

Module : Services Web

Version 3

Formateur :

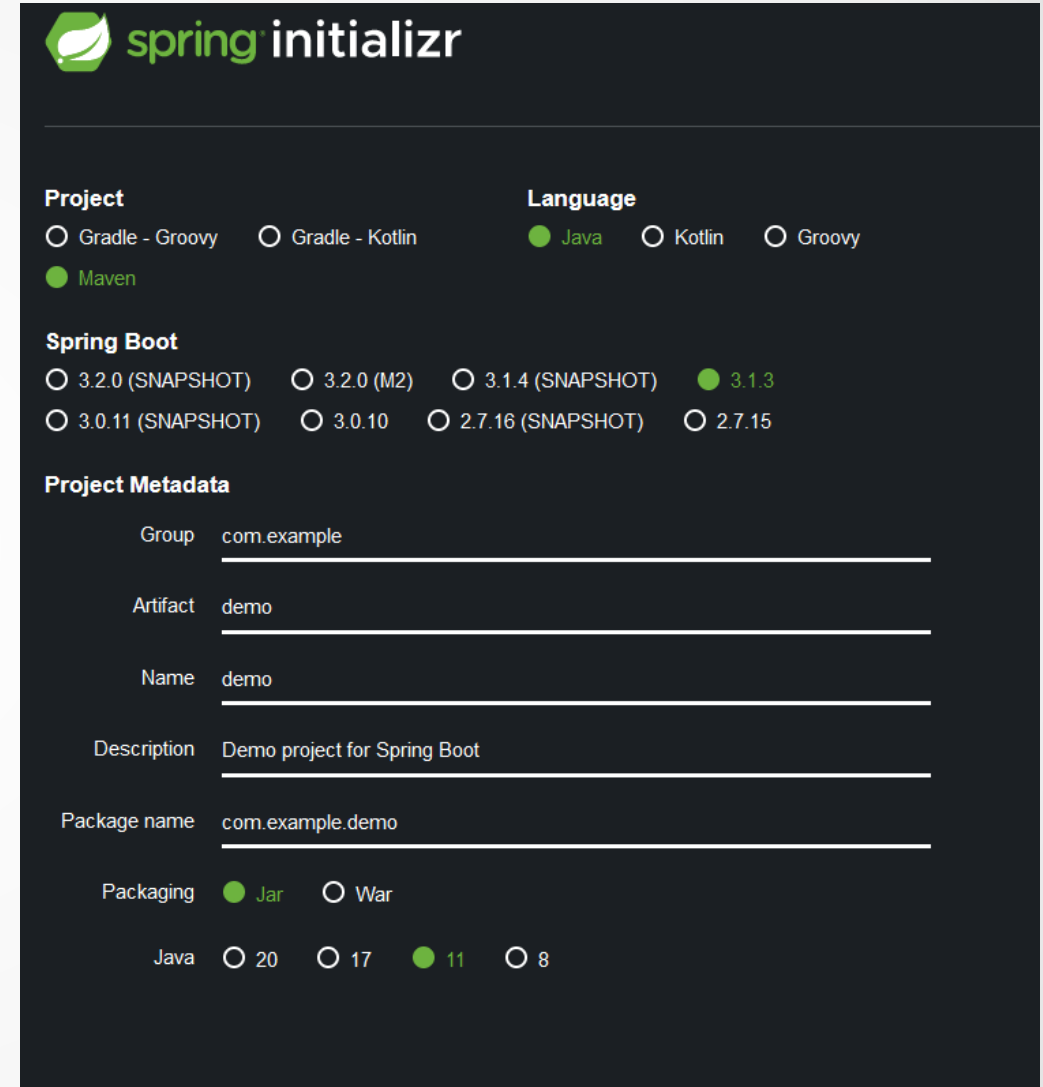
DANGLA Loïc
loic.dangla1@reseau-cd.net

API

Spring initializr

Utilisation de Spring initializr :

- Project : Vous pouvez choisir Maven ou Gradle en fonction de vos affinités.
- Language : Java
- Spring Boot : 3.1.3
- Project Metadata : Permet de définir :
 - le group,
 - le nom de l'artefact,
 - le nom du projet,
 - la description du projet,
 - le nom du package,
 - le type de packaging,
 - la version de JavaPour initialiser et configurer automatiquement le projet Spring.



