



Atelier Complet Full Stack : Spring Boot & React JS

Khalid Nafil
ENSIAS

Dans ce Lab, nous considérons la mise en place d'un projet Full Stack où la partie Backend va être réalisée sous Spring Boot et la partie Frontend va être réalisée sous React JS. Pour la partie Backend, nous allons utiliser **Spring Web**, **Spring Web Devtools**, **Spring Data JPA**, **Spring Rest**, **Spring Security** et **Lombok**. Nous utilisons l'IDE STS pour la création du projet Spring Boot. Le projet consiste à mettre en place une application commerciale qui met en jeu deux entités : une Voiture et un Propriétaire. Pour le reste du Lab, vous allez suivre les étapes suivantes.

1. Créez un projet Spring Boot Starter sous STS sous le nom 'SpringDataRest'.
2. Choisir les dépendances **Web** et **Devtools** pour commencer.
3. Pour cet atelier, veuillez installer le serveur de base de données MariaDB <https://downloads.mariadb.org/>.
4. Faites : **sudo mysql -u root -p** et vous tapez le mot de passe root.
5. Une fois connectée à Mariadb et créé la base de données 'springboot' faites : **use springboot** (Entrée) et après **select * from Voiture;** pour afficher la liste des voitures déjà existantes dans la base.
6. Nous allons commencer par utiliser d'abord la base de données intégrée (in-memory database) à Spring Boot qui est la base H2 en mode développement ou test.

-
7. Pour pouvoir utiliser **H2** et **JPA** vous devez les ajouter en faisant un bouton droit sur le nom du projet et choisir Spring Starter et puis cochez ces deux dépendances, elles vont être ajoutées au fichier pom.xml.
 8. Ajoutez aussi la dépendance **Lombok**, au cas où.
 9. Vous créez le fichier de propriétés 'application.properties' qui contient les informations suivantes :

```
spring.h2.console.enabled=true  
spring.datasource.platform=h2  
spring.datasource.url=jdbc:h2:mem:testbd
```

10. Créez la classe Voiture dans le package 'modele' du root package 'org.cours'.

```
package org.cours.modele;  
  
...  
  
import lombok.Data;  
  
import lombok.NoArgsConstructor;  
  
import lombok.NonNull;  
  
import lombok.RequiredArgsConstructor;  
  
@Entity  
  
@Data  
  
@RequiredArgsConstructor  
  
@NoArgsConstructor  
  
public class Voiture {  
  
    @Id  
  
    @GeneratedValue(strategy=GenerationType.AUTO)  
  
    private long id;  
  
    @NonNull
```

```

private String marque;

@NotNull

private String modele;

@NotNull

private String couleur;

@NotNull

private String immatricule;

@NotNull

private int annee;

@NotNull

private int prix;

}

```

11. Créez maintenant l'interface 'VoitureRepo' qui va étendre CrudRepository dans le package 'org.cours.modele'.
12. CrudRepository fournit plusieurs méthodes dont ci-après la liste de celles parmi les plus utilisées :
 - a. long count(), retourne le nombre d'entités;
 - b. Iterable<T> findAll(), retourne tous les items d'un certain type;
 - c. Optional<T> findById(ID id), retourne un item par id;
 - d. void delete(T entity), supprime une entité;
 - e. void deleteAll(), supprime toutes les entités de cette repository;
 - f. <S extends T> save(S entity), sauvegarde une entité. Si la méthode retourne un seul élément, Optional<T> est retournée au lieu de T. Optional est un conteneur d'une seule valeur qui soit possède une valeur soit ne possède aucune. En utilisant Optional, on peut prévoir des exceptions de pointeur null.

```

package org.cours.modele;

public interface VoitureRepo extends CrudRepository<Voiture, Long> {

}

```

13. Nous allons après ajouter des données de test à notre base de données H2 en utilisant l'interface **CommandLineRunner** de Spring Boot, qui va nous permettre d'exécuter un code additionnel avant que l'application ne puisse démarrer complètement.

14. @SpringBootApplication

```
public class SpringDataRest {

    @Autowired
    private VoitureRepo repository;
    public static void main(String[] args) {
        SpringApplication.run(SpringDataRest.class, args);
    }
    @Bean
    CommandLineRunner runner(){
        return args -> {
            repository.save(new Voiture("Toyota", "Corolla", "Grise", "A-1-9090", 2018, 95000));
            repository.save(new Voiture("Ford", "Fiesta", "Rouge", "A-2-8090", 2015, 90000));
            repository.save(new Voiture("Honda", "CRV", "Bleu", "A-3-7090", 2016, 140000));
        };
    }
}
```

15. Lancez l'exécution du projet et via la console de H2, vous pouvez consulter les voitures ajoutées à la base (<http://localhost:8080/h2-console>).

16. Pour les opérations de manipulation des éléments de l'entité Voiture, vous pouvez via l'interface VoitureRepo définir vos propres requêtes comme suit :

```
package org.cours.modele;
public interface VoitureRepo extends CrudRepository<Voiture, Long> {
```

```

// Sélectionnez les voitures par marque

List<Voiture> findByMarque(String marque);

// Sélectionnez les voitures par couleur

List<Voiture> findByCouleur(String couleur);

// Sélectionnez les voitures par année

List<Voiture> findByAnnee(int annnee);

// Sélectionnez les voitures par marque et modèle

List<Voiture> findByMarqueAndModele(String marque, String modele);

// Sélectionnez les voitures par marque ou couleur

List<Voiture> findByMarqueOrCouleur(String marque, String couleur);

// Sélectionnez les voitures par marque et trier par année

List<Voiture> findByMarqueOrderByAnneeAsc(String marque);

// Sélectionnez les voitures par marque en utilisant SQL

@Query("select v from Voiture v where v.marque = ?1")

List<Voiture> findByMarque(String marque);

@Query("select v from Voiture v where v.marque like %?1")

List<Voiture> findByMarqueEndsWith(String marque);

}

```

17. Nous créons maintenant l'entité **Proprietaire** au niveau du package 'modele' :

```

...
import lombok.Data;

import lombok.NoArgsConstructor;

import lombok.NonNull;

```

```

import lombok.RequiredArgsConstructor;

@Entity
@Data
@RequiredArgsConstructor
@NoArgsConstructor
public class Proprietaire {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    @NotNull
    private String nom;

    @NotNull
    private String prenom;

}

```

18. Nous créons après l'interface **ProprietaireRepo** comme suit :

19. package org.cours.modele;

```

public interface ProprietaireRepo extends CrudRepository<Proprietaire, Long> {
}

```

20. Exécutez pour voir si tout marche bien.

21. Maintenant on va relier les deux entités **Voiture** et **Proprietaire** par une association de type **ManyToOne** côté Voiture. Pour cela nous allons ajouter un attribut propriétaire au niveau de la classe **Voiture** et l'annoter par l'annotation **@ManyToOne** comme suit :

22. **@NotNull @ManyToOne(fetch = FetchType.LAZY)**

```

@JoinColumn(name = "proprietaire")
private Proprietaire propriétaire;

```

23. Dans la classe **Proprietaire**, on apporte les modifications suivantes :

24. @OneToMany(cascade = CascadeType.ALL, mappedBy="proprietaire")

```
private List<Voiture> voitures;
```

25. Exécutez pour voir les nouveaux changements.

26. Nous allons à ce niveau ajouter deux propriétaires comme pour les voitures au niveau de CommandLineRunner, après avoir ajouté @Autowired ProprietaireRepo proprietaireRepo :

27. Proprietaire proprietaire1 = new Proprietaire("Ali", "Hassan");

```
Proprietaire proprietaire2 = new Proprietaire("Najat", "Bani");
```

```
proprietaireRepo.save(proprietaire1);
```

```
proprietaireRepo.save(proprietaire2);
```

```
repository.save(new Voiture("Toyota", "Corolla", "Grise", "A-1-9090", 2018, 95000),  
proprietaire1);
```

```
repository.save(new Voiture("Ford", "Fiesta", "Rouge", "A-2-8090", 2015, 90000,  
proprietaire1));
```

```
repository.save(new Voiture("Honda", "CRV", "Bleu", "A-3-7090", 2016, 140000,  
proprietaire2));
```

28. Pour passer après sur *MariaDB*, il faut chercher sa dépendance sur le site *mvnrepository* et la placer sur le fichier pom.xml. Pour ceux qui utilisent *Postgresql*, essayez d'ajouter sa dépendance depuis Spring Boot.

