

Spring Boot au quotidien

Découvrir Spring Boot

Configurer Spring Boot

Utiliser les fonctionnalités de Spring Boot

Tester une application Spring Boot

Sur 2 jours

- Objectifs :
 - Comprendre le développement d'applications Spring Boot avec des configurations minimales.
 - Décrire les principales caractéristiques de Spring Boot:
 - Starters,
 - Auto-configuration,
 - Beans,
 - Actuator
 - Et plus ...
- Prérequis
 - Java, Maven, Spring, REST

1. Qu'est ce qu'un micro service

- Spring Boot est un framework Java open source utilisé pour créer un micro-service.
 - Développé par l'équipe Pivotal, il est utilisé pour créer des applications de autonomes et prêtes à la production.
- Micro Service est une architecture qui permet aux développeurs de développer et de déployer des services de manière indépendante.
- Chaque service en cours d'exécution a son propre processus, ce qui permet d'obtenir le modèle léger pour prendre en charge les applications métier.
- Micro services offrent les avantages suivants à ses développeurs:
 - Déploiement facile
 - Evolutivité simple
 - Compatible avec les conteneurs
 - Configuration minimale
 - Temps de production moindre

1. Qu'est-ce que Spring Boot?

- Spring Boot fournit aux développeurs Java une plate-forme pour développer une application Spring autonome et de niveau production simple à exécuter.
 - Un développeur utilise des configurations minimales sans avoir besoin d'une configuration complète de la configuration Spring.
- Spring Boot offre à ses développeurs les avantages suivants:
 - Facile à comprendre et à développer des applications Spring
 - Augmente la productivité
 - Réduit le temps de développement
- Spring Boot est conçu avec les objectifs suivants -
 - Pour éviter une configuration XML complexe de Spring
 - Développer plus facilement des applications Spring prêtes pour la production
 - Pour réduire le temps de développement et exécuter l'application indépendamment
 - Offrir un moyen plus simple de démarrer avec l'application

1. Pourquoi Spring Boot?

- Spring Boot a les caractéristiques et les avantages suivants:
 - Il fournit un moyen flexible de configurer Java Beans, des configurations XML et des transactions de base de données.
 - Il fournit un traitement par batch puissant et gère les endpoints REST.
 - Dans Spring Boot, tout est configuré automatiquement. aucune configuration manuelle n'est nécessaire.
 - Il offre une application Spring basée sur des annotations
 - Facilite la gestion des dépendances
 - Il comprend un conteneur de servlets intégré

1. Comment ça marche?

- Spring Boot configure automatiquement une application en fonction des dépendances ajoutées au projet à l'aide de l'annotation `@EnableAutoConfiguration`.
 - Par exemple, si la base de données MySQL se trouve sur le chemin d'accès aux classes, mais que l'on n'a configuré aucune connexion de base de données, Spring Boot configure automatiquement une base de données en mémoire.
- Le point d'entrée de l'application de démarrage printanière est la classe qui contient l'annotation `@SpringBootApplication` et la méthode principale.
- Spring Boot analyse automatiquement tous les composants inclus dans le projet en utilisant l'annotation `@ComponentScan`.

1. Starters de Spring Boot

- La gestion de la dépendance est une tâche difficile pour les grands projets. Spring Boot résout ce problème en fournissant un ensemble de dépendances pour simplifier la création d'un projet.
 - Par exemple, pour utiliser Spring et JPA pour l'accès à la base de données, il suffit d'inclure la dépendance `spring-boot-starter-data-jpa` dans votre projet.
- Notez que tous les starters Spring Boot suivent le même modèle de nommage `spring-boot-starter-*`, où `*` indique qu'il s'agit d'un type d'application.
- Exemple
 - La dépendance Spring Boot Starter Actuator est utilisée pour surveiller et gérer une application. Son code est indiqué ci-dessous -

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

- La dépendance Spring Boot Starter Security est utilisée pour Spring Security. Son code est indiqué ci-dessous -

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

Exemples Starter Spring Boot

- La dépendance Web Spring Boot Starter est utilisée pour écrire un endpoint REST. Son code est indiqué ci-dessous -

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

- Spring Boot Starter La dépendance de Thyme Leaf est utilisée pour créer une application Web. Son code est indiqué ci-dessous -

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

- La dépendance Spring Boot Starter Test est utilisée pour écrire des scénarios de test. Son code est indiqué ci-dessous -

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


2. Configuration automatique

- Observez le code suivant pour une meilleure compréhension –

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

@EnableAutoConfiguration
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

2. Application Spring Boot

- Le point d'entrée de l'application Spring Boot est la classe qui contient l'annotation `@SpringBootApplication`.
- Cette classe doit avoir la méthode principale pour exécuter l'application Spring Boot.
- L'annotation `@SpringBootApplication` inclut la configuration automatique, l'analyse des composants et la configuration de démarrage de Spring.
- Si on ajoute une annotation `@SpringBootApplication` à la classe, on n'a pas besoin d'ajouter les annotations `@EnableAutoConfiguration`, `@ComponentScan` et `@SpringBootConfiguration`.
- L'annotation `@SpringBootApplication` inclut toutes les autres annotations.
- Observez le code suivant pour une meilleure compréhension –

```
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication  
public class DemoApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
}
```

2. Component Scan

- L'application Spring Boot analyse tous les déclarations de beans et de packages lors de son initialisation.
- Il faut ajouter l'annotation `@ComponentScan` à votre fichier de classe pour analyser les composants ajoutés au projet.
- Observez le code suivant pour une meilleure compréhension –

```
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

3. CLI de Spring Boot

- Spring CLI est un outil de ligne de commande pour exécuter les scripts Groovy.
 - C'est le moyen le plus simple de créer une application Spring Boot à l'aide de l'interface de ligne de commande Spring Boot.
 - Il faut créer, exécuter et tester l'application dans l'invite de commande elle-même.
- La distribution Spring CLI est téléchargeable depuis <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#getting-started-installing-spring-boot>
- Pour une installation manuelle, on utilise les 2 dossiers suivants -
 - spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.zip
 - spring-boot-cli-2.0.0.BUILD-SNAPSHOT-bin.tar.gz
- Après le téléchargement, décompresser le fichier archive et suivre les étapes décrites dans le fichier install.txt. Non pas que cela ne nécessite aucune configuration de l'environnement.

3. CLI de Spring Boot

- Sous Windows, accès au répertoire `bin` de l'interface de ligne de commande Spring Boot dans l'invite de commande,
 - exécuter la commande `spring --version` pour vous assurer que cette interface est correctement installée.
 - On peut voir la version CLI de Spring comme indiqué ci-dessous –

```
C:\spring-2.1.1.BUILD-SNAPSHOT\bin>spring.bat --version
Spring CLI v2.1.1.BUILD-SNAPSHOT
```

3. Exemple d'exécution de script Groovy

- Après la création un fichier groovy simple contenant le script Rest Endpoint et exécutez-le avec la CLI Spring Boot.

```
@Controller
class Example {
    @RequestMapping("/")
    @ResponseBody
    public String hello() {
        "Hello Spring Boot"
    }
}
```

```
C:\spring-2.1.1.BUILD-SNAPSHOT\bin>spring run hello.groovy  
Resolving dependencies.....  
  
