


Fiche 10 : Test Driven Development & DI

Table des matières

1	Objectifs.....	2
2	Concepts.....	2
2.1	Introduction au Test Driven Development.....	2
2.2	TDD - Etape 1 : écrire des tests spécifiant ce que le code doit faire	3
2.2.1	Spécification des tests	3
2.2.2	Implémentation des tests.....	5
2.3	TDD – Etape 2 : écrire du code simple passant le test	13
2.4	Injection de dépendances à l'aide d'un framework.....	14
2.4.1	Introduction à l'injection de dépendances.....	14
2.4.2	Introduction à Jersey hk2	15
2.4.3	Injection de notre premier service via hk2.....	16
2.4.4	Visualisation de l'injection de dépendances	18
2.4.5	 En savoir plus sur l'injection via hk2	21
2.5	TDD – Etape 3 : refactoring	21
2.6	TDD : continuer les itérations.....	23
2.7	Les mocks	25
2.8	TDD en résumé	25
3	Exercices.....	26
3.1	Introduction.....	26
3.2	Exercice 1 : Création de la liste de tests.....	26
3.3	Exercice 2 : Lecture de blagues	27
3.4	Exercice 3 : Création de blagues.....	27
3.5	Challenges optionnels	28
3.5.1	Exercice 4 : Lecture aléatoire de blagues	28
3.5.2	Exercice 5 : Limitation des accès aux opération à une plage horaire.....	28

1 Objectifs

ID	Objectifs
AJ01	Test Driven Development (TDD)
AJ02	Refactor de code selon les bonnes pratiques de l'OO
AJ03	Injection de dépendances à l'aide d'un framework

2 Concepts

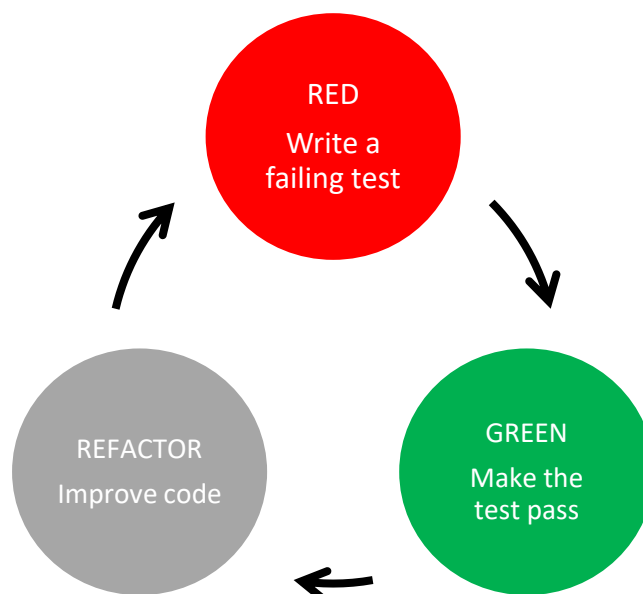
2.1 Introduction au Test Driven Development

Le TDD est un processus de développement de logiciel où les tests sont utilisés comme une spécification pour concevoir et écrire le code.

Les 3 étapes du TDD :

1. **Ecrire un test qui échoue** : on écrit un test qui décrit ce que notre code devra faire.
2. **Ecrire le code pour faire passer ce test** : on écrit le code le plus simple permettant le succès du test.
3. **Refactor du code** : on va considérer une mise à jour du code selon les bonnes pratiques de l'OO. S'il y a des aspects qui peuvent être améliorées, on va faire un refactor du code pour qu'il soit plus facilement maintenable, en éliminant les duplications, en diminuant les dépendances, en renommant des méthodes, variables, en augmentant l'efficacité... tout cela sans changer le comportement du code : le test doit toujours réussir.

Voici donc comment on schématise le TDD :



Voici certains des avantages du TDD :

- Le programmeur reçoit un rapide feedback sur ce qu'il produit :
 - o Tout changement de code peut se faire en toute confiance car si tous les tests passent, c'est que les modifications n'ont rien changé !
 - o Cela diminue donc la peur de changer son code et encourage à l'améliorer.
- Le programmeur doit se mettre dans la peau de l'utilisateur lors de l'écriture du test permettant de produire du code répondant mieux aux besoins.
- Le programmeur doit se focaliser sur l'écriture de classes concrètes, sans en faire trop, en évitant ainsi des généralisations et optimisations du code prématurées.

Si vous souhaitez découvrir le TDD en Java via des vidéos, n'hésitez pas, en dehors des cours, à jeter un œil aux vidéos de Jason Gorman qui ont inspiré cette fiche :

- [Test-Driven Development \(TDD\) in Java #1 - The 3 Steps of TDD](#)
- [Test-Driven Development \(TDD\) in Java #2 - Good TDD Habits](#)
- [Test-Driven Development \(TDD\) in Java #3 - What Tests Should We Write?](#)
- [Test-Driven Development \(TDD\) in Java #4 - Duplication & Design](#)
- [Test-Driven Development \(TDD\) in Java #5 - Part I - Inside-Out TDD](#)
- [Test-Driven Development in Java #6 - Scaling TDD with Stubs & Mocks](#)

2.2 TDD - Etape 1 : écrire des tests spécifiant ce que le code doit faire

2.2.1 Spécification des tests

Point-clé : écrire des tests qui ont du sens

OK, nous devons commencer par écrire un premier test... Mais à quel niveau commencer ce test ?

Voici les différents niveaux de tests et les questions auxquelles ils répondent :

- **Acceptance tests ou end-to-end tests** : est-ce que l'entière du système fonctionne ?
- **Integration tests** : est-ce que notre code fonctionne en intégration avec du code qui n'est pas à nous (du code que nous ne pouvons pas changer) ?
- **Unit tests** : est-ce que nos objets font ce qu'il est attendu d'eux ?

Il y a deux approches principales en TDD :

- L'approche « Inside-out » (aussi appelée « Classic », « Chicago style ») : on démarre avec un unit test, et on étend notre solution de l'intérieur vers l'extérieur, en ajoutant des fonctions.
- L'approche « Outside-in » (aussi appelée « Mockist TDD », « London style ») : on démarre avec un test end-to-end (ou acceptance test), un test ayant du sens. Ce premier test doit permettre de mettre en place tout le squelette de l'application, et doit la tester dans son environnement de production dès le début. Puis on écrit des unit tests, on les fait passer, jusqu'à ce que finalement, le test end-to-end passe lui aussi.

Il existe des fervents combattant de chacune de ces deux approches. Ces deux approches sont utiles et il est particulièrement intéressant de les mixer en fonction de notre connaissance de l'architecture de notre application.

Quand nous sommes en phase de recherche de comment architecturer notre application, d'identifier ses composants, nous devrions plutôt faire du « Outside-in TDD ». Puis, quand nous savons comment

nous allons construire notre application, que nous connaissons son architecture, nous allons plutôt faire du « Inside-out TDD ».

Voici donc le processus que nous recommandons d'appliquer :

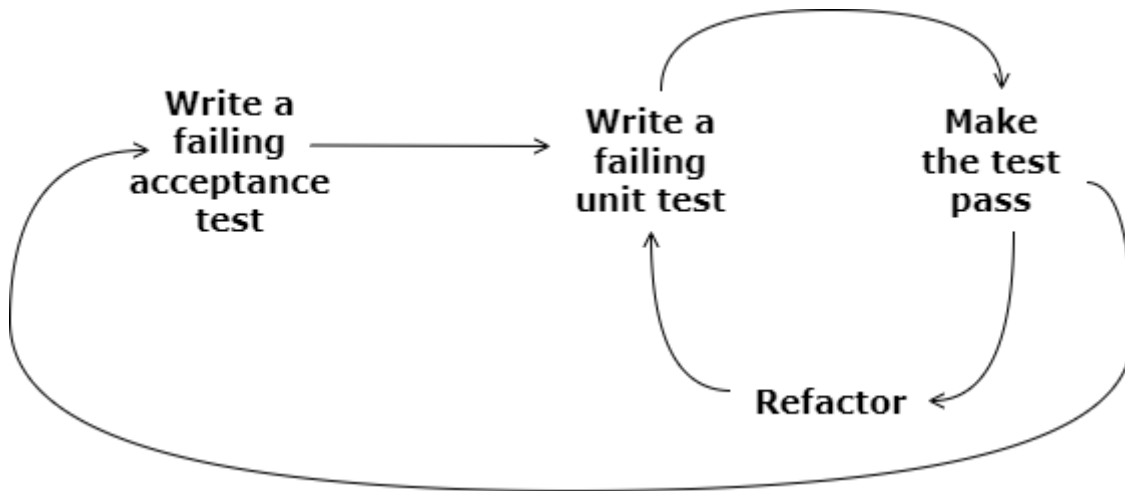


Figure 1: boucles de feedback interne & externe en TDD [Growing Object-Oriented Software, Guided by tests, Steve Freeman & Nat Pryce]

Dans un premier temps, nous allons faire du « Outside-in TDD » à l'aide d'un acceptance test qui doit nous forcer à mettre en place un squelette fonctionnelle de notre application, d'identifier les composants principaux. Puis nous ferons du « Inside-in TDD » à l'aide de Unit Tests sur chacun des composants connus de notre application.