29. Au niveau du fichier application.properties vous allez placer les lignes suivantes (où compagnie c'est le nom de la BD) et enlever ceux de H2 :

```
spring.datasource.url=jdbc:mariadb://localhost:3306/compagnie
```

```
spring.datasource.username=root
```

```
spring.datasource.password=YOUR_PASSWORD
```

```
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
```

```
spring.jpa.generate-ddl=true
```

```
spring.jpa.hibernate.ddl-auto=create-drop
```

```
//pour Postgresql

spring.datasource.url=jdbc:postgresql://localhost:5432/springboot

spring.datasource.username=postgres

spring.datasource.password=admin

spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.generate-ddl=true

spring.jpa.hibernate.ddl-auto=create-drop

//pour générer le path /api

spring.data.rest.base-path=/api
```

30. Vous pouvez exécuter le projet et voir les tables créées au niveau de *MariaDB*.

Spring Rest

1. A ce stade, nous allons créer un webservice RESTful. Nous commençons par créer la classe *VoitureController* dans le package 'org.cours.web'.
2. Cette classe va être annotée par `@RestController` pour indiquer qu'elle va jouer le rôle du contrôleur du webservice RESTfull.
3. Nous allons ajouter une méthode `Iterable` qui va être annotée par `@RequestMapping` qui définit le 'endpoint' ("voitures") avec lequel la méthode va être mappée. Cette annotation permet de traiter toutes les méthodes HTTP (GET, POST, PUT, PATCH, DELETE, etc), il suffit de faire `@RequestMapping("/voitures", method=GET)` pour définir le endpoint 'voitures' qui va être manipulé par la méthode GET. Nous allons nous contenter maintenant de la méthode GET, qui est la méthode utilisée par défaut. On va injecter ensuite la classe *VoitureRepo* pour pouvoir utiliser sa méthode `findAll()` qui va nous retourner toutes les voitures de la Repository. Le code est comme suit :

```
@RestController

public class VoitureController {
```

```

@Autowired

private VoitureRepo voitureRepo;

@RequestMapping("/voitures")

public Iterable<Voiture> getVoitures(){

    return voitureRepo.findAll();

}

}

```

- Allez après sur Postman pour faire les tests. Choisir la méthode 'GET' et 'localhost:8080/voitures'. Appuyez sur 'Send' et vous allez remarquer qu'il y a un effet de boucle dû à la sérialisation répétée entre Voiture et Proprietaire. Pour stopper cet effet, nous allons ajouter l'annotation `@JsonIgnore` avant l'attribut de jointure dans les deux classes Voiture et Proprietaire, comme suit :

```

//Proprietaire.java

@OneToMany(cascade = CascadeType.ALL, mappedBy = "proprietaire")

@JsonIgnore

private List<Voiture> voitures;

//-----

//Voiture.java

@JoinColumn(name = "proprietaire")

@JsonIgnore

private Proprietaire proprietaire;

```

- Maintenant vous pouvez ré-exécuter et voir le résultat qui affiche la liste des voitures sur 'localhost:8080/voitures', avec POSTMAN.
- Vous pouvez par après essayer d'ajouter deux classes services l'une pour voiture et l'autre pour proprietaire pour pouvoir assurer les autres méthodes POST, PUT, DELETE, etc.
- Voilà !!!

8. Maintenant nous allons intégrer la dépendance **Jpa Data Rest** dans notre projet. Faites un bouton droit sur le nom du projet, ensuite 'Spring', ensuite 'Edit Starters'.
9. Vous allez après ajouter au fichier 'application.properties' la ligne suivante : `spring.data.rest.base-path=/api` , cela va permettre de définir un endpoint nommé 'api' pour gérer toutes les méthodes HTTP (GET, POST, PUT, etc).
10. Redémarrez le serveur et tapez l'adresse <http://localhost:8080/api>
11. Vous allez voir l'affichage des données JSON dans le format HAL (HyperText Application Language). Le format HAL fournit un ensemble de conventions pour exprimer les hyperliens sous JSON et faire en sorte que votre webservice RESTfull soit facile à utiliser par les développeurs Frontend.
12. Si vous choisissez l'url <http://localhost:8080/api/voitures> vous allez pouvoir récupérer la liste de toutes les voitures.
13. Le service Spring Data Rest fournit toutes les opérations CRUD. GET: Read, POST: Create, PUT/PATCH: Update, DELETE: Delete.
14. Pour supprimer la voiture par exemple dont l'id est 3, on tape l'url suivante : <http://localhost:4545/api/voitures/3> via l'outil POSTMAN par exemple. Amusez vous à tester toutes les autres méthodes : GET, POST, PATCH et PUT.
15. Pour inclure maintenant ces requêtes dans notre service, nous allons ajouter l'annotation `@RepositoryRestResource` à la classe repository. Les paramètres de la requête sont annotés par l'annotation `@Param`. Nous allons appliquer ça à la classe VoitureRepo :
16. `@RepositoryRestResource`

```

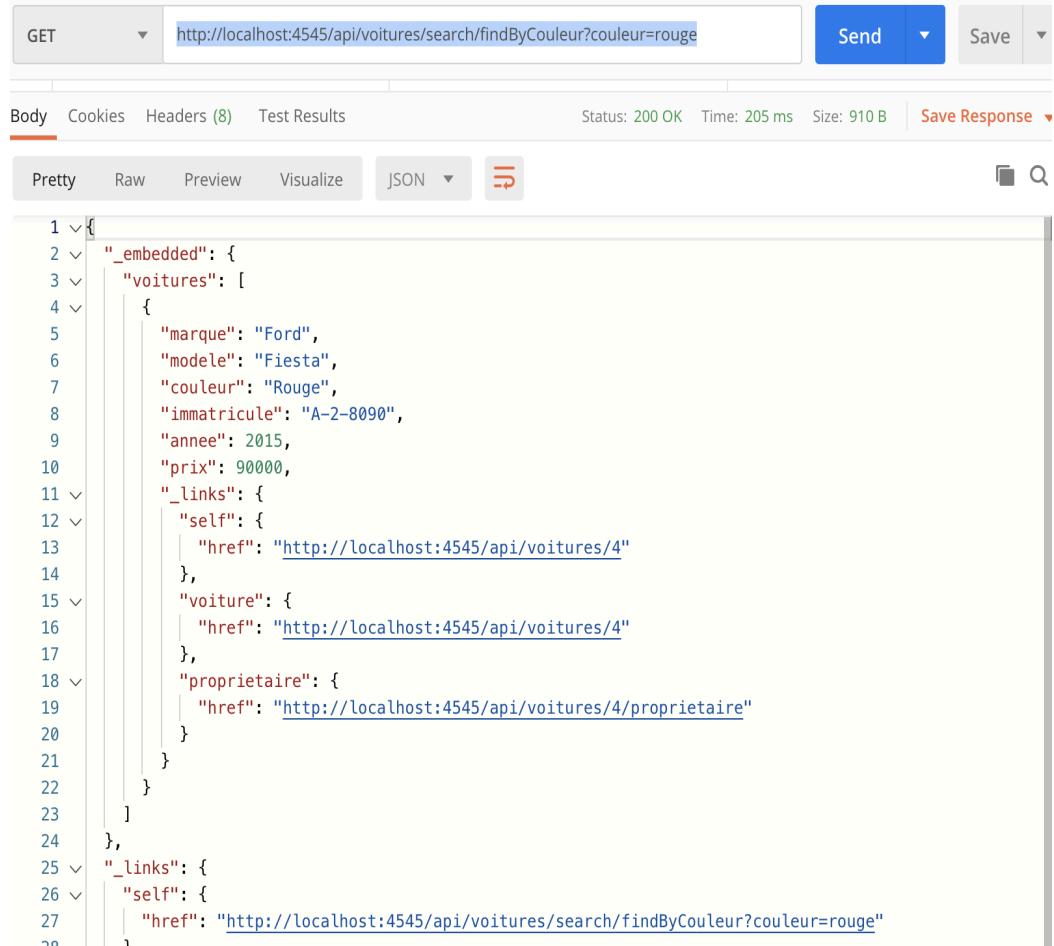
public interface VoitureRepo extends CrudRepository<Voiture, Long>{
    //Lister Voitures par marque
    List<Voiture> findByModele(@Param("modele") String modele);
    //Lister Voitures par couleur
    List<Voiture> findByCouleur(@Param("couleur") String couleur);
}
```

17. On exécute le projet avec GET pour l'api voitures, on va s'apercevoir qu'une nouvelle branche 'search' est ajoutée que vous pouvez explorer. Cliquez sur cette branche et valider en appuyant sur le bouton 'Send' de Postman.

18. Veuillez sélectionner par exemple la branche des couleurs et s'assurer que dans l'url de Postman vous avez ça :

<http://localhost:4545/api/voitures/search/findByCouleur?couleur=Rouge>

19. Si vous validez vous allez obtenir la réponse suivante :



```

1 <!
2   "_embedded": {
3     "voitures": [
4       {
5         "marque": "Ford",
6         "modele": "Fiesta",
7         "couleur": "Rouge",
8         "immatricule": "A-2-8090",
9         "annee": 2015,
10        "prix": 90000,
11        "_links": {
12          "self": {
13            "href": "http://localhost:4545/api/voitures/4"
14          },
15          "voiture": {
16            "href": "http://localhost:4545/api/voitures/4"
17          },
18          "proprietaire": {
19            "href": "http://localhost:4545/api/voitures/4/proprietaire"
20          }
21        }
22      }
23    ],
24    "_links": {
25      "self": {
26        "href": "http://localhost:4545/api/voitures/search/findByCouleur?couleur=rouge"
27      }
28    }
29  }
30

