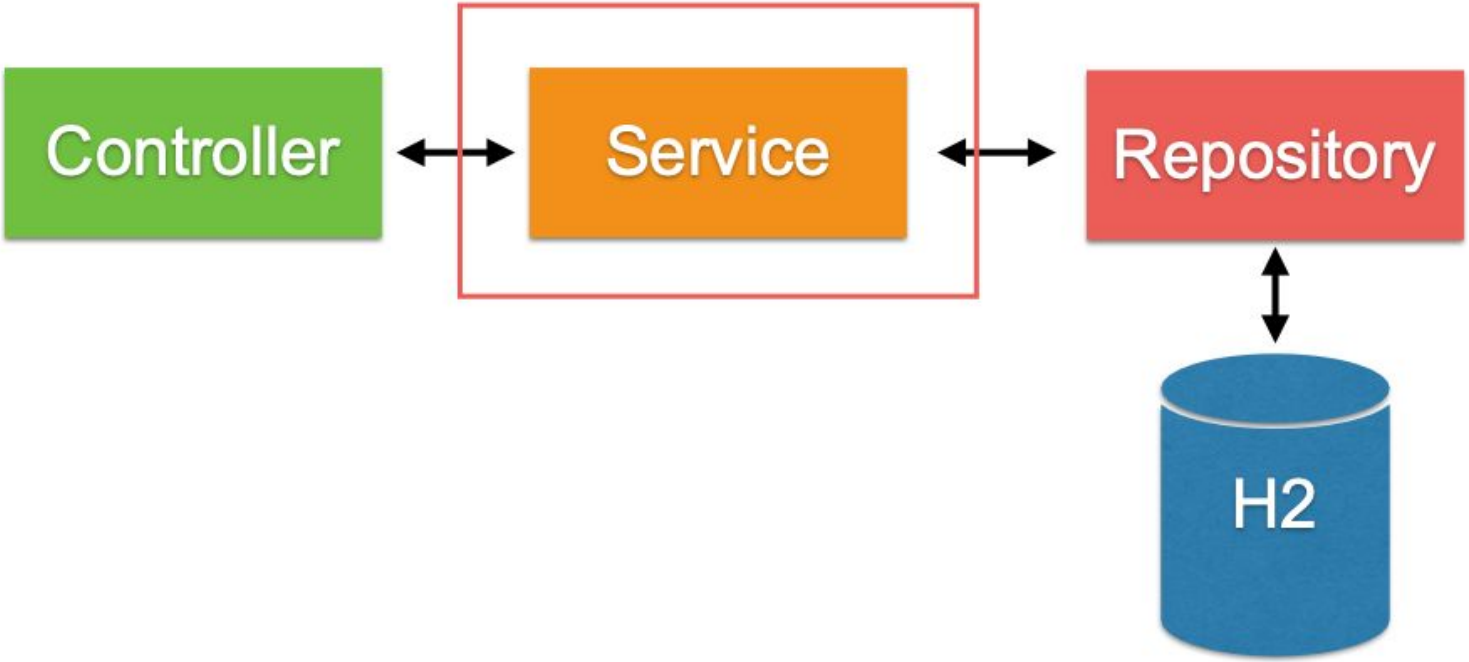
$mvnw clean test

```
[INFO]
[INFO] Results:
[INFO]
[WARNING] Tests run: 10, Failures: 0, Errors: 0,
[INFO]
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------
[INFO] Total time: 18.299 s
[INFO] Finished at: 2018-08-20T23:36:31+07:00
[INFO] ------------------------------------------
```

How to improve the speed of testing ?


CLOSER TO THE USER
$$$ SLOWER / FEWER
USER JOURNEY
Connection of UI tests to a persona like flow through the system
UI
Testing on the level of the users - incl. browser, network, databases, external systems, ...
INTEGRATION
Tests of multiple integrated method
UNIT
Tests of single methods without dependencies
CLOSER TO THE DEVELOPER
$ FASTER / MORE

Service Testing ?