The screenshot shows the Spring Initializr web interface with the following configuration:

- Project:** ☒ Maven, ☐ Gradle - Groovy, ☐ Gradle - Kotlin
- Language:** ☒ Java, ☐ Kotlin, ☐ Groovy
- Spring Boot:** ☒ 3.1.3, ☐ 3.2.0 (M2), ☐ 3.1.4 (SNAPSHOT), ☐ 3.0.11 (SNAPSHOT), ☐ 3.0.10, ☐ 2.7.16 (SNAPSHOT), ☐ 2.7.15
- Project Metadata:**
 - Group: com.example
 - Artifact: demo
 - Name: demo
 - Description: Demo project for Spring Boot
 - Package name: com.example.demo
 - Packaging: ☒ Jar, ☐ War
 - Java: ☒ 11, ☐ 20, ☐ 17, ☐ 8

API

Spring initializr

Dépendance Spring initializr :

Vous avez la possibilité d'ajouter des dépendances qui vont être automatiquement initialisées et téléchargées dans le package de votre projet.

Dans notre cas, nous utiliserons les dépendances :

- **Spring Web** : comme la description l'indique, permet de faire du RESTful, ce qui correspond à notre API pour exposer des endpoints.
- **Lombok** : est une librairie pour optimiser certaines classes afin d'éviter d'écrire les get/set par exemple.
- **H2 Database** : vous permet d'avoir une base de données sans installer un MySQL ou autres.
- **Spring Data JPA** : permet de gérer la persistance des données avec la base de données.
- **Spring Security** : Optionnel

Dependencies

ADD DEPENDENCIES... CTRL + B

H2 Database

SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Lombok

DEVELOPER TOOLS

Java annotation library which helps to reduce boilerplate code.

Spring Data JPA

SQL

Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

API

Base de données H2

Configuration de la base de données H2 :

Pour configurer la base de données H2, diverses méthodes sont disponibles sur le Web. Cependant, je vous recommande de conserver la configuration par défaut et d'ajouter simplement une propriété pour activer la console de visualisation de la base de données, dont les instructions sont facilement accessibles sur Internet.

Pour insérer des données dans la base de données, vous pouvez créer un fichier « database.sql » contenant la structure de la base de données et ses données:

```
DROP TABLE IF EXISTS students;
```

```
CREATE TABLE students (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  firstname VARCHAR(250) NOT NULL,  
  lastname VARCHAR(250) NOT NULL,  
  mail VARCHAR(250) NOT NULL,  
  school VARCHAR(250) NOT NULL  
);
```

```
INSERT INTO students (firstname, lastname , mail, school) VALUES  
('Jean', 'Durant', 'jean@mail.com', 'EPSI'),  
('Laura', 'Ladier', 'laura@mail.com', 'EPSI'),  
('Florian', 'Germa', 'florian@mail.com', 'WIS');
```

API

Base de données H2

Configuration de la base de données H2 :

Vous n'avez qu'à placer le fichier SQL dans le répertoire "src/main/resources". Spring Boot gère automatiquement l'exécution de ce script au démarrage de l'application, créant ainsi la table et insérant les données nécessaires sans intervention supplémentaire.

La puissance de Spring Boot simplifie ce processus, vous laissant vous concentrer sur le développement de votre application.

