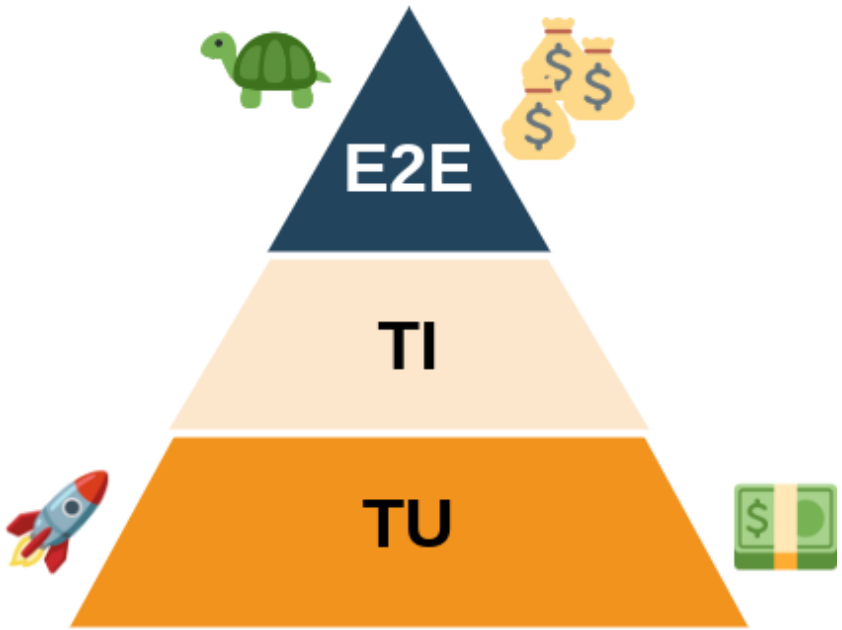
ALOM

✅ TESTS

TYPES DE TESTS

- Tests unitaires (TU)
- Test d'intégration (TI)
- Tests end-to-end (E2E)

LA PYRAMIDE DES TESTS



DURÉE DES TESTS

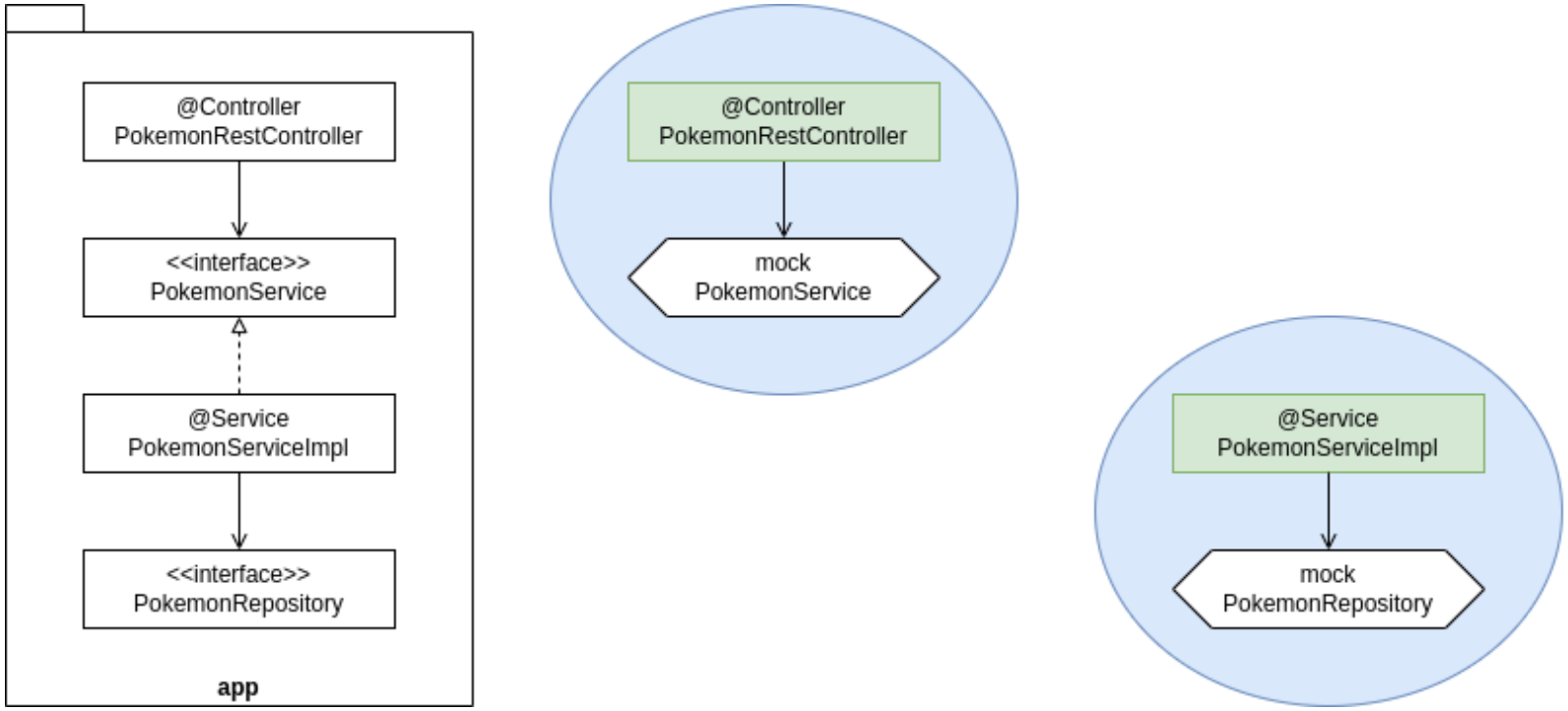
- TU ~ milliseconds
- TI ~ seconds / minutes
- E2E ~ minutes