Etape 1.1 : Nous allons commencer par **identifier une liste d'exigences fonctionnelles** pour notre application, qui contiendra les **cas d'utilisation** (use cases ou UC) de notre application.

Nous allons maintenant réaliser un tutoriel pour le TDD d'une application permettant de mettre à disposition des CD se trouvant dans une médiathèque. Nous allons décrire une version minimaliste des services que doit offrir notre application. Nous sommes dans le cadre du développement d'une HTTP API uniquement, notre application n'offrira pas d'IHM. Elle sera donc appelée directement par des clients HTTP.

Voici les exigences fonctionnelles que nous avons identifiées pour notre application :

- Chercher des CD : les clients peuvent chercher des CD sur base du titre ou de l'artiste ;
- Louer des CD ;
- Restituer des CD à la médiathèque ;
- Stocker une ou plusieurs copies d'un CD ;
- Supprimer une ou plusieurs copies d'un CD.

Point-clé : écrire une liste de tests pour identifier la roadmap de votre développement

Etape 1.2 : Puis nous **allons identifier une liste de tests**. Pour chaque UC, nous allons identifier les « Test cases », c'est-à-dire les scénarios de tests que notre code devra gérer, et le résultat attendu pour chaque test :

- Commençons par spécifier le « happy path » : quand l'exécution d'un UC fournit un résultat normal, habituel.
- Puis identifions les scénarios alternatifs : quand l'exécution d'un UC suit un chemin moins habituel, comme par exemple quand une exception doit être lancée.

Pour notre application de location de CD, voici la liste de tests que nous avons identifiée :

- Chercher un CD
 - o (« happy path » :) le CD est dans le catalogue – (résultat :) retourne l'info associée au CD (titre, artiste, stock, id).
 - o («scénario alternatif » :) Pas de CD trouvé – (résultat :) retourne "No cds".
 - o Multiples CD trouvés (si un CD a plusieurs versions) – retourne la liste de CD dans l'ordre dans lequel les CD ont été ajoutés
- Louer un CD
 - o CD en stock dans le catalogue – enlever une copie du stock du CD, indiquer l'emprunteur et la date d'emprunt et renvoyer la date de retour attendue (2 semaines plus tard)
 - o CD pas en stock dans le catalogue – retourne un « status code » d'erreur et un message d'erreur ;
- Restituer des CD
 - o ...
- ...

Etape 1.3 : Sur base de cette liste de tests, nous allons **identifier le scénario de test le plus important** : ça sera l'acceptance test qui semble avoir le plus de valeur, par exemple en tant qu'utilisateur du système, ou pour lequel nous apprendrons le plus en tant que développeurs. Ce scénario doit nous forcer à mettre en place le squelette de notre application.

Ici, tant le service de chercher un CD que de louer un CD semblent importants. Nous avons ici fait le choix que si un CD n'est pas trouvé, ce qui arrive souvent quand on visite une médiathèque, il ne pourra pas être loué... Dès lors, nous préférons donc commencer par ce scénario : « Chercher un CD, le CD est dans le catalogue – retourne l'info associée au CD (titre, artiste, stock, id). ».

2.2.2 Implémentation des tests

2.2.2.1 Acceptance test

Etape 1.4 : Nous allons écrire un **acceptance test** permettant de développer le squelette de notre application, d'identifier les principaux composants de celles-ci.

⚡ Attention, la mise en place d'un squelette fonctionnel, n'étant pas quelque chose d'évident, prend du temps. Le squelette de votre application doit pouvoir capturer les entrées fournies par les utilisateurs de votre application. Dans le cadre d'API, il s'agit de requêtes HTTP. Dans le cadre d'applications contenant une IHM, il s'agit des événements soulevés par les utilisateurs au contact de l'application. Comme nous développons une HTTP API en Java, nous allons vous aider à mettre en place celle-ci avec le minimum de code possible, et en aidant à identifier les composants attendus.

2.2.2.2 Implémentation du test à partir des assertions

Pour l'implémentation du test, on va travailler à l'envers par rapport à ce qu'on fait dans des tests traditionnels.

Point-clé : travailler à l'envers à partir des assertions

Nous allons d'abord identifier ce que l'on veut vérifier, en écrivant les asserts attendus. On écrira donc nos assertions pour vérifier le résultat de l'appel d'une méthode sur un objet pour une méthode et un objet qui souvent n'existent pas encore.

Puis nous allons utiliser notre Editeur de Code pour nous aider dans la génération du contexte de notre test, souvent appelé le setup du test, pour créer les classes nécessaires et les objets.

Si vous voulez plus de détails sur les fonctionnalités offertes par IntelliJ pour faire du TDD, vous pouvez consulter la documentation : [Test-driven development](#). Nous allons reprendre ces fonctionnalités dans notre tutoriel.

Sous IntelliJ, veuillez créer un projet **Maven** nommé **AJ_Atelier_10**. Pensez à mettre dans **Advanced Setting**, comme **GroupId** : **be.vinci**

Avant de créer la classe qui va permettre de tester l'UC « Chercher un CD », faites un clic droit sur le dossier racine de tests (répertoire « **/src/test/java** ») de votre projet, et créez le package **acceptance**. Dans ce package, créez la classe pour votre 1^{er} acceptance test : **New | Java Class** et donnez-lui le nom de « **CdApiTest** ».

Nous allons maintenant créer notre premier TC (Test Case ou scénario de test) que nous allons nommer **findCdInStockFromTitle**.

Pour ce faire, mettez votre curseur dans le corps de la classe, **Alt + Insert, Test...**

Puis, cliquez une fois sur **Fix** si JUnit5 n'est pas trouvé. Cela ajoutera la dépendance nécessaire dans votre fichier **pom.xml**. Cliquez sur **OK**. L'import des assertions a été fait automatiquement dans votre classe de tests.

Pour créer une méthode de test : mettez votre curseur dans le corps de la classe, **Alt + Insert, Test Method**. Il ne vous reste plus qu'à donner le nom **findCdInStockFromTitle** à votre méthode de test.

Imaginons ici que notre HTTP API offre, via une requête HTTP de type **GET** sur **/cds?title=young mystic**, l'information du CD dont le titre est « young mystic ». La réponse à cette requête devra être renvoyée au format texte sous cette forme : **" title:Young Mystic, artist:Bob Marley, stock:3, price:12.0€, id:666-777"**

Commençons donc par écrire cette assertion :

```
@Test
void findCdInStockFromTitle() {

    String expectedResponse = "title:Young Mystic, artist:Bob Marley,
stock:3, price:12.0€, id:666-777";
    String actualResponse = getRequest("cds", "title=young mystic");
    assertEquals(expectedResponse, actualResponse);
}
```

A ce stade-ci, nous n'avons aucune idée de comment nous allons faire la requête à l'API, c'est pourquoi nous avons identifié la méthode **getRequest**.

2.2.2.3 Exécution d'un test

Est-il intéressant à ce stade-ci de voir le test échouer ?

Point-clé : voir le test échouer pour les bonnes raisons

On va tester notre test avant de le faire réussir... Mais ça veut dire quoi ?

Cela signifie qu'avant d'écrire du code qui va faire réussir notre test, on doit s'assurer que le test échoue pour de bonnes raisons. On ne veut pas que le test échoue juste parce que le setup du test est mauvais.

Ici le setup du test est clairement mauvais : la méthode **getRequest** n'existe pas, et il n'y a même pas de server qui permet de traiter des requêtes !

Nous allons continuer à écrire le contexte du test, le setup. Veuillez générer automatiquement la méthode **getRequest** :

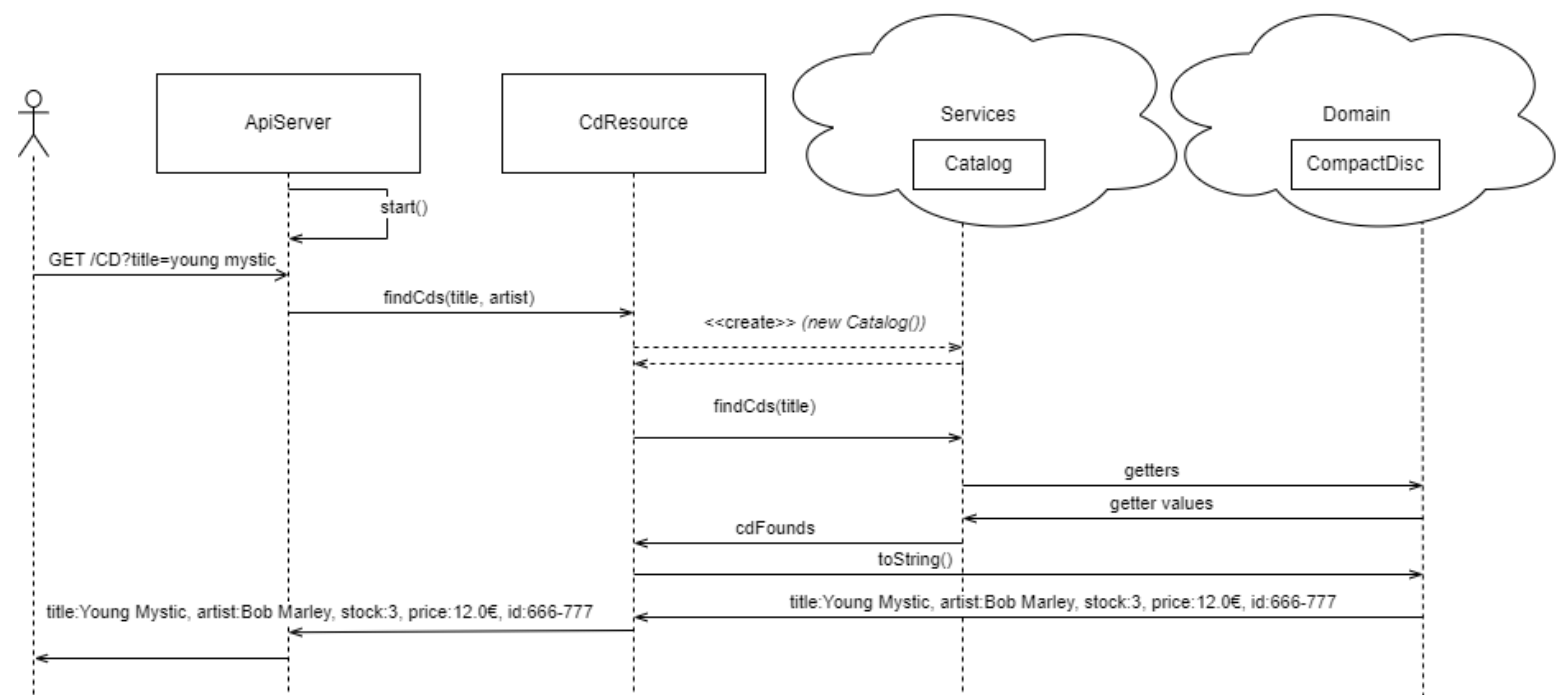
- soit clic droit sur **getRequest**, **Show Context Actions**, **Create method 'getRequest' in 'CdApiTest'**
- soit si le curseur est sur **getRequest**, **Alt + Enter**, **Create method 'getRequest' in 'CdApiTest'**