      .  
    /\ \ /____' _ _ _ _ ( _ ) _ _ _ _ \\\ \ \ \ \ \\  
  ( ( ) \___| | ' _ | | | | ' _ \ V _ \ | \ \ \ \ \ \  
  \ \ / ____| |_) | | | | | | | | | | | | ( _ | | ) ) ) )  
    ' | ____| . _ | | | | | | | | | | | | \__, | / / / / /  
=====|_|=====|_____/=//_/_/_/_/  
:: Spring Boot :: (v2.1.1.BUILD-SNAPSHOT)
```

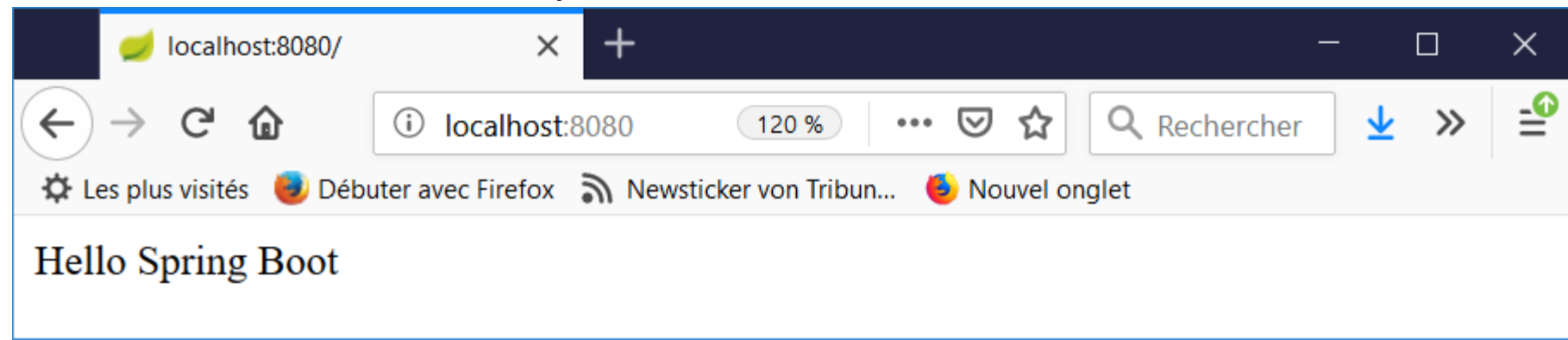
- Enregistrer le fichier groovy sous le nom `hello.groovy`.
- Dans cet exemple, le fichier groovy est enregistré dans le répertoire `bin` de Spring Boot CLI.
- Lancez l'application en utilisant la commande `spring run hello.groovy` comme indiqué dans la capture d'écran ci-dessous -

3. Exemple d'exécution de script Groovy

- Une fois le fichier groovy interprété, les dépendances requises seront automatiquement téléchargées et l'application démarrera dans le port Tomcat 8080

```
2018-11-10 19:55:39.583 INFO 20988 --- [runner-0] o.s.boot.SpringApplication : Starting application on fabrice-HP  
with PID 20988 (started by fabrice in C:\spring-2.1.1.BUILD-SNAPSHOT\bin)  
2018-11-10 19:55:39.589 INFO 20988 --- [runner-0] o.s.boot.SpringApplication : No active profile set, falling bac  
k to default profiles: default  
2018-11-10 19:55:41.514 INFO 20988 --- [runner-0] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8  
080 (http)  
2018-11-10 19:55:41.565 INFO 20988 --- [runner-0] o.apache.catalina.core.StandardService : Starting service [Tomcat]  
2018-11-10 19:55:41.565 INFO 20988 --- [runner-0] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache To  
mcat/9.0.12  
2018-11-10 19:55:41.585 INFO 20988 --- [runner-0] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native
```

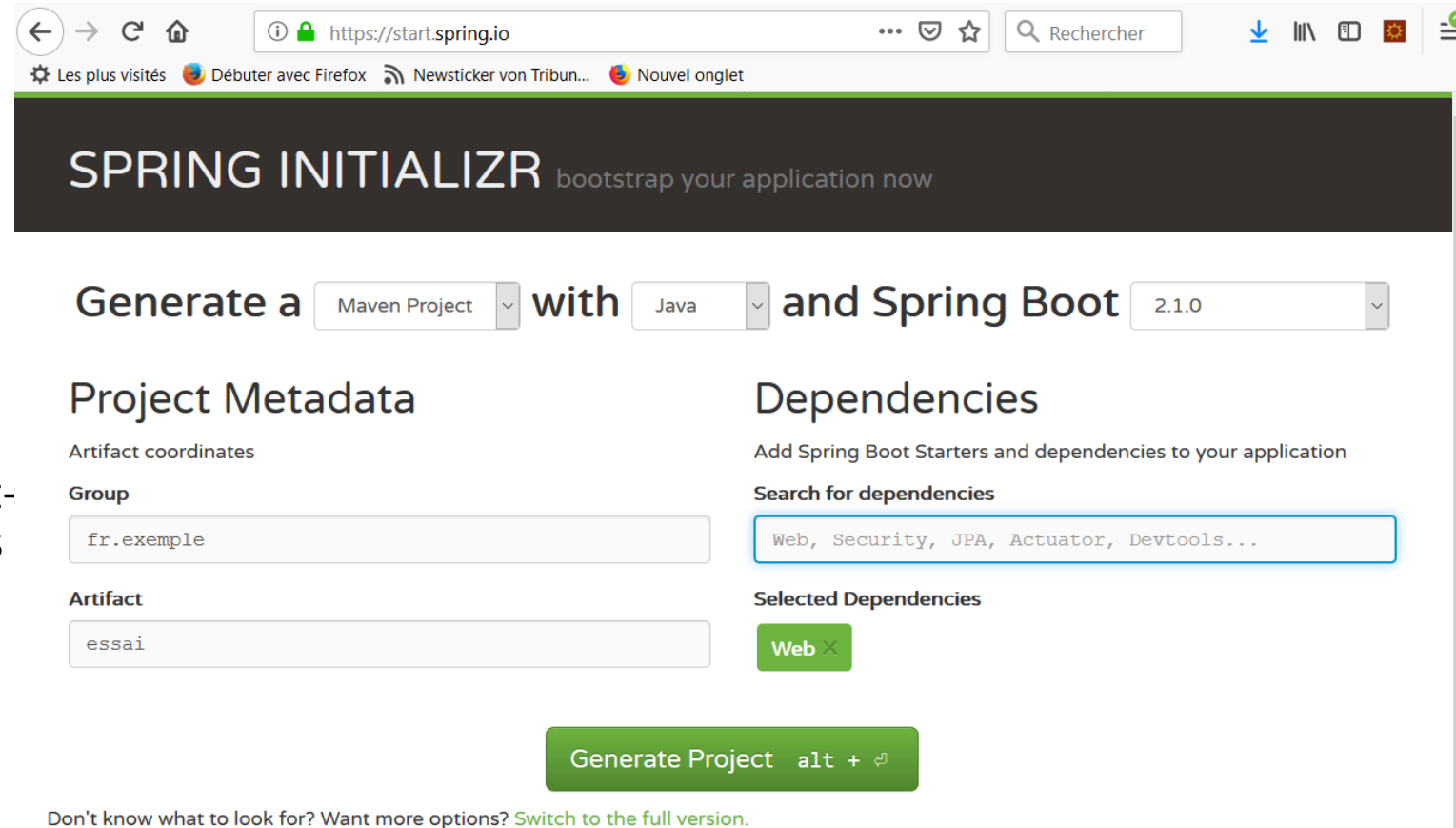
- Une fois Tomcat démarré, accédez au navigateur Web et tapez l'adresse URL `http://localhost:8080/` pour voir la sortie.



4. Spring Boot - Bootstrapping

- **Spring Initializer**

- L'un des moyens de démarrer une application Spring Boot consiste à utiliser Spring Initializer.
 - visiter la page Web de Spring Initializer <https://start.spring.io/>
 - choisir une version et une plate-forme.
 - fournir un groupe, un artefact et les dépendances requises pour exécuter l'application.
- Exemple où la dépendance spring-boot-starter-web est ajoutée pour écrire des endpoints REST.
- Une fois fourni les groupes, artefacts, dépendances, génération de projet, plate-forme et version, un clic sur le bouton Générer un projet. Le fichier zip sera téléchargé et les fichiers seront extraits.



The screenshot shows the Spring Initializer web application in a browser. The URL bar shows <https://start.spring.io>. The page has a dark header with the text "SPRING INITIALIZR" and "bootstrap your application now". Below the header, there are dropdown menus to "Generate a" (Maven Project), "with" (Java), and "and Spring Boot" (2.1.0). The main content is divided into two columns: "Project Metadata" and "Dependencies". Under "Project Metadata", there are input fields for "Group" (fr.exemple) and "Artifact" (essai). Under "Dependencies", there is a search bar with the text "Web, Security, JPA, Actuator, Devtools..." and a "Selected Dependencies" section showing "Web" with a close button. At the bottom, there is a large green button labeled "Generate Project" with a keyboard shortcut "alt + ⌘". A footer note says "Don't know what to look for? Want more options? [Switch to the full version.](#)"

4. pom.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>

<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>fr.exemple</groupId>
  <artifactId>essai</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
  </properties>
```