INTÉRÊT / PÉRIMÈTRE

Type	Scope	Intérêt
TU	Méthode/ Classe	Boucle de feedback rapide
TI	Composant	Intégration avec d'autres composants (Frameworks, BDD, APIs)
E2E	Application	Tests complets (comportement, non-régression, performance, sécurité)

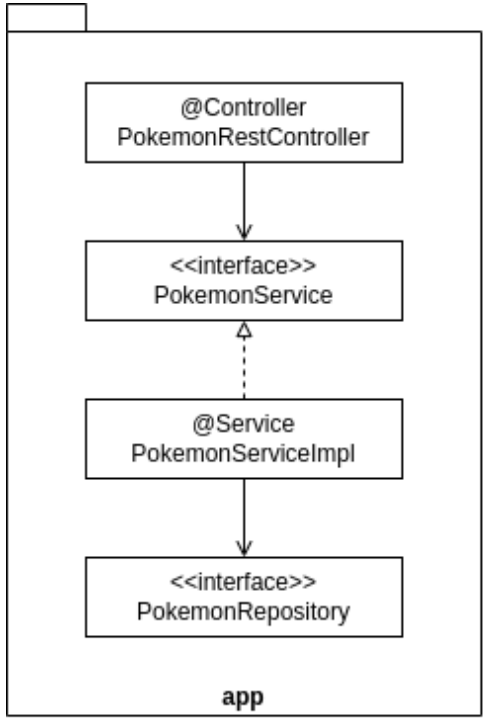
PÉRIMÈTRE DES TESTS - TU

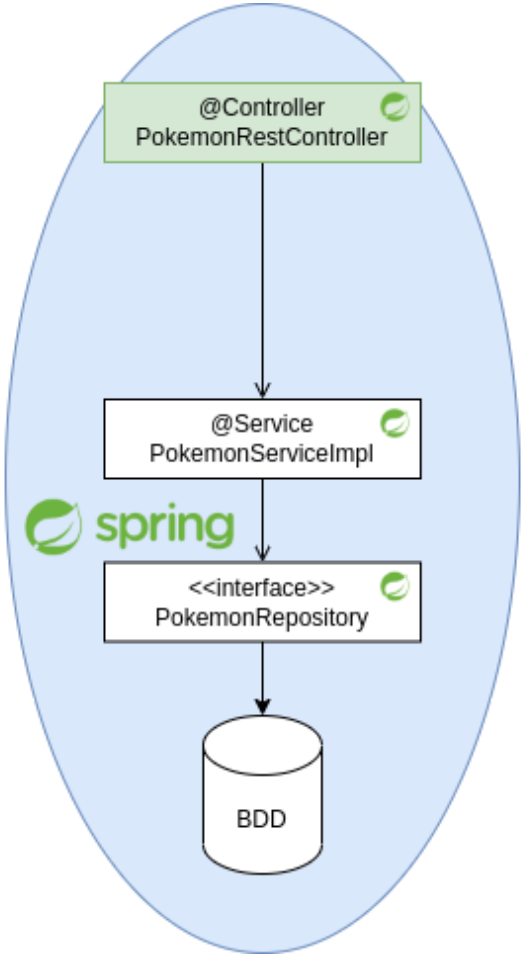
Une classe ou un composant uniquement.

