Le modèle M.V.C. de Spring 2/2

1 Utiliser des données en session

Il existe trois solutions pour travailler facilement avec des données stockées en session.

1.1 Gérer facilement les données en session

Voila un exemple de contrôleur qui travaille sur un compteur stocké en session et récupéré via les paramètres des méthodes :

Étape 1 : Le POJO :

```
package mybootapp.web;

import lombok.Data;

@Data
public class CounterBean {
    private int value = 0;
}
```

Étape 2 : La vue :

Fichier WEB-INF/jsp/counter.jsp

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<h1>Counter is <c:out value="${counter.value}" default="None" /></h1>

<c:url var="init" value="/counter/init" />
<c:url var="inc" value="/counter/inc" />
<c:url var="show" value="/counter" />

<p>
    <a class="btn btn-primary mx-2" href="${show}">Show</a>
    <a class="btn btn-primary mx-2" href="${init}">Init</a>
    <a class="btn btn-primary mx-2" href="${inc}">Increment</a>
</p>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Étape 3 : Le contrôleur :

```java
package mybootapp.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttribute;
import org.springframework.web.servlet.ModelAndView;

import jakarta.servlet.http.HttpSession;

@Controller
@RequestMapping("/counter")
public class CounterController {

    @GetMapping("")
    public ModelAndView showCounter(//
            @SessionAttribute(required = false, name = "counter") CounterBean counter) {
        return new ModelAndView("counter", "counter", counter);
    }

    @GetMapping("/init")
    public ModelAndView init(HttpSession session) {
        var counter = new CounterBean();
        session.setAttribute("counter", counter);
        return new ModelAndView("counter", "counter", counter);
    }

    @GetMapping("/inc")
    public ModelAndView incCounter(@SessionAttribute("counter") CounterBean counter) {
        counter.setValue(counter.getValue() + 1);
        return new ModelAndView("counter", "counter", counter);
    }

}
```

1.2 Utiliser la portée dans Spring

Il est facile de récupérer des données placées en session, mais Spring nous offre le moyen d'injecter directement dans nos contrôleurs des données de portée session.

Étape 1 : définissez un nouveau bean pour représenter l'utilisateur courant :

```java
package mybootapp.web;

import org.springframework.stereotype.Component;
import org.springframework.web.context.annotation.SessionScope;

import lombok.Data;

@Component
@SessionScope
@Data
public class User {

    private String name;

}
```

Note : L'annotation `Component` indique que c'est un composant géré par Spring. L'annotation `SessionScope` donne la portée des instances (une par session). Les portées `RequestScope` et `ApplicationScope` sont également disponibles. Ce n'est pas directement une instance qui va être injectée, mais un proxy qui va sélectionner la bonne instance (dans la bonne session) en fonction du contexte.

Étape 2 : La vue :

Fichier `WEB-INF/jsp/user.jsp`

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<c:url var="login" value="/user/login" />
<c:url var="logout" value="/user/logout" />
<c:url var="show" value="/user" />

<h1>User <c:out value="${user.name}" default="no name" /></h1>

<p>
    <a class="btn btn-primary mx-2" href="${show}">Show</a>
    <a class="btn btn-primary mx-2" href="${login}">Login</a>
    <a class="btn btn-primary mx-2" href="${logout}">Logout</a>
</p>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Étape 3 : définissez un contrôleur qui utilise l'injection du bean `User` :

```java
package mybootapp.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/user")
public class UserController {

    @Autowired
    User user;

    @ModelAttribute("user")
    public User newUser() {
        return user;
    }

    @GetMapping("")
    public String show() {
        return "user";
    }

    @GetMapping("/login")
    public String login() {
        user.setName("It's␣me");
        return "user";
    }

    @GetMapping("/logout")
    public String logout() {
        user.setName("Anonymous");
        return "user";
    }
}
```

Note : Le contrôleur (qui est un singleton exécuté par plusieurs `threads`) utilise le `proxy` pour sélectionner **automatiquement** l'instance du bean `User` qui correspond à la requête courante et à la session courante. La liaison se fait par le `thread`. C'est le même `thread` qui traite toute la requête (`Dispatcher`, contrôleur, vue). Le `thread` courant est donc utilisé comme une sorte de variable globale qui permet de faire des liaisons implicites.