Veuillez donner des noms parlant aux arguments, et une valeur de retour qui fera d'office échouer le test :

```
private String getRequest(String path, String queryParams) {  
    return null;  
}
```

2.2.2.4 Identification de la structure de l'application

A ce stade-ci, le test échoue toujours, mais pas pour de bonnes raisons. En effet, nous n'avons pas encore mis en place le squelette de l'application. Après réflexion sur l'architecture, voici comment nous souhaiterions démarrer la découpe de l'application en composants :



Nous souhaitons avoir un serveur web lancé par la méthode **main** de notre classe **ApiServer**, qui écoute sur un port en particulier, et qui met à disposition les opérations offertes par notre **HTTP API** qui devront se trouver dans **CdResource**.

Nous souhaitons avoir un service nommé **Catalog** qui permettra d'accéder aux données de type **CompactDisc**. Plus tard, probablement que **Catalog** fera appel à une base de données. Mais pour ce tutoriel, nous simplifions et imaginons que le service **Catalog** s'occupera de créer une représentation des données sous forme d'une liste de CD.

Pour l'instant, nous n'avons pas besoin d'aller plus loin dans les couches logicielles de notre application, nous verrons l'application de bonnes pratiques de l'orienté objet plus tard, lorsque le besoin s'en fera ressentir.

2.2.2.5 *Création des composants nécessaires pour voir le test échouer pour les bonnes raisons*

Complétons notre scénario de test pour intégrer le démarrage d'**ApiServer** avant de lancer chaque scénario de test, et la fermeture du serveur à chaque scénario de test.

- Mettez votre curseur dans le corps de **CdApiTest** et appuyez sur **Alt + Insert, SetUp Method**
- Mettez votre curseur dans le corps de **CdApiTest** et appuyez sur **Alt + Insert, TearDown Method** :

```
@BeforeEach
void setUp() {
    ApiServer.start();
}

@AfterEach
void tearDown() {
    ApiServer.stop();
}
```

Puis nous générons automatiquement la classe **ApiServer** dans le package **main** :

- soit clic droit sur **ApiServer**, **Show Context Actions, Create class 'ApiServer'**
- soit si le curseur est sur **ApiServer**, **Alt + Enter, Create class 'ApiServer'**
- **Destination package**, écrivez : **main**
- **Target destination directory**, sélectionnez : « ... \src\main\java\main ».
- Cliquez sur **OK**.

Puis nous pouvons générer automatiquement la méthode **start** et **stop** (en cliquant sur ces méthodes dans **CdApiTest**, **Create method 'start' in 'ApiServer'...**) :

```
public static void start() {
}

public static void stop() {
}
```

2.2.2.6 *Introduction à un serveur Grizzly et à Jersey*

La méthode **start** doit démarrer un **serveur HTTP**.

Nous souhaitons utiliser **Grizzly2** et le framework **Jersey** pour gérer notre API et avoir un serveur HTTP intégré à notre application. Pour ce faire, nous devons ajouter deux dépendances à notre

pom.xml : jersey-container-grizzly2-http et jersey-hk2. Nous pouvons ajouter ces deux dépendances à la main, juste avant la balise `</dependencies>`, dans **pom.xml** :

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-grizzly2-http</artifactId>
  <version>3.1.0</version>
</dependency>

<dependency>
  <groupId>org.glassfish.jersey.inject</groupId>
  <artifactId>jersey-hk2</artifactId>
  <version>3.1.0</version>
</dependency>
```

Il est aussi possible d'utiliser **Alt + Insert** dans **pom.xml** pour ajouter ces deux dépendances.

N'oubliez pas d'actualiser vos dépendances en cliquant sur **Load Maven Changes** et d'importer les classes dans la méthode **start** (par exemple en faisant **Alt + Shift + Enter** sur les classes en rouge).

Pour information :

- **jersey-container-grizzly2-http** : offre un serveur http intégré à une application Java ; le framework **Jersey** met à disposition des annotations et méthodes afin de créer facilement des **RESTful web services**.
- **jersey-hk2** : package permettant de faire de l'**injection de dépendances** via des annotations et des binders. Nous verrons cela plus en détail plus tard.

En consultant [la documentation de Jersey pour un serveur http intégré appelé grizzly2](#), nous arrivons facilement à ce code-ci pour démarrer le serveur HTTP :

```
public class ApiServer {

    private static HttpServer server;

    public static void main(String[] args) {
        start();
    }

    public static void start() {
        URI baseUri =
UriBuilder.fromUri("http://localhost/").port(9998).build();
        ResourceConfig config = new ResourceConfig();
        config.registerClasses(CdResource.class);
        server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        System.out.println("API server started on " + baseUri.getHost() + ":" +
baseUri.getPort());
    }

    public static void stop() {
        server.shutdownNow();
    }
}
```

Pour les classes à importer, importez bien le **HttpServer** de **grizzly**.

Attention, le serveur HTTP ici va écouter sur le port **9998** et mettra à disposition les opérations de notre HTTP API qui se trouveront dans la classe **CdResource** : le lien entre le serveur web et les ressources de type CD offertes par notre future API est fait via cette ligne de code :

```
config.registerClasses(CdResource.class);
```

Générons automatiquement la classe **CdResource** (soit par exemple clic droit sur **ApiServer**, **Show Context Actions**, **Create class 'CdResource'**) dans le package **api** que nous créons automatiquement :

```
public class CdResource {  
}
```

2.2.2.7 Gestion d'opérations HTTP via Jersey

Nous allons maintenant rajouter le minimum de code pour offrir une méthode qui récupère deux paramètres de requête (le **titre** et l'**artiste**) et qui renvoie un message bidon pour faire échouer le test. Pour ce faire, **Jersey** met à disposition des annotations qui permettent de facilement traiter les requêtes transitant par le serveur HTTP. Notons que dans le code qui suit, si dans la requête le titre ou l'artiste n'est pas donné, le framework, par le biais d'annotations, donnera des valeurs par défaut aux paramètres **title** et **artist**. Veuillez mettre à jour **CdResource** :

```
@Path("/cds")  
public class CdResource {  
  
    @GET  
    @Produces(MediaType.TEXT_PLAIN)  
    public String findCds(@DefaultValue("") @QueryParam("title") String  
title, @DefaultValue("") @QueryParam("artist") String artist) {  
        System.out.println("find my CD ? " + title + " artist" );  
        return "No CD found";  
    }  
}
```

NB : quelques explication sur le fonctionnement de Jersey :

- **@Path** permet de spécifier le chemin pris en compte par une opération de notre API donnée dans la **Resource**.
- **@GET** : l'opération est appelée que si la requête est de type **GET**.
- **@Produces (MediaType.TEXT_PLAIN)** : indique que l'opération renvoie du texte au client.
- **@QueryParam** permet de récupérer un paramètre de la requête ;
 - o par exemple, on récupère la valeur "**young mystic**" dans le paramètre **title** si un client fait une requête **GET /cds?title=young mystic** à notre API.
 - o **@DefaultValue** est la valeur d'un paramètre si celui-ci n'est pas donné dans la requête : par exemple pour une requête **GET /cds title** vaudra "".
- Si vous souhaitez plus d'info, vous pouvez jeter un œil à la [documentation de Jersey pour les Resources](#).

2.2.2.8 Finalisation du code pour que le test échoue pour les bonnes raisons

Maintenant que nous avons une API qui écoute les requêtes de type **GET /cds**, il nous reste à compléter le code permettant de faire une requête HTTP dans **CdApiTest** :