```

Springdoc-openapi

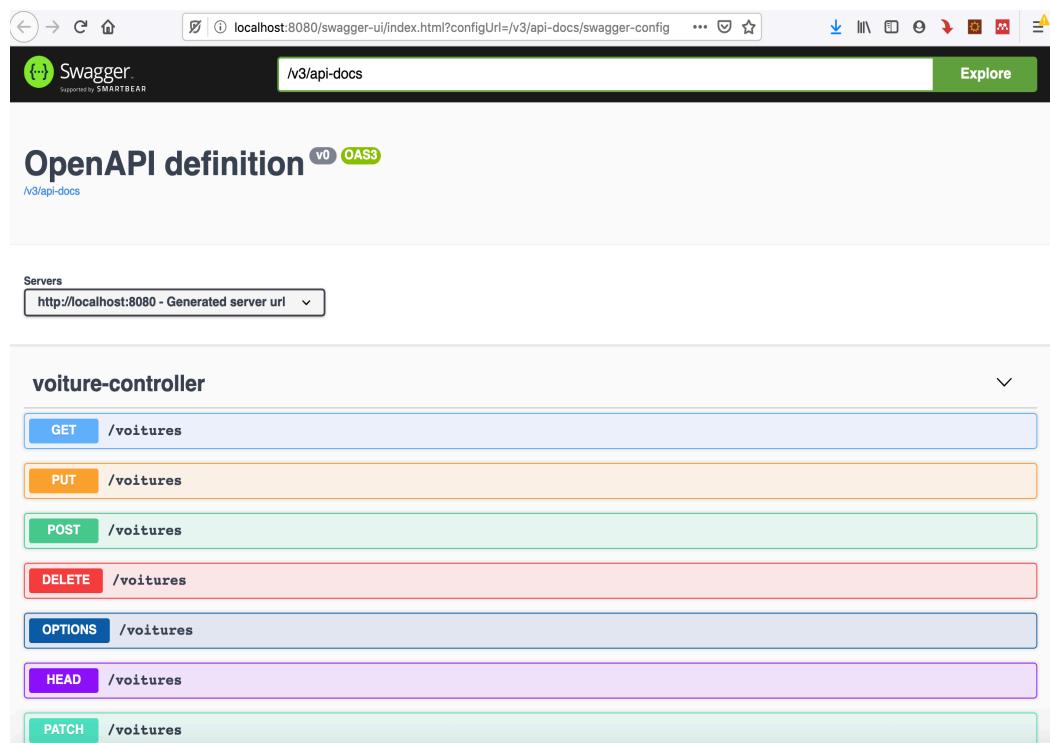
1. La librairie java **springdoc-openapi** aide à automatiser la génération de la documentation API utilisant les projets spring boot. Cette librairie fonctionne en examinant une application en mode exécution pour inférer les sémantiques de l'API en se basant sur les configurations de Spring, la structure des classes et les différentes annotations. Springdoc-openapi se veut une bonne alternative au célèbre Swagger.
2. Springdoc-openapi génère automatiquement la documentation dans les formats APIs JSON/Yaml et HTML. Cette documentation peut

éventuellement être complétée par l'usage des annotations de Swagger. Cette librairie supporte OpenAPI 3, Spring-boot (v1 et v2), JSR-303, Swagger-ui et OAuth 2.

- Pour utiliser cette librairie dans notre projet, il suffit d'ajouter la dépendance suivante au niveau du fichier pom.xml :

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.4.0</version>
</dependency>
```

- Relancez votre projet et consultez les deux chemins : <http://localhost:8080/swagger-ui.html> qui va vous permettre de consulter votre documentation Rest sous une forme agréable bien sûr en demandant les services de swagger-ui.



The screenshot shows the Swagger UI interface. At the top, it displays the URL localhost:8080/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config. Below the header, there's a navigation bar with icons for back, forward, search, and refresh, followed by a green "Explore" button. The main content area is titled "OpenAPI definition v0 OAS3". It shows the path [/v3/api-docs](http://localhost:8080/v3/api-docs). Underneath, there's a "Servers" dropdown set to "http://localhost:8080 - Generated server url". The main part of the screen is titled "voiture-controller" and lists various HTTP methods for the "/voitures" endpoint: GET, PUT, POST, DELETE, OPTIONS, HEAD, and PATCH. Each method is represented by a colored button (blue for GET, orange for PUT, green for POST, red for DELETE, light blue for OPTIONS, purple for HEAD, and light green for PATCH) followed by its corresponding URL path.

L'autre chemin <http://localhost:8080/v3/api-docs> qui va lui vous permettre

de générer la documentation dans sa forme basique.

```

{
  "openapi": "3.0.1",
  "info": {
    "title": "OpenAPI definition",
    "version": "v0"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Generated server url"
    }
  ],
  "paths": {
    "/voitures": {
      "get": {
        "tags": [
          "voiture-controller"
        ],
        "operationId": "getVoitures_3",
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "/*/*": {
                "schema": {}
              }
            }
          }
        }
      },
      "post": {
        "tags": [
          "voiture-controller"
        ],
        "operationId": "getVoitures_1",
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "/*/*": {
                "schema": {}
              }
            }
          }
        }
      },
      "put": {},
      "delete": {},
      "options": {},
      "head": {},
      "patch": {}
    }
  }
}

```

Spring Security

1. **Spring Security** fournit les services de sécurité pour les applications web à base de Java.
2. Commençons par ajouter la dépendance Spring Security à notre projet Spring boot.
3. Lorsque vous lancez votre application, un utilisateur va être ajouté avec le nom 'user' et le password vous allez le voir sur la console, par exemple : 263ab9a7-c5da-40c2-9660-aecd09b1937b
4. Si vous essayez maintenant de faire le GET, vous aurez un message d'erreur comme quoi vous n'êtes pas autorisé.

-
5. Pour pouvoir faire l'appel du GET, on va faire une authentification basique. Sur Postman, sélectionnez au niveau de l'onglet 'Authorization' le Type 'Basic Auth' et renseigner après votre 'user' et 'password'.

Le Test sous Spring Boot

1. Commencez par ajouter la dépendance suivante au niveau du pom.xml pour pouvoir faire le test sous H2 :

```
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>test</scope>
</dependency>
```
2. Veuillez ouvrir la classe générée par défaut par Spring Boot et qui va s'occuper de réaliser les tests : 'SpringbootReactPart1ApplicationTests' et la modifier de cette façon:

```
package org.cours;
import static org.assertj.core.api.Assertions.assertThat;
import org.cours.web.VoitureController;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
@RunWith(SpringRunner.class)
@SpringBootTest
//indique que c'est une classe de test régulier qui exécute les tests à base de Spring boot
class SpringbootReactPart1ApplicationTests {
    @Autowired
    VoitureController voitureController;
    @Test
    //indique que cette méthode peut être exécutée comme un cas de test
    void contextLoads() {
        assertThat(voitureController).isNotNull();
        //on teste si l'instance du contrôleur a été créé et injectée avec succès.
    }
}
```

-
3. Faites bouton droit sur cette classe de test et choisir Run As -> Junit Test.
Si tout passe bien, une barre verte va s'afficher, cela veut dire techniquement que le test passe.
 4. Maintenant nous allons créer une classe de test pour les opérations CRUD de notre repository Voiture. Pour cela on crée la classe de test VoitureRepoTest comme suit :

```
package org.cours;
import static org.assertj.core.api.Assertions.assertThat;
import org.cours.modele.Voiture;
import org.cours.modele.VoitureRepo;
import org.junit.jupiter.api.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import org.springframework.test.context.junit4.SpringRunner;
@RunWith(SpringRunner.class)
@DataJpaTest
//Si le test concerne uniquement les composantes JPA
//Lorsque cette annotation est utilisée, H2, Hibernate et Spring Data sont configurés
//automatiquement pour le test.
public class VoitureRepoTest {
    @Autowired
    private TestEntityManager entityManager;
    //TestEntityManager est utilisée pour manipuler les entités persistantes
    @Autowired
    VoitureRepo voitureRepo;
    @Test
    public void ajouterVoiture() {
        Voiture voiture = new Voiture ("MiolaCar" "Uber" "Blanche" "M-2020", 2021, 180000);
        entityManager.persistAndFlush(voiture);
        //permet de faire persister ce tuple de l'entité Voiture
        assertThat(voiture.getId()).isNotNull();
        //permet de tester qu'un tuple de Voiture a bien été ajoutée en mémoire H2
    }
    @Test
    public void supprimerVoiture() {
```

```

entityManager.persistAndFlush(new Voiture ("MiolaCar","Uber","Blanche","M-2020", 2021,
180000));
entityManager.persistAndFlush(new Voiture ("MiniCooper","Uber","Rouge","C-2020", 2021,
180000));
voitureRepo.deleteAll();
assertThat(voitureRepo.findAll()).isEmpty();
//permet de tester si tous les tuples ont été supprimés
}
}

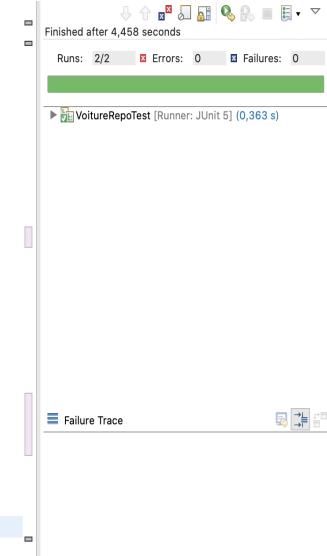
```

- Exécutez par Run As -> Junit Test et si tout marche bien vous allez avoir une barre verte comme ci dessous :

