1. TP "Spring-Mvc" (jsp, thymeleaf , rest)

1.1. Configuration d'un projet spring-mvc

Créer un nouveau projet appliSpringMvc via spring initializr (https://start.spring.io/).

Java >=17 , maven , SpringBoot 3.x (stable) .
Group : **tp** , Artifact : ***appliSpringMvc*** , packaging jar
Dependencies : Web, JPA et h2 , ...

Extraire dans un répertoire de travail (ex : c:\tp) le contenu du .zip généré et téléchargé.
Charger le projet dans votre IDE favori (IntelliJ , eclipse , VSCode ou autre) .

Ajouter la configuration Suivante dans **application.properties** :

```
server.servlet.context-path=/appliSpringMvc
server.port=8080
```

Ajouter un premier embryon de **index.html** dans le répertoire **src/main/resources/static**

```
<html>
<head>
<title>index</title>
</head>
<body>
    <h1>appliSpringMvc (main index/menu)</h1>
</body>
</html>
```

Ajouter ceci à la **fin** de la méthode ***main***() de ***AppliSpringMvcApplication.java***

```
System.out.println("http://localhost:8080/appliSpringMvc");
```

Effectuer un premier démarrage de l'application pour vérifier son bon démarrage et fonctionnement au sein d'un navigateur internet .

1.2. Récupération de la partie "core / backend" des Tps

Recopier du projet "debutAppliSpringMvc" vers votre projet toutes les parties essentielles manquantes pour la série de Tps :

src/main/java
tp.appliSpringMvc.converter
tp.appliSpringMvc.core (.dao , .entity , .exception , .service , ...)
tp.appliSpringMvc.dto
tp.appliSpringMvc.init
…

src/main/resources
application-dev.properties

src/test/resources
tp.appliSpringMvc.core.service

NB : ceci sera utile au sein d'une série ultérieure de Tps
(points communs pour les besoins en mode "JSP" ou "Tymeleaf ou "Api Rest") .

2. TP "Spring-Mvc" en mode "JSP"

2.1. Configuration et exemple simple en mode "JSP"

Ajouter ce bloc de configuration complémentaire au sein du fichier pom.xml :

```
<!-- selon la phase du Tp , commenter et decommenter une des 2 parties  thymeleaf ou JSP -->
		<!-- partie jsp -->
		<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
			<scope>provided</scope>
		</dependency>
		<dependency>
			<groupId>jakarta.servlet.jsp.jstl</groupId>
			<artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
		</dependency>
		<dependency>
			<groupId>org.glassfish.web</groupId>
			<artifactId>jakarta.servlet.jsp.jstl</artifactId>
		</dependency>
		<!-- partie thymeleaf -->
		<!--
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-thymeleaf</artifactId>
		</dependency>
		<dependency>
			<groupId>nz.net.ultraq.thymeleaf</groupId>
			<artifactId>thymeleaf-layout-dialect</artifactId>
		</dependency>
		-->
```

Créer le nouveau package "**tp.appliSpringMvc.web.controller**"
Au sein de ce package , coder en premier la classe suivante ***BasicController*** :

```
package tp.appliSpringMvc.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/site/basic")
public class BasicController {

	@RequestMapping("helloWorld")
	public String helloWorld(Model model) {
		model.addAttribute("message", "hello world");
		return "displayBasicMessage" ;//.jsp ou .html(thymeleaf)
	}
}
```

Ajouter le fichier **application-jsp.properties** au sein de **src/main/resources** :

```
#will be search in src/main/resources/... (... can be META-INF/resources or /resources ou /static or /public )
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp
```

Ajouter ceci au **début** de la méthode ***main***() de ***AppliSpringMvcApplication.java***

```
//activation ou pas au démarrage du profil spring pour vues en version ".jsp"
System.setProperty("spring.profiles.default", "dev,jsp");
```

Créer les nouveaux sous répertoires suivants au sein de **src/main/resources**
META-INF/resources/views

Ajouter le fichier displayBasicMessage.jsp suivant
au sein de src/main/resources/META-INF/resources/views

```
<html>
<head>
<title>displayBasicMessage (jsp)</title>
</head>
<body>
  <span>${message}</span>
  <hr/>
  <a href="../../index-site.html"> retour menu</a> <br/>
</body>
</html>
```