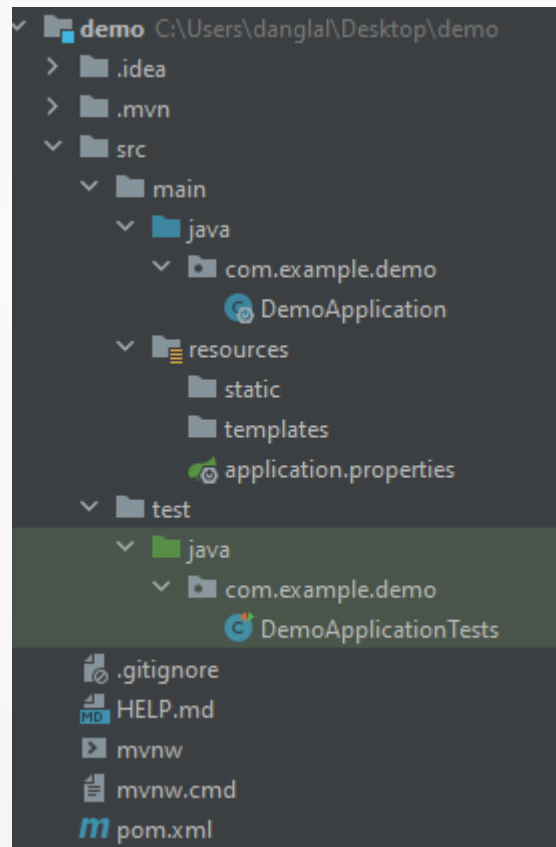
API

Création des packages

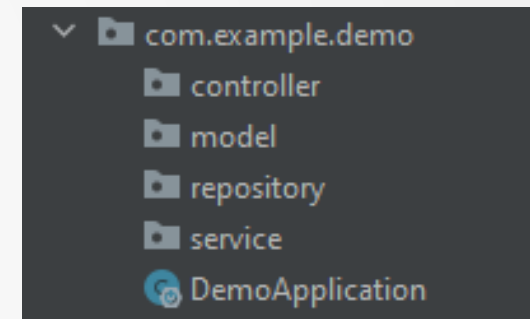
Création des packages nécessaires :

Pour construire l'arborescence du projet API, il faut ajouter les packages suivants :

Avant



Après



| Couche | Objectif |
|------------|---|
| controller | Réceptionner la requête et fournir la réponse |
| service | Exécuter les traitements métiers |
| repository | Communiquer avec la source de données |
| model | Contenir les objets métiers |

API

Propriétés

Définissons les propriétés :

Ajouter dans le fichier « application.properties » les éléments suivants :

```
application.properties x
1
2 #Global configuration
3 spring.application.name=demo
4
5 #Tomcat configuration
6 server.port=9000
```

- **spring.application.name**=demo : pour définir un nom à l'application.
- **server.port**=9000 : pour ne pas être sur le port par défaut 8080.

•logging.level :

- root=ERROR : par défaut, seules les traces en ERROR s'affichent. L'idée est simple : réduire les affichages dans la console de toutes les "3rd party",
- com.openclassrooms=INFO : pour ce qui est de notre code, on est en INFO pour avoir du détail,
- org.springframework.boot.autoconfigure.h2=INFO : permet de voir dans la console l'URL jdbc de la base H2,
- org.springframework.boot.web.embedded.tomcat : permet de voir dans la console le port utilisé par Tomcat au démarrage ;

•**spring.h2.console.enabled**=true : correspond à la propriété pour activité de la console H2.

```
7
8 #Log level configuration
9 logging.level.root=ERROR
10 logging.level.com.openclassrooms=INFO
11 logging.level.org.springframework.boot.autoconfigure.h2=INFO
12 logging.level.org.springframework.boot.web.embedded.tomcat=INFO
13
14 #H2 Configuration
15 spring.h2.console.enabled=true
```

API

Propriétés

Définissons les propriétés :

Ajouter dans le fichier « application.properties » les éléments suivants :

```
application.properties x
1
2 #Global configuration
3 spring.application.name=demo
4
5 #Tomcat configuration
6 server.port=9000
```

- **spring.application.name**=demo : pour définir un nom à l'application.
- **server.port**=9000 : pour ne pas être sur le port par défaut 8080.