- **Maven**
- **Après avoir téléchargé le projet, décompressez le fichier. Le fichier pom.xml configure le futur projet**

```
<dependencies>

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

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

</dependencies>

<build>

  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>

</build>

</project>
```

4. Explication du pom.xml

- **Class Path Dependencies**

- Spring Boot fournit un certain nombre de starters pour ajouter les fichiers jar dans le chemin de CLASSPATH.
- Par exemple, pour écrire un endpoint Rest. On ajoute la dépendance `spring-boot-starter-web` dans le CLASSPATH.

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

4. Explication du projet généré

- **Méthode main**

- La méthode `main` doit être écrite dans la classe principale Spring Boot Application. Cette classe doit être annotée avec `@SpringBootApplication`.
 - C'est le point d'entrée de l'application Spring Boot à démarrer.
 - Le fichier de classe principale est sous les répertoires `src/java/main` dans le package par défaut.

```
package fr.exemple.essai;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EssaiApplication {
    public static void main(String[] args) {
        SpringApplication.run(EssaiApplication.class, args);
    }
}
```

4. Explication du projet généré

- **Ecrire un endpoint REST**
 - Pour écrire un endpoint Hello World Rest dans le package principal de l'application Spring Boot, il faut:
 - Modifier la classe, ajoutez l'annotation `@RestController` en haut de la classe.
 - Ecrire une méthode annotée `@RequestMapping`, qui renvoie la chaîne Hello World.

```
package fr.exemple.essai;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class EssaiApplication {
    public static void main(String[] args) {
        SpringApplication.run(EssaiApplication.class, args);
    }
    @RequestMapping(value = "/")
    public String hello() {
        return "Hello World";
    }
}
```

4. Explication de la création du livrable

- Création du jar exécutable

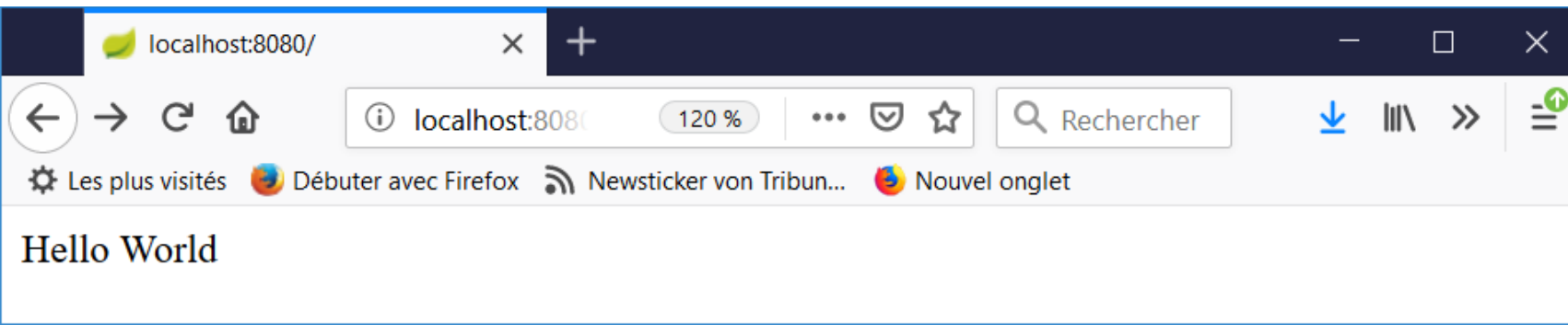
- Pour créer un fichier JAR exécutable et exécuter l'application Spring Boot en utilisant les commandes Maven dans l'invite de commande: charger les dépendances

```
H:\sac\essai>c:\apache-maven-3.3.9\bin\mvn dependency:resolve
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.1.0.RELEASE/
```

- Construire le livrable

```
H:\sac\essai>c:\apache-maven-3.3.9\bin\mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building essai 0.0.1-SNAPSHOT
[INFO] -----
```

- Le jar se trouve dans le répertoire target: `java -jar essai-0.0.1-SNAPSHOT.jar`



5. Déploiement sous tomcat

- En utilisant l'application Spring Boot, on peut créer un fichier war à déployer sur le serveur Web.
- **Spring Boot Servlet Initializer**
- La méthode de déploiement traditionnelle consiste à étendre la classe `SpringBootServletInitializer` par la classe `Spring Boot Application` annotée `@SpringBootApplication`.
- La classe `SpringBootServletInitializer` permet de configurer l'application lorsqu'elle est lancée à l'aide du conteneur Servlet.
- Par défaut le code pour une application Spring Boot est le suivant

```
package fr.exemple.essai;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class EssaiApplication {
    public static void main(String[] args) {
        SpringApplication.run(EssaiApplication.class, args);
    }
}
```

5. Deploiement sous tomcat

- Résultat

```
package fr.exemple.essai;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

@SpringBootApplication
public class EssaiApplication extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(EssaiApplication.class);
    }
    public static void main(String[] args) {
        SpringApplication.run(EssaiApplication.class, args);
    }
}
```

5. Modification du pom.xml

- Configuration de la classe principale

- Définir une propriété

```
<start-class>fr.exemple.essai.EssaiApplication</start-class>
```

- Mise à jour du jar en war

- Modification du packaging

```
<packaging>war</packaging>
```

- On ajoute un endpoint REST pour renvoyer la chaîne «Hello World from Tomcat». Il faut ajouter la dépendance de starter Web Spring Boot dans le pom.xml.

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
```

```
</dependency>
```


5. Modification de la classe principale en endpoint

```
package fr.exemple.essai;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController

public class EssaiApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(EssaiApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(EssaiApplication.class, args);
    }

    @RequestMapping(value = "/")
    public String hello() {
        return "Hello World from Tomcat";
    }
}
```

Compiler et déployer

- Calculer les nouvelles dépendances

```
H:\sac\essai>c:\apache-maven-3.3.9\bin\mvn dependency:resolve
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-starter-parent/2.1.0.RELEASE/
```

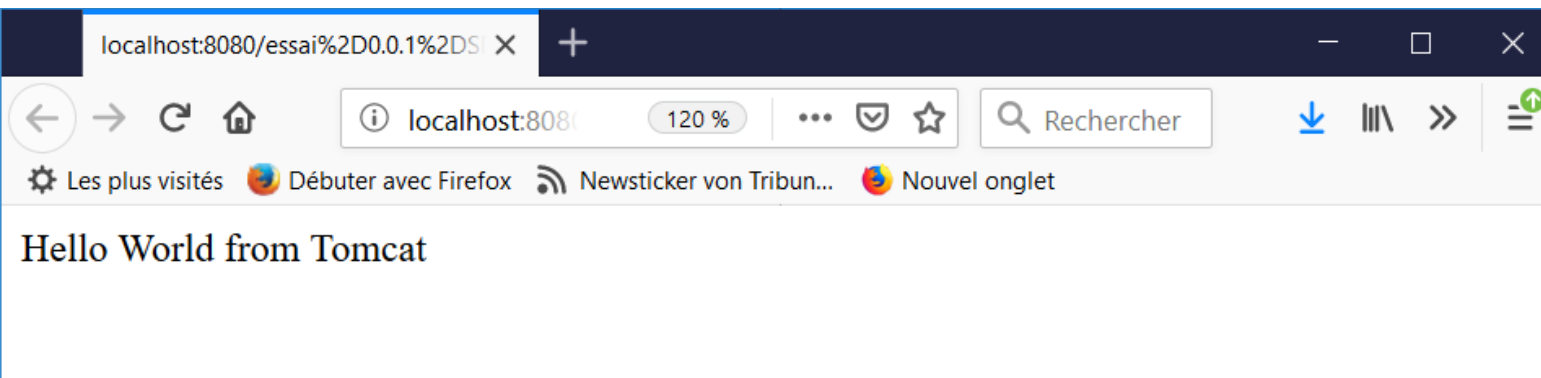
- Fabriquer un livrable