Service Testing

```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private AccountRepository accountRepository;

    @Test
    public void getAccount() {
        // Stub
        Account account = new Account();
        account.setUserName("user");
        account.setPassword("pass");
        account.setSalary(1000);
        given(accountRepository.findById(1))
                .willReturn(Optional.of(account));

        UserService userService = new UserService(accountRepository);
        Account actualAccount = userService.getAccount(1);
        assertNotNull(actualAccount);
    }

}
```

1

Service Testing

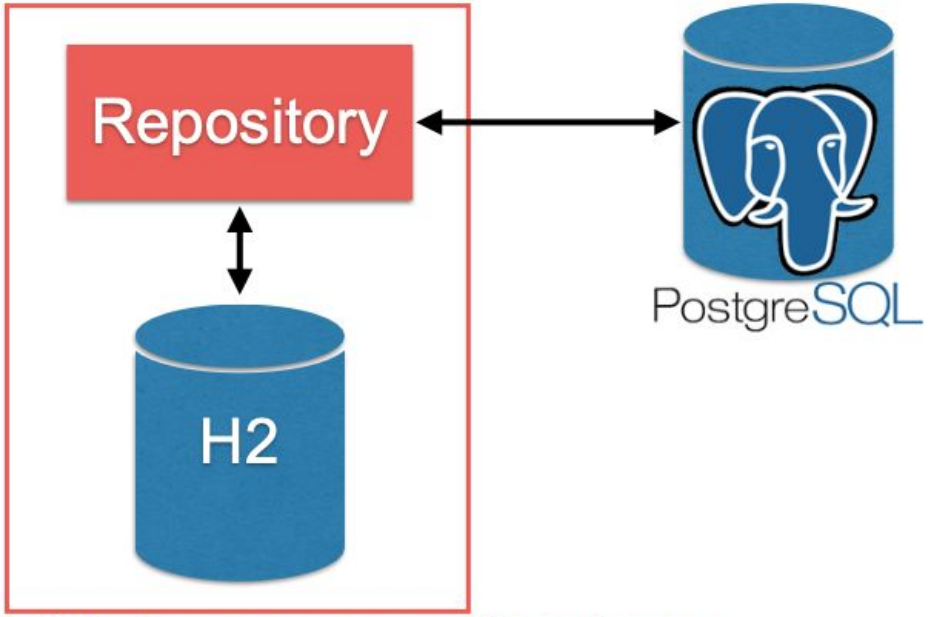
```
@ExtendWith(MockitoExtension.class)
public class UserServiceTest {

    @Mock
    private AccountRepository accountRepository;

    @Test
    public void getAccount() {
        // Stub
        Account account = new Account();
        account.setUserName("user");
        account.setPassword("pass");
        account.setSalary(1000);
        given(accountRepository.findById(1))
                .willReturn(Optional.of(account));

        UserService userService = new UserService(accountRepository);
        Account actualAccount = userService.getAccount(1);
        assertNotNull(actualAccount);
    }

}
```

2

How to testing repository ?

Repository Testing

Using @DataJpaTest (slice testing)



Working with In-memory database

Repository Testing #1

Setup test with @DataJpaTest

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```

Repository Testing #2

Auto wired repository for testing

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```

Repository Testing #3

Clear data in table after executed each test case

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class PersonRepositoryTest {

    @Autowired
    private PersonRepository repository;

    @After
    public void tearDown() throws Exception {
        repository.deleteAll();
    }

}
```

Repository Testing #3

Write your first test case

```
@Test
public void should_save_fetch_a_person() {

    Person  person  = new Person(" Person ", " B  ");
    repository.save( person );

    Optional<Person>  maybeBob
                        = repository.findByLastName(" B  ");

    assertEquals( maybeBob   , Optional.of( person ));

}
```

Run test

$mvnw clean test

```
Hibernate: drop table person if exists
Hibernate: drop sequence if exists hibernate_sequence

