Organisation des tests en Spring

GLG 203/Architectures Logicielles Java

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2021–2022

Démonstrations

Essentiellement dans les cours précédents :

- https://gitlab.cnam.fr/gitlab/glg203_204_demos/05_spring_web2.git
- https://gitlab.cnam.fr/gitlab/glg203_204_demos/06_spring_jpa.git
- https://gitlab.cnam.fr/gitlab/glg203_204_demos/07_spring_security.git
- https://github.com/spring-projects/spring-petclinic : appli « standard » de démo de Spring.

Introduction

Comment configurer les tests en Spring

Plusieurs approches :

- « vrais » tests unitaires : pas besoin de Spring !
- configuration complète de **Spring Boot** avec **Spring BootTest** ;
- configuration partielle de **Spring Boot** ;
- mise en place d'une configuration **Spring** *ad-hoc* ;
- test unitaires avec *simulation* de certaines couches : les simulacres ;

Spring, mocks, interfaces et tests

Pour tester une classe A qui utilise une classe B, c'est plus simple si B est une interface : au lieu de devoir déployer B, on peut la remplacer par une implémentation simple.

```
public class MonControleur {
@Autowired PersonneService service;

@GetMapping("/{id}")
@ResponseBody
public String get(Long id) {
  Personne p = service.getPersonne(id);
  ...
}
```

Pour tester `MonControleur`, pas besoin de la vraie classe `PersonneService` : un service qui renverrai toujours la même personne conviendrait.

Mock/Dummy/Simulacre

Définition (Mock ou Simulacre)

Dans le test unitaire d'une classe `A`, Un *Mock* ou *Simulacre* est un objet java, écrit spécifiquement pour remplacer un objet de classe **B** qui interagit avec `A`.

Typiquement, B est une interface, et le simulacre implémente uniquement les méthodes appelées par A, généralement avec des valeurs prédéfinies.

```
public class PersonneServiceMock implements PersonneService {
  public Personne getPersonne(int id) {
    return new Personne(id, "a", "b");
  }
  public void savePersonne(Personne p) {
    throw new RunTimeException("non écrit");
  } ...
}
```

Simuler ou pas les objets liés ?

Pour Martin Fowler (o.c.) :

- *The **classical TDD** style is to use real objects if possible and a double if it's awkward to use the real thing. So a classical TDDer would use a real warehouse and a double for the mail service. The kind of double doesn't really matter that much.*
- *A **mockist TDD** practitioner, however, will always use a mock for any object with interesting behavior. In this case for both the warehouse and the mail service.*

(il se range lui-même dans le style « classique » — pragmatique, quoi.

Et Spring là dedans ?

- passer par une interface n'est raisonnable que si les objets sont créés par un tiers ;
- Spring permet de le faire de manière simple et quasi déclarative.

Mockito

Bibliothèque permettant de créer très facilement des simulacres :

05_spring_web2/demoTestJunit5Mock

```
@WebMvcTest(CalcController.class)
public class CalcControllerTest {
 @MockBean CalcService calcService;
 ...

 @Test
 public void testNominal() throws Exception {
   when(calcService.somme(5, 7)).thenReturn(12);
   ...
 }
```

- CalcService est une interface ;
- Configure un « faux » objet CalcService, dont la méthode somme renverra « 12 » pour les arguments 5 et 7.

(Mockito a aussi une syntaxe *bahaviour driven* où « given » remplace « when »)

Tests des composants Spring : les *slices*

Les *slices*

Ce sont des tests qui n'utilisent *qu'une partie* de la configuration de `SpringBoot`. Par exemple, les tests de base de données ne mettent pas en place la couche MVC.
L'appendice D de la documentation indique quelles partie de la configuration est mise en place.