```
private String getRequest(String path, String queryParams1) {
    Client c = ClientBuilder.newClient();
    WebTarget target = c.target("http://localhost:9998/");
    String key = queryParams1.split("=")[0];
    String value = queryParams1.split("=")[1];
    String responseBody = target.path(path).queryParam(key, value)
        .request().get(String.class);
    return responseBody;
}
```

NB: ce code a été écrit en utilisant la [documentation de Jersey pour la "Client API"](#).

Point-clé : donner une seule raison à un test d'échouer

On souhaite être capable de pouvoir facilement détecter la méthode qui ne produirait pas les effets attendus. De ce fait, nos tests doivent se focaliser sur généralement une seule assertion, même si cela amène à un peu de duplication au niveau du setup lors de la mise en place du contexte de chacun des tests.

C'est pour cette raison que dans notre premier scénario de test, nous n'avons qu'une seule assertion.

Il nous reste à identifier les objets utiles à notre premier test, afin de compléter **CdResource**. Nous aurons besoin d'un catalogue qui contiendra des CD.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String findCds(@DefaultValue("") @QueryParam("title") String title,
    @DefaultValue("") @QueryParam("artist") String artist) {
    System.out.println("find my CD ? " + title + " artist");
    Catalog catalog = new Catalog();
    List<CompactDisc> cdsFound = catalog.findCds(title, artist);
    return cdsFound.toString();
}
```

Veuillez générer automatiquement **Catalog** et sa méthode **findCds** dans un nouveau package nommé **services**.

```
public class Catalog {

    public List<CompactDisc> findCds(String title, String artist) {
        return new ArrayList<CompactDisc>();
    }
}
```

Veuillez générer automatiquement **CompactDisc** dans un nouveau package nommé **domain**.

```
public class CompactDisc {

}
```

Il ne reste plus qu'à exécuter ce test pour voir que le test échoue bien, pour de bonnes raisons !
Voilà, le squelette principal de notre application est en place 🎉 !

NB : Il y a des warnings en rouge qui sont affichés dans votre terminal. Ca n'est pas grave, ceux-ci pourraient être supprimés en configurant de manière plus soignée Grizzly.

Nous avons volontairement simplifié le squelette afin de ne pas prendre en compte de base de données ou d'objets faisant abstraction des opérations sur des données... 😊

Nous allons maintenant commencer à faire des tests unitaires sur notre **Catalog**. Il ne semble pas intéressant ici de faire des tests unitaires sur un **CompactDisc** qui n'offre pas de comportements « intelligents ».

Nous recommençons donc un cycle de TDD et allons spécifier un test pour **Catalog**.

Pour générer la classe de tests de **Catalog**, mettez le curseur sur le nom de la classe : **Show Context Actions, Create Test**. Donnez le nom **CatalogTest** à la classe de tests et générez une méthode de test. Nous allons tester que le CD est trouvé.

Comme précédemment, on démarre de l'assertion, puis on identifie le setup de test dans la nouvelle classe **CatalogTest** :

```
@Test
void cdIsFound() {
    Catalog catalog = new Catalog();
    CompactDisc cd1 = new CompactDisc("Young Mystic", "Bob Marley", 3, 12,
    "666-777");
    CompactDisc cd2 = new CompactDisc("Time Out", "Dave Brubeck Quartet", 10,
    111.1, "123-456");

    catalog.initCatalog(cd1, cd2);

    assertEquals(1, catalog.findCds("young mystic", "").size()),
    () -> assertEquals(cd1, catalog.findCds("young mystic", "").get(0))
    );
}
```

🤔 Ici, comme nous sommes dans un test unitaire de **Catalog**, avons-nous le droit de créer un objet d'une autre classe (**CompactDisc**) sans passer par un **Stub** ou un **Mock** ?

Dans ce cas-ci, comme l'objet que nous créons n'a pas d'intelligence, que tout ce qu'il va offrir ce sont des getters et des setters, ou en tout cas à ce stade-ci, il n'est pas utile de créer un mock de **CompactDisc**. Néanmoins, si vous souhaitez vous amuser, vous pourriez le faire en créant un mock de chaque **CompactDisc**.

🤔 Normalement, la classe **Catalog** serait connectée à une DB. Les données seraient donc disponibles en faisant des requêtes. Ici, pour maîtriser nos tests, nous souhaitons pouvoir initialiser les données d'un **Catalog**. C'est pourquoi nous avons identifié la méthode **initCatalog** dans le setup du test.

Ecrivons le code minimum pour que le test échoue pour des bonnes raisons. D'abord créons un constructeur dans la classe **CompactDisc** :

```
public class CompactDisc {
    private String title;
    private String artist;
    private int stock;
    private double price;
    private String id;

    public CompactDisc(String title, String artist, int stock, double price,
```

```
String id) {
    this.title = title;
    this.artist = artist;
    this.stock = stock;
    this.price = price;
    this.id = id;
}
}
```

Puis créons la méthode **initCatalog** :

```
public void initCatalog(CompactDisc... defaultItems) {
}
}
```

Il ne reste plus qu'à exécuter ce test pour voir que le test échoue bien, pour de bonnes raisons !

Ici, le test échoue car **findCds** renvoie un tableau vide. En effet, nous n'avons pas encore le corps de la méthode initialisant un **Catalog**.

2.3 TDD – Etape 2 : écrire du code simple passant le test

OK, nous devons écrire le code minimum permettant de passer le test...

Mais ça veut dire quoi ? Nous allons le découvrir au fur et à mesure de l'implémentation du prochain test.

Nous allons initialiser un **Catalog**. De plus, nous allons coder la fonction **findCds**. Pour le scénario considéré, cette fonction doit retourner qu'un seul CD.

Point-clé : écrire l'implémentation qui est évidente en évitant les étapes triviales de tests

Il est inutile d'écrire du code trop trivial comme, par exemple, renvoyer une valeur hardcodée dans une fonction quand on sait pertinemment que cette valeur va changer au cours du temps. Dans ce cas, autant directement créer un attribut. Idem si on doit ajouter un élément à une liste, autant créer cette liste. Il est notamment inutile de tester les getters & setters. Et quand une implémentation n'est pas si évidente, alors n'écrivez que le code permettant de passer le test... Lors des tests qui suivront, vous mettrez à jour l'implémentation pour qu'elle devienne complète.

Ainsi, il semblerait que plus les développeurs deviennent expérimentés, plus ils sentent les implémentations évidentes qui ne nécessitent pas de tests, et donc moins ils écrivent de tests inutiles.

Ici, en gros, il serait trop évident de renvoyer une liste contenant le CD hardcodé dans **findCds**. Nous allons d'abord traiter de la fonction **initCatalog** afin d'initialiser une liste de CD.

```
public class Catalog {
    private List<CompactDisc> cds = new ArrayList<>();

    public List<CompactDisc> findCds(String title, String artist) {
        return new ArrayList<CompactDisc>();
    }
}
```

```

public void initCatalog(CompactDisc... defaultItems) {
    if (defaultItems.length == 0) {
        cds = new ArrayList<>();
    } else {
        cds = Arrays.asList(defaultItems);
    }
}
}

```

Et ensuite, nous pouvons mettre à jour la méthode **findCds** de **Catalog** :

```

public List<CompactDisc> findCds(String title, String artist) {
    if (!title.isBlank()) {
        return cds.stream().filter(cd ->
cd.getTitle().equalsIgnoreCase(title)).toList();
    }
    if (!artist.isBlank()) {
        return cds.stream().filter(cd ->
cd.getArtist().equalsIgnoreCase(artist)).toList();
    }
    return null;
}

```

Nous devons donc créer les getters **getTitle** et **getArtist** dans **CompactCd** :

```

public String getTitle() {
    return title;
}

public String getArtist() {
    return artist;
}

```

Après ces changements, le test unitaire passe 🎉

Nous pouvons voir si le test parent, l'acceptance test donné dans **CdApiTest**, passe lui aussi ?

Non, il échoue toujours, car dans **CdResource**, nous créons un catalogue vide, sans CD. Cela semble être un comportement par défaut normal.

En effet, il ne serait pas logique, à chaque requête, d'hardcoder des CD dans un catalogue. Plus tard, l'API devra offrir l'option de créer des CD. Néanmoins, on se rend compte que, pour les tests d'acceptance, il serait utile de pouvoir bypasser tant le besoin de faire une requête pour créer des données, que de devoir agir sur une éventuelle DB afin d'initialiser les données que l'on souhaiterait en entrée de notre test.

Nous allons voir comment l'injection de dépendances va nous permettre d'injecter des données là où c'est nécessaire et nous permettre de rendre nos tests indépendants.

2.4 Injection de dépendances à l'aide d'un framework

2.4.1 Introduction à l'injection de dépendances

Dans certains scénarios, il est très intéressant de pouvoir, au démarrage d'une application, configurer les services qui vont être injectés dans des attributs ou arguments de certaines classes. Ces attributs ou arguments injectés doivent correspondre à un type abstrait / un contrat / une interface.

La librairie d'injection de dépendances va faire en sorte que, pour les variables associées à une interface pour lesquelles vous indiquez via une annotation qu'elles doivent être injectées, un objet sera créé en utilisant le service associé à cette interface. Le constructeur nécessaire à la création du service sera donc appelé à un moment décidé par le framework, afin de rendre ce service disponible dans les méthodes des classes.