Hibernate: create sequence hibernate_sequence start with 1 increment by 1
Hibernate: create table person (id varchar(255) not null, first_name varchar(255),
last_name varchar(255), primary key (id))
```

Insert data

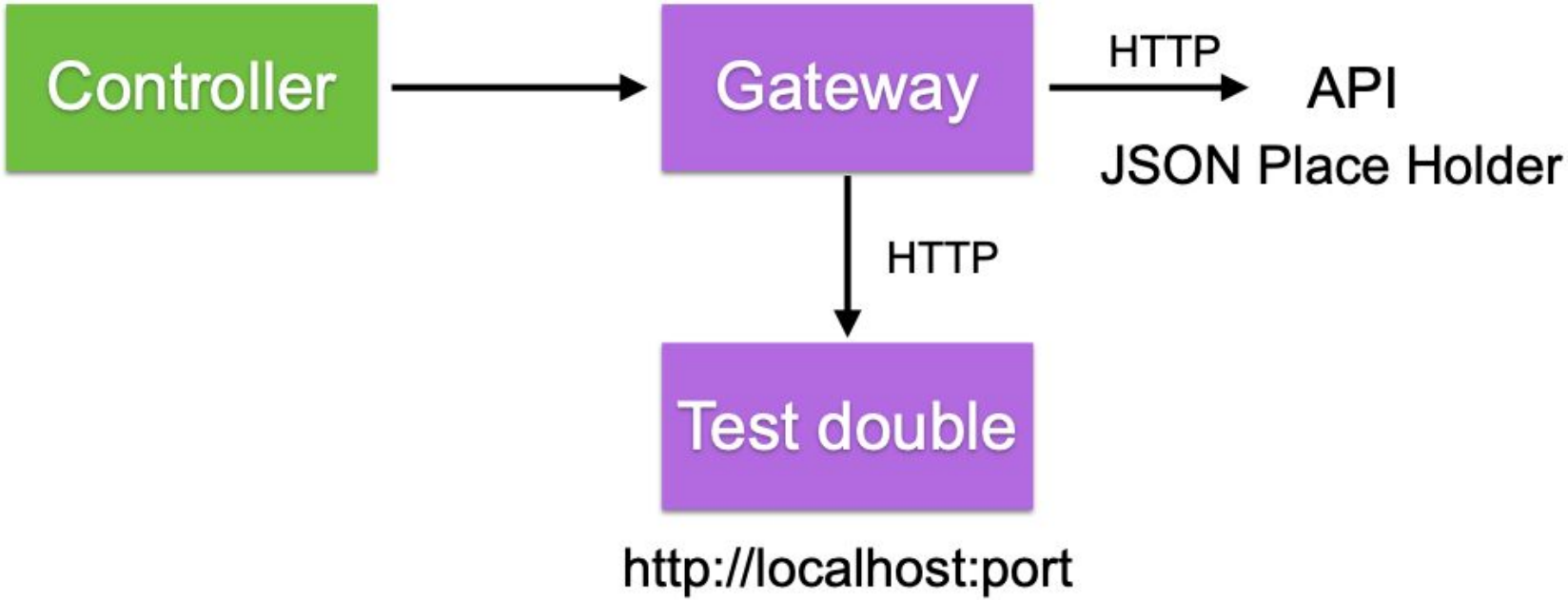
```
Hibernate: call next value for hibernate_sequence
Hibernate: insert into person (first_name, last_name, id) values (?, ?, ?)
```

```
Hibernate: select person0_.id as id1_0_, person0_.first_name as first_na2_0_,
person0_.last_name as last_nam3_0_ from person person0_ where
person0_.last_name=?
Hibernate: select person0_.id as id1_0_, person0_.first_name as first_na2_0_,
person0_.last_name as last_nam3_0_ from person person0_
2
```

Query data