- @WebMvcTest : web seul ;
- @JsonTest : pour tester que la sérialisation JSON fonctionne bien ;
- @DataJpaTest : fourni les repositories JPA configurés ;
- @JdbcTest, @DataMongoTest...pour diverses autres solutions de mapping vers des bd ;
- @DataLdapTest ;
- @RestClientTest

Les propriétés

- Normalement, `application.properties` dans les tests remplace le fichier du code principal (et le cache complètement) ;
- ça repose sur le classpath utilisé par l'environnement ;
- pour certains IDE, par exemple VSCode, le comportement est différent
- les tests réagissent différemment selon la manière dont on les lance ;
- solutions possibles :
  - **utiliser** '@TestPropertySource' : fichier utilisé *en complément* de `application.properties`
  - utiliser **@ActiveProfiles**
  - exemples dans le package `glg203.demoConfig.demoProperties`

Utilisation de @ActiveProfiles

classe de test

```
@ActiveProfiles({ "testA" })
@SpringBootTest
public class TestPropertyWithProfile {
...
}
```

... utilise (en **plus** de `application.properties`) le fichier `application-testA.properties`

Tests d'intégration avec @SpringBootTest

- teste dans le cadre d'une **application Spring Boot complète** ;
- les couches qui sont activées dépendent des parties de Spring Boot incluses dans le classpath !
- pour JUnit 4 (pas 5) : annoter la classe de test avec `@RunWith(SpringRunner.class)`
- remonte dans les packages jusqu'à trouver une classe annotée avec `@SpringBootConfiguration` ou `@SpringBootApplication` ;
- sauf si la classe de test contient une classe **statique** annotée avec `@Configuration`.

Annotation SpringBootTest

configure *a priori* une application *complète*, avec les *mêmes* couches que l'application d'origine (et donc, assez coûteuse à lancer) ;

glg203.demoConfig.fullApp.TestConfigComplete

```java
@SpringBootTest
public class TestConfigComplete {
    // Test très coûteux :
    // pour vérifier que le bean est injecté, on
    // "monte" toute l'application !
    @Autowired
    MyBean myBean;

    @Test
    public void demoInjectionBean() {
        String expected = "bean de main";
        assertEquals(expected, myBean.getText());
    }
}
```

Configuration fine des tests

- Un test @SpringBootTest, de base, construit tous les beans ;
- on souhaite se concentrer sur ceux qui sont nécessaires, en restant quand même le plus simple possible.

but : écrire un test en injectant uniquement les éléments nécessaires

- on peut utiliser `@SpringBootTest` ;
- mais fournir une classe de configuration pour éviter de tout charger :
  - classe statique de configuration à l'intérieur du test ;
  - classe de configuration externe (annotation `@ContextConfiguration(TestConfig.class)` sur la classe de test).
- ne pas négliger les autres options possibles :
  - tests unitaires sans Spring quand c'est possible ;
  - annotations spécifiques (transparents précédents) ;
  - la mise en place d'une configuration Spring de tests (sans Spring boot) reste possible.

Remplacement de la configuration

On peut remplacer *complètement* la configuration d'un test Spring Boot en incluant dans la classe de test une classe **statique** annotée avec **@Configuration**

glg203.demoConfig.testFullInternalConfig

```java
@SpringBootTest
public class TestFullInternalConfig {
    @Autowired MyBean myBean;

    @Test
    public void demoInjectionBean() {...}

    @Configuration
    static class InnerTestInternalConfig {
        @Bean
        public MyBean myBean() {
            return new MyBean();
        };
    }
}
```

Remplacement de la configuration

- Avec @Configuration, la configuration de l'application d'origine est complètement oubliée ;
- Les beans définis seront ceux impliqués par les bibliothèques `SpringBoot`, et ceux définis explicitement dans notre nouvelle configuration ;
- les composants dus au classpath (jpa, web...) sont quand même construits ;
- autres solutions : voir `TestFullExternalConfig`.