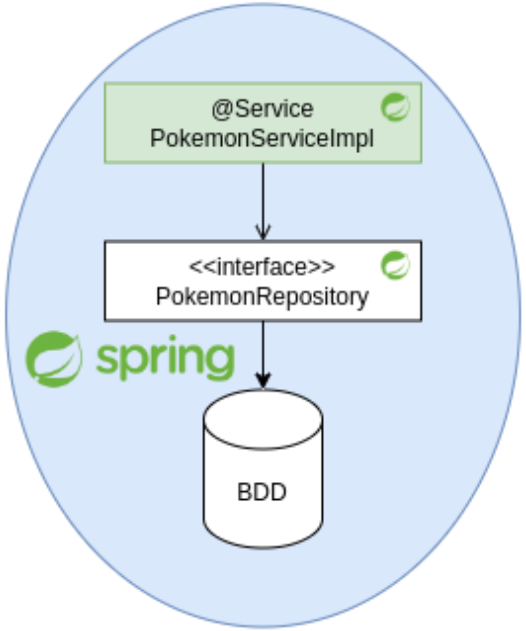


PÉRIMÈTRE DES TESTS - TI / E2E

L'application entièrement démarrée, ou en partie







OUTILLAGE

- TU : JUnit / Mockito / AssertJ
- TI : Tests Spring / TestContainers / WireMock
- E2E : Cucumber / Selenium
- Performance : JMeter / Gatling

STRUCTURE D'UN TEST

1. Le setup 🔧
2. L'exécution ▶
3. Les assertions ✅
4. Le nettoyage 🧹

STRUCTURE D'UN TEST

```
@Test
void shouldCalculateHealthCorrectly(){
    // setup 🔧
    var calculator = new StatsCalculator();
    var pikachu = new Pokemon("pikachu", 50);

    // exécution ▶
    var health = calculator.calculateHealth(pikachu);

    // assertions ✅
    assertEquals(95, health);
}
```

STRUCTURE D'UN TEST

```
@Test
void shouldCalculateHealthCorrectly(){
    // setup 🔧
    var calculator = new StatsCalculator();
    var pikachu = new Pokemon("pikachu");

    // exécution ▶ & assertions ✅
    assertThat(calculator.calculateHealth(pikachu.atLevel(6)).
    assertThat(calculator.calculateHealth(pikachu.atLevel(18))
    assertThat(calculator.calculateHealth(pikachu.atLevel(50))
    assertThat(calculator.calculateHealth(pikachu.atLevel(100)
}
```

STRUCTURE D'UN TEST

⚠️ Un test sans assertions :

- "couvre le code source"
- ne valide pas son comportement

> *"On sait juste que ça ne plante pas"*
>
> *-- Un(e) dev qui a pas écrit d'assertion*

DANS CE COURS

- JUnit 5
- Mockito
- AssertJ
- `@SpringBootTest`
- `@MockBean`
- MockMVC
- TestContainers
- WireMock
- Cucumber

JUNIT 5 (DOC)

JUNIT-JUPITER-API

- `@Test` : déclare une méthode de test
- `@Nested` : permet de grouper des tests dans une classe
- `@BeforeEach`, `@AfterEach` : méthode à exécuter avant/après chaque test
- `@BeforeAll`, `@AfterAll` : méthode statique à exécuter avant/après la classe de test

JUNIT-JUPITER-API

```
import org.junit.jupiter.api.*;

class DummyTest {

    @BeforeAll
    static void beforeAll() {}

    @AfterAll
    static void afterAll() {}

    @BeforeEach
    void setUp() {}

    @AfterEach
    void tearDown() {}
```

MOCKITO (DOC)

- `Mockito.mock(Class<T> classToMock)` : crée un mock
- `Mockito.when(T methodCall)` : ajoute du comportement à un mock (stubbing)
- `Mockito.verify(T mock)` : vérifie qu'un mock a été appelé