```

package org.cours;
import static org.assertj.core.api.Assertions.assertThat;
@RunWith(SpringRunner.class)
@DataJpaTest
public class VoitureRepoTest {
    @Autowired
    private TestEntityManager entityManager;
    @Autowired
    VoitureRepo voitureRepo;
    @Test
    public void ajouterVoiture() {
        Voiture voiture = new Voiture ("MiolaCar", "Uber", "Blanche", "M-2020",
2021, 180000);
        entityManager.persistAndFlush(voiture);
        assertThat(voiture.getId()).isNotNull();
    }
    @Test
    public void supprimerVoiture() {
        entityManager.persistAndFlush(new Voiture ("MiolaCar", "Uber",
"Blanche", "M-2020", 2021, 180000));
        entityManager.persistAndFlush(new Voiture ("MiniCooper", "Uber",
"Blanche", "M-2020", 2021, 180000));
        voitureRepo.deleteAll();
        assertThat(voitureRepo.findAll()).isEmpty();
    }
}

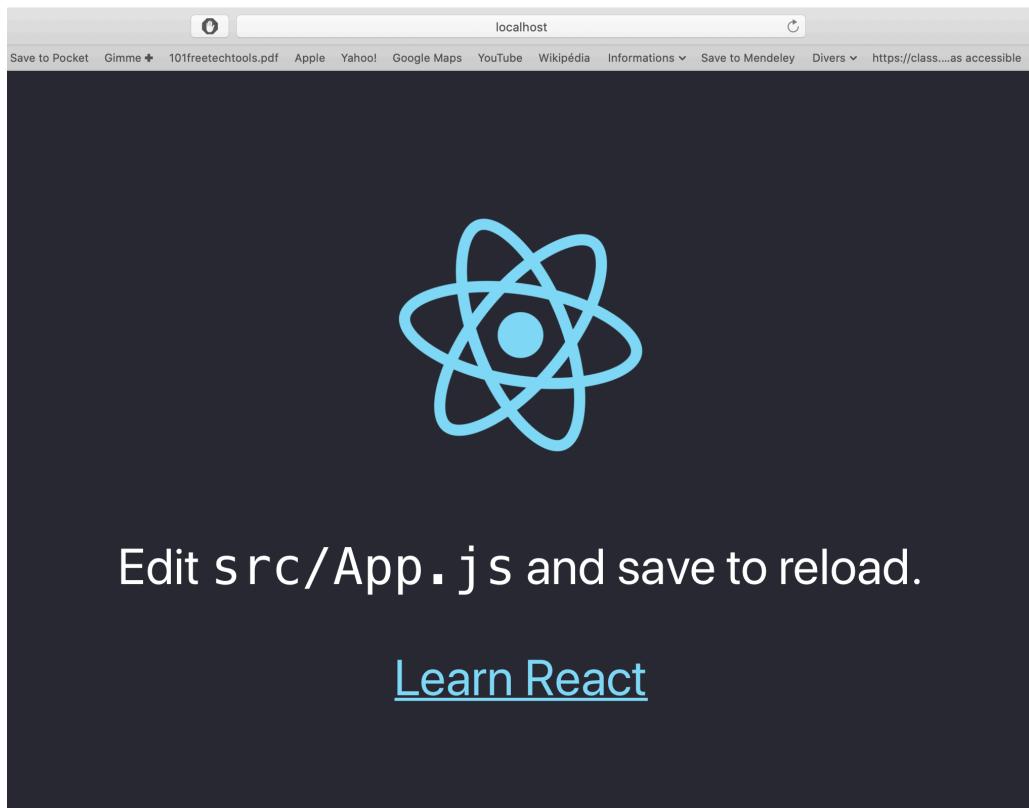
```



Frontend avec React

- On commence par installer les outils [Node.js](#) et [VS Code](#).
- Ouvrir le terminal et tapez `node -v` et `npm -v` pour voir si ces deux environnements sont bien installés.
- Pour installer React vous tapez cette commande : `npm install -g create-react-app`
- Une fois l'installation terminée, nous allons créer notre projet react 'myapp' en tapant : `create-react-app myapp`
- Une fois créée vous tapez : `cd myapp` pour changer de dossier.

6. À partir de ce dossier on va pouvoir exécuter notre application sur le port 3000 en tapant la commande : **npm start** (ou **yarn start**)
7. Vous allez avoir le rendu suivant :



8. Pour arrêter le serveur, faites Ctrl-C.
9. Ouvrir le projet react 'myapp' sous l'outil Visual Code et après ouvrir le fichier App.js pour le modifier ainsi :
10.

```
import React from 'react';
import './App.css';
function App() {
  return (
    <div className="App">
      Hello React!
    </div>
  );
}

export default App;
```
11. Nous supposons que le projet Spring Boot est déjà créé sous un outil tel que STS ou autre. Au niveau de l'arborescence du projet sous STS vous allez créer un dossier 'webapp' sous le dossier 'main' de 'src'.
12. Faites un bouton droit sur le dossier 'webapp' et lancez le terminal. À partir du terminal vous tapez la commande pour créer un projet react de nom 'reactjs' : **create-react-app reactjs**
13. **cd reactjs**
14. **npm start**

-
15. Nous allons maintenant installer react bootstrap. Ce dernier remplace bootstrap Javascript. Chaque composant de bootstrap a été construit from scratch.
 16. Au niveau du moteur de recherche, tapez **react bootstrap** et cliquez sur le premier lien 'react-bootstrap.github.io' et appuyez sur le bouton 'Get Started'. Vous copiez la commande d'installation.
 17. Arrêtez le serveur et tapez en ligne de commande : **npm install react-bootstrap bootstrap**
 18. Nous allons prendre le code suivant du site 'react-bootstrap.github.io' et le placer au niveau du fichier index.html juste avant la rubrique <title> comme suit :

```
<link
  rel="stylesheet"
  href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
  integrity="sha384-Vkoo8x4CGs03+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiF
  eWPGFN9MuhOf23Q9Ifjh"
  crossorigin="anonymous" />
<title>Voiture Shop</title>
```
 19. Enlevez tout le code indésirable de ce fichier et relancer le serveur react avec la commande **npm start**.
 20. Changez le code du fichier 'App.js' de telle sorte de ne garder que ce script :

```
function App() { return ( <div className="App"><p>Bienvenue au
  Magasin des Voitures</p></div>);} export default App;
```
 21. Modifiez légèrement au niveau du fichier 'index.html' et y placez cette balise :

```
<link rel="icon" href="https://upload.wikimedia.org/wikipedia/commons/1/17/Tata_Tamo_Racemo.jpg" />
```
 22. Si jamais vous voyez pas le dossier nodes_modules au niveau de l'arborescence du projet sous STS alors au niveau du terminal arrêtez le projet avec Ctrl C, ensuite faites un **npm install** pour installer les packages manquants.
 23. Lancez la commande : **npm install**
 24. Enlevez le contenu du fichier 'index.css'.

- 25.Au niveau du dossier ‘src’ nous créons un sous dossier intitulé ‘Components’.
- 26.Nous créons dans ‘Components’ le fichier ‘NavigationBar.js’.
- 27.Dans ‘NavigationBar.js’ tapez le code suivant : import React from ‘react’; class NavigationBar extends React.Component { render() { return (<div>Navigation Bar</div>); } } export default NavigationBar;
- 28.Dans le fichier ‘App.js’ nous allons importer le composant ‘NavigationBar’ comme suit : ...import NavigationBar from ‘.:Components/NavigationBar’; function App() { ...<div className="APP"><NavigationBar/><p>Bienvenue.
- 29.Sauvegardez et observez le changement.
- 30.On revient sur le site react bootstrap vu ci dessus et y chercher le composant Navbar (vous donne un effet de branding au niveau de votre barre de navigation Navbar) que nous allons utiliser dans ‘NavigationBar.js’ comme suit :

```
import React from 'react';
import { Navbar, Nav } from 'react-bootstrap';
class NavigationBar extends React.Component {
  render() {
    return (
      <Navbar bg="dark" variant="dark">
        <Navbar.Brand href="/">
          
        </Navbar.Brand>
        <Nav className="mr-auto">
          <Nav.Link href="#"> Ajouter Voiture</Nav.Link>
          <Nav.Link href="#">
            Liste Voitures
          </Nav.Link>
        </Nav>
      </Navbar>
    );
  }
}
```

```
export default NavigationBar;
```

31. Le composant [Navbar](#) permet de définir une barre de navigation puissante au niveau de votre page. Nav.Link permet de définir des liens au niveau de cette barre de navigation.

32. Sauvegardez et observez le changement.

33. Cherchons maintenant le composant Jumbotron toujours dans le même site react bootstrap. On copie le code de Jumbotron dans 'App.js' comme suit :

```
import React from 'react';
import './App.css';
import { Container, Row, Jumbotron } from 'react-bootstrap';
import NavigationBar from './Components/NavigationBar';
function App() {
    render() {
        return (
            <div className="App">
                <NavigationBar/>
                <Container>
                    <Row>
                        <Jumbotron>
                            <h1>Hello World!</h1>
                            <p>
                                Ceci est le corps de la page.
                            </p>
                        </Jumbotron>
                    </Row>
                </Container>
            </div>
        );
    }
}
export default App;
```

34. Le composant Jumbotron permet de mettre en avant le corps de votre page avec un rendu remarquable.

35. Le composant [Container](#) permet de centrer le contenu, son sous-élément Row permet de définir des lignes à l'intérieur du Container avec

éventuellement la possibilité d'intégrer des colonnes au niveau de chaque Row.

36. Sauvegardez et observez le changement.
37. Ajoutons des marge pour Jumbotron en ajoutant une constante juste avant 'return' comme ça :