Extension de la configuration

On veut conserver la configuration de l'application principale dans les tests, mais **remplacer** certains beans.
Deux solutions :

- fichier de configuration interne annoté par **@TestConfiguration** ;
- *s'ajoute* à la configuration standard ;
- peut en cacher certains éléments.
- démo : `TestInternalConfig`

- fichier de configuration externe
- importé par `@Import`
- *s'ajoute* à la configuration standard ;
- peut en cacher certains éléments.
- démo : `TestExternalConfig`

Infrastructure partielle

12_spring_aop/testJunit5

- Spring Boot utilise le `classpath` pour déterminer l'infrastructure à créer
- on peut configurer le test pour **enlever une partie de l'infrastructure.**

glg203.demoConfig.fullApp.TestConfigElaguee

```
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@EnableAutoConfiguration(exclude = {
    @JpaRepositoriesAutoConfiguration.class,
    @HibernateJpaAutoConfiguration.class})
public class TestConfigElaguee {
    @Autowired MyBean myBean;
    @Test public void demoInjectionBean() {...}
    @Configuration
    @ComponentScan("glg203.beans")
    static class MyOwnConfig {}
}
```

Exemple

```java
@SpringBootTest
@Transactional
public class DvdServiceTest {
    @Autowired
    private DvdService service;

    ...
    @Configuration // classe de configuration
    @Import(DvdService.class) // ce service et pas les autres
    @EnableAutoConfiguration // simplifie la vie
    static class Config { // statique ! (important)
    }
}
```

Tests en Spring (sans Spring Boot)

12_spring_aop/testJunit5

- On refait *toute* la configuration de Spring à la main ;
- On peut évidemment utiliser une partie des fichiers/classes de configuration de l'application principale ;
- on ne bénificie pas de la création automatique des éléments créés par Spring Boot, comme l'entity manager, et on doit les créer soi-même...
- Les mécanismes sont les mêmes...
- ici, on se lie à une classe `Config.class`, qui crée des **beans**.

glg203.demoConfig.springNotBoot.SpringNotBoot

```
@SpringJUnitConfig(classes = {Config.class})
// @ContextConfiguration(classes = Config.class) (alternative)
public class SpringNotBoot {

}
```

Tests de couches spécifiques

Tests et Bases de données

- on peut simuler les Repositories par des Mocks ;
- souvent, on utilise à la place une véritable base ;
- si la classe de test est `@Transactional`, un rollback est effectué à la fin de chaque test ;
- on peut mettre en place les entités à travers un repository ;
- possibilité d'annotations `@SqlGroup` et `@Sql` pour manipuler la base en SQL avant un test ;
- utilisation des propriétés (voir plus haut) pour différencier base de tests et base de développement ;
- utilisation de h2 (ou similaire) : si le moteur de BD de test et celui de déploiement sont différents, risques sur la validité des tests...

Tests et Contrôleurs

- test simple possible (on crée le contrôleur et on appelle ses méthodes) ;
- test en contexte : utilisation de MockMvc ;
- test d'intégration : HttpUnit

WebMvcTest

- les tests portent uniquement sur Spring MVC ;
- pas d'auto configuration des composants, services ou repositories ;
- Spring Security est a priori activé ;
- permet l'injection d'un objet MockMvc ou d'un composant HtmlUnit pour les tests ;
- on **simule** les services.

Mock MVC, mise en place des tests

```java
import static org.springframework.test.web.servlet
               .request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet
                 .result.MockMvcResultMatchers.*;


@WebMvcTest(CalcController.class)
public class CalcControllerTest {
 @MockBean CalcService calcService;
 @Autowired  MockMvc mockMvc;

 @Configuration
 @ComponentScan(basePackageClasses = CalcController.class)
 static class Config {

 }
 ...
}
```

Mock MVC, mise en place des tests

```
import static org.springframework.test.web.servlet
               .request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet
                 .result.MockMvcResultMatchers.*;


@WebMvcTest(CalcController.class)
public class CalcControllerTest {
 @MockBean CalcService calcService;
 @Autowired  MockMvc mockMvc;

 @Configuration
 @ComponentScan(basePackageClasses = CalcController.class)
 static class Config {

 }
 ...
}
```

prépare un test limité à la couche web ; les autres services, composants et repositories ne sont pas automatiquement créés