MOCKITO USAGE

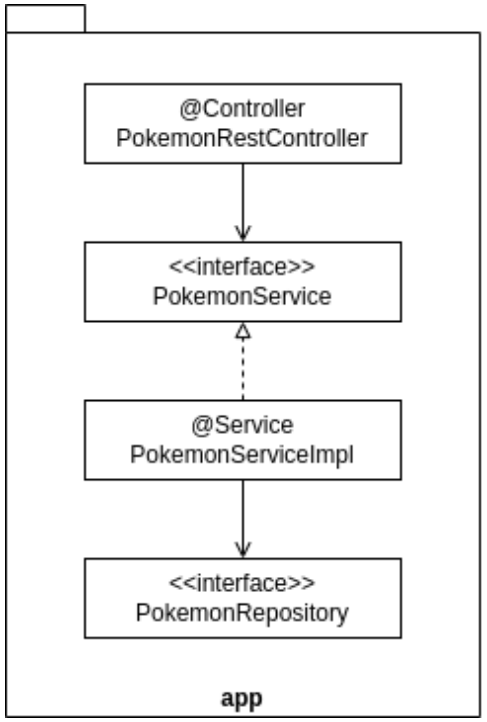
```
void showPokedex_shouldListThePokemonTypes_usingTheService(){
    // setup 🔧
    var serviceMock = Mockito.mock(PokemonTypeService.class);
    var controller = new PokemonTypeController(pokemonTypeServ

    // exécution ▶
    controller.showPokedex();

    // assertions ✅
    Mockito.verify(serviceMock).listPokemonsTypes();
}
```

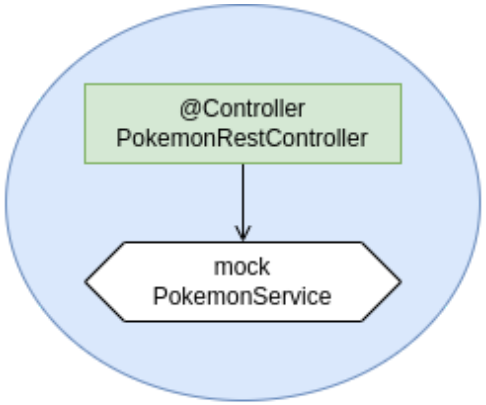
MOCKITO USAGE WITH ANNOTATIONS

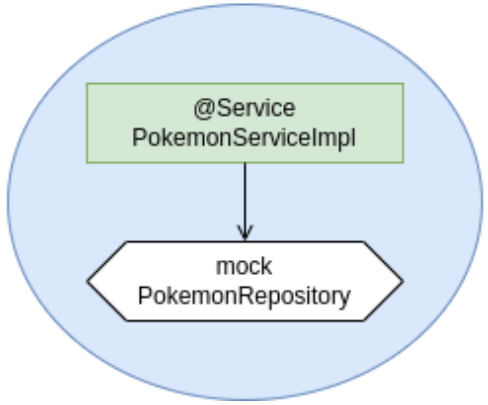
```
@ExtendWith(MockitoExtension.class)
class MockitoTest {

    // setup 🔧 - crée l'objet, et injecte les champs annotés
    @InjectMocks PokemonTypeController controller;
    // setup 🔧 - équivalent à `Mockito.mock`
    @Mock PokemonTypeService service;

    void showPokedex_shouldListThePokemonTypes_usingTheService
        // exécution ▶
        controller.showPokedex();

        // assertions ✅
        Mockito.verify(serviceMock).listPokemonTypes();
    }
```

PÉRIMÈTRE DES TESTS - TU

⚠️ L'objet annoté `@InjectMock` est celui qu'on teste







MOCKITO USAGE STUBBING

Donner du comportement aux mocks avec

`when().thenReturn()` et

`when().thenThrow()`

```
@ExtendWith(MockitoExtension.class)
class MockitoTest {

    @InjectMocks PokemonTypeController controller;
    @Mock PokemonTypeService service;

    void showPokedex_shouldListThePokemonTypes_usingTheService
        // setup 🔧 - le mock retourne une liste à un seul élé
        var pikachu = new Pokemon("pikachu");
        when(service.listPokemonTypes()).thenReturn(List.of(pi

        // exécution ▶
        controller.showPokedex();

        // assertions ✅
```

MOCKITO USAGE VERIFICATIONS

Donner du comportement aux mocks avec

`when().thenReturn()` et

`when().thenThrow()`

```
@ExtendWith(MockitoExtension.class)
class MockitoTest {

    @InjectMocks PokemonTypeController controller;
    @Mock PokemonTypeService service;

    void showPokedex_shouldListThePokemonTypes_usingTheService
        // setup 🔧 - le mock retourne une liste à un seul élé
        var pikachu = new Pokemon("pikachu");
        when(service.listPokemonTypes()).thenReturn(List.of(pi

        // exécution ▶
        controller.showPokedex();

        // assertions ✅
```