```
H:\sac\essai>c:\apache-maven-3.3.9\bin\mvn clean install
[INFO] Scanning for projects...
[INFO]
```

- Déployer sous tomcat le war

/essai-0.0.1-SNAPSHOT	None specified		true	0	Démarrer Arrêter Recharger Retirer
					Expirer les sessions inactives depuis ≥ 30 minutes

- Effectuer un test via un navigateur <http://localhost:8080/essai-0.0.1-SNAPSHOT/>



6. Automatisation avec Maven

- Pour Spring Boot, Maven ou Gradle sont recommandés
 - ils fournissent un bon support pour la gestion des dépendances. Spring ne supporte pas bien les autres systèmes de build.
- Gestion des dépendances
- L'équipe Spring Boot fournit une liste de dépendances permettant de prendre en charge la version Spring Boot.
 - Il n'est pas nécessaire de fournir une version pour les dépendances dans le fichier de configuration de la construction.
 - Spring Boot configure automatiquement la version des dépendances en fonction de la version. N'oubliez pas que lorsque vous mettez à niveau la version Spring Boot, les dépendances sont également mises à niveau automatiquement.
- Remarque - Si vous souhaitez spécifier la version de la dépendance, vous pouvez l'indiquer dans votre fichier de configuration. Cependant, l'équipe Spring Boot recommande vivement qu'il ne soit pas nécessaire de spécifier la version pour la dépendance.

6. Automatisation avec Maven

- Dépendance Maven
- Pour la configuration Maven, le projet hérite du projet parent Spring Boot Starter afin de gérer les dépendances de Spring Boot Starters. Pour cela, nous pouvons simplement hériter du parent de départ dans le fichier pom.xml, comme indiqué.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.0.RELEASE</version>
</parent>
```

- Nous devrions spécifier le numéro de version de la dépendance Spring Boot Parent Starter. Ensuite, pour les autres dépendances de démarrage, il n'est pas nécessaire de spécifier le numéro de version de Spring

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

7. Code structure

- Spring Boot ne dispose pas de template de code pour commencer.
 - Cependant, certaines pratiques sont utiles.
- Utiliser le package par défaut n'est pas recommandée. Spring Boot provoquera des problèmes lors de la configuration automatique ou de l'analyse des composants, si on utilise le package par défaut.
 - La convention de dénomination recommandée par Java pour la déclaration de package est le nom de domaine inversé. Par exemple - `fr.spring.exemple1`
- La structure logicielle typique d'une l'application Spring Boot est illustrée

Le fichier `Application.java` doit déclarer la méthode principale avec `@SpringBootApplication`.

```
fr
+- spring
    +- exemple
        +- Application.java
        |
        +- model
        | +- Product.java
        +- dao
        | +- ProductRepository.java
        +- controller
        | +- ProductController.java
        +- service
        | +- ProductService.java
```

```
package fr.spring.exemple1;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

8. Beans et injection de dépendances

- Dans Spring Boot, on utilise Spring Framework pour définir les beans et leur injection de dépendance.
 - L'annotation `@ComponentScan` est utilisée pour rechercher les beans et l'annotation `@Autowired` correspondante injectée.
- Si on suit la structure logicielle typique de Spring Boot, inutile de spécifier des arguments pour l'annotation `@ComponentScan`.
 - Tous les fichiers de classe de composants sont automatiquement enregistrés avec Spring Beans.

```
@Bean
public RestTemplate getRestTemplate() {
    return new RestTemplate();
}
```

8. Beans et injection de dépendances

- RestTemplate permet d'envoyer une requête REST
- L'attribut restTemplate est automatiquement initialisé par l'appel de la méthode annotée @Bean

```
package fr.spring.exemple1;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class DemoApplication {

    @Autowired
    RestTemplate restTemplate;

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }

}
```

9. Runners

- Les interfaces `ApplicationRunner` et `CommandLineRunner` permettent d'exécuter le code après le démarrage de l'application Spring Boot.

- On utilise ces interfaces pour effectuer toute action immédiatement après le démarrage de l'application.

- **ApplicationRunner**

- C'est une interface utilisée pour exécuter le code après le démarrage de l'application Spring Boot.

```
package fr.spring.eemple1;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements ApplicationRunner {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Override
    public void run(ApplicationArguments arg0) throws Exception {
        System.out.println("Hello World from Application Runner");
    }
}

[main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.s
[main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
[main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)

[main] fr.spring.eemple1.DemoApplication : Started DemoApplication in 2.501 seconds (JVM running for 2.839)
```

```
2018-11-11 17:49:23.969 INFO 15036 --- [
2018-11-11 17:49:24.110 INFO 15036 --- [
2018-11-11 17:49:24.169 INFO 15036 --- [
Hello World from Application Runner
2018-11-11 17:49:24.174 INFO 15036 --- [
```


9. Runners

- **CommandLineRunner**
 - C'est une interface pour exécuter le code après le démarrage de l'application Spring Boot.

```
package fr.spring.exemple1;

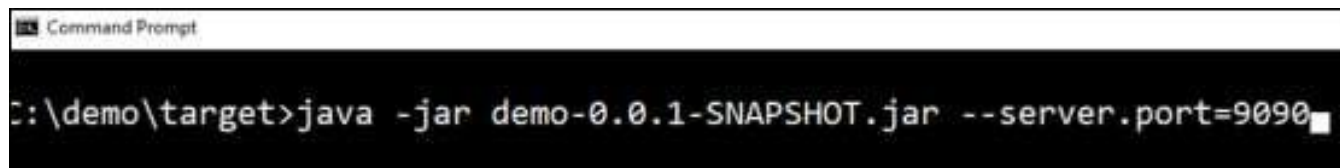
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Hello world from Command Line Runner");
    }
}
```

```
2018-11-11 18:06:49.272 INFO 5008 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.sp
2018-11-11 18:06:49.600 INFO 5008 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2018-11-11 18:06:49.761 INFO 5008 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
Hello world from Command Line Runner
2018-11-11 18:06:49.769 INFO 5008 --- [main] fr.spring.eemple1.DemoApplication : Started DemoApplication in 3.15 seconds (JVM running for 3.505)
```

10. Application properties

- **Propriétés de la ligne de commande**
- L'application Spring Boot convertit les propriétés de ligne de commande en propriétés Spring Boot Environment.
 - Les propriétés de la ligne de commande ont priorité sur les autres sources de propriétés.
 - Par défaut, Spring Boot utilise le numéro de port 8080 pour démarrer Tomcat.
 - Comment changer le numéro de port en utilisant les propriétés de la ligne de commande.
- Étape 1 - Après avoir créé un fichier JAR exécutable, exécutez-le à l'aide de la commande `java -jar <JARFILE>`.
- Étape 2 - Utilisez la commande fournie dans la capture d'écran ci-dessous pour modifier le numéro de port de l'application Spring Boot à l'aide des propriétés de ligne de commande.



```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --server.port=9090
```

10. Application properties

- **Fichier de propriétés**
- Les fichiers de propriétés sont utilisés pour conserver les propriétés dans un seul fichier afin d'exécuter l'application dans un environnement différent.
 - Dans Spring Boot, les propriétés sont conservées dans le fichier `application.properties` sous le chemin de classe.
- Le fichier `application.properties` se trouve dans le répertoire `src/main/resources`.