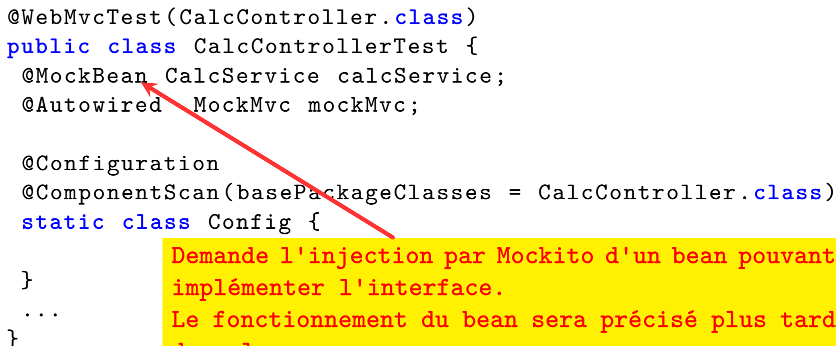
Mock MVC, mise en place des tests

```java
import static org.springframework.test.web.servlet
                .request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet
                  .result.MockMvcResultMatchers.*;


@WebMvcTest(CalcController.class)
public class CalcControllerTest {
 @MockBean CalcService calcService;
 @Autowired  MockMvc mockMvc;

 @Configuration
 @ComponentScan(basePackageClasses = CalcController.class)
 static class Config {

 }
 ...
}
```

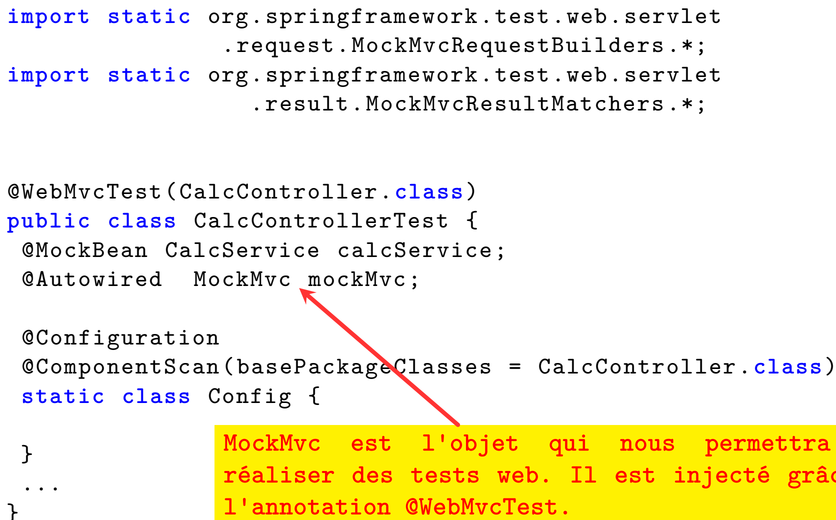
Du coup, cette configuration permet de trouver notre contrôleur.
On trouvera souvent utilisé :
@Import(CalcController.class)

Mock MVC, mise en place des tests

```java
import static org.springframework.test.web.servlet
                .request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet
                .result.MockMvcResultMatchers.*;
```



Mock MVC, mise en place des tests



```
import static org.springframework.test.web.servlet
                .request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet
                  .result.MockMvcResultMatchers.*;


@WebMvcTest(CalcController.class)
public class CalcControllerTest {
 @MockBean CalcService calcService;
 @Autowired  MockMvc mockMvc;

 @Configuration
 @ComponentScan(basePackageClasses = CalcController.class)
 static class Config {

 }
 ...
}
```

MockMvc est l'objet qui nous permettra de réaliser des tests web. Il est injecté grâce à l'annotation @WebMvcTest.
Dans un test complet, annoté par @SpringBootTest, il faut utiliser @AutoConfigureMockMvc.