Ajouter ceci au sein de **index.html**

```
  <hr/>
  <a href="index-site.html"> site (dynamic html , jsp ou thymeleaf)</a> <br/>
```

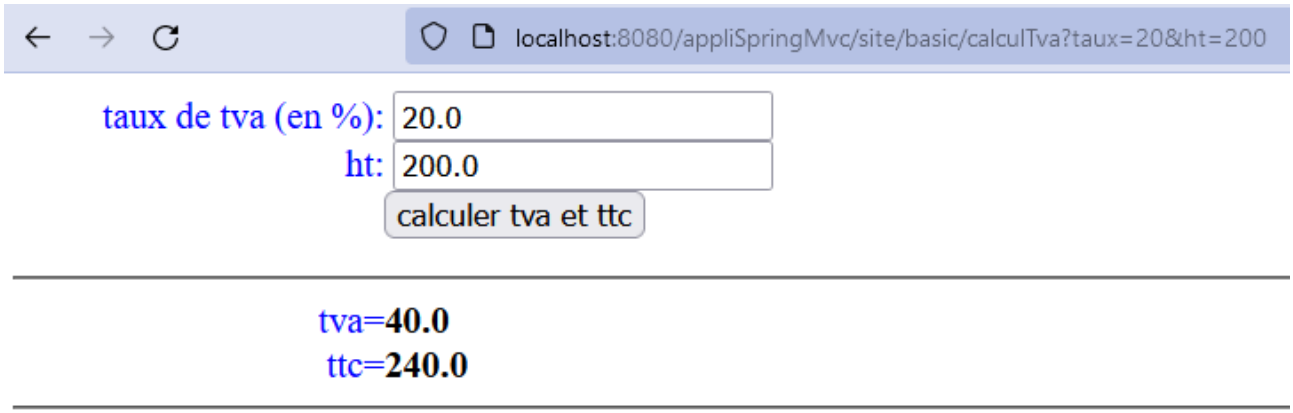
Ajouter cet embryon de ***index-site.html*** à coté de *index.html* :

```
<html>
<head>
<title>index</title>
</head>
<body>
  <h1>index/menu de la partie "site (dynamic html , jsp ou thymeleaf)"</h1>
  <hr/>
  <a href="index.html"> retour index/menu principal</a> <br/>
  <a href="site/basic/helloWorld"> helloWorld</a> <br/>
</body>
</html>
```

Effectuer les navigations nécessaires pour **tester le bon fonctionnement de ce premier exemple** (index → index-site → helloWorld).

2.2. Calcul de Tva en version "jsp"

Coder un calcul de Tva



Indications :

- **BasicController.calculTva(…** model, …. ht , … **taux**)
  préfixé par *@RequestMapping("calculTva")*
- paramètres ht et taux préfixés par **@RequestParam(name="...",defaultValue = "0")**
- formules de calculs : tva=ht*taux/100; et ttc=ht+tva;
- on stocke tout ("ht" , "taux" , "tva" , "ttc") dans les attributs du model
  et on retourne le besoin d'afficher la vue de nom "calcul_tva"

- dans index-site.html , nouveau lien hypertexte calculTva

- dans ***src/main/resources/META-INF/resources/views*** ajouter la nouvelle vue **calcul_tva.jsp**
- structure possible de **calcul_tva.jsp :**

```
<body>
        <form action="calculTva" method="GET">
                <label>…. :</label> <input type="text" name="taux" value="${taux}" /> <br />
                 ...
                <input type="submit" value="calculer tva et ttc" /> <br />
        </form>
        <hr/>
        ….<span>${tva}</span><br />… <hr/>
        <a href="../../index-site.html"> retour menu</a> <br/>
</body>
```

Facultativement (pour peaufiner) :

- Ajouter une feuille de styles css (ex : src/main/resources/static/**css/styles.css**)

exemple :

```
label { color: blue ; }
.highlight { font-weight: bold; }
.simpleAlign { color: blue ; display: inline-block; width: 10em; text-align: right; }
```

- Ajouter un lien avec une feuille de style dans la page jsp

```
<link href="../../css/styles.css" rel="stylesheet" />
```

- Ajouter quelques classes css dans la page JSP
(ex : <label *class="simpleAlign"*> , <span *class="highlight"* > , …)