Par exemple, dans notre tutorial :

- Lorsque notre API est lancée normalement, via le main, nous souhaitons utiliser un **Catalog** « standard » comme attribut de **CdResource**.
- Lorsque notre API est lancée dans un test acceptance, nous souhaitons utiliser un **Catalog** qui est peuplé d'éléments par défaut comme attribut de **CdResource**.
- NB : Deux schémas sont donnés plus tard pour montrer comment on peut injecter un **Catalog** à l'aide de la librairie Jersey hk2 : §2.4.4.

2.4.2 Introduction à Jersey hk2

Lorsqu'on utilise **Jersey** et **hk2** comme framework pour gérer l'injection de dépendances dans un service web, il faut configurer son application pour indiquer comment l'injection de dépendances doit créer vos instances.

On fait le « binding » entre des variables de type abstrait, associées à une interface, et leur implémentation, via la redéfinition d'un **AbstractBinder**. Il existe plusieurs moyens de définir un **AbstractBinder**, voici la façon la plus directe :

Voilà à quoi ressemble le binding via **Jersey hk2** :

```
config.register(new AbstractBinder() {
    @Override
    protected void configure() {
        bind(Service.class).to(Contract.class).in(Singleton.class);
    }
});
```

Le binding se fait donc via **bind(service).to(contrat)** et on peut chaîner avec **.in(scope)** et aussi éventuellement **ranked(number)**.

Les **scopes** disponibles sont :

- **PerLookup** : pour chaque appel du contrat, une instance du service est créée.
- **RequestScoped** : une instance est créée pour chaque requête à votre API.
- **Singleton** : une seule instance de votre service est créée par contrat.

Notons qu'il est possible :

- Comme dans le morceau de code ci-dessus, d'indiquer la classe implémentant l'interface (le service) à utiliser lors de l'injection dans une variable associée à cette interface. Dans ce cas-là, l'injection de dépendance se fera automatiquement en utilisant le constructeur par défaut du service.
- D'indiquer un objet qui répond au contrat de l'interface, à utiliser lors de l'injection de variables abstraites. Le code ressemblerait à quelque chose ainsi :

```
config.register(new AbstractBinder() {
    @Override
    protected void configure() {
```

```

        bind(new Service()).to(Contract.class);
    }
});

```

Pour injecter un service, une fois le « binding » réalisé, il suffit d'utiliser l'annotation **@Inject**, comme par exemple :

```

@Inject
private Contract service;

```

2.4.3 Injection de notre premier service via hk2

Dans notre tutoriel, nous souhaitons injecter un **Catalog**.

Auparavant, nous allons créer un type abstrait **Catalog**, une interface, en extrayant cette interface de la classe **Catalog** et en renommant cette classe **Catalog** en **CatalogImpl** (**Refactor, Extract Interface...**). Pensez à ajouter toutes les méthodes présentes dans la classe dans l'interface.

Voici comment configurer de manière basique l'injection d'un **Catalog** via **ApiServer** :

```

public static void start() {
    URI baseUri = UriBuilder.fromUri("http://localhost/").port(9998).build();
    ResourceConfig config = new ResourceConfig();
    config.registerClasses(CdResource.class);

    config.register(new AbstractBinder() {
        @Override
        protected void configure() {
            bind(CatalogImpl.class).to(Catalog.class).in(Singleton.class);
        }
    });

    server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
    System.out.println("API server started on " + baseUri.getHost() + ":" +
baseUri.getPort());
}

```

Et voici pour injecter un **Catalog** dans **CdResource** :

```

@Path("/cds")
public class CdResource {
    @Inject
    private Catalog catalog;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String findCds(@DefaultValue("") @QueryParam("title") String
title,
        @DefaultValue("") @QueryParam("artist") String artist) {
        System.out.println("find my CD ? " + title + " artist");

        List<CompactDisc> cdsFound = catalog.findCds(title, artist);
        return cdsFound.toString();
    }
}

```


Cela récupère la classe **CatalogImpl** qui est configurée dans la méthode **main** et gère l'injection d'une instance dans **catalog** quand nécessaire.

Maintenant, il faut que lorsque nous lançons l'**ApiServer** via **CdApiTest**, nous puissions configurer une injection différente pour le **Catalog**. Nous allons donc créer un **AbstractBinder** (provenant de la librairie **hk2**) qui sera seulement utilisé lors de tests. Nous ajoutons un setter dans **ApiServer** afin de pouvoir mettre à jour l'**AbstractBinder** de l'application si nécessaire : (voir les ajouts associés à l'attribut **applicationBinder**). Voici le code d'**ApiServer** :

```
public class ApiServer {

    private static HttpServer server;

    private static AbstractBinder applicationBinder = new AbstractBinder() {
        @Override
        protected void configure() {
            bind(CatalogImpl.class).to(Catalog.class).in(Singleton.class);
        }
    };

    public static void main(String[] args) {
        start();
    }

    public static void start() {
        URI baseUri =
UriBuilder.fromUri("http://localhost/").port(9998).build();
        ResourceConfig config = new ResourceConfig();
        config.registerClasses(CdResource.class);

        config.register(applicationBinder);

        server = GrizzlyHttpServerFactory.createHttpServer(baseUri, config);
        System.out.println("API server started on " + baseUri.getHost() + ":" +
baseUri.getPort());
    }

    public static void stop() {
        server.shutdownNow();
    }

    public static void setApplicationBinder(
        AbstractBinder applicationBinder) {
        ApiServer.applicationBinder = applicationBinder;
    }
}
```

Et voici le nouveau setup de notre Acceptance Test dans **CdApiTest** :

```
@BeforeEach
void setUp() {
    defaultCatalog = new CatalogImpl();
    CompactDisc cd1 = new CompactDisc("Young Mystic", "Bob Marley", 3, 12,
"666-777");
    CompactDisc cd2 = new CompactDisc("Time Out", "Dave Brubeck Quartet", 10,
111.1, "123-456");
    defaultCatalog.initCatalog(cd1, cd2);
}
```

```

ApiServer.setApplicationBinder((new AbstractBinder() {
    @Override
    protected void configure() {
        bind(defaultCatalog).to(Catalog.class);
    }
}));

ApiServer.start();
}

```

Après ces changements, l'injection est fonctionnelle. Veuillez exécuter le test. Le test ne passe pas encore ! Nous devons écrire le code minimum pour que **CdResource** renvoie une String qui corresponde au format attendu.

Commençons par ajouter un **toString** à **CompactDisc** :

```

@Override
public String toString() {
    return "title:" + title +
        ", artist:" + artist +
        ", stock:" + stock +
        ", price:" + price + "€" +
        ", id:" + id;
}

```

En exécutant l'acceptance test, on se rend compte qu'il y a encore un traitement à faire au niveau de la String à renvoyer. Voici le code de **findCds** mis à jour dans **CdResource** :

```

@GET
@Produces(MediaType.TEXT_PLAIN)
public String findCds(@DefaultValue("") @QueryParam("title") String title,
    @DefaultValue("") @QueryParam("artist") String artist) {
    System.out.println("find my CD ? " + title + " artist");

    List<CompactDisc> cdsFound = catalog.findCds(title, artist);
    String cdsFoundSerialized = cdsFound.stream().map(compactDisc ->
compactDisc.toString())
        .collect(
            Collectors.joining("\n"));
    return cdsFoundSerialized;
}

```

Après ces changement, l'acceptance test passe 🎉.

Exécutons tous les tests pour nous assurer que notre modification de code n'a rien cassé. Pour exécuter tous les tests, faites un clic droit sur le répertoire **/src/test/java** et **Run 'All Tests'**.