Exemple d'utilisation de MockMvc

(05_spring_web2/demoTestJunit5Mock)

```
@Test public void testNominal() throws Exception {
 when(calcService.somme(5, 7)).thenReturn(12);
 mockMvc.perform(post("/")
                   .param("a", "5")
                   .param("b", "7"))
        .andExpect(view().name("calcForm"))
        .andExpect(model().attribute("resultat", 12));
}
@Test public void testMauvaiseValidation() throws Exception {
  mockMvc.perform(post("/")
                   .param("a", "23")
                   .param("b", "ds"))
         .andExpect(status().isOk())
         .andExpect(model().attributeHasFieldErrors("calcForm", "b"))
         .andExpect(view().name("calcForm"));
}
```

tests qui vérifient `ModelAndView` plus que le contenu HTML de la page.
(notez le DSL et l'utilisation massive de méthodes statiques)

Exemple de test sur un contrôleur REST

tiré de Pet Clinic

```
@Test
void testShowResourcesVetList() throws Exception {
  ResultActions actions =
      mockMvc
        .perform(get("/vets")
                   .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk());
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.vetList[0].id").value(1));
 }
```

- on vérifie que le résultat est bien du JSON ;
- … et on utilise jsonPath pour explorer les données reçues
- … l'id du premier vétérinaire de la liste doit être 1.

WebMvcTest et HtmlUnit

On peut aussi simuler un navigateur avec `HtmlUnit` : permet de tester *aussi* les interactions avec `Javascript`, par exemple.

Inclure : `testImplementation 'net.sourceforge.htmlunit:htmlunit'` dans le `build.gradle`.

```
// On configure uniquement le Contrôleur MessageControl
@WebMvcTest(MessageControl.class)
public class WebTestWithHtmlUnit {
    // WebClient : simule un navigateur
    @Autowired WebClient client;
    // On simule le service qu'on utilise
    @Autowired @MockBean MessageService messageService;

    @Test
    public void testListeVide() throws Exception {
        // voir transparent suivant...
    }
}
```

HtmlUnit (suite)

```java
// On configure uniquement le Contrôleur MessageControl
@WebMvcTest(MessageControl.class)
public class WebTestWithHtmlUnit { ...
  @Test
  public void testListeVide() throws Exception {
    ...
    when(messageService.findAll())
        .thenReturn(Collections.emptyList());
    Page page = client.getPage("/");
    assertEquals(200, page.getWebResponse().getStatusCode());
    assertTrue(page.isHtmlPage());
    HtmlPage htmlPage = (HtmlPage)page;
    assertEquals("Liste des messages", htmlPage.getTitleText());
    assertEquals("", htmlPage.querySelector("ul")
                       .getTextContent().trim());
    assertEquals("", ((DomNode)htmlPage.getFirstByXPath("//ul"))
                      .getTextContent().trim());
  }
}
```

SpringBootTest et le web

- par défaut, `SpringBootTest` ne lance pas de « vrai » serveur ;
- on peut définir l'argument `webEnvironment` de `@SpringBootTest` :
  - **MOCK** : valeur par défaut ; les tests se feront sans « vrai » serveur ; convient cependant pour la majorité des tests web ;
  - **DEFINED_PORT** : lance un serveur (tomcat...) sur un port prédéfini, ou à défaut sur le port 8080 ;
  - **RANDOM_PORT** : lance un vrai serveur, sur un port aléatoire, qui peut être injecté grâce à l'annotation `@LocalServerPort`
  - **NONE** sans émulation de la couche web.
- pour obtenir un MockMvc, il faut l'annotation `@AutoConfigureMockMvc` ;

Web et @SpringBootTest

```
@SpringBootTest
@AutoConfigureMockMvc // permet l'injection de mockMvc
public class DemoTestIntegrationMockMvc {
 @Autowired
 MockMvc mockMvc;

 @Test
 public void testEnglishLocale() throws Exception {
   // Pour utiliser xpath, le résultat doit être du XML bien formé !
   // Dans un premier temps, ce test a échoué parce que la balise <meta>
   // ne se terminait pas par "/>" !
   mockMvc.perform(get("http://localhost:8080/")
                          .locale(Locale.ENGLISH))
          .andExpect(xpath("//h1/text()")
                          .string("Computation Form"));
 }
```

Ici, tout est configuré : les services sont les « vrais », en particulier.