•logging.level :

- root=ERROR : par défaut, seules les traces en ERROR s'affichent. L'idée est simple : réduire les affichages dans la console de toutes les "3rd party",
- com.openclassrooms=INFO : pour ce qui est de notre code, on est en INFO pour avoir du détail,
- org.springframework.boot.autoconfigure.h2=INFO : permet de voir dans la console l'URL jdbc de la base H2,
- org.springframework.boot.web.embedded.tomcat : permet de voir dans la console le port utilisé par Tomcat au démarrage ;

•**spring.h2.console.enabled**=true : correspond à la propriété pour activité de la console H2.

```
7
8 #Log level configuration
9 logging.level.root=ERROR
10 logging.level.com.openclassrooms=INFO
11 logging.level.org.springframework.boot.autoconfigure.h2=INFO
12 logging.level.org.springframework.boot.web.embedded.tomcat=INFO
13
14 #H2 Configuration
15 spring.h2.console.enabled=true
```


API

Table en entité Java

Entité Java : Student

- @Data est une annotation Lombok. Evite d'ajouter les getters et les setters. La librairie Lombok s'en charge pour nous. Très utile pour alléger le code.
- @Entity est une annotation qui indique que la classe correspond à une table de la base de données.
- @Table(name="students") indique le nom de la table associée.
- L'attribut id correspond à la clé primaire de la table, et est donc annoté @Id. D'autre part, comme l'id est auto-incrémenté, j'ai ajouté l'annotation @GeneratedValue(strategy = GenerationType.IDENTITY).
- Enfin, firstname et lastname sont annotés avec @Column pour faire le lien avec le nom du champ de la table.

```
Student.java x
1 package com.example.demo.model;
2
3 import javax.persistence.Column;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8 import javax.persistence.Table;
9
10 import lombok.Data;
11
12 @Data
13 @Entity
14 @Table(name = "students")
15 public class Student {
16
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     private Long id;
20
21     @Column(name = "firstname")
22     private String firstname;
23
24     @Column(name = "lastname")
25     private String lastname;
26
27     @Column(name = "email")
28     private String email;
29
30     @Column(name = "school")
31     private String school;
32 }
```

API

Interface

Interface : StudentRepository

- Une fois que l'entité est créée, nous devons implémenter le code qui déclenche les actions pour communiquer avec la base de données.

Une interface ne contient pas de code, comment elle peut exécuter des requêtes ?

- Grâce au composant **Spring Data JPA**, Il nous permet d'exécuter des requêtes SQL, sans avoir besoin de les écrire.
- `@Repository` est une annotation Spring pour indiquer que la classe est un bean, et que son rôle est de communiquer avec une source de données (en l'occurrence la base de données).
- **CrudRepository** est une interface fournie par Spring. Elle fournit des méthodes pour manipuler votre entité. Elle utilise la généricité pour que son code soit applicable à n'importe quelle entité, d'où la syntaxe "`CrudRepository<Student, Long>`"

```
1 package com.example.demo.repository;  
2  
3 import org.springframework.data.repository.CrudRepository;  
4 import org.springframework.stereotype.Repository;  
5  
6 import com.example.demo.model.Student;  
7  
8 @Repository  
9  
10 public interface StudentRepository extends CrudRepository<Student, Long> {  
11  
12 }
```

API

Service

Service : StudentService

- **@Service** : est une spécialisation de **@Component** comme l'annotation **@Repository**.
- **Ajout des actions :**
 - **getStudent**
 - **getStudents**
 - **deleteStudent**
 - **saveStudent**

```
1 package com.example.demo.service;
2
3 import java.util.Optional;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7 import com.example.demo.model.Student;
8 import com.example.demo.repository.StudentRepository;
9
10 import lombok.Data;
11
12 @Data
13 @Service
14 public class StudentService {
15
16     4 usages
17     @Autowired
18     private StudentRepository studentRepository;
19
20     public Optional<Student> getStudent(final Long id) {
21         return studentRepository.findById(id);
22     }
23
24     public Iterable<Student> getStudents() {
25         return studentRepository.findAll();
26     }
27
28     public void deleteStudent(final Long id) {
29         studentRepository.deleteById(id);
30     }
31
32     public Student saveStudent(Student student) {
33         Student savedStudent = studentRepository.save(student);
34         return savedStudent;
35     }
36 }
```