```
function App() { const marginTop = { marginTop:"20px"};...<Container><Row><Col lg={12} style = {marginTop}><Jumbotron>...</Jumbotron></Col>
```
38. Sauvegardez et observez le changement.
39. Nous voulons rendre le composant Jumbotron noir, on applique alors le changement suivant : `<Jumbotron className="bg-dark text-white">`. Sauvegardez et observez le changement.
40. Nous allons vouloir modifier aussi la couleur du fond. Cela nécessite un changement au niveau du fichier 'index.html' dans la balise 'body' comme suit : `<body style="background-color:#272B30">`.
41. Créons maintenant un autre composant 'Bienvenue.js' toujours dans le dossier 'Components' et y placer le code suivant :
42. `import React from 'react'; import {Jumbotron} from 'react-bootstrap'; class Bienvenue extends React.Component { render() { return (<Jumbotron>...</Jumbotron>); } } export default Bienvenue . La partie <Jumbotron> on va la prendre du fichier App.js et la coller ici et la supprimer dans l'emplacement initial.`
43. Dans 'App.js', après avoir enlevé le code `<Jumbotron>...</Jumbotron>` on va le remplacer par le code : `<Bienvenue/>` après avoir importé le composant Bienvenue au niveau de 'App.js' comme suit : `import Bienvenue from './components/Bienvenue';`
44. Modifions maintenant 'Bienvenue.js' comme suit : `<Jumbotron...><h1> Bienvenue au Magasin des Voitures</h1> <blockquote className= "blockquote mb-0"> <p>Le meilleur de nos voitures est exposé près de chez vous</p> <footer className="blockquote-footer">Master MIOLA </footer></blockquote></Jumbotron>.`
45. Sauvegarder et observer le changement.
46. Ajoutons maintenant le composant 'Footer.js' que nous allons insérer juste après `</Container><Footer/>` au niveau de 'App.js'.

47. Le code de 'Footer.js' est comme suit : import React from 'react'; import {Navbar, Container, Col} from 'react-bootstrap'; class Footer extends React.Component { render() { return { <Navbar fixed="bottom" bg="dark" variant="dark"><Container><Col lg={12} className="text-center text-muted"><div>All Rights Reserved by Master MIOLA</div> </Col> </Container></Navbar>}; } }
48. Container permet de centrer le contenu. Sauvegarder et observer le changement. Pour ajouter l'année nous créons une variable 'fullYear' de type Date() comme ci après : render(){ let fullYear = new Date().getFullYear(); return (...<div>{fullYear} - {fullYear+1}, All Rights...);}
49. Le code de 'Footer.js' ressemble à celui-ci :

```
import React from 'react';
import {Navbar, Container, Col} from 'react-bootstrap';
class Footer extends React.Component {
  render() {
    let fullYear = new Date().getFullYear();
    return (
      <Navbar fixed="bottom" bg="dark" variant="dark">
        <Container>
          <Col lg={12} className="text-center text-muted">
            <div>
              {fullYear}-{fullYear+1}, All Rights Reserved by Master MIOLA
            </div>
          </Col>
        </Container>
      </Navbar>
    );
  }
}
export default Footer;
```

50. En exécutant, vous allez avoir le rendu suivant :



51. Créons maintenant deux composants Voiture.js et VoitureListe.js et ajoutez les dans 'App.js' comme suit : <Bienvenue/> <Voiture/> <VoitureListe/>, et n'oubliez pas d'importer ces composants dans 'App.js'
52. Pour 'VoitureListe.js' insérez le code suivant : import React from 'react'; class VoitureListe extends React.Component { render() { return (<div className="text-white">Liste Voitures</div>);}} export default VoitureListe;
53. Même chose pour 'Voiture.js'
54. Pour assurer une bonne navigation nous allons installer le composant react-router-dom comme suit :
55. Arrêtez le serveur et lancez la commande : **npm install --save react-router-dom**. Lancez ensuite **npm start** pour exécuter le serveur.
56. Dans 'App.js' faites : import {BrowserRouter as Router, Switch, Route} from 'react-router-dom'; puis remplacer la balise <div className="App"> par <Router>. Le composant [Router](#) va nous permettre d'instaurer la navigabilité au niveau de notre page 'App.js', il va jouer le rôle d'un routeur. Nous choisissons ici la configuration [Router -> Row -> Switch](#).
57. Après nous apportons la modification suivante dans 'App.js' comme suit : <Col...><Switch> <Route path="/" exact component={Bienvenue}/> <Route

```
path="/add" exact component={Voiture}/> <Route path="/list" exact component={VoitureListe}/> </Switch></Col>...
```

58. Les liens au niveau de 'path' vont être définis par 'Link' au niveau de la page 'NavigationBar.js'.

59. Nous ajoutons des liens au niveau de 'NavigationBar.js' comme suit :

```
import {Link} from 'react-router-dom'; ...<Navbar bg="dark" variant="dark">
<Link to={"/"} className="navbar-brand">  </Link>
<Nav className="mr-auto"> <Link to={"add"} className="nav-link">
Ajouter Voiture </Link><Link to={"list"} className="nav-link"> Liste Voiture </Link> </Nav> </Nav>...Sauvegardez et observez le changement.
```

60. Le code de 'App.js' est comme suit :

```
import React from 'react';
import { Navbar, Nav } from 'react-bootstrap';
import {Link} from 'react-router-dom';
class NavigationBar extends React.Component {
  render() {
    return [
      <Navbar bg="dark" variant="dark">
        <Link to={"/"} className="nav-link">
          
          Ajouter une Voiture
        </Link>
        <Link to={"add"} className="nav-link">
          Liste des Voitures
        </Link>
      </Navbar>
    ];
  }
  export default NavigationBar;
```

61. Le rendu est comme suit :



62. Dans 'VoitureListe.js' apportez les changements suivants: import React, {Component} from 'react'; export default class VoitureListe extends Component { render() { return...} }
63. Nous ajoutons maintenant le composant 'Card' au niveau de 'VoitureListe.js' comme suit : import {Card} from 'react-bootstrap'; ...return { <Card className=""> <Card.Header> Liste Voitures </Card.Header> </Card>...Sauvegardez et observez le changement. Après dans className de Card mettez : className={"border border-dark bg-dark text-white"} et observez le changement.
64. [Card](#) est un composant qui permet de concevoir des cartes d'affichage très sophistiquées.
65. Ajoutons maintenant le Card.Body comme suit :
- 66....</Card.Header>
- ```
<Card.Body><Table><thead><tr><th>Marque</th><th>Modele</th><th>Couleur</th><th>Annee</th><th>Prix</th></tr></thead><tbody><tr align="center"><td colSpan="6">Aucune Voiture n'est disponible</td></tr></tbody></Table></Card.Body></Card>.
```
67. Sauvegardez et observez le changement.

68. Le code de 'App.js' est comme suit :

```
import React from 'react';
import NavigationBar from './Components/NavigationBar';
import {BrowserRouter as Router, Switch, Route} from 'react-router-dom';
import { Container, Row, Col} from 'react-bootstrap';
import './App.css';
import Bienvenue from './Components/Bienvenue';
import Footer from './Components/Footer';
import Voiture from './Components/Voiture';
import VoitureListe from './Components/VoitureListe';
function App() {
 const marginTop = { marginTop:"20px"}
 return (
 <Router>
 <NavigationBar/>
 <Container>
 <Row>
 <Col lg={12} style={marginTop}>
 <Switch>
 | <Route path="/" exact component={Bienvenue}/>
 | <Route path="/add" exact component={Voiture}/>
 | <Route path="/list" exact component={VoitureListe}/>
 </Switch>
 </Col>
 </Row>
 </Container>
 <Footer/>
 </Router>
);
}
export default App;
```

69. Le code de 'Voiture.js' est comme suit :