```
Hibernate: drop table person if exists
Hibernate: drop sequence if exists hibernate_sequence
```

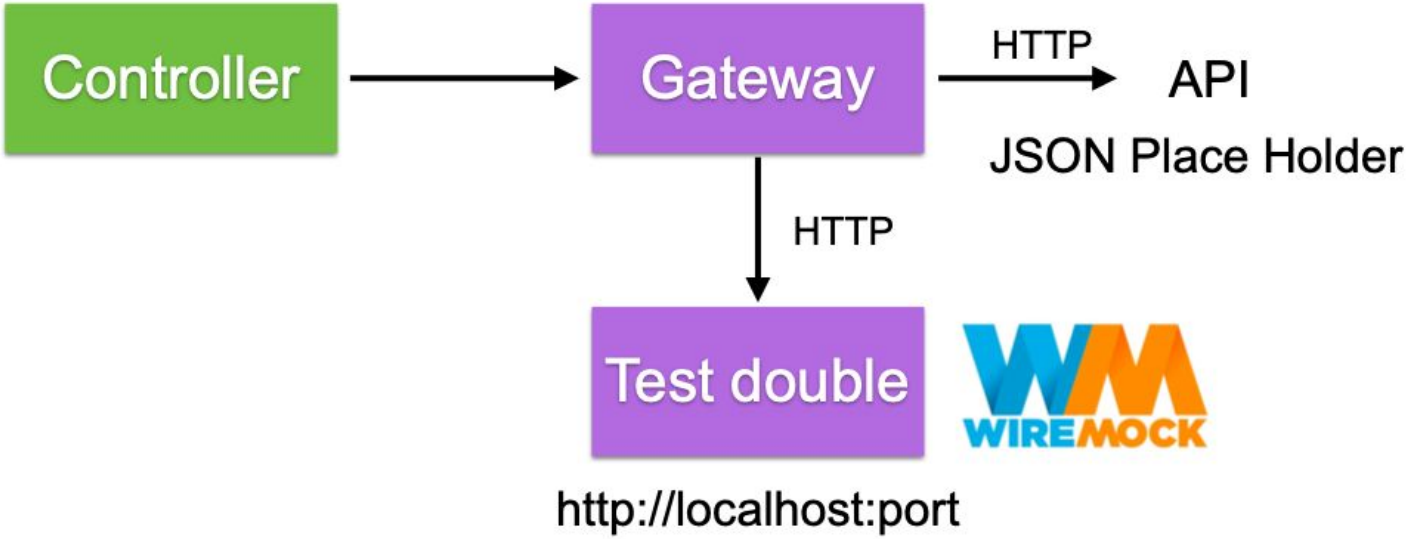
Testing with API



Testing with API

Unit testing with Mockito
Component testing with WireMock
Consumer testing with Pact

Component testing with WireMock



/src/test/resources/application.properties

```
post.api.url=http://localhost:9999
```

Component testing #1

Working with WireMock

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = 9999)
public class PostGatewayComponentTest {

    @Autowired
    private PostGateway postGateway;
```

“Default port = 9999”

Component testing #2

Success case

Stub response

```
@Test
public void getPostById() throws IOException {
    stubFor(get(urlPathEqualTo("/posts/1"))
            .willReturn(aResponse()
                    .withBody(read("classpath:postApiResponse.json"))
                    .withHeader(CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
                    .withStatus(200)));

    Optional<PostResponse> postResponse = postGateway.getPostById(1);
    assertEquals(11, postResponse.get().getId());
    assertEquals(11, postResponse.get().getUserId());
    assertEquals("Test Title", postResponse.get().getTitle());
    assertEquals("Test Body", postResponse.get().getBody());
}
```

Component testing #3

Read data from resources folder

```
public static String read(String filePath) throws IOException {
    File file = ResourceUtils.getFile(filePath);
    return new String(Files.readAllBytes(file.toPath()));
}
```

Component testing #4

File postApiResponse.json

```
{
  "userId": 11,
  "id": 11,
  "title": "Test Title",
  "body": "Test Body"
}
```

Group selfie