API

Endpoints REST

Les endpoints REST :

- L'un des avantages de Spring et Spring Boot est de vous fournir les composants logiciels qui évitent de faire du code complexe !
- Le starter "spring-boot-starter-web" nous fournit justement tout le nécessaire pour créer un endpoint :
 - une classe Java annotée **@RestController** ;
 - que les méthodes de la classe soient annotées. Chaque méthode annotée devient alors un endpoint grâce aux annotations **@GetMapping**, **@PostMapping**, **@PatchMapping**, **@PutMapping**, **@DeleteMapping**, **@RequestMapping**.

```
1 package com.example.demo.controller;  
2  
3 import org.springframework.beans.factory.annotation.Autowired;  
4 import org.springframework.web.bind.annotation.GetMapping;  
5 import org.springframework.web.bind.annotation.RestController;  
6  
7 import com.example.demo.model.Student;  
8 import com.example.demo.service.StudentService;  
9  
10 @RestController  
11 public class StudentController {  
12  
13     1 usage  
14     @Autowired  
15     private StudentService studentService;  
16  
17     /**  
18      * Read - Get all students  
19      * @return - An Iterable object of Student full filled  
20      */  
21     @GetMapping("/students")  
22     public Iterable<Student> getStudents() {  
23         return studentService.getStudents();  
24     }  
25 }
```

API

Explications

Un peu d'explications supplémentaires :

- @RestController atteint 2 objectifs :
 - @Component, elle permet d'indiquer à Spring que cette classe est un bean.
 - Elle indique à Spring d'insérer le retour de la méthode au format JSON dans le corps de la réponse HTTP. Grâce à ce deuxième point, les applications qui vont communiquer avec votre API accéderont au résultat de leur requête en parsant la réponse HTTP.
- On a injecté une instance de StudentService afin de permettre l'appel des méthodes pour communiquer avec la base de données.
- On a créé une méthode getStudents() annotée @GetMapping("/students"). Cela signifie que les requêtes HTTP de type GET à l'URL /students exécuteront le code de cette méthode. Et ce code est tout simple : il s'agit d'appeler la méthode getStudents() du service, ce dernier appellera la méthode findAll() du repository, et nous obtiendrons ainsi tous les employés enregistrés en base de données.

API

Récapitulatif

Ajouter des actions supplémentaires :

| Annotation | Type HTTP | Objectif |
|-----------------|-----------|--|
| @GetMapping | GET | Pour la lecture de données. |
| @PostMapping | POST | Pour l' envoi de données. Cela sera utilisé par exemple pour créer un nouvel élément. |
| @PatchMapping | PATCH | Pour la mise à jour partielle de l'élément envoyé. |
| @PutMapping | PUT | Pour le remplacement complet de l'élément envoyé. |
| @DeleteMapping | DELETE | Pour la suppression de l'élément envoyé. |
| @RequestMapping | | Intègre tous les types HTTP. Le type souhaité est indiqué comme attribut de l'annotation. Exemple : @RequestMapping(method = RequestMethod.GET) |

API

Résumé

Synthèse :

- Notre entité du **model** est modélisée, et **@Entity** est l'annotation obligatoire !
- La **communication aux données** s'effectue via une classe annotée **@Repository**.
- La classe annotée **@Service** se charge **des traitements métiers**.
- Nos controllers **@RestController** nous permettent de **définir des URL et le code à exécuter** quand ces dernières sont requêtées.



EPSSI

l'École
d'ingénierie
informatique

Fin du cours