```
server.port = 9090  
spring.application.name = demoservice
```

- le service `demoservice` de l'application Spring Boot démarre sur le port 9090.

10. Application properties

- **Fichier YAML**
- Spring Boot prend en charge les configurations de propriétés basées sur YAML pour exécuter l'application.
 - Au lieu de `application.properties`, nous pouvons utiliser le fichier `application.yml`. Ce fichier YAML doit être conservé dans le classpath.

```
spring:
  application:
    name: demoservice
  server:
port: 9090
```

10. Application properties

- **Propriétés externalisées**
- Au lieu de conserver le fichier de propriétés sous classpath, on conserve les propriétés dans un chemin différent.
 - Lors de l'exécution du fichier JAR, on spécifie le chemin du fichier de propriétés. On utilise la commande suivante pour spécifier l'emplacement du fichier de propriétés lors de l'exécution du fichier JAR

`-Dspring.config.location = C:\application.properties`



```
Command Prompt
C:\demo\target>java -jar -Dspring.config.location=C:\application.properties demo-0.0.1-SNAPSHOT.jar
```

10. Application properties

- Utilisation de l'Annotation `@Value`
- `@Value` est utilisée pour lire la valeur d'une propriété d'environnement ou d'application en code Java.

```
@Value("${property_key_name}")
```

```
@Value("${spring.application.name}")
```

10. Application properties

```
package fr.spring.eemple1;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController

public class DemoApplication {

    @Value("${spring.application.name}")
    private String name;

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

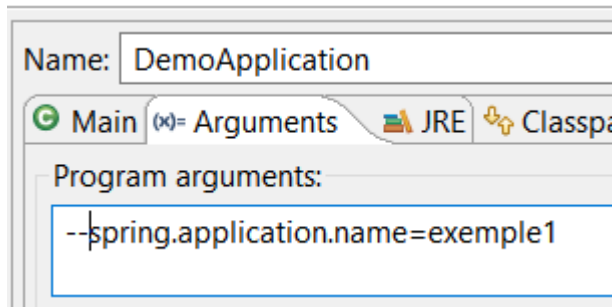
    @RequestMapping(value = "/")
    public String name() {
        return name;
    }
}
```

- Si la propriété est introuvable lors de l'exécution de l'application, Spring Boot lève l'exception `Illegal Argument`

- Pour résoudre le problème de clé réservée, on définit la valeur par défaut de la propriété.

```
@Value("${property_key_name:default_value}")
```

```
@Value("${spring.application.name:demo-service}")
```



Caused by: [java.lang.IllegalArgumentException](#): Could not resolve placeholder 'spring.application.name' in value "\${spring.application.name}"
at org.springframework.util.PropertyPlaceholderHelper.parseStringValue([PropertyPlaceholderHelper.java:174](#)) ~[spring-core-4.3.12.RELEASE.jar:4.3.12.RELEASE]
at org.springframework.util.PropertyPlaceholderHelper.replacePlaceholders([PropertyPlaceholderHelper.java:126](#)) ~[spring-core-4.3.12.RELEASE.jar:4.3.12.RELEASE]

10. Application properties

- **Profil actif de Spring Boot**
- Spring Boot prend en charge différentes propriétés basées sur le profil actif.
 - Par exemple, on peut conserver 2 fichiers distincts pour le développement et la production afin d'exécuter l'application Spring Boot.
- Spring profil actif dans `application.properties`
 - Par défaut, `application.properties` sont utilisées pour exécuter l'application Spring Boot.
 - Si on utilise des propriétés basées sur un profil, on les conserve un fichier de propriétés séparé pour chaque profil
- **application.properties**

```
server.port = 8080
spring.application.name = demoservice
```
- **application-dev.properties**

```
server.port = 9090
spring.application.name = demoservice
```
- **application-prod.properties**

```
server.port = 4431
spring.application.name = demoservice
```


10. Application properties

- Lors de l'exécution du fichier JAR, on spécifie le profil actif Spring en fonction de chaque fichier de propriétés.
 - Par défaut, l'application Spring Boot utilise le fichier `application.properties`.



```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
```

- 2018-11-26 08:13:16.322 INFO 14028 --- [main] fr.spring.exemple1.DemoApplication : The following profiles are active: dev
- Tomcat a maintenant démarré sur le port 9090 (http),
- 2018-11-26 08:13:20.185 INFO 14028 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 9090 (http)

10. Application properties

- **Profil actif Spring pour application.yml**
 - On peut conserver les propriétés du profil actif de Spring dans le seul fichier application.yml.
 - Pas besoin d'utiliser le fichier séparé comme application.properties.
 - Le délimiteur (---) est utilisé pour séparer chaque profil dans le fichier application.yml.
 - La commande pour définir le profil de développement actif est



```
Command Prompt
C:\demo\target>java -jar demo-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev
```

```
2018-11-26 08:41:37.202 INFO 14104 --- [main]
fr.spring.exemple1.DemoApplication : The
following profiles are active: dev
2018-11-26 08:41:46.650 INFO 14104 --- [main]
s.b.c.e.t.TomcatEmbeddedServletContainer :
Tomcat started on port(s): 9090 (http)
```

```
spring:
  application:
    name: demoservice
server:
  port: 8080
```

```
---
spring:
  profiles: dev
  application:
    name: demoservice
server:
  port: 9090
```

```
---
spring:
  profiles: prod
  application:
    name: demoservice
server:
  port: 4431
```

11. Logging

- Spring Boot utilise Apache Commons Logging pour toutes les journalisations internes.
 - Les configurations par défaut de Spring Boot prennent en charge l'utilisation de Java Util Logging, Log4j2 et Logback.
 - Avec ceux-ci, on configure la journalisation de la console et des fichiers.
- Si on utilise un starter de Spring Boot, Logback fournira un support pour la journalisation.
 - Logback permet d'utiliser correctement les fonctions Common Logging, Util Logging, Log4J et SLF4J.
 - Le format par défaut des logs de Spring Boot
 - Date et Time qui donne l'heure et la date du message de log
 - Log level affiche INFO, ERROR ou WARN
 - Process ID
 - Le --- qui est le séparateur
 - Thread name est placé entre []
 - Logger Name qui affiche le nom de la classe source
 - Le message de Log

```
2017-11-26 09:30:27.873 INFO 5040 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-11-26 09:30:27.895 INFO 5040 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2017-11-26 09:30:27.898 INFO 5040 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2017-11-26 09:30:28.040 INFO 5040 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2759 ms
```

11. Logging

- **Sortie de log sur la console**

- Les messages de log par défaut seront imprimés dans la fenêtre de la console.
- Par défaut, les messages de log «INFO», «ERROR» et «WARN» sont imprimés dans le log.

- Pour changer le niveau de log, il est mieux de le faire au démarrage de l'application

```
java -jar demo.jar --debug
```

- On peut ajouter le mode débogage au fichier `application.properties`,
`debug = true`

11. Logging

- **Sortie de log dans un fichier**

- Par défaut, tous les logs sont imprimés dans la fenêtre de la console et non dans les fichiers.
- Pour imprimer les log dans un fichier, on définit la propriété `logging.file` ou `logging.path` dans le fichier `application.properties`.

- Vous pouvez spécifier le chemin du fichier de log alors le nom du fichier journal est `spring.log`.