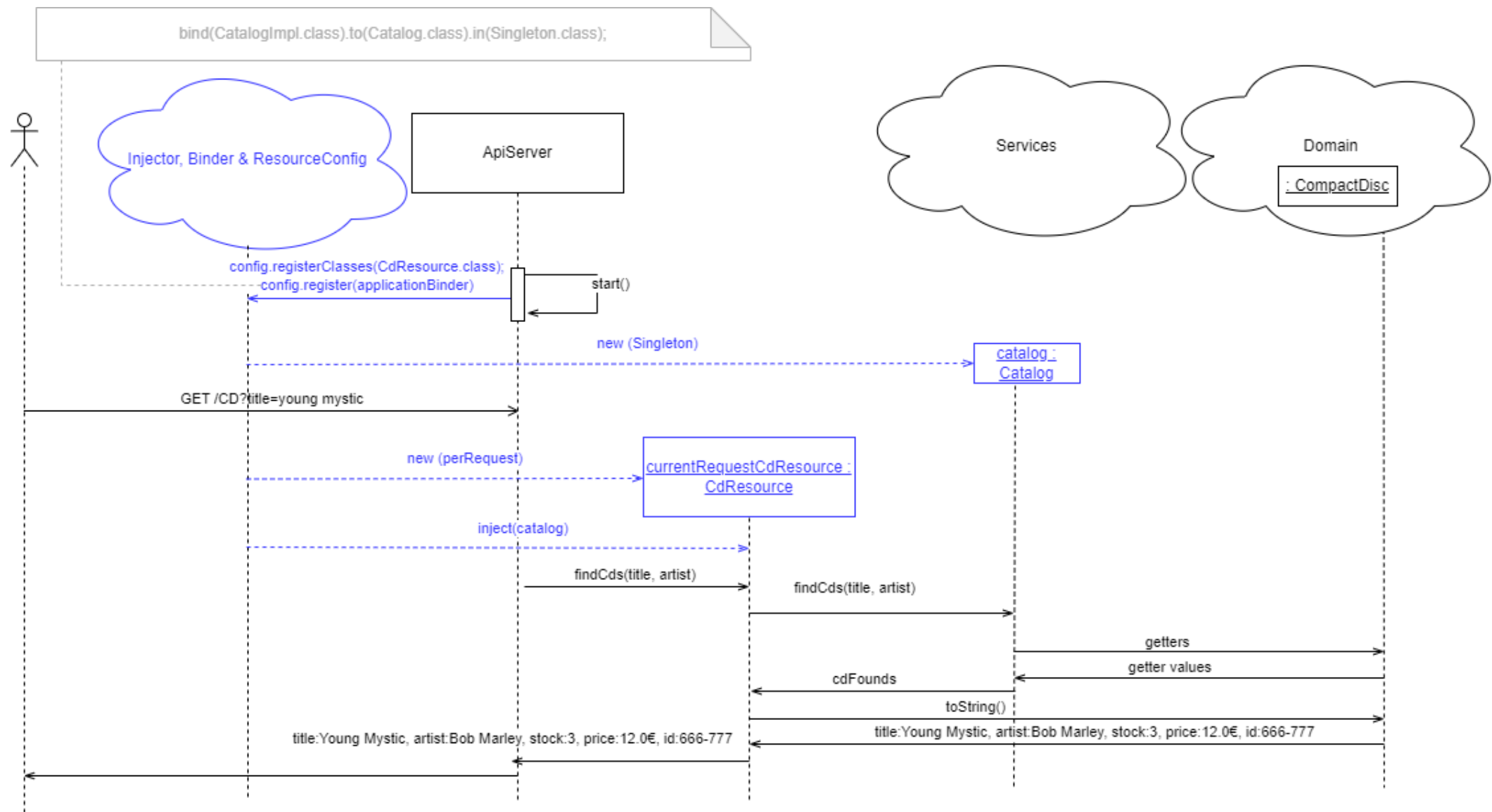
Tout passe 🎉 !

2.4.4 Visualisation de l'injection de dépendances

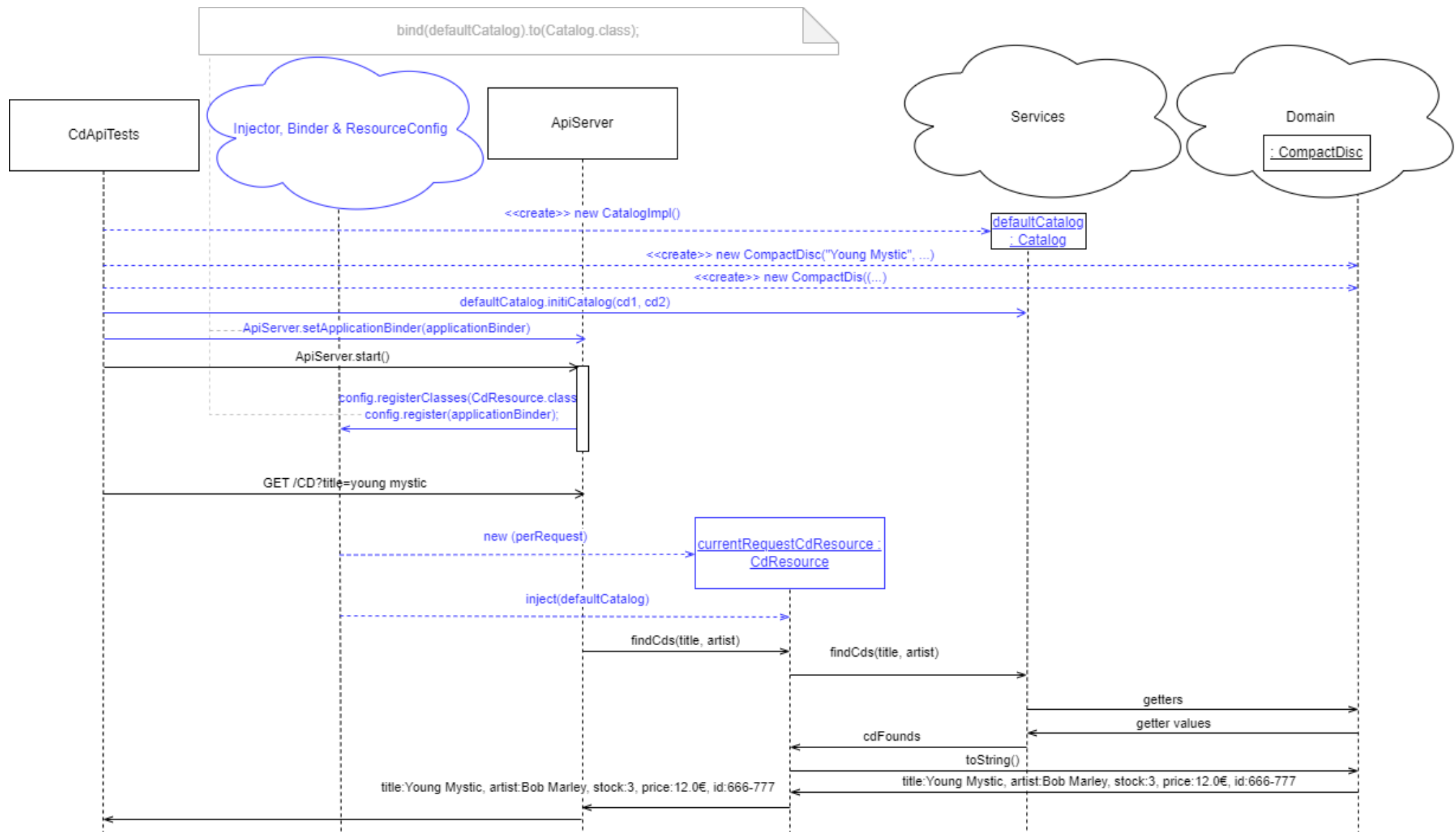
🗨 Avez-vous compris le fonctionnement de l'injection de dépendances ?

Voici deux schémas qui expliquent ce que fait **Jersey hk2** pour injecter un **Catalog** lorsque le **main** est lancé ou lorsqu'un acceptance test est lancé.

Les parties associées à l'injection de dépendances sont données **en bleu** dans les deux diagrammes qui suivent.



Lorsque le main est lancé, on voit que l'injection de dépendances se fait de manière cachée pour créer une instance de **CdResource** à chaque requête faite à l'API ! Une fois une instance de **CdResource** créée, une instance de **CatalogImpl**, en appelant le constructeur par défaut, sera injectée en attribut de l'instance de **CdResource**.



Dans le setting du test d'acceptance, le binding d'un catalogue contenant des données par défaut est fait par le biais d'un setter. Ce catalogue par défaut sera injecté le moment venu comme attribut de l'instance de **CdResource**.

2.4.5 En savoir plus sur l'injection via hk2

NB : Il est possible d'injecter des services d'autres façons, notamment en injectant le ou les paramètre(s) d'un constructeur. Si vous souhaitez en apprendre plus sur l'injection de dépendances à l'aide de **Jersey HK2**, vous pouvez jeter un œil à la documentation : [Custom Injection and Lifecycle Management](#).

Il est notamment possible de lier plusieurs services à une même interface, principalement pour pouvoir redéfinir une implémentation sans devoir changer tous les liens faits entre des classes et des interfaces. Dans ce cas-ci, les **ranked(number)** peuvent être utilisés :

- Pour identifier le service qui sera injecté si plusieurs **bind** sont enregistrés sur un même service, le **number** le plus haut étant le plus prioritaire.
- Dans l'exemple qui suit, ça n'est que **Service2** qui sera utilisé pour injecter une variable de type **Contract** :

```
config.register(new AbstractBinder() {
    @Override
    protected void configure() {
        bind(Service1.class).to(Contract.class).in(Singleton.class).ranked(1);
    }
});

config.register(new AbstractBinder() {
    @Override
    protected void configure() {
        bind(Service2.class).to(Contract.class).in(Singleton.class).ranked(2);
    }
});
```

2.5 TDD – Etape 3 : refactoring

OK, il est temps de mettre en place toutes les bonnes pratiques de l'orienté objet.

Quand est-ce qu'on enlève de la duplication dans notre code ?

Point-clé : appliquer la règle de trois quand il y a de la duplication dans notre code avant de créer du code à réutiliser

La réutilisation de code gagne généralement sur la duplication. Car s'il y a plus de code, il y a plus de risque de bugs, et le code est moins lisible.

Néanmoins, afin de trouver un bon compromis entre créer de l'abstraction trop rapidement, et attendre trop longtemps avant de le faire, en moyenne, on conseille d'attendre qu'il y ait 3 duplications dans notre code avant de faire un refactoring.

Ceci doit être pris comme une moyenne, si vous sentez qu'il y a un pattern qui va souvent revenir dans votre code, vous pouvez bien sûr le faire plus rapidement.

Notons que la duplication peut se trouver tant dans le code des scénarios de test, que dans les objets à tester.

En cas de duplication dans vos scénarios de tests, vérifiez que vos tests restent facilement lisibles. Parfois cela amène à préférer de la duplication dans nos scénarios de tests.

