

Organisation des tests en Spring

GLG 203/Architectures Logicielles Java

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

2019–2020

Démonstrations

Essentiellement dans les cours précédents (attention, mises à jours éventuelles) :

- 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.

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 MonContrôleur {
    @Autowired PersonneService service;

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

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

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.

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.

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 où « given » remplace « when »)

Tests et Bases de données

- on peut simuler les Repositories par des Mocks ;
- souvent, on utilise à la place une véritable base ;
- configuration possible dans `application.properties` : on peut dupliquer ce dernier dans `test` ;
- si la classe de test est `@Transactional`, un rollback est effectué à la fin de chaque test ;
- 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

Mock MVC, mise en place des tests

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

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

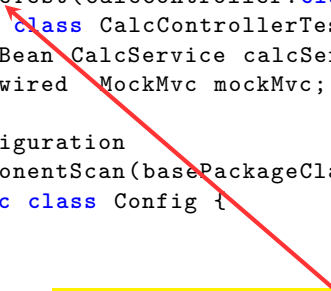
    }
    ...
}
```

Mock MVC, mise en place des tests

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

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

    }
    ...
}
```



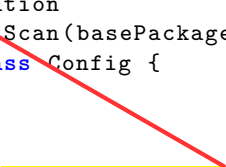
prépare un test limité à la couche web ; les autres services, composants et repositories ne sont pas automatiquement créés

Mock MVC, mise en place des tests

```
@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 :

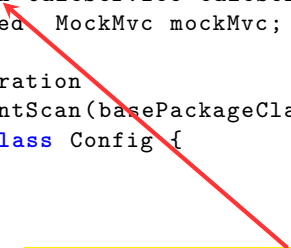
`@Import(CalcController.class)`

Mock MVC, mise en place des tests

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

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

    }
    ...
}
```



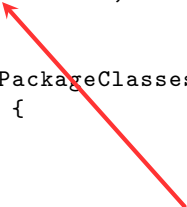
Demande l'injection par Mockito d'un bean pouvant implémenter l'interface.
Le fonctionnement du bean sera précisé plus tard dans le programme.

Mock MVC, mise en place des tests

```
@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.

Avec @SpringBootTest

```
@SpringBootTest
@WebAppConfiguration
@AutoConfigureMockMvc
public class DemoTestIntegrationMockMvc {
    @Autowired
    MockMvc mockMvc;

    @Test
    public void testEnglishLocale() throws Exception {
        // Note : 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 », thymeleaf est actif.


```
@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.

Tests avec htmlunit

(05_spring_web2/demoTestJUnit5)

Inclure :

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

dans le build.gradle.

Configurer les fichiers de test :

```
@SpringBootTest
@WebAppConfiguration
@AutoConfigureMockMvc
public class DemoTestIntegrationHtmlUnit {

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

Tests avec htmlunit

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());  
}
```

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

Note

(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.

L'annotations des tests : @SpringBootTest

- Ce n'est pas la seule manière de faire du test en Spring boot ;
- crée une `SpringBootApplication` ;
- attribut `webEnvironment` : sa présence demande un environnement web, réel ou simulé ;
- pour obtenir un `MockMvc`, il faut l'annotation `@AutoConfigureMockMvc` ;

Tests spécifiques

Pour des tests plus spécifiques, on dispose des annotations :

- `@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`
- ...liste complète dans la documentation de Spring Boot.

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 de tests complète (sans Spring boot) reste possible.

Exemple

```
@RunWith(SpringRunner.class)
@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)
    }
}
```

Annotations pour une configuration simple

@EnableAutoConfiguration : sans cette annotation, il faut préciser explicitement tous les beans dont on a besoin, y compris les beans d'infrastructure (entityManager, etc.)

@Import : normalement destiné à importer des fichiers de configuration, il permet ici de récupérer le composant DvdService (et pas les autres services qui seraient dans le même package);

avec cette configuration, seuls les beans nécessaires à la construction de DvdService sont créés;

On peut préciser un fichier de propriétés à charger :

```
@TestPropertySource(locations =  
    "classpath:application-test.properties")
```

mais le plus simple est de redéfinir
application.properties dans la partie test.

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.