Travail à faire : Testez le bon fonctionnement de cet exemple.

1.3 Placer des données en session

Une deuxième solution consiste à indiquer, dans le contrôleur, les instances du modèle que Spring devra placer dans la session. Reprenons le même exemple mais avec un utilisateur simple (pas annoté) :

```java
package mybootapp.web;

import lombok.Data;

@Data
public class SimpleUser {

    private String name;

}
```

Nous pouvons maintenant définir un nouveau contrôleur :

```java
package mybootapp.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

@Controller()
@RequestMapping("/simple-user")
@SessionAttributes("simpleUser")
public class SimpleUserController {


    @ModelAttribute("simpleUser")
    public SimpleUser newUser() {
        var user = new SimpleUser();
        return user;
    }

    @GetMapping("/show")
    public String show(@ModelAttribute("simpleUser") SimpleUser user) {
        return "simple-user";
    }

    @GetMapping("/login")
    public String login(//
            @ModelAttribute("simpleUser") SimpleUser user, //
            RedirectAttributes attributes) {
        user.setName("It's␣me");
        attributes.addFlashAttribute("message", "Bienvenue␣!");
        return "redirect:show";
    }

    @GetMapping("/logout")
    public String logout(//
            @ModelAttribute("simpleUser") SimpleUser user, //
            RedirectAttributes attributes) {
        user.setName("Anonymous");
        attributes.addFlashAttribute("message", "Au␣revoir.");
        return "redirect:show";
    }
}
```

Note : L'annotation `@SessionAttributes` permet d'indiquer quelles sont les instances du modèle qui doivent être placées en session. L'instance en question (`simpleUser`) est produite par la méthode `newUser` . Les méthodes traitants les requêtes peuvent récupérer cette instance pour la modifier.

Note : Je profite de cet exemple pour introduire les données **flash** qui sont destinées à être utilisées dans la requête suivante. C'est précisément le cas car, contrairement à la première version, les actions de `/login` et `/logout` renvoient une redirection vers `/show` .

Il ne reste plus qu'à définir la vue :

Fichier `WEB-INF/jsp/simple-user.jsp`

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<c:url var="login" value="/simple-user/login" />
<c:url var="logout" value="/simple-user/logout" />
<c:url var="show" value="/simple-user/show" />

<h1>Simple User</h1>

<c:if test="${message != null}">
    <div class="alert alert-success" role="alert">
        <c:out value="${message}" />
    </div>
</c:if>

<p>name: <c:out value="${simpleUser.name}" default="no name" /></p>

<p>
    <a class="btn btn-primary mx-2" href="${show}">Show</a>
    <a class="btn btn-primary mx-2" href="${login}">Login</a>
    <a class="btn btn-primary mx-2" href="${logout}">Logout</a>
</p>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Travail à faire : Testez le bon fonctionnement de cet exemple.

2 Des fonctionnalités en plus

2.1 Tester vos contrôleurs

Voila un exemple simple de test unitaire basé `MockMvc` qui permet de vérifier le bon fonctionnement des contrôleurs :

- Créez dans le répertoire `test` le package `mybootapp.web` .
- Créez la classe de test unitaire ci-dessous :

```java
package mybootapp.web;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.model;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.web.servlet.MockMvc;

import mybootapp.web.Starter;

@SpringBootTest
@ContextConfiguration(classes = Starter.class)
@AutoConfigureMockMvc
public class TestWebApp {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testCourseList() throws Exception {
        mvc.perform(get("/course/list"))//
                // afficher
                .andDo(print())//
                // vérifier le statut
                .andExpect(status().isOk())//
                // vérifier le nom de la vue
                .andExpect(view().name("course"))//
                // vérifier le modèle
                .andExpect(model().attributeExists("courses"));
    }

}
```

Travail à faire : Tester quelques possibilités des instances renvoyées par les méthodes statiques `status()`, `view()` et `model()`.