N'hésitez pas à considérer des tests paramétrés en cas de duplication dans vos scénarios de tests pour appeler les mêmes méthodes avec différents arguments.

Suite à l'écriture du code pour faire passer nos deux premiers tests, il n'y a pas tant de duplication de code. Est-ce qu'il y aurait une bonne pratique de l'OO que nous pourrions appliquer ? Est-ce qu'il y aurait une duplication de code comme candidat à du refactoring ?

La création d'un **Catalog** a déjà été optimisée grâce à l'injection de dépendances.

La création de **CompactDisc** est faite actuellement 2 fois dans **CdApiTest** et deux fois dans **CatalogTest**. Afin de centraliser la création de CD, il serait intéressant de créer une factory du domaine.

Pour commencer, nous allons créer un type abstrait **CompactDisc**, en extrayant une interface de la classe **CompactDisc** et en renommant cette classe **CompactDisc** en **CompactDiscImpl**. Notons que le **toString** ne doit pas être rajouté dans l'interface car il est accessible grâce à l'héritage d'**Object**.

Il faut ensuite nous assurer que le type abstrait **CompactDisc** est appelé là où nécessaire au lieu de **CompactDiscImpl**, notamment dans la méthode **intiCatalog**.

Après cela, nous pouvons relancer tous les tests pour nous assurer que tout continue à fonctionner.

Nous allons maintenant créer la **DomainFactory** dans le package **domain** :

```
public class DomainFactory {  
  
    public CompactDisc createCompactDisc(String title, String artist, int  
stock, double price,  
        String id) {  
        return new CompactDiscImpl(title, artist, stock, price, id);  
    }  
}
```

Et maintenant, nous allons utiliser la factory dans **CdApiTest** :

```
class CdApiTest {  
  
    private Catalog defaultCatalog;  
    private DomainFactory factory = new DomainFactory();  
  
    @BeforeEach  
    void setUp() {  
        defaultCatalog = new CatalogImpl();  
        CompactDisc cd1 = factory.createCompactDisc("Young Mystic", "Bob  
Marley", 3, 12, "666-777");  
        CompactDisc cd2 = factory.createCompactDisc("Time Out", "Dave Brubeck  
Quartet", 10, 111.1, "123-456");  
    }  
}
```

Et voici la mise à jour de **CatalogTest** pour utiliser la factory :

```
class CatalogTest {  
    private DomainFactory factory = new DomainFactory();  
  
    @Test  
    void cdIsFound() {  
        Catalog catalog = new CatalogImpl();  
        CompactDisc cd1 = factory.createCompactDisc("Young Mystic", "Bob  
Marley", 3, 12, "666-777");  
        CompactDisc cd2 = factory.createCompactDisc("Time Out", "Dave Brubeck  
Quartet", 10, 111.1, "123-456");  
    }  
}
```

Une fois ces changements faits, veuillez-vous assurer que tous les tests continuent de passer !

2.6 TDD : continuer les itérations

Le but est de continuer à développer sur base de tests jusqu'à ce que tous les UC soient implémentés.

Pour ce tutoriel, nous allons simplement implémenter les deux scénarios restant associés à l'UC « Chercher un CD ».

Prochain scénario à tester : « Chercher un CD : Pas de CD trouvé - retourne "No cds" ».

Voici l'acceptance test associé à ajouter dans **CdApiTest** :

```
@Test
void cdNotFoundInCatalog() {
    String actualResponse = getRequest("cds", "title=there once was a song");
    assertEquals("No cds", actualResponse);
}
```

Le test ne passe pas.

Il faut mettre à jour la méthode **findCds** de **CdResource** :

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String findCds(@DefaultValue("") @QueryParam("title") String title,
    @DefaultValue("") @QueryParam("artist") String artist) {
    System.out.println("find my CD ? " + title + " artist");

    List<CompactDisc> cdsFound = catalog.findCds(title, artist);
    if(cdsFound.isEmpty())
        return "No cds";
    String cdsFoundSerialized = cdsFound.stream().map(compactDisc ->
compactDisc.toString())
        .collect(
            Collectors.joining("\n"));
    return cdsFoundSerialized;
}
```

Après ce changement, l'acceptance test passe 🎉 !

Est-ce qu'il y a du refactoring à faire ? La règle de 3 est respectée...

🗯 Est-il utile de réaliser un unit test de **Catalog** si le scénario d'acceptance passe ? Dans ce cas-ci, il est difficile de trouver une valeur ajoutée à un test de la méthode **findCds** de **Catalog** pour vérifier que la méthode renverrait **une liste vide** dans certains cas limites.

🗯 Il est intéressant de se demander s'il est approprié de créer le format de la String (**cdsFoundSerialized** ci-dessus) à renvoyer au client directement dans **CdResource**. Quelle classe serait la meilleure pour avoir cette responsabilité ? Si on ne souhaitait plus renvoyer du texte au client mais du JSON... On sent qu'il y aurait un intérêt à abstraire ce code. Néanmoins, si nous n'allons pas plus loin dans le développement de cette application, cela n'est pas nécessaire. En effet,

actuellement, la règle de 3 est respectée, nous n'avons pas 3 fois le code formatant des CD au format texte.

NB : Selon la documentation [JAX-RS Entity Providers](#), nous pourrions nous rediriger vers la création d'un **MessageBodyWriter**. En effet, Jersey met à disposition la possibilité d'écrire ses « **entity provider** » personnalisés : dans ce cas, ça serait une classe qui étendrait **MessageBodyWriter<CompactDisc>**,

Nous allons donc passer au dernier scénario d'acceptance que nous souhaitons tester ensemble : « Chercher un CD : Multiples CD trouvés (si un CD a plusieurs versions) – retourne la liste de CD ».

Voici l'acceptance test associé dans **CdApiTest** :

```
@Test
void multipleCdsFoundInCatalog() {
    String expectedResponse = "title:Time Out, artist:Dave Brubeck Quartet,
stock:10, price:111.1€, id:123-456";
    expectedResponse += "\ntitle:Time Out, artist:Dave Brubeck Quartet,
stock:1, price:98.0€, id:123-457";
    expectedResponse += "\ntitle:Time Out, artist:Dave Brubeck Quartet,
stock:150, price:28.5€, id:123-458";
    String actualResponse = getRequest("cds", "title=time out");
    assertEquals(expectedResponse, actualResponse);
}
```

Pour que ce test passe, nous devons aussi ajouter deux nouveaux CD au catalogue par défaut qui sera injecté dans **CdResource**. Dans **CdApiTest**, veuillez ajouter les deux derniers CD :

```
@BeforeEach
void setUp() {
    defaultCatalog = new CatalogImpl();
    CompactDisc cd1 = factory.createCompactDisc("Young Mystic", "Bob Marley",
3, 12, "666-777");
    CompactDisc cd2 = factory.createCompactDisc("Time Out", "Dave Brubeck
Quartet", 10, 111.1, "123-456");
    CompactDisc cd3 = factory.createCompactDisc("Time Out", "Dave Brubeck
Quartet", 1, 98, "123-457");
    CompactDisc cd4 = factory.createCompactDisc("Time Out", "Dave Brubeck
Quartet", 150, 28.5, "123-458");
    defaultCatalog.initCatalog(cd1, cd2, cd3, cd4);

    ApiServer.setApplicationBinder((new AbstractBinder() {
        @Override
        protected void configure() {
            bind(defaultCatalog).to(Catalog.class);
        }
    }));

    ApiServer.start();
}
```

Le test passe directement sans devoir modifier le code 🎉!

Pour ce tutoriel, nous n'allons pas aller plus loin dans le développement de notre application de location de CD.

2.7 Les mocks

Lorsque vous faites des tests unitaires sur une classe, vous ne devez pas tester les objets qui ne correspondent pas au type de la classe que vous êtes en train de tester. Bien sûr, si les collaborateurs de la classe que vous testez sont des objets sans « intelligence », il n'est pas utile de créer des Mocks pour ces objets.

De même, lors de tests d'intégration ou d'acceptance tests, on souhaite parfois se détacher de l'utilisation d'un service, et utiliser un Mock à la place. Imaginez par exemple que l'on souhaite se détacher de l'horloge système... Nous pourrions créer un Mock object qui permettrait de configurer l'heure lors de l'appel d'une méthode...

Si vous devez créer des « Mocks objects », nous vous recommandons d'utiliser la librairie **Mockito**.

Pour votre exercice, vous allez devoir utiliser Mockito. Veuillez ajouter Mockito à votre application en ajoutant cette dépendance à votre **pom.xml** : **mockito-junit-jupiter**.

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>4.8.0</version>
</dependency>
```

N'oubliez pas d'actualiser vos dépendances en cliquant sur **Load Maven Changes**.