MOCKITO USAGE VERIFICATIONS - ARGUMENTMATCHERS

Matcher les arguments pour `when()` et `verify()`

```
import static org.mockito.ArgumentMatchers.*;

void matchersDemo(){
    // eq()
    when(service.getPokemon(eq(25))).thenReturn(new Pokemon("p
    // any() - anyInt() - anyBoolean() - any(T) ...
    when(service.getPokemon(anyInt())).thenReturn(new Pokemon(
    // something notNull() or isNull()
    when(service.levelUpPokemon(isNull())).thenThrow(new Illeg
    // something that contains a string
    when(service.getPokemonFromType(contains("electric")).then
    // etc
}
```

ASSERTJ (DOC)

ASSERTIONS "FLUENT"

Les assertions JUnit sont assez limitées :

- `assertTrue(boolean)` / `assertFalse(boolean)`
- `assertEquals(Object, Object)`
- `assertNull(Object)` / assertNotNull(Object)`

ASSERTIONS "FLUENT"

```java
import static org.assertj.core.api.Assertions.*;

// assertion simple ✅
assertThat(pikachu.level()).isEqualTo(14);

// assertions ✅ chaînées sur un même objet
assertThat(pikachu.name())
    .isNotNull()
    .isEqualTo("Pikachu");
```

SPRINGBOOT TESTS (DOC)

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

Importe :

- JUnit 5
- Mockito
- AssertJ

@SPRINGBOOTTEST

Déclare un test Spring !

Démarre l'application entière (comme avec le `@SpringBootApplication`), sans la partie serveur.

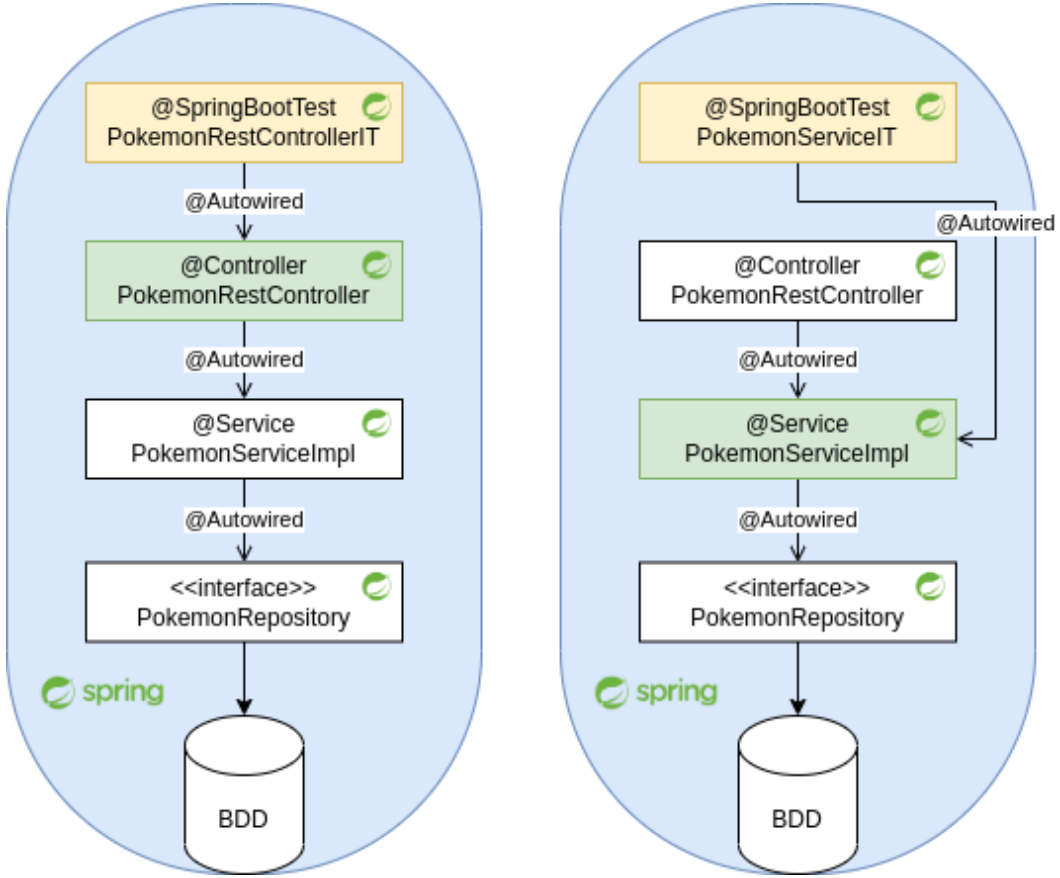
```
@SpringBootTest
class PokemonTypeServiceIntegrationTest {

    // on peut recevoir des beans Spring en injection de dépen
    @Autowired PokemonTypeService service;

    @Test
    void doSomething(){
        // tester !
    }
}
```