2.2 Utiliser des intercepteurs

Vous pouvez très facilement installer des classes d'interception afin d'ajouter des opérations avant et après le traitement des requêtes. Définissez la classe suivante (elle vérifie l'adresse du client) :

```java
package mybootapp.web;

import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class LoggerInterceptor implements HandlerInterceptor {

    private static Log log = LogFactory.getLog(LoggerInterceptor.class);

    @Override
    public boolean preHandle(HttpServletRequest request, //
            HttpServletResponse response, Object handler) throws Exception {
        var client = request.getRemoteAddr();
        log.info("Inside␣pre␣handle␣from␣" + client);
        switch (client) {
        case "127.0.0.1":
        case "0:0:0:0:0:0:0:1":
            return true;
        }
        response.getWriter().printf("Only␣127.0.0.1");
        return false;
    }

    @Override
    public void postHandle(HttpServletRequest request, //
            HttpServletResponse response, //
            Object handler, //
            ModelAndView modelAndView) throws Exception {
        log.info("Inside␣post␣handle");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, //
            HttpServletResponse response, Object handler, //
            Exception exception) throws Exception {
        log.info("Inside␣after␣completion");
    }
}
```

Pour installer cet intercepteur (il peut y avoir plusieurs), ajoutez la méthode ci-dessous à votre classe `Starter` :

```java
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new LoggerInterceptor());
}
```

Travail à faire : Vérifiez que l'application n'est plus accessible à partir de l'adresse publique.

2.3 Gestion des erreurs

La récupération des erreurs est simplement réalisée par l'ajout d'un contrôleur spécifique dans lequel nous sommes capable de traiter plusieurs causes d'exception :

```java
package mybootapp.web;

import org.springframework.boot.web.servlet.error.ErrorController;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
public class ErrorControler implements ErrorController {

    @ResponseBody
    @ExceptionHandler({NullPointerException.class})
    public String handleNullPointerException(Exception e) {
        System.err.println("--␣NullPointerException:");
        e.printStackTrace(System.err);
        return "Null␣Pointer␣Error";
    }

    @ResponseBody
    @ExceptionHandler
    public String handleOtherException(Exception e) {
        System.err.println("--␣Other␣Exception:");
        e.printStackTrace(System.err);
        return "Other␣Error";
    }

}
```

Note : Je ne rentre pas dans plus de détails, vous trouverez plus de détails sur cette page [a].

a. https://www.baeldung.com/spring-boot-custom-error-page

3 Introduction à Spring security

3.1 Mise en place

Nous pouvons maintenant ajouter Spring Security [1] : une couche des gestion de la sécurité. Pour ce faire, suivez les étapes ci-dessous :

- **Étape n°1 - Les dépendances** : Ajoutez au fichier `pom.xml` les dépendances de Spring Security.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
</dependency>
```

1. https://spring.io/guides/topicals/spring-security-architecture

- **Étape n°2 - Définir les utilisateurs** : Nous allons créer une nouvelle entité afin de représenter les utilisateurs capables de s'authentifier. Ajoutez, dans le package `mybootapp.model`, la classe ci-dessous, ainsi que le dépôt **spring-data** :

Représentation d'un utilisateur et ses rôles

```
package mybootapp.model;

import java.util.Collection;

import jakarta.persistence.Basic;
import jakarta.persistence.ElementCollection;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class XUser {

    @Id
    String userName;

    @Basic
    String password;

    @ElementCollection(fetch = FetchType.EAGER)
    Collection<String> authorities;

}
```

La `Repository` **Spring-data**

```
package mybootapp.repo;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import mybootapp.model.XUser;

@Repository
@Transactional
public interface XUserRepository extends JpaRepository<XUser, String> {

}
```

- **Étape n°3 - Définir l'utilisateur authentifié** : Commencez par créer le package `mybootapp.web.security`. Nous allons maintenant ajouter une classe qui permet de construire un utilisateur authentifié (un `UserDetails` de Spring Security). Vous remarquerez que toutes les possibilités **ne sont pas exploitées** (compte verrouillé, expiré, désactivé). Il faudrait, pour cela, enrichir notre classe `XUser`.