2.8 TDD en résumé

- Commencez par identifier une liste d'exigences fonctionnelles contenant les cas d'utilisation (use cases ou UC).
- Identifiez une liste de tests.
- Parmi la liste de tests, commencez par le scénario de test le plus intéressant, celui qui forcera à mettre en place le squelette de l'application.
- Pour chaque test que vous avez identifié, appliquez ce processus :
 - 1. Ecrire un test qui échoue
 - Travaillez à l'envers à partir des assertions.
 - Visualisez le test qui échoue pour de bonnes raisons (pas de **NullPointerException** par exemple).
 - Donnez une seule raison à un test d'échouer.
 - 2. Ecrire le code minimum pour faire passer le test

Ecrivez une implémentation évidente tout en évitant les étapes triviales de tests (par exemple, évitez de hardcoder une réponse, préférez par exemple la création d'un attribut)
 - 3. Refactor du code
 - Appliquez la règle de trois : dès qu'il y a 3 duplications dans votre code, remplacez celui-ci par du code réutilisable (par exemple via la création et l'utilisation de fonction, de factory, l'injection de dépendances...) ; sinon, laissez votre code en l'état.
 - Assurez-vous que tous les tests continuent à passer après le refactor.
- Il est normal de ne pas faire réussir un acceptance test dès le début. L'acceptance test va permettre d'identifier les composants principaux de l'application. Pour chaque composant

créé, on peut passer à des tests unitaires. Une fois tous les composants créés via des scénarios de tests unitaires, l'acceptance test devrait réussir !

3 Exercices

3.1 Introduction

Hello chers développeurs, la société FREE WEB SERVICES rêve de pouvoir faire appels aux étudiants pour développer sa prochaine HTTP API, un service très sérieux de blagues.

Le côté très sérieux de ce service : FREE WEB SERVICES s'est rendu compte qu'il est utile de faire rire ses clients, mais il faut aussi mettre en place un mécanisme pour s'assurer de la santé d'esprit de leurs lecteurs. En effet, certains utilisateurs deviennent parfois totalement accros au service et ne peuvent plus s'arrêter.

De plus, FREE WEB SERVICES s'est rendu compte de l'importance d'avoir un code qui ne fait que ce qui est demandé, et pas plus ! Dès lors, cette société impose à leurs développeurs de faire du TDD !

Vous allez développer **en TDD** la HTTP API mettant en œuvre ces **cas d'utilisation** :

- **Envoyer toutes les blagues**

Une blague contient une catégorie, un titre, un contenu et un auteur. Suite à une requête **GET /jokes**, les blagues seront affichées dans l'ordre dans lequel elles ont été ajoutées, ainsi :
"id:..., category:..., title:..., content:..., author:..."
id:..., category:..., title:..., content:..., author:..."
id:..., category:..., title:..., content:..., author:..."
S'il n'y avait pas de blague, l'API renverrait : "No jokes".

Créer une blague

Une blague peut être créée en faisant une requête **GET /jokes/add?category=value1&title=value2&content=value3&author=value4** en incluant donc ces paramètres de requêtes : **category**, **title**, **content**, **author**.
Si un de ces paramètres manque, l'API renverra l'info suivante dans le corps de la réponse : "Bad request". Si tous les paramètres sont donnés, l'API répondra : "Joke added with id : XYZ". XYZ est à remplacer par un **id** généré à l'aide de la librairie **UUID**.

3.2 Exercice 1 : Création de la liste de tests

Veuillez commencer par identifier la liste **des acceptance tests** sur base des UC demandés !

N'hésitez donc pas à créer un document Word reprenant vos scénarios de tests, selon la forme proposée au §2.2.1.

3.3 Exercice 2 : Lecture de blagues

Vous allez développer **en TDD** l'UC « Envoyer toutes les blagues » qui répond aux requêtes de type **GET /jokes**.

Vous ne devez pas recréer un nouveau squelette d'application. Veuillez utiliser le même projet que celui associé au tutoriel donné dans cette fiche : vous allez simplement y ajouter des classes dans les bons packages, tout en réutilisant le code de la classe **ApiServer** et éventuellement d'autres classes.

En cas de souci lors de la mise en place du code de votre tutoriel, vous pouvez créer un nouveau projet en repartant de ce code-ci : [AJ Atelier 10](#) .

Pensez à créer votre premier acceptance test, en partant des assertions, et en identifiant ensuite le setup du test : vous allez créer des classes et des méthodes en utilisant la génération automatique offerte par IntelliJ. A ce stade-ci, vous devriez avoir créé le squelette d'un service de « Jokes ».

Puis, vous allez créer une ressource Jersey de type « Joke » qui offrira l'opération de lecture des blagues ; cette ressource utilisera les notations présentées au §2.2.2.7 ; elle permettra d'écouter les requêtes HTTP et s'occupera des réponses à ces requêtes en faisant à votre service de « Jokes ».

Pensez à configurer le serveur (**config.registerClasses** dans **ApiServer**) en lui indiquant la nouvelle ressource que vous créez pour répondre aux requêtes des clients, comme par exemple : **config.registerClasses(CdResource.class, JokeResource.class);**

N'hésitez pas à créer une classe **Joke** et à l'utiliser dans le service de « Jokes » qui possédera une liste de **Joke**. Pour vos tests, il n'est pas utile de créer un **Mock** de **Joke** quand il s'agit d'un objet qui n'a pas vraiment d'intelligence.

Démarrez le test unitaire de votre service de « Jokes » et implémenter le code pour que le test unitaire passe.

Une fois réussi, vous pouvez passer à l'injection de dépendances via Jersey HK2 tel qu'expliqué au §2.4. L'idée est d'injecter, pour chaque acceptance test, une service de « Jokes » qui renverra les données auxquelles on s'attend.


3.4 Exercice 3 : Création de blagues

Veuillez développer **en TDD** l'UC « Créer une blague » qui répond aux requêtes de type **GET /jokes/add?category=value1&title=value2&content=value3&author=value4**.

Attention, si vous initialisez votre liste de blagues directement via **Arrays.asList(defaultItems)**, votre liste sera immuable. N'hésitez pas à créer une nouvelle liste via : **new ArrayList<>(Arrays.asList(defaultItems));**

Votre service gérant votre liste de blagues est à utiliser dans la ressource de blagues. Attention pour les tests d'acceptance, vous devez vérifier, après avoir ajouté une blague, qu'à la prochaine requête de lecture de blagues, cette blague existe toujours !

Pour la génération d'identifiant, pensez à jeter un œil aux fonctions offertes par la classe **UUID**.

 **Optionnel** : Pour le binding, si vous ne souhaitez pas inclure à chaque test toutes les classes à binder, vous pouvez jouer avec la priorité & **ranked**, comme présenté au §2.4.5. N'hésitez pas à ajouter **ranked(1)** pour tous les **bind** que vous faites dans **ApiServer** pour **applicationBinder**. Si vous ajoutiez un **additionalApplicationBinder** et un **setter** associé dans **ApiServer**, vous pourriez utiliser ce setter pour juste mettre à jour un **bind** dans un scénario de test en faisant appel à ce nouveau

ApiServer.setAdditionalApplicationBinder : mettez **ranked(2)** pour tous les **bind** que vous passeriez à **ApiServer** via **setAdditionalApplicationBinder**. Pour que cela fonctionne, vous devriez aussi faire un **register** du **additionalApplicationBinder** si celui-ci n'est pas nul, au sein d'**ApiServer**.

3.5 Challenges optionnels

3.5.1 Exercice 4 : Lecture aléatoire de blagues

Veillez compléter **en TDD** la HTTP API précédente en mettant en œuvre ces **cas d'utilisation** :

- **Envoyer une blague aléatoire**

Une blague contient une catégorie, un titre, un contenu et un auteur. Suite à une requête **GET /jokes/random**, la blague aléatoire sera affichée ainsi :

"id:..., category:..., title:..., content:..., author:...".

S'il n'y avait pas de blague, l'API renverrait : "No jokes".

Pour sélectionner une blague au sein d'une liste de manière aléatoire, vous pouvez générer un index aléatoire : découvrez la classe **Random**.

Si vous souhaitez ajouter une route répondant à **/jokes/random**, vous pouvez taguer la classe ressource par **@Path("/jokes")** et l'opération offrant une blague aléatoire par **@Path("/random")**.

3.5.2 Exercice 5 : Limitation des accès aux opération à une plage horaire

Veillez compléter **en TDD** la HTTP API précédente en mettant en œuvre ces **cas d'utilisation** :

- **Limiter toutes les opérations de l'API à certaines périodes de la journée**

Pour le cas d'utilisation précédent, ceux-ci ne doivent renvoyer les services attendus qu'entre 6AM et 12PM heure belge. Entre 12PM et 6 AM, les opérations de l'API doivent toujours renvoyer ce message :

"At FREE WEB SERVICES we are doing our best to provide serious jokes. To protect your from addiction, we don't provide any services at night. See you later in the day."

Pour pouvoir agir sur la plage horaire lors de vos tests, il est intéressant de pouvoir fixer l'heure associée à l'heure actuelle. Pour ce faire, voici ce que nous vous proposons :

- Créez un service qui offrira comme méthode l'heure associée à l'heure actuelle (un entier, par exemple **17** s'il est 17h46), et un autre service qui vérifiera si l'heure associée à l'heure actuelle est dans les heures autorisées pour les accès aux opérations des API ;
- Faites en sorte que le service indiquant l'heure associée à l'heure actuelle soit injecté là où il est utile.

Vous devez donc extraire une interface de votre nouvelle classe fournissant l'heure associée à l'heure actuelle, et faire le binding de cette classe à votre interface tant dans **ApiServer** que dans vos classes de tests où vous souhaitez « **mock**er » le temps.

- Au niveau des acceptance tests, cela vous permettra d'injecter un mock stipulant l'heure associée à l'heure actuelle.