@SPRINGBOOTTEST

La configuration chargée est celle présente dans `src/test/resources` en priorité.

Possibilité de définir des beans annotés `@TestConfiguration` pour surcharger des beans ou de la conf.

@SPRINGBOOTTEST



@MOCKBEAN

Insère un mock Mockito dans le contexte Spring

```
@SpringBootTest
class PokemonTypeControllerIT {

    // déclaration d'un mock bean !
    @MockBean PokemonTypeAPIService service;

    // le contrôleur recevra le mock en injection de dépendanc
    @Autowired PokemonTypeController controller;

    @Test
    void doSomething(){
        // setup 🔧 - le mock retourne une liste à un seul élé
        var pikachu = new Pokemon("pikachu");
        when(service.listPokemonTypes()).thenReturn(List.of(pi
```

@MOCKBEAN

⚠️ L'utilisation de `@MockBean` occasionnera la re-création du contexte Spring à la sortie de la classe de tests

⚠️ Préférer un TU avec Mockito si possible.

@MOCKBEAN


@SpringBootTest
PokemonRestControllerIT
@Autowired
@Controller
PokemonRestController
@Autowired
@Service
PokemonServiceImpl
@Autowired
<<interface>>
PokemonRepository
spring
BDD
@SpringBootTest
PokemonRestControllerIT
@Autowired
@Controller
PokemonRestController
@Autowired
@MockBean
PokemonServiceImpl
<<interface>>
PokemonRepository
spring
BDD

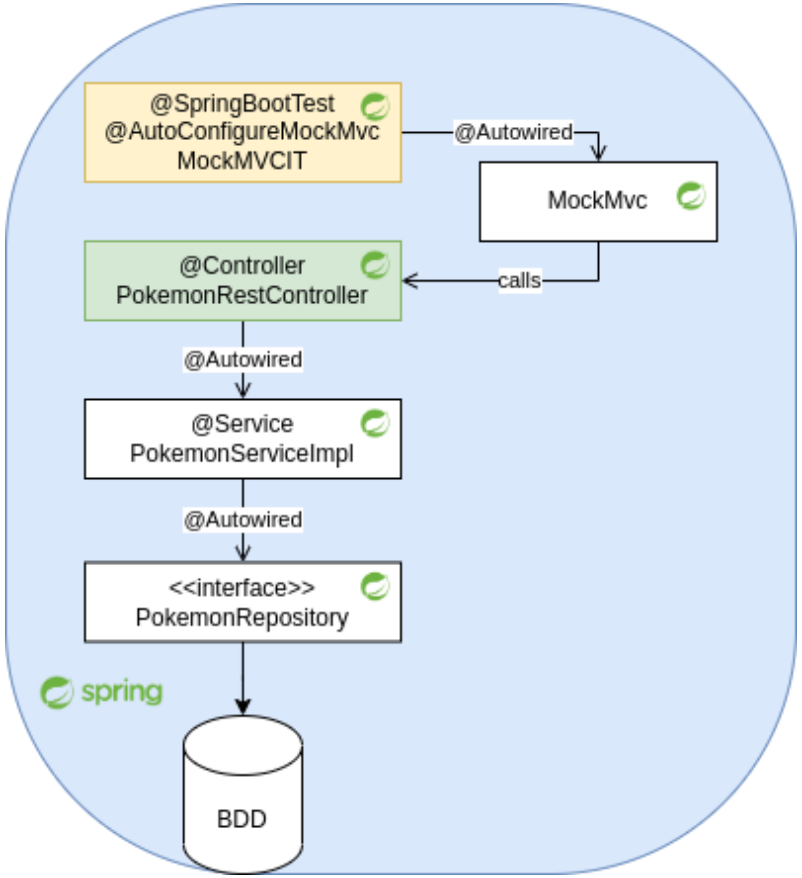
MOCKMVC

Tests d'intégration de la couche Contrôleur.