```java
package mybootapp.web.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import mybootapp.model.XUser;
import mybootapp.repo.XUserRepository;

@Service
public class MyUserDetailsService implements UserDetailsService {

    @Autowired
    private XUserRepository userRepository;

    /*
     * À partir d'un XUser, chargé depuis la base, nous allons construire une
     * instance de la classe User offerte par Spring Security (même si nous
     * n'utilisons pas toutes les possibilités). La classe User est une
     * implémentation de l'interface UserDetails qui représente un utilisateur
     * authentifié dans Spring Security.
     *
     */
    @Override
    public UserDetails loadUserByUsername(String username) {
        XUser xuser = userRepository.findById(username)//
                .orElseThrow(() -> new UsernameNotFoundException(username));
        return User.withUsername(xuser.getUserName())// Compte userName
                .password(xuser.getPassword())// Mot de passe
                .authorities(xuser.getAuthorities().toArray(new String[0]))// Autorisations
                .disabled(false)// Compte toujours actif
                .accountExpired(false)// Compte jamais expiré
                .accountLocked(false)// Compte jamais verrouillé
                .credentialsExpired(false)// mot de passe jamais expiré
                .build();
    }
}
```

- **Étape n°4 - Configurer Spring Security** : Ajoutez ensuite une classe de configuration :

```java
package mybootapp.web.security;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityCustomizer;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

import jakarta.annotation.PostConstruct;
import jakarta.servlet.DispatcherType;
import mybootapp.model.XUser;
import mybootapp.repo.XUserRepository;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
public class SpringSecurity {

    @Autowired
    XUserRepository userRepo;

    @Bean
    WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> {
            web.ignoring().requestMatchers("/webjars/**");
        };
    }

    @Bean
    SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        String[] anonymousRequests = { "/", //
                "/webjars/**", //
                "/hello**", //
                "/product/**", //
                "/course/**", //
                "/user", "/user/**", //
                "/counter", "/counter/**", //
                "/simple-user/**", //
                "/home", "/login", "/calculator/**" //
        };
        String[] adminRequests = { //
                "/NOT-simple-user/**" };

        http.authorizeHttpRequests(config -> {//
            config.dispatcherTypeMatchers(DispatcherType.FORWARD).permitAll();
            // Pour tous
            config.requestMatchers(anonymousRequests).permitAll();//
            // Pour les admins
            config.requestMatchers(adminRequests).hasAnyAuthority("ADMIN");//
            // Pour les autres
            config.anyRequest().authenticated();
        });
        // Nous autorisons un formulaire de login
        http.formLogin(config -> {
            config.permitAll();
        });
        // Nous autorisons un formulaire de logout
```

Note : Cette classe va définir les règles de sécurité et la base de l'authentification.

Travail à faire :

- Redémarrez votre application et vérifiez que les règles sont bien respectées. Réalisez plusieurs phases de login/logout.
- Limitez l'accès à `/simple-user` aux administrateurs : faites **glisser** l'URL `/simple-user/**` de la catégorie `anonymousRequests` à la catégorie `adminRequests`.
- Vous pouvez écrire un test unitaire pour valider cette limitation et utiliser l'annotation ci-dessous (sur la méthode de test) pour vous mettre dans le bon contexte :

```
@WithMockUser(username = "user1",  authorities = { "ADMIN" })
```

3.2 Les tag de Spring Security

Travail à faire : Utilisez cette documentation [a] pour faire varier le contenu des pages JSP en fonction de l'authentification : par exemple, faites apparaître sur certaines pages un lien vers `/logout` seulement si l'utilisateur est connecté.

a. https://www.baeldung.com/spring-security-taglibs

3.3 Protection CSRF

Travail à faire : Activez Spring Security pour les URL `/product/**` (la même chose que `/simple-user/**`) et vérifiez que tout fonctionne bien. Le formulaire d'édition des produits (géré par une balise Spring) fonctionne car il intègre maintenant un jeton CSRF. Vérifiez la présence de ce jeton dans le code source de la page HTML.

Travail à faire :

- Activez Spring Security pour les URL `/course/**` (la même chose que `/simple-user/**`). À ce stade, votre formulaire de rechercher des UE ne doit plus fonctionner. Le jeton CSRF est absent (nous gérons ce formulaire à la main).
- En utilisant la balise ci-dessous (déjà vu dans la librairie de balises Spring Security), faites en sorte que le formulaire de recherche des UE fonctione à nouveau.