```
import React from 'react';
export default class Voiture extends React.Component {
 render() {
 return (
 <div className="text-white">
 | Liste Voiture
 </div>
);
 }
}
```

70. Le code de 'VoitureListe.js' est comme suit :

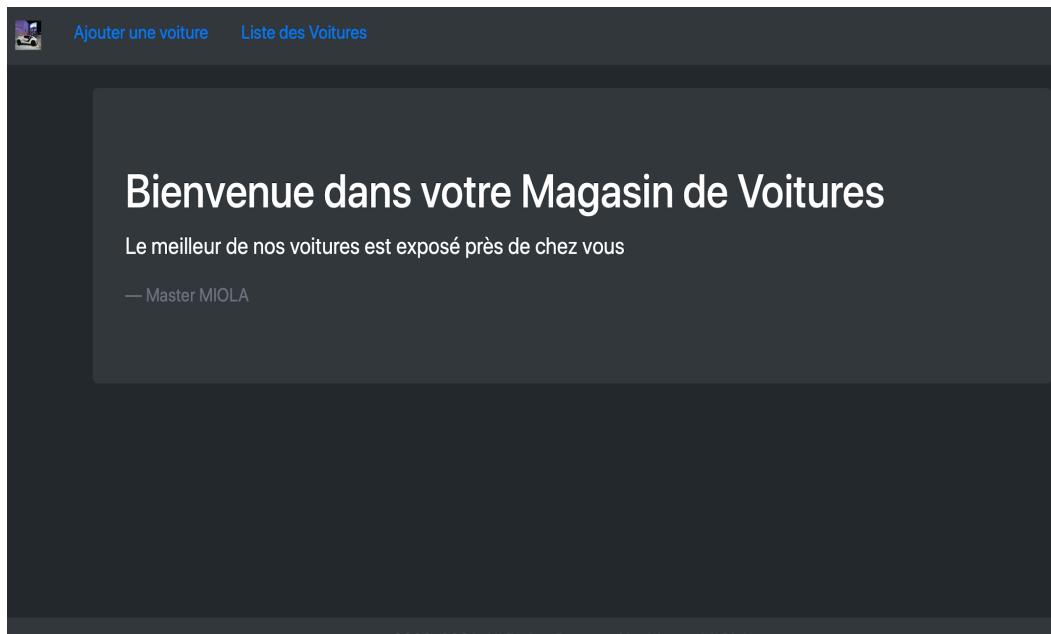
```
import React from 'react';
import { Card, Table } from 'react-bootstrap';
export default class VoitureListe extends React.Component {
 render() {
 return [
 <Card className={"border border-dark bg-dark text-white"}>
 <Card.Header>Liste des Voitures</Card.Header>
 <Card.Body>
 <Table><thead>
 <tr>
 <th>Marque</th>
 <th>Modele </th>
 <th>Couleur</th>
 <th>Annee</th>
 <th>Prix</th></tr>
 </thead>
 <tbody>
 <tr align="center">
 <td colSpan="6">Aucune Voiture n'est disponible</td>
 </tr>
 </tbody>
 </Table>
 </Card.Body>
 </Card>
];
 }
}
```

71. Le code de 'NavigationBar.js' est comme suit :

```
import React from 'react';
import { Navbar, Nav } from 'react-bootstrap';
import {Link} from 'react-router-dom';
export default class NavigationBar extends React.Component {
 render() {
 return [
 <Navbar bg="dark" variant="dark">
 <Link to={"/"} className="navbar-brand">

 </Link>
 <Link to={"add"} className="nav-link">
 Ajouter une voiture
 </Link>
 <Link to={"list"} className="nav-link">
 Liste des Voitures
 </Link>
 </Navbar>
];
 }
}
```

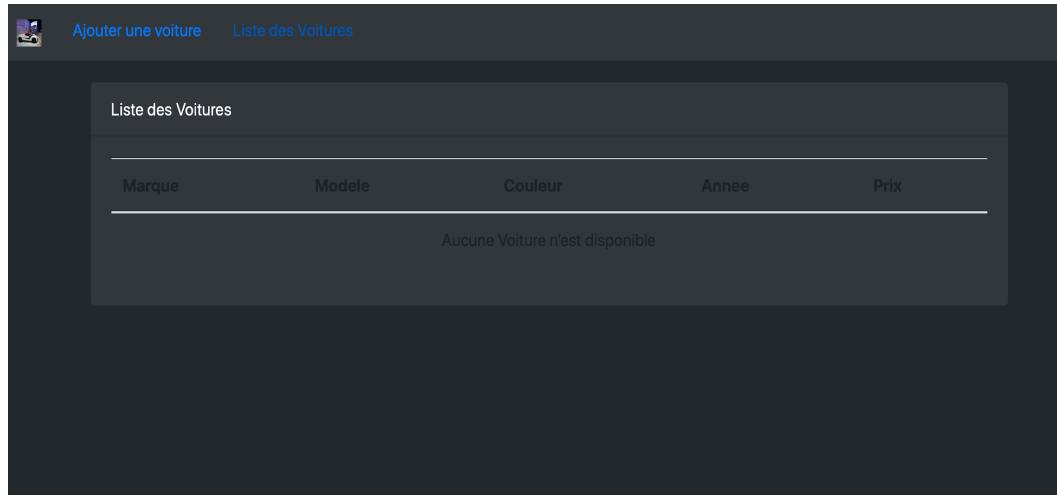
72. Ce qui nous donne le rendu suivant :



73. Si on clique sur le menu de navigation ‘Ajouter une Voiture’ on aura :



74. Et si on clique sur ‘Liste des Voitures’ on obtient ça :



75. Modifions légèrement le look de la table en ajoutant des propriétés au composant Table dans ‘VoitureListe.js’ comme suit : <Table bordered hover striped variant="dark">

76. Nous allons maintenant utiliser le composant Card au niveau de ‘Voiture.js’ et la modifiez ainsi : import React, {Component} from ‘react’; import {Card} from ‘react-bootstrap’; export default class Voiture extends Component { render() { return ( <Card className={"border border-dark bg-dark text-white"}> <Card.Header> Ajouter

---

Voiture</Card.Header><Card.Body>    </Card.Body>    </Card>);   }   }.

Sauvegardez et observez le changement.

77. Nous allons après utiliser le composant [Form](#) depuis react bootstrap. Copiez le code Form depuis le site et collez le dans 'Voiture.js' comme suit : <Form><Form.Group><From.Label>Title</From.Label>   <From.Control type="text" className={"bg-dark text-white"} placeholder= "Entrez Marque Voiture"/></From.Group> <Button size="sm" variant="success" type = "submit"> Submit</Button></Form> et testez.
78. La structure classique du composant Form est comme suit : Form -> Form.Group -> Form.Label -> Form.Control -> Form.Text en plus du Button. Dans un [Form](#) on peut avoir plusieurs Form.Group et un Form.Group contient un Form.Label pour afficher l'étiquette du groupe, un Form.Control pour pouvoir contrôler le type des données : password, email, etc, et un Form.Text pour écrire un texte quelconque.
79. Now, nous changeons dans 'Voiture.js' comme suit :<Form onSubmit={this.submitVoiture} id="VoitureFormId">    <Card.Body> <Form.Row> <Form.Group as={Col}> <Form.Label> Marque </Form.Label> <Form.Control name="marque" type="text" className={"bg-dark text-white"} placeholder= "Entrez Marque Voiture"/> </Form.Group> <Form.Group as={Col}>   <Form.Label> Modele </Form.Label> <Form.Control name="modele" type="text" className={"bg-dark text-white"} placeholder= "Entrez Modele Voiture"/></Form.Group> </Form.Row>   </Card.Body>   <Card.Footer style={{"textAlign":"right"}}> <Button size="sm" variant="success" type="submit"> Submit </Button> </Card.Footer> </Form>
80. Ajoutons un [constructeur](#) à 'App.js' comme suit : ...extends Component { constructor(props) { super(props); this.state={marque:"", modele:"", couleur:"", annee:"", prix:""}; } et avant render() nous ajoutons la fonction submitVoiture comme suit : submitVoiture(event) { alert(this.state.marque); event.preventDefault(); } render()...et testez après.
81. Ajoutez aussi des propriétés à la balise Form.Control : <Form.Control required type=.. name=.. value = {this.state.marque} onChange = {this.voitureChange} , puis on crée la fonction comme suite :

- ```
voitureChange(event) { this.setState ({ [event.target.name]:event.target.value }) ; }
```
82. Au niveau de 'Voiture.js' ajouter le paramètre 'controlId' dans chaque <From.Group> comme suit : <From.Group as={Col} controlId="formGridMarque"> et dupliquer le même paramètre sur les autres champs.
83. Au niveau du constructor ajouter ces deux instructions à la fin de ce bloc :
- ```
this.voitureChange = this.voitureChange.bind(this);
this.submitVoiture = this.submitVoiture.bind(this);
```
84. Ajoutez ces deux paramètres au niveau de chaque balise <Form.Control> et faire les adaptations nécessaires : value={this.state.marque} onChange={this.voitureChange}
85. Cherchons le composant 'font awesome react github' sur notre moteur de recherche pour pouvoir utiliser des icônes pertinentes et ouvrons le premier lien qui s'affiche.
86. Allez sur la rubrique **installation** et exécutez ces trois commandes une après une après avoir arrêté le serveur:

```
$ npm i --save @fortawesome/fontawesome-svg-core
$ npm i --save @fortawesome/free-solid-svg-icons
$ npm i --save @fortawesome/react-fontawesome
```

87. **npm start**, pour relancer le serveur.
88. Pour utiliser ces icônes nous allons copier le script <FontAwesomeIcon icon="coffee" /> que vous trouverez que le même site et le placer au

niveau de 'VoitureListe.js'. Le fichier se présente comme suit :

```
import React from 'react';
import { Card, Table } from 'react-bootstrap';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import { faList } from '@fortawesome/free-solid-svg-icons';
export default class VoitureListe extends React.Component {
 render() {
 return (
 <Card className="border border-dark bg-dark text-white">
 <Card.Header><FontAwesomeIcon icon={faList} /> Liste des Voitures</Card.Header>
 <Card.Body>
 <Table bordered hover striped variant="dark"><thead>
 <tr>
 <th>Marque</th>
 <th>Modele </th>
```

89. On aura le rendu suivant :



90. Observez le bouton 'Hamburger' devant 'Liste des Voitures'.

91. On refais la même chose dans 'Voiture.js' pour ajouter deux icônes de sauvegarde avec le script suivant :

```
render() {
 return [
 <Card className="border border-dark bg-dark text-white">
 <Card.Header><FontAwesomeIcon icon={faPlusSquare} /> Ajouter une Voiture</Card.Header>
 <Form onSubmit={this.submitVoiture} id="VoitureFormId">
```