```
logging.path = /var/tmp/
```

- On peut spécifier le nom du fichier de log :

```
logging.file = /var/tmp/mylog.log
```

- Les fichiers sont configurés automatiquement pour ne pas dépasser la taille 10 Mo.

11. Logging

- Niveaux de log

- Spring Boot prend en charge tous les niveaux de log tels que “TRACE”, “DEBUG”, “INFO”, “WARN”, “ERROR”, “FATAL”, “OFF”.

- On définit le journal dans le fichier `application.properties`
`logging.level.root = WARN`

- Logback ne prend pas en charge le log de niveau «FATAL». Il est mappé sur le log de niveau «ERROR».

11. Configuration Logback

- Logback prend en charge la configuration basée sur XML pour gérer les configurations du log de Spring Boot.

- Le fichier logback.xml doit être placé sous le CLASSPATH.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <root level = "INFO">
    </root>
  </configuration>
```

- Pour faire un echo dans la console

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
  <appender name = "STDOUT" class =
"ch.qos.logback.core.ConsoleAppender"></appender>
  <root level = "INFO">
    <appender-ref ref = "STDOUT"/>
  </root>
</configuration>
```

11. Configuration Logback

- On peut nommer le fichier appender dans le fichier Logback.xml

- On doit spécifier le chemin d'accès au fichier log dans l'appender.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>
    <appender name = "FILE" class = "ch.qos.logback.core.FileAppender">
        <File>/var/tmp/mylog.log</File>
    </appender>
    <root level = "INFO">
        <appender-ref ref = "FILE"/>
    </root>
</configuration>
```

- On peut définir un template de log dans le fichier logback.xml

```
<pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}] [%C] [%t] [%L] [%-5p]
%m%n</pattern>
```


11. Exemple Logback

```
<?xml version = "1.0" encoding = "UTF-8"?>
<configuration>

  <appender name = "STDOUT" class = "ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}} [%C] [%t] [%L] [%-5p] %m%n</pattern>
    </encoder>
  </appender>

  <appender name = "FILE" class = "ch.qos.logback.core.FileAppender">
    <File>/var/tmp/mylog.log</File>
    <encoder>
      <pattern>[%d{yyyy-MM-dd'T'HH:mm:ss.sss'Z'}} [%C] [%t] [%L] [%-5p] %m%n</pattern>
    </encoder>
  </appender>

  <root level = "INFO">
    <appender-ref ref = "FILE"/>
    <appender-ref ref = "STDOUT"/>
  </root>
</configuration>
```

```
package fr.spring.exemple1;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    private static final Logger logger = LoggerFactory.getLogger(DemoApplication.class);

    public static void main(String[] args) {
        logger.info("this is a info message");
        logger.warn("this is a warn message");
        logger.error("this is a error message");
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

```
[2018-11-26T08:41:37.202][fr.spring.exemple1.DemoApplication][main][14][INFO] this is a info message
[2018-11-26T08:41:37.202][fr.spring.exemple1.DemoApplication][main][15][WARN] this is a warn message
[2018-11-26T08:41:37.202][fr.spring.exemple1.DemoApplication][main][16][ERROR] this is a error message
```

12. Construction de Web service REST

- Spring Boot est un support pour la création de services Web RESTful.
 - Il faut ajouter la dépendance Spring Boot Starter Web dans le fichier de configuration pom.xml.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```
 - **@RestController**
 - L'annotation `@RestController` est utilisée pour définir les services Web RESTful. Il traite un flux JSON, XML avec une réponse personnalisée.

```
@RestController
public class ProductServiceController {
}
```
 - **@RequestMapping**
 - L'annotation `@RequestMapping` est utilisée pour définir l'URI de demande permettant d'accéder aux endpoints REST. On définit une méthode pour consommer et produire des objets. La méthode est par défaut associée à GET.

```
@RequestMapping(value = "/products")
public ResponseEntity<Object> getProducts() { }
```

12. Construction de Web service REST

- **@RequestBody**

- Elle est utilisée pour définir le contenu de la requête par un type

```
public ResponseEntity<Object> createProduct (@RequestBody  
Product product) {  
}
```

- **@PathVariable**

- Elle est utilisée pour définir des paramètres dans l'URI. La variable est définie comme des accolades

```
public ResponseEntity<Object> updateProduct (@PathVariable ("id")  
String id) {  
}
```

- **@RequestParam**

- Elle est utilisée pour lire les paramètres de la requête à partir de l'URL de la requête. Par défaut, il s'agit d'un paramètre obligatoire. Nous pouvons également définir la valeur par défaut pour les paramètres de requête

```
public ResponseEntity<Object> getProduct (@RequestParam (value =  
"name", required = false, defaultValue = "honey") String name) {  
}
```

12. pom.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.spring.exemple1</groupId>
  <artifactId>essai</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>demo</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/>
  </parent>
```

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

12. La méthode GET

- La méthode HTTP par défaut est GET.
 - Cette méthode ne nécessite aucun corps de requête.
 - On envoie des QueryParam et des PathVariable pour définir l'URL.
 - Exemple de code avec une requête HTTP GET.
 - On utilise une HashMap pour stocker un produit.
 - On utilise une classe POJO en tant que produit à stocker.
- L'URI de la requête est `/products` et elle renvoie la liste des produits du référentiel HashMap.

```
package fr.spring.exemple1.controller;

import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import fr.spring.exemple1.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {
        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);
    }

    @RequestMapping(value = "/products")
    public ResponseEntity<Object> getProduct() {
        return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);
    }
}
```

```
package fr.spring.exemple1.controller;
```

12. La méthode POST

```
import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import fr.spring.exemple1.model.Product;

@RestController
public class ProductServiceController {
    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value = "/products", method = RequestMethod.POST)
    public ResponseEntity<Object> createProduct(@RequestBody Product product)
    {
        productRepo.put(product.getId(), product);
        return new ResponseEntity<>("Product is created successfully",
            HttpStatus.CREATED);
    }
}
```

- La requête HTTP POST est utilisée pour créer une ressource.
- Cette méthode contient le corps de la demande.
- On envoie des QueryParam et des PathVariable pour définir l'URL personnalisée.
 - Exemple de code avec une requête HTTP POST.
 - On utilise une HashMap pour stocker un produit.
 - On utilise une classe POJO en tant que produit à stocker.
- L'URI de la requête est /products et elle renvoie la liste des produits du référentiel HashMap.

12. La méthode PUT

- La méthode HTTP PUT est utilisée pour mettre à jour une ressource existante.
- Cette méthode contient un corps de requête. On envoie des `ParamQuery` et des `PathVariable` pour définir l'URL personnalisée.
- L'exemple définit la requête HTTP PUT. On utilise une `HashMap` pour mettre à jour le produit existant, le produit étant une classe POJO.
- L'URI de la demande est `/products/{id}` qui retourne la chaîne après que le produit ait été mis à jour dans un référentiel `HashMap`. On utilise la `PathVariable {id}`, qui définit l'ID de produits devant être mis à jour.

```
package fr.spring.exemple1.controller;

import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import fr.spring.exemple1.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String
id, @RequestBody Product product) {

        productRepo.remove(id);

        product.setId(id);

        productRepo.put(id, product);

        return new ResponseEntity<>("Product is updated successssfully",
HttpStatus.OK);

    }

}
```

12. La méthode DELETE

- La méthode HTTP DELETE est utilisée pour supprimer une ressource existante.
- Cette méthode ne contient aucun corps de requête. On envoie des ParamQuery et des PathVariable pour définir l'URL personnalisée.
- L'exemple définit la méthode HTTP DELETE. On utilise HashMap pour supprimer le produit existant, qui est une classe POJO.
- L'URI est /products/{id} et il renvoie la chaîne après avoir supprimé le produit du référentiel HashMap.
- On utilise la PathVariable {id}, qui définit l'ID de produit à supprimer.

```
package fr.spring.exemple1.controller;
import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import fr.spring.exemple1.model.Product;

@RestController
public class ProductServiceController {
    private static Map<String, Product> productRepo = new HashMap<>();

    @RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {
        productRepo.remove(id);

        return new ResponseEntity<>("Product is deleted successssfully",
            HttpStatus.OK);
    }
}
```


12. Récapitulatif

DemoApplication.java

```
package fr.spring.exemple1;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