(suite)

```
@Test
public void testNominal() throws Exception {
  mockMvc.perform(post("/")
                  .param("a", "23")
                  .param("b", "76"))
          .andExpect(xpath("//*[@id='resultat']")
                  .string(Integer.toString(23+76)));
}
```

On n'a pas de « vrai » navigateur : pas de test du javascript des pages.

@SpringBootTest et HtmlUnit
(05_spring_web2/demoTestJunit5)

Inclure :

```
testImplementation 'net.sourceforge.htmlunit:htmlunit'
```

dans le build.gradle.
Configurer les fichiers de test :

```
@SpringBootTest
@AutoConfigureMockMvc
public class DemoTestIntegrationHtmlUnit {

 @Autowired
 WebClient webClient; // simulateur de navigation !
```

@SpringBootTest

On simule ensuite les ordres qu'on donne au navigateur, en utilisant le DOM pour accéder aux champs :

```
@Test
public void testNominal() throws IOException {
 HtmlPage form = webClient.getPage("http://localhost:8080/");
 HtmlTextInput inputA = (HtmlTextInput) form.getElementById("a");
 HtmlTextInput inputB = (HtmlTextInput) form.getElementById("b");
 inputA.type("10"); // on saisit 10 dans le champ a
 inputB.type("5"); // on saisit 5 dans le champ b
 HtmlSubmitInput button =
      (HtmlSubmitInput) form.getElementById("submit");
 // Un clic sur ce bouton expédie le formulaire
 // page2 est la page suivante, résultat du clic :
 HtmlPage page2 = button.click();
 assertEquals(Integer.toString(15),
             page2.getElementById("resultat").getTextContent());
}
```

WebClient et Javascript

WebClient permet de manipuler le DOM de la page résultat - le code javascript s'y exécute

Test de messages d'erreur en Javascript

```
@Test
public void testCheck() throws IOException {
  // On vérifie seulement que les champs ne sont pas vides.
  HtmlTextInput inputA = (HtmlTextInput) form.getElementById("a")
  HtmlTextInput inputB = (HtmlTextInput) form.getElementById("b")
  inputA.type(""); // est vide !
  inputB.type("5"); // on saisit 5 dans le champ b
  HtmlSubmitInput button =
     (HtmlSubmitInput) form.getElementById("submit");
  button.click(); // On clique et reste sur la même page
  assertEquals(
          "les champs ne doivent pas être vide",
          form.getElementById("erreur").getTextContent());
}
```

@WebAppConfiguration

Cette annotation est éventuellement utile pour une *application web classique* (un `war` intégré sur un serveur applicatif externe).

Tests de Spring Security

- balises spécifiques abordées dans le cours de Spring security ;
- démonstration dans l'archive `https://gitlab.cnam.fr/gitlab/glg203_204_demos/07_spring_security.git`

Périmètres des tests

(probablement à débattre) : il faut sans doute rester sur des tests plus atomiques... ici, on croise :

- les services ;
- l'interface utilisateur au niveau abstrait ;
- l'internationalisation et le détail des messages.

Le risque est grand de devoir modifier énormément de tests pour des raisons cosmétiques si on va trop loin.

Bibliographie

- https://martinfowler.com/articles/mocksArentStubs.html : discussion intéressante de diverses approches ;
- https://docs.spring.io/spring-boot/docs/current/reference/pdf/spring-boot-reference.pdf : documentation de Spring boot — plutôt bien faite ;
- https://docs.spring.io/spring/docs/current/spring-framework-reference/pdf/testing.pdf : le test en Spring, documentation officielle.
- https://www.baeldung.com/spring-boot-testing