Permet de simuler des appels HTTP (GET/POST...) avec une API "fluent", et de faire des assertions sur le résultat (code HTTP, réponse, headers...).

```
@SpringBootTest
@AutoConfigureMockMvc
class MockMvcTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void dummy(){
        // test code
    }
```

MOCKMVC


@SpringBootTest
@AutoConfigureMockMvc
MockMVCIT
@Autowired
MockMvc
@Controller
PokemonRestController
calls
@Autowired
@Service
PokemonServiceImpl
@Autowired
<<interface>>
PokemonRepository
spring
BDD

MOCKMVC

```
import static org.springframework.test.web.servlet.request.Moc
import static org.springframework.test.web.servlet.result.Mock

var expectedJson = """
    {
        "id": 1,
        "name": "Bulbizarre"
    }
    """;

// Envoie une requête GET à /pokemon-types/1
mockMvc.perform(get("/pokemon-types/1"))
        .andExpect(status().isOk())  // Attend une réponse HTT
        .andExpect(content().contentType(MediaType.APPLICATION
        .andExpect(header().string(HttpHeaders.CONTENT_LANGUAG
```

MOCKMVC

Permet d'automatiser les tests au niveau de la couche API / MVC

⚠️ Tests parfois compliqués à écrire et maintenir, mais ça vaut le coup.

💡 Peut raisonnablement remplacer les TU/TI sur la couche Contrôleur.

TESTCONTAINERS (DOC)

Propose d'utiliser des containers Docker "jetables" pour exécuter les tests d'intégration.

Pratique pour tester sur une vraie BDD, ou service externe (en remplacement d'un `h2` par exemple).

TESTCONTAINERS

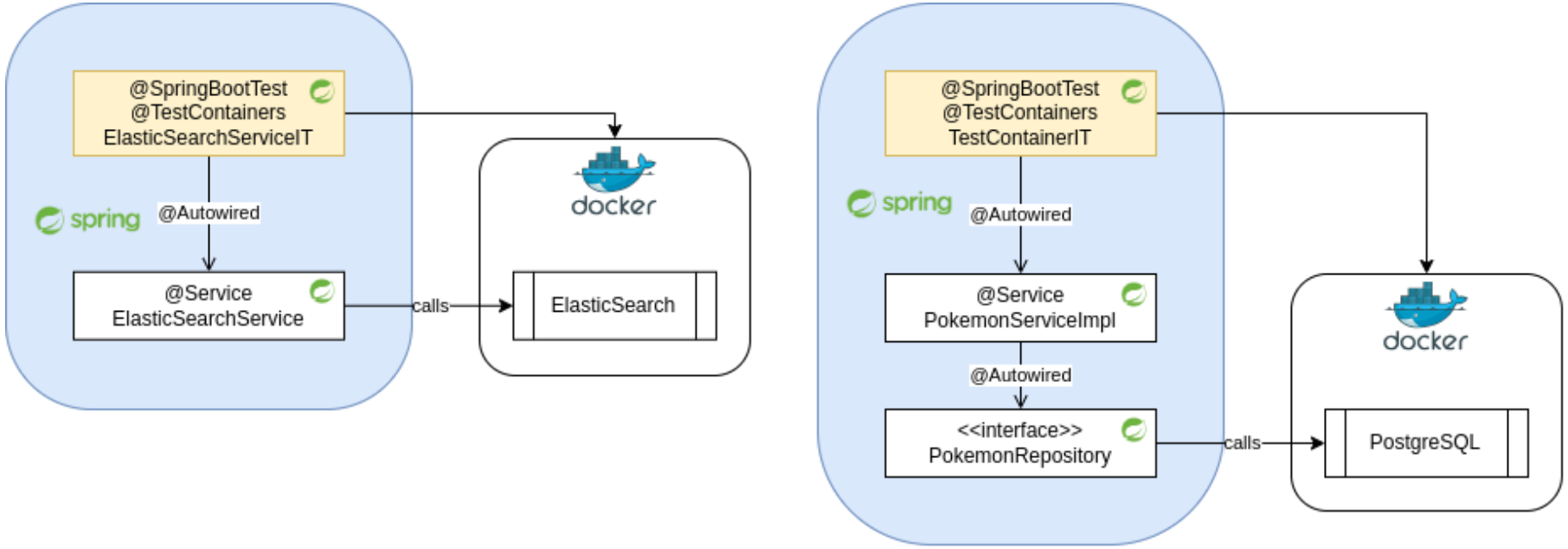
Supporte :