```
package fr.spring.exemple1.demo.model;

public class Product {
    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```

package fr.spring.exemple1.controller;

import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
import fr.spring.exemple1.model.Product;

@RestController
public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();

    static {
        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);
    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {
        productRepo.remove(id);
        return new ResponseEntity<>("Product is deleted successssfully",
        HttpStatus.OK);
    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
    @RequestBody Product product) {
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
        return new ResponseEntity<>("Product is updated successssfully",
        HttpStatus.OK);
    }

    @RequestMapping(value = "/products", method = RequestMethod.POST)
    public ResponseEntity<Object> createProduct(@RequestBody Product product) {
        productRepo.put(product.getId(), product);
        return new ResponseEntity<>("Product is created successfully",
        HttpStatus.CREATED);
    }

    @RequestMapping(value = "/products")
    public ResponseEntity<Object> getProduct() {
        return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);
    }
}

```

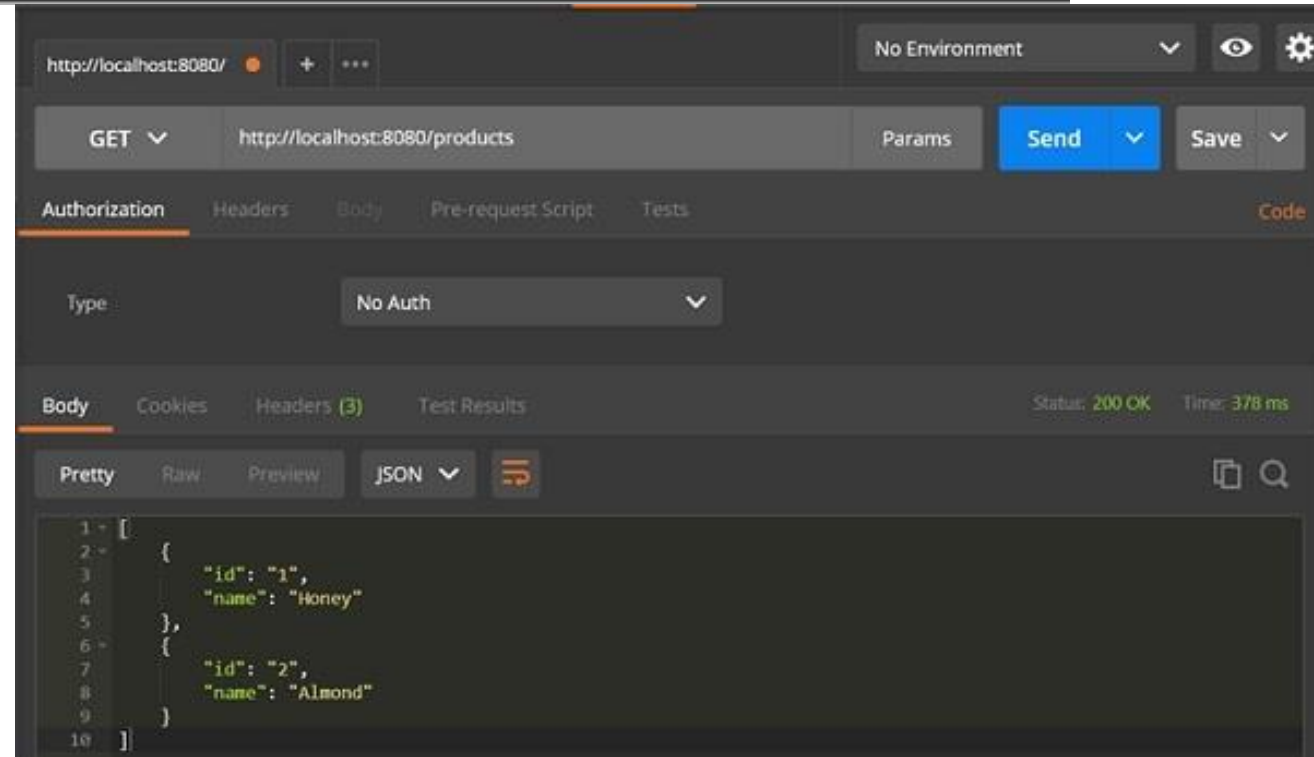
ProductServiceController.java

12. récapitulatif

- mvn clean install
- java -jar <JARFILE>

```
2017-11-26 13:56:27.912 INFO 13204 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 13:56:27.922 INFO 13204 --- [main] com.tutorialspoint.demo.DemoApplication : Started DemoApplication in 5.961 seconds (T
M running for 6.933)
```

- Utiliser PostMan
- <http://localhost:8080/products>



13. Gestion des exceptions

- **@ControllerAdvice**

- Elle permet de gérer les exceptions globalement

```
import org.springframework.web.bind.annotation.ControllerAdvice;
```

```
@ControllerAdvice
```

```
public class ProductExceptionHandler {  
}
```

- **@ExceptionHandler**

- Elle est utilisée pour gérer les exceptions spécifiques et envoyer les réponses personnalisées au client.

```
package fr.spring.exemple1.exception;
```

```
package fr.spring.exemple1.exception;
```

```
public class ProductNotFoundException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
}
```

13. Gestion des exceptions

- **@ExceptionHandler** pour gérer les exceptions.
 - Cette méthode doit être utilisée pour écrire le fichier de classe ControllerAdvice.

```
@ExceptionHandler(value = ProductNotFoundException.class)
public ResponseEntity<Object>
exception(ProductNotFoundException exception) {
}
```

- Le code qui lève une exception depuis l'API REST

```
@RequestMapping(value = "/products/{id}", method =
RequestMethod.PUT)
public ResponseEntity<Object> updateProduct() {
    throw new ProductNotFoundException();
}
```

13. Gestion des exceptions

```
package fr.spring.exemple1.exception;
```

- L'API PUT pour mettre à jour un produit.
- Lors de la mise à jour du produit, si le produit est introuvable, retournez le message d'erreur de réponse sous la forme «Produit non trouvé».
 - NB la classe d'exception `ProductNotFoundException` doit étendre l'exception `RuntimeException`.

```
package fr.spring.exemple1.exception;

public class ProductNotFoundException extends
    RuntimeException {

    private static final long serialVersionUID = 1L;

}
```

```
import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import
    org.springframework.web.bind.annotation.ControllerAdvice;

import
    org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice

public class ProductExceptionHandler {

    @ExceptionHandler(value = ProductNotFoundException.class)

    public ResponseEntity<Object>
        exception(ProductNotFoundException exception) {

        return new ResponseEntity<>("Product not found",
            HttpStatus.NOT_FOUND);

    }

}
```

13. Exception

```
package fr.spring.exemple1;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class DemoApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

}

package fr.spring.model.exemple1;