```
<sec:csrfInput />
```

3.4 Interroger l'utilisateur connecté

Travail à faire : Nous créons des utilisateurs par JPA et la couche `UserDetails` est utilisée par Spring Security pour récupérer les informations d'authentification. Vous pouvez même créer une entrée pour récupérer les données de l'utilisateur connecté :

```
package mybootapp.web;

import java.security.Principal;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller()
@RequestMapping("/principal")
public class ShowPrincipal {

    protected final Log logger = LogFactory.getLog(getClass());

    @ResponseBody
    @RequestMapping("")
    public String show(Principal p) {
        logger.info("show user " + p);
        return p.toString();
    }

}
```

3.5 Contrôler les méthodes

Spring Security n'est pas seulement utile pour les contrôleurs Il est également capable de contrôler l'accès aux méthodes d'un service.

Voila un service Spring qui est sécurisé par Spring Security. La méthode `helloAdmin` est réservée aux administrateur et la méthode `helloForUser` n'est accessible qui si le paramètre `userName` correspond à l'utilisateur courant.

```
package mybootapp.web.security;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Service;

@Service
public class SecureService {

    @PreAuthorize("hasAuthority('ADMIN')")
    public String helloAdmin() {
        return "Hello";
    }

    @PreAuthorize("#userName == principal.username")
    public String helloForUser(String userName) {
        return "Hello " + userName;
    }

}
```

Travail à faire : Préparez un contrôleur pour tester cette sécurisation :

```
package mybootapp.web.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller()
@RequestMapping("/secure")
public class SecureController {

    @Autowired
    SecureService ss;

    @ResponseBody
    @RequestMapping("/hello")
    public String hello() {
        return ss.helloAdmin();
    }

    @ResponseBody
    @RequestMapping("/aaa")
    public String helloForUser() {
        return ss.helloForUser("aaa");
    }

}
```

Travail à faire : Pour aller plus loin [a].

a. https://www.baeldung.com/spring-security-method-security

3.6 Définir ses conditions

Nous avons quelquefois besoin d'écrire le code de vérification d'un droit (par exemple pour savoir si un utilisateur a le droit d'agir sur une donnée précise). Nous devons, dans ce cas, définir un service de vérification :

```
package mybootapp.web.security;

import org.springframework.stereotype.Service;

@Service("securityChecker")
public class SecurityChecker {

    public boolean isOk(String userName) {
        return "aaa".equals(userName);
    }

}
```

Note : Ce service est trivial, mais nous pourrions faire des accès BD et des vérifications plus compliquées.

Travail à faire :

- Nous pouvons maintenant définir une nouvelle méthode du service sécurisé qui utilise cette vérification :

```
...

@Service
public class SecureService {

    ...

    @PreAuthorize("@securityChecker.isOk(principal.username)")
    public String helloSecuredByCode() {
        return "helloSecuredByCode␣is␣OK␣";
    }

}
```

- Ajoutez une entrée à votre contrôleur pour tester cette vérification.
- Pour aller plus loin [a].

a. https://docs.spring.io/spring-security/site/docs/5.0.7.RELEASE/reference/html/el-access.html

4 Très légère introduction aux API RESTfull

L'idée est simple :

- Proposer une API WEB pour récupérer et agir sur les données d'un serveur.
- Utiliser les méthodes HTTP (`GET` , `POST` , `DELETE` , etc) pour coder les actions sur les données.
- Utiliser des langages normalisés pour la description des données (XML et surtout Json).
- Offrir ainsi un service complet accessible aux clients (des applications WEB, des téléphones portables, des dispositifs nomades, etc.)

4.1 Mise en place d'un controleur REST

Commencez par ajouter les dépendances pour Jackson (outils pour transformer une instance Java en Json et vice-versa) :

```
...
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <!-- <version>2.9.8</version> NOUVEAU -->
</dependency>
...
```

Créez ensuite une contrôleur REST (une calculatrice à pile) :

```java
package mybootapp.web;

import java.util.Stack;

import jakarta.annotation.PostConstruct;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/calculator")
public class RestCalculator {

    protected final Log logger = LogFactory.getLog(getClass());
    private Stack<Integer> numbers = new Stack<>();

    @PostConstruct
    public void init() {
        numbers.push(100);
        numbers.push(200);
        numbers.push(300);
    }

    @GetMapping("/show")
    public Stack<Integer> show() {
        return numbers;
    }