- les BDD relationnelles (Oracle, MySQL, PostgreSQL)
- les systèmes "NoSQL" (Cassandra, Couchbase, Elasticsearch, MongoDb, Neo4j, Redis)
- des message brokers (Kafka, Pulsar, RabbitMQ)
- des mocks de cloud (Google Cloud, LocalStack, Minio)
- ... tout ce qui peut tourner dans un container

TESTCONTAINERS

```
import org.junit.jupiter.api.Test;
import org.testcontainers.containers.ElasticsearchContainer;
import org.testcontainers.junit.jupiter.*;

@Testcontainers
@SpringBootTest
class ElasticsearchIT {

    @Container
    @ServiceConnection
    static ElasticsearchContainer<?> elastic = new Elasticsear

    @Autowired
    // contiendra les infos de connection au container
    private ElasticsearchConnectionDetails connectionDetails;
```

TESTCONTAINERS



WIREMOCK (DOC)

Permet de créer des Mock d'API.

Utile pour développer quand une API n'existe pas ou est compliquée à appeler dans des tests (ex: l'API de GitHub !).

Configuration des requêtes / réponses du mock via des fichiers JSON.

Pas de support officiel de Spring Boot, à configurer manuellement.

WIREMOCK STUBS

`src/test/resources/mappings/bulbizarre.json`

```
{
  "request": {
    "method": "GET",
    "url": "/pokemon-types/1"
  },
  "response": {
    "status": 200,
    "jsonBody": {
      "id": 1,
      "name": "Bulbizarre"
    },
    "headers": {
        "Content-Type": "application/json"
    }
  }
```

WIREMOCK

Setup

```
<dependency>
    <groupId>org.wiremock</groupId>
    <artifactId>wiremock</artifactId>
    <version>3.3.1</version>
    <scope>test</scope>
</dependency>
```

WIREMOCK

Injection dynamique de properties dans l'environnement de test

```
@WireMockTest
@SpringBootTest
class WireMockIT {

    @Autowired RestTemplate restTemplate;

    @BeforeAll
    static void injectProperties(
            WireMockRuntimeInfo wmRuntimeInfo,
            @Autowired ConfigurableEnvironment environment) {
        TestPropertyValues
            .of("pokemon-type.service.url="+wmRuntimeInfo.getH
            .applyTo(environment);
    }
```

CUCUMBER (DOC)

Behaviour Driver Development

Language d'écriture de tests fonctionnels : *Gherkin*

```
Feature: Pokemon API

  Scenario: Retrieving Pokemon Information
    Given the Pokemon API is available
    When a user requests information for Pokemon with ID 25
    Then the API should respond with status code 200
    And the response should contain the following details:
      | Field          | Value          |
      | id             | 25             |
      | name           | Pikachu        |

  Scenario: Retrieving Pokemon List
    Given the Pokemon API is available
    When a user requests the list of all Pokemon
    Then the API should respond with status code 200
```

CUCUMBER

Développement d'une *glue* pour interpréter le *Gherkin*.

```java
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.And;
import io.cucumber.java.en.But;

public class PokemonApiStepDefinitions {

    @Given("the Pokemon API is available")
    public void givenThePokemonAPIIsAvailable() {
        // Code pour configurer l'état initial
    }

    @When("a user requests information for Pokemon with ID {in
    public void whenAUserRequestsInformationForPokemonWithID(i
```

CUCUMBER

```
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-suite</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>7.15.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-spring</artifactId>
    <version>7.15.0</version>
```

CUCUMBER

Les fichiers *gherkin* vont dans `src/test/resources/features`.

Intégration avec Spring Boot :

```
@CucumberContextConfiguration // fait le lien avec Cucumber !
@SpringBootTest
public class CucumberGlue {

    @Given("the Pokemon API is available")
    public void givenThePokemonAPIIsAvailable() {
        ...
    }

    ...
}
```

CONFIGURATION JUNIT

```java
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("features")
public class CucumberRunnerTest {
    // cette classe ajoute les tests Cucumber aux tests que JU
}
```

FIN DU COURS !