```
</Card.Body>
<Card.Footer style={{"textAlign":"right"}}>
 <Button size="sm" variant="success" type="submit"> <FontAwesomeIcon icon={faSave} /> Submit </Button>
</Card.Footer>
</Form>
</Card>
```

92. Le rendu ressemble à ça avec les deux icônes ajoutées :

The screenshot shows a user interface for adding a car. At the top, there are two buttons: "Ajouter une Voiture" (Add Car) and "Liste des Voitures" (List of Cars). Below this, a modal window titled "Ajouter une Voiture" (Add Car) is displayed. The modal has six input fields labeled "Marque" (Brand), "Modele" (Model), "Couleur" (Color), "Immatricule" (Registration Plate), "Prix" (Price), and "Annee" (Year). Each field has a placeholder text: "Entrez Marque V", "Entrez Modele V", "Entrez Couleur V", "Entrez Immatriculu", "Entrez Prix Voitu", and "Entrez Annee Vo". A green "Submit" button is located at the bottom right of the modal.

## Librairie Axios

1. Axios est une librairie qui nous permet d'établir des requêtes Http à des ressources externes. Elle supporte plusieurs méthodes telles que HEAD, GET, POST, PUT, PATCH, DELETE et OPTIONS.
2. Pour installer axios faites: **npm install axios**
3. Lancez le serveur : **npm start**
4. On rappelle les trois méthodes de cycle de vie constructor() : est appelé avant d'être monté, render() : est la seule méthode qui doit absolument exister dans un composant classe, componentDidMount() : est invoquée immédiatement après que le composant est monté, componentDidUpdate() : est invoquée chaque fois qu'il y a une modification.

- 
- 5. Au niveau de 'VoitureListe.js' modifiez ainsi :

```
export default class VoitureListe extends Component {
 constructor(props) {
 super(props);
 this.state = {
 voitures : []
 };
 }
 componentDidMount(){
 axios.get("http://localhost:8080/voitures")
 .then(response => console.log(response.data));
 }
}
```

- 6. En cas de CROSS Origin error, veuillez appliquer le changement suivant au niveau du projet Spring dans le contrôleur de Voiture comme ici

```
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.RestController;

 @RestController
 @CrossOrigin(origins="http://localhost:3000")
 public class VoitureController {
 @Autowired
```

- 7. Si tout marche bien, vous allez pouvoir afficher les trois voitures initiales au niveau de la console du navigateur.

8. It's Ok, on va maintenant appliquer les changements suivants pour charger les données :

```
componentDidMount(){
 axios.get("http://localhost:8080/voitures")
 .then(response => response.data)
 .then((data) => {
 this.setState({voitures: data});

 });

}
```

9. Aussi un changement en bas au niveau de la balise <tbody> comme ici :

```
<tbody>
 <tr align="center">
 <td colSpan="6">
 {this.state.voitures.length} Voitures disponibles</td>
 </tr>
</tbody>
</Table>
</Card.Body>
```

10. Et vous aurez alors le rendu suivant :



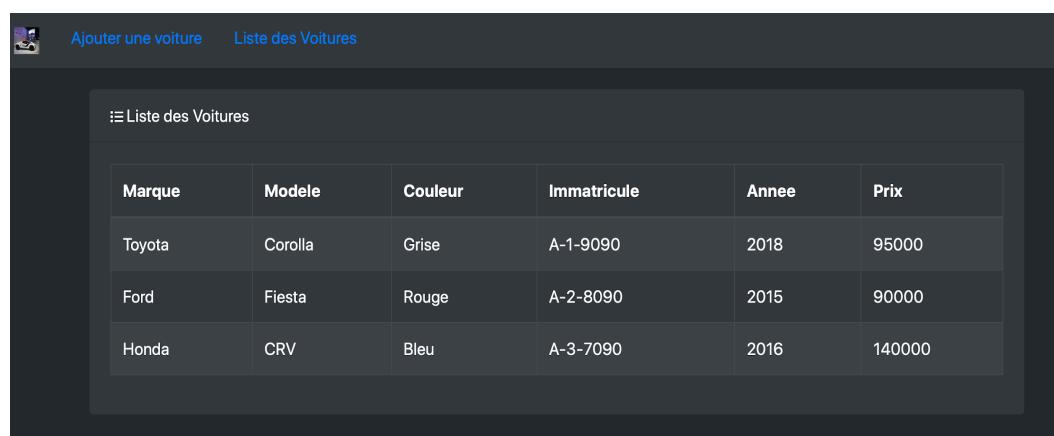
11. Alors on va vérifier si la liste est vide des voitures sinon on affiche la liste en appliquant le changement suivant au niveau de la balise <tbody> de

'VoitureListe.js' :

```

<tbody>
{
 this.state.voitures.length ===0 ?
 <tr align="center">
 <td colSpan="6">
 Voitures disponibles.</td>
 </tr> :
 this.state.voitures.map((voiture) => (
 <tr key={voiture.id}>
 <td>{voiture.marque}</td>
 <td>{voiture.modele}</td>
 <td>{voiture.couleur}</td>
 <td>{voiture.immatricule}</td>
 <td>{voiture.annee}</td>
 <td>{voiture.prix}</td>
 </tr>
)));
}
</tbody>
```

12. Vous aurez le rendu suivant :



The screenshot shows a web application interface. At the top, there is a navigation bar with three items: a camera icon, 'Ajouter une voiture' (Add a car), and 'Liste des Voitures' (List of cars). Below the navigation bar, the title 'Liste des Voitures' is displayed. A table is shown with the following data:

Marque	Modele	Couleur	Immatricule	Annee	Prix
Toyota	Corolla	Grise	A-1-9090	2018	95000
Ford	Fiesta	Rouge	A-2-8090	2015	90000
Honda	CRV	Bleu	A-3-7090	2016	140000

13. Nous allons à ce niveau ajouter deux boutons : édition et suppression comme ci après :

```
<td>{voiture.années}</td>
<td>{voiture.prix}</td>
<td>
<ButtonGroup>
<Button size="sm" variant="outline-primary"><FontAwesomeIcon icon={faEdit} /> </Button>{' '}
<Button size="sm" variant="outline-danger"><FontAwesomeIcon icon={faTrash} /> </Button>
</ButtonGroup>
</td>
</tr>
))
```

14. Et vous aurez le rendu suivant :



The screenshot shows a dark-themed user interface for managing a list of vehicles. At the top, there are two buttons: "Ajouter une voiture" (Add vehicle) and "Liste des Voitures" (List of vehicles). Below this is a header bar with the title "Liste des Voitures". The main area contains a table with the following data:

Marque	Modele	Couleur	Immatricule	Année	Prix	
Toyota	Corolla	Grise	A-1-9090	2018	95000	 
Ford	Fiesta	Rouge	A-2-8090	2015	90000	 
Honda	CRV	Bleu	A-3-7090	2016	140000	 

15. Nous allons apporter des modifications à 'Voiture.js' notamment ajouter un bouton 'Reset' plus d'autres modifications légères comme suit :

```
export default class Voiture extends React.Component {
 constructor(props) {
 super(props);
 this.state=this.initialState;
 this.voitureChange = this.voitureChange.bind(this);
 this.submitVoiture = this.submitVoiture.bind(this);
 }
 initialState = {
 marque:'',
 modele:'',
 couleur:'',
 immatricule:'',
 prix:'',
 annee:''
 }
 resetVoiture = () => {
 this.setState(() => this.initialState
);
}
 submitVoiture =event => {
 event.preventDefault();
 const voiture={
 marque:this.state.marque,
 modele:this.state.modele,
 couleur:this.state.couleur,
 immatricule:this.state.immatricule,
 annee:this.state.annee,
 prix:this.state.prix
 };
}
 voitureChange = event => {
```

16. Ensuite au niveau de 'voitureChange' :

```
voitureChange = event => {
 this.setState({
 [event.target.name]:event.target.value
 })
}
```

17. Au niveau de render() nous allons définir une nouvelle constante en plus de l'évènement 'onReset' au niveau de la balise <Form> comme ci après :

```
return (
 <Card className={"border border-dark bg-dark text-white"}>
 <Card.Header><FontAwesomeIcon icon={faPlusSquare}/> Ajouter une Voiture</Card.Header>
 <Form onReset={this.resetVoiture} onSubmit={this.submitVoiture} id="VoitureFormId">
 <Card.Body>
```

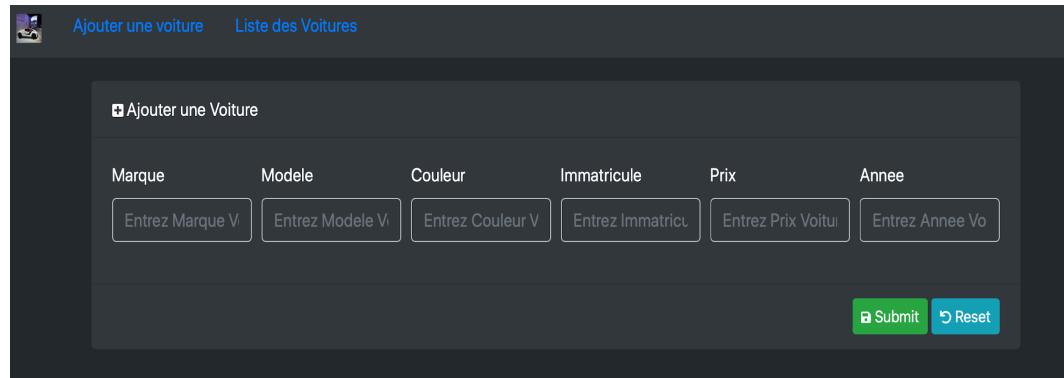
18. Ensuite appliquer les changements suivants au niveau de chaque balise <Form.Control> respectivement pour chaque champ pour les attributs 'value', 'autoComplete' et 'onChange' comme ici :