    @GetMapping("/add")
    @ResponseStatus(HttpStatus.OK)
    public void add() {
        if (numbers.size() < 2)
            throw new IllegalStateException();
        Integer val1 = numbers.pop();
        Integer val2 = numbers.pop();
        numbers.push(val1 + val2);
    }

    @PostMapping(value = "/put", consumes = "application/json")
    @ResponseStatus(HttpStatus.OK)
    public void put(@RequestBody() Integer id) {
        numbers.push(id);
        logger.info(String.format("put␣%d", id));
    }

}
```

Travail à faire : Testez l'API Rest avec des requêtes directes :

```
http://localhost:8081/calculator/show
http://localhost:8081/calculator/add
http://localhost:8081/calculator/show
```

La première donne les trois éléments de la pile. La second remplace les deux éléments de la tête de pile par leur addition. etc.

Travail à faire : Vous pouvez ensuite déposer des données dans la pile en lancant une requete `POST` à l'aide de l'outil en ligne de commande `curl` :

Commandes à taper dans un shell

```
URL="http://localhost:8081/calculator/put"
curl -X POST -H "Content-Type: application/json" --data '222' $URL
curl -X POST -H "Content-Type: application/json" --data '333' $URL
curl http://localhost:8081/calculator/show
```

4.2 Une petite application REST

Nous allons maintenant créer un code javaScript coté client qui va interagir avec cette API REST. Commencez par le fichier javaScript suivant :

Fichier src/main/webapp/functions.js

```
function showStack() {
    var base = ($('<a␣href=".">')[0].href);
    $.ajax({
        type : 'GET',
        url : (base + "calculator/show"),
        data : '200',
        timeout : 3000,
        success : function(data) {
            $('#numbers').hide();
            $('#numbers').html("");
            jQuery.each(data, function(i, val) {
                $("#numbers").append(val + ",␣");
            });
            $('#numbers').show();
        }
    });
}

function show() {
    showStack();
    $('#message').html("");
}

function add() {
    var base = ($('<a␣href=".">')[0].href);
    $.ajax({
        type : 'GET',
        url : (base + "calculator/add"),
        timeout : 3000,
        error : function() {
            $('#message').html('Addition␣impossible');
            showStack();
        },
        success : function(data) {
            $('#message').html('Addition␣r&eacute;alis&eacute;e');
            showStack();
        }
    });
}

function put() {
    var base = ($('<a␣href=".">')[0].href);
    var value = ($('#input').val());
    $.ajax({
        type : 'POST',
        url : (base + "calculator/put"),
        data : value,
        timeout : 3000,
        dataType : "text",
        contentType : "application/json",
        success : function(data) {
            showStack();
            $('#message').html('donn&eacute;e␣ajout&eacute;e.');
            $('#input').val("");
        },
        error : function() {
            showStack();
            $('#message').html('donn&eacute;e␣invalide.');
            $('#input').val("");
        }
    });
}
```

Ces fonctions JavaScript vont utiliser la méthode `ajax` de `JQuery` pour envoyer des requêtes asynchrones vers l'API REST.

Nous pouvons maintenant préparer une page JSP qui va produire une page HTML à destination d'un navigateur. La page chargée va utiliser `JQuery` et proposer une interface minimale pour lister les éléments de la pile, ajouter un nombre et calculer une addition.

Fichier `src/main/webapp/rest-app.jsp`

```
<%@ include file="/WEB-INF/jsp/header.jsp"%>

<c:url var="function" value="/functions.js" />

<script src="${function}"></script>
<h1>Simple stack calculator (rest application)</h1>
<p>
    <button class="mx-2 btn btn-primary" onclick="show();">Les valeurs</button>
    <input class="mx-2" id="input" size="10" />
    <button class="mx-2 btn btn-primary" onclick="put();">Ajouter</button>
    <button class="mx-2 btn btn-primary" onclick="add();">+</button>
    <span class="mx-2" style="color: blue;" id="message"></span>
</p>
<p>La pile : <span id="numbers"></span></p>

<%@ include file="/WEB-INF/jsp/footer.jsp"%>
```

Travail à faire :Prévoir l'opération de soustraction (fonction JavaScript, bouton html et requête `/calculator/sub` sur l'API REST.