public class Product {

    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

```
package fr.spring.exemple1.controller;
import java.util.HashMap;
import java.util.Map;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import fr.spring.exemple1.exception.ProductNotFoundException;
import fr.spring.exemple1.model.Product;

@RestController

public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();
    static {

        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);

    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)
    public ResponseEntity<Object> updateProduct(@PathVariable("id") String id,
    @RequestBody Product product) {

        if(!productRepo.containsKey(id))throw new ProductNotFoundException();
        productRepo.remove(id);
        product.setId(id);
        productRepo.put(id, product);
        return new ResponseEntity<>("Product is updated successfully", HttpStatus.OK);

    }

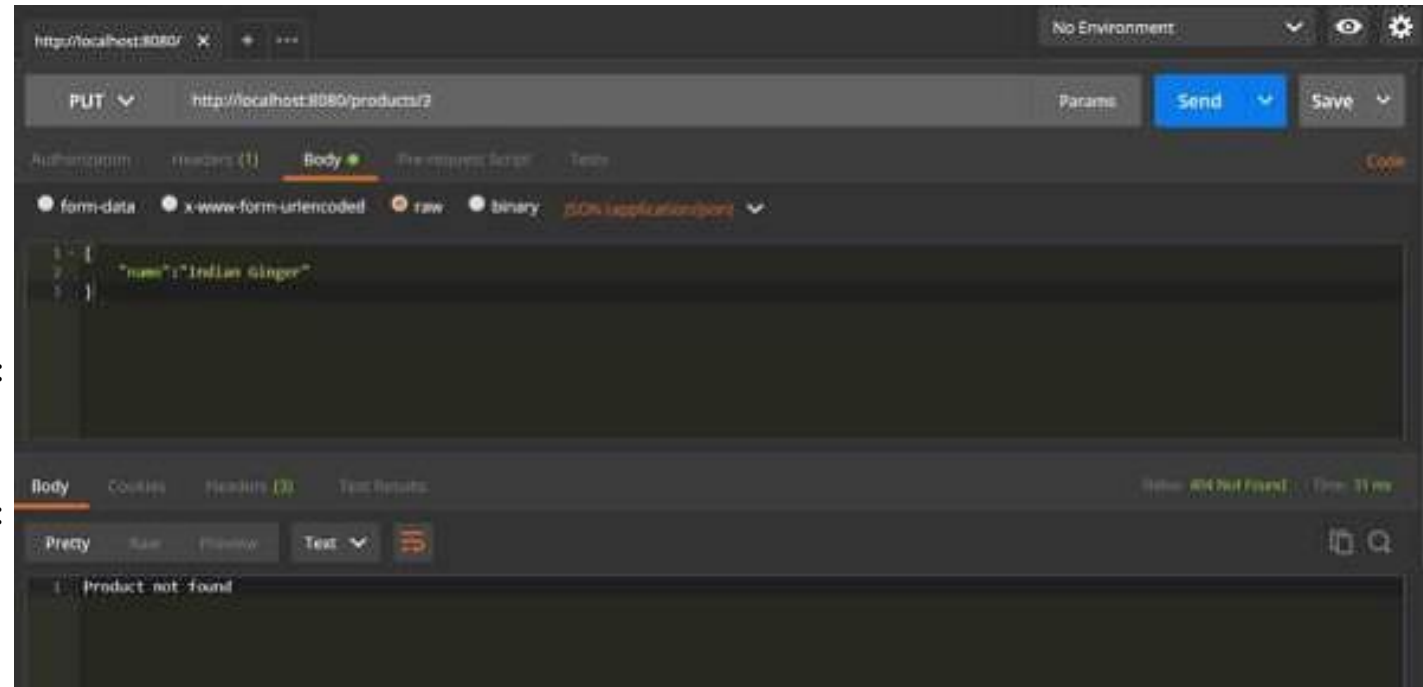
}
```

13. Run avec exception

```
mvn clean install  
java -jar <JARFILE>
```

```
2018-11-26T08:41:37.202 INFO 13204 main :  
s.b.c.e.t.TomcatEmbeddedServletContainer :  
Tomcat started on port(s): 8080 (http)  
2018-11-26T08:41:37.202 INFO 13204 main :  
fr.spring.exemple1.DemoApplication :  
Started DemoApplication in 5.421 seconds
```

Update URL:
<http://localhost:8080/products/3>



14. Intercepteur

- Un intercepteur en Spring Boot effectue des opérations :
 - Avant d'envoyer la demande au contrôleur
 - Avant d'envoyer la réponse au client
- Par exemple, un intercepteur peut ajouter une en-tête avant d'envoyer une requête au contrôleur et pour ajouter une en-tête avant d'envoyer la réponse au client.
- Pour faire un interceptor, il faut une classe `@Component` qui la prend en charge et implémenter l'interface `HandlerInterceptor`: 3 méthodes
 - `preHandle ()` - pour effectuer des opérations avant d'envoyer la demande au contrôleur.
Cette méthode doit renvoyer `true` pour renvoyer la réponse au client.
 - `postHandle ()` - pour effectuer des opérations avant d'envoyer la réponse au client.
 - `afterCompletion ()` - Utilisée pour effectuer des opérations après avoir complété la demande et la réponse.

14. Intercepteur

```
@Component
public class ProductServiceInterceptor implements
HandlerInterceptor {
    @Override
    public boolean preHandle(
        HttpServletRequest request, HttpServletResponse response,
        Object handler) throws Exception {

        return true;
    }
    @Override
    public void postHandle(
        HttpServletRequest request, HttpServletResponse response,
        Object handler,
        ModelAndView modelAndView) throws Exception {}

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response,
        Object handler, Exception exception) throws Exception {}
}
```

- On doit enregistrer cet intercepteur avec `InterceptorRegistry` en utilisant `WebMvcConfigurerAdapter`

```
@Component
public class ProductServiceInterceptorAppConfig extends
WebMvcConfigurerAdapter {
    @Autowired
    ProductServiceInterceptor productServiceInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry
registry) {
        registry.addInterceptor(productServiceInterceptor);
    }
}
```

15. Unit Test

- Les tests unitaires sont les tests effectués par les développeurs pour s'assurer que les fonctionnalités des classes individuelles fonctionnent correctement.
 - Comment écrire un scénario de test d'unité à l'aide de Mockito et Web Controller.
 - Pour injecter de Mocks dans Spring Beans, on ajoute la dépendance Mockito-core dans le pom.xml.

```
<dependency>
```

```
  <groupId>org.mockito</groupId>
```

```
  <artifactId>mockito-core</artifactId>
```

```
  <version>2.13.0</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
```

```
  <artifactId>spring-boot-starter-test</artifactId>
```

```
  <scope>test</scope>
```

```
</dependency>
```

15. Unit Test

- La classe Service (CUT) contenant une méthode renvoyant la valeur String

```
package fr.spring.exemple1;
```

```
import org.springframework.stereotype.Service;
```

```
@Service
```

```
public class ProductService {  
    public String getProductName() {  
        return "Honey";  
    }  
}
```

- On injecte la classe ProductService dans un autre fichier de classe de service

```
package fr.spring.exemple1;
```

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;
```

```
@Service
```

```
public class OrderService {  
    @Autowired  
    ProductService productService;  
  
    public OrderService(ProductService productService) {  
        this.productService = productService;  
    }  
    public String getProductName() {  
        return productService.getProductName();  
    }  
}
```

15. Unit Test runner

- Le fichier principal de la classe d'application Spring Boot

```
package fr.spring.exemple1;

import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MockitoDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(MockitoDemoApplication.class,
args);
    }
}
```

- Configuration du contexte d'application pour les tests.
- L'annotation `@Profile («test»)` est utilisée pour configurer la classe lorsque les scénarios de test sont en cours d'exécution.

```
package fr.spring.exemple1;

import org.mockito.Mockito;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.context.annotation.Profile;

@Profile("test")
@Configuration
public class ProductServiceTestConfiguration {
    @Bean
    @Primary
    public ProductService productService() {
        return Mockito.mock(ProductService.class);
    }
}
```

15. Unit Test runner

- mvn clean install

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

- un scénario de test unitaire pour OrderService sous le package src/test/resources.

```
package fr.spring.exemple1;
```

```
import org.junit.Assert;
```

```
import org.junit.Test;
```

```
import org.junit.runner.RunWith;
```

```
import org.mockito.Mockito;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.test.context.SpringBootTest;
```

```
import org.springframework.test.context.ActiveProfiles;
```

```
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
```

```
@SpringBootTest
```

```
@ActiveProfiles("test")
```

```
@RunWith(SpringJUnit4ClassRunner.class)
```

```
public class MockitoDemoApplicationTests {
```

```
    @Autowired
```

```
    private OrderService orderService;
```

```
    @Autowired
```

```
    private ProductService productService;
```

```
    @Test
```

```
    public void whenUserIdIsProvided_thenRetrievedNameIsCorrect() {
```

```
        Mockito.when(productService.getProductName()).thenReturn("Mock Product Name");
```

```
        String testName = orderService.getProductName();
```

```
        Assert.assertEquals("Mock Product Name", testName);
```

```
    }
```

```
}
```