```
<Form.Group as={Col} controlId="formGridMarque">
 <Form.Label> Marque </Form.Label>
 <Form.Control required name="marque" type="text" value={marque} autoComplete="off"
 onChange = {this.voitureChange} className={"bg-dark text-white"} placeholder= "Entrez Marque Voiture" />
</Form.Group>
<Form.Group as={Col} controlId="formGridModele">
 <Form.Label> Modele </Form.Label>
 <Form.Control required value={this.state.modele} name="modele" autoComplete="off"
 type="text" value={modele} onChange = {this.voitureChange} className={"bg-dark text-white"} placeholder=>
</Form.Group>
<Form.Group as={Col} controlId="formGridCouleur">
 <Form.Label> Couleur </Form.Label>
 <Form.Control required value={this.state.couleur} name="couleur" autoComplete="off"
 type="text" value={couleur} onChange = {this.voitureChange} className={"bg-dark text-white"} placeholder=>
</Form.Group>
<Form.Group as={Col} controlId="formGridImmatricule">
 <Form.Label> Immatricule </Form.Label>
 <Form.Control required value={this.state.immatricule} name="immatricule" autoComplete="off"
 type="text" value={immatricule} onChange = {this.voitureChange} className={"bg-dark text-white"} placeholder=>
</Form.Group>
<Form.Group as={Col} controlId="formGridPrix">
```

19. En bas du formulaire, nous allons ajouter le bouton ‘Reset’ comme ici :

```
</Card.Body>
<Card.Footer style={{'textAlign':'right'}}>
 <Button size="sm" variant="success" type="submit"> <FontAwesomeIcon icon={faSave}/> Submit </Button>{' '}
 <Button size="sm" variant="info" type="reset"> <FontAwesomeIcon icon={faUndo}/> Reset </Button>
</Card.Footer>
</Form>
```

20. L’exécution va donner le rendu suivant :



21. Maintenant nous essayons de procéder à l’insertion des données au niveau de la base. Pour ce faire, nous importons le composant ‘axios’, et nous modifions comme suit :

```
axios.post("http://localhost:8080/voitures", voiture)
.then(response => {
 if(response.data != null){
 this.setState(this.initialState);
 alert("Voiture enregistrée avec succès");
 }
})
```

22. On essaie maintenant d'enregistrer une nouvelle voiture :

Marque	Modele	Couleur	Immatricule	Prix	Année
Fiat	Uno	Bleu	A2-234	12345	2010

23. Vous appuyez sur le bouton 'Submit' et après en appuyant sur le lien 'Liste des Voitures' vous allez pouvoir afficher la nouvelle liste des voitures :

Marque	Modele	Couleur	Immatricule	Année	Prix	
Toyota	Corolla	Grise	A-1-9090	2018	95000	
Ford	Fiesta	Rouge	A-2-8090	2015	90000	
Honda	CRV	Bleu	A-3-7090	2016	140000	
Fiat	Uno	Bleu	A2-234	2010	12345	

24. Cherchons le composant 'Toast' au niveau du site react bootstrap. Nous allons utiliser ce composant pour afficher un bon rendu lorsqu'une voiture est enregistrée avec succès comme par ici :

Marque	Modele	Couleur	Immatricule	Prix	Année
Entrez Marque V	Entrez Modele V	Entrez Couleur V	Entrez Immatricul	Entrez Prix Voitu	Entrez Année Vo

25. Pour ce faire nous allons créer un nouveau composant myToast.js et apporter une modification légère à la page 'Voiture.js'.

26. Le code de 'myToast.js' se présente comme suit :

```
import React, {Component} from 'react';
import {Toast} from 'react-bootstrap';
export default class MyToast extends Component {
 render() {
 const toastCss = {
 position: 'fixed',
 top: '20px',
 right: '20px',
 zIndex: '1',
 boxShadow: '0 4px 8px 0 rgba(0, 0, 0, 0.2), 0 6px 20px 0 rgba(0, 0, 0, 0.19)'
 };
 return (
 <div style={this.props.children.show ? toastCss : null}>
 <Toast className={"border border-success bg-success text-white"} show={this.props.children.show}>
 <Toast.Header className={"bg-success text-white"} closeButton={false}>
 <strong className="mr-auto">Success
 </Toast.Header>
 <Toast.Body>
 {this.props.children.message}
 </Toast.Body>
 </Toast>
 </div>
);
 }
}
```

27. Nous procédons maintenant à la suppression des voitures. Commençons par ajouter l'événement onClick au bouton de suppression dans 'VoitureListe.js' comme ici :

```
<Button size="sm" variant="outline-danger"
 onClick={this.deleteVoiture.bind(this,voiture.id)}>
```

28. Ensuite nous définissons la fonction deleteVoiture juste avant render :

```
....

deleteVoiture = (voitureId) => {
 alert(voitureId);
};

render() {


```

29. Exécutez, appuyez sur le lien 'Liste des Voitures', ensuite appuyez sur le bouton de suppression relatif à une voiture donnée, la fenêtre alert doit

normalement s'afficher en retournant l'Id de la voiture.

Liste des Voitures					
Marque	Modele	Couleur	Immatricule	Année	Prix
Toyota	Corolla	Grise	3	2015	95000
Ford	Fiesta	Rouge		Fermer	90000
Honda	CRV	Bleu	A-3-7090	2016	140000

30. Nous allons après remplacer 'alert' par le script suivant :

```
deleteVoiture = (voitureId) => {
 axios.delete("http://localhost:8080/voitures"+voitureId)
 .then(reponse => {
 if(response.data != null){
 alert("Voiture supprimée avec succès.");
 }
 })
};
```

31. Vous procédez à la suppression, le message de confirmation s'affiche mais la voiture est toujours affichée. Alors pour rafraîchir la page nous ajoutons le script suivant juste après 'alert' qui va nous permettre de ne garder que les voitures dont l'Id est différent de du paramètre **voitureId**:

```
if(response.data != null){
 alert("Voiture supprimée avec succès.");
 this.setState({
 voitures: this.state.voitures.filter(voiture => voiture.id !== voitureId)
 })
}
```

32. Maintenant nous allons afficher un message de type Toast. Nous prenons donc le script **this.setState({ "show":true}); setTimeout(() => this.setState({ "show":false}), 3000);** au niveau de 'Voiture.js' et le coller au niveau de 'VoitureListe.js' et après supprimer la fonction **alert**. Copiez aussi le code **else { this.setState({ "show":false}); }** et le placer dans 'VoitureListe.js'.

33. Copiez aussi le code suivant et importer MyToast dans la page 'Voiture.js':

```
<div style={{display:this.state.show ? "block" : "none"}}>
```

```

 <MyToast children = {{show:this.state.show, message:"Voiture supprimée avec
succès."}}/>
</div>

```

34. Le script se présente ainsi :

```

render() {
 return (
 <div>
 <div style={{display:this.state.show ? "block" : "none"}}>
 <MyToast children = {{show:this.state.show, message:"Voiture supprimée avec succès."}}/>
 </div>

 <Card className={"border border-dark bg-dark text-white"}>

```

35. La balise <div> placée juste après 'return' vous allez la fermer en bas :

```

 }
 </tbody>
</Table>
</Card.Body>

</Card>
</div>

```

36. On va après personnaliser ces messages Toast de telle sorte à avoir une boîte verte pour l'ajout et une boîte rouge pour la suppression. Pour cela, nous allons appliquer trois changements dans les trois pages Voiture.js, VoitureListe.js et myToast.js

37. Dans Voiture.js :

```

render() {
 const {marque, modèle, couleur, immatricule, prix, année} = this.state;
 return (
 <div>
 <div style={{display:this.state.show ? "block" : "none"}}>
 <MyToast children = {{show:this.state.show, message:"Voiture enregistrée avec succès.", type:"success" }}/>
 </div>

```

38. Dans VoitureListe.js :

```

render() {
 return (
 <div>
 <div style={{display:this.state.show ? "block" : "none"}}>
 <MyToast children = {{show:this.state.show, message:"Voiture supprimée avec succès.", type:"danger" }}/>
 </div>

 <Card className={"border border-dark bg-dark text-white"}>

```

39. Dans myToast.js :

```
return (
 <div style={this.props.children.show ? toastCss : null}>
 <Toast className={`border text-white ${this.props.children.type === "success" ?
 "border-success bg-success" : "border-danger bg-danger"}`}
 show={this.props.children.show}>
 <Toast.Header className={`text-white ${this.props.children.type === "success" ?
 "bg-success" : "bg-danger"}`} closeButton={false}>
 <strong className="mr-auto">Success
 </Toast.Header>
 <Toast.Body>
 {this.props.children.message}
 </Toast.Body>
 </Toast>
```

40. Vous utilisez le caractère : ` plutôt que le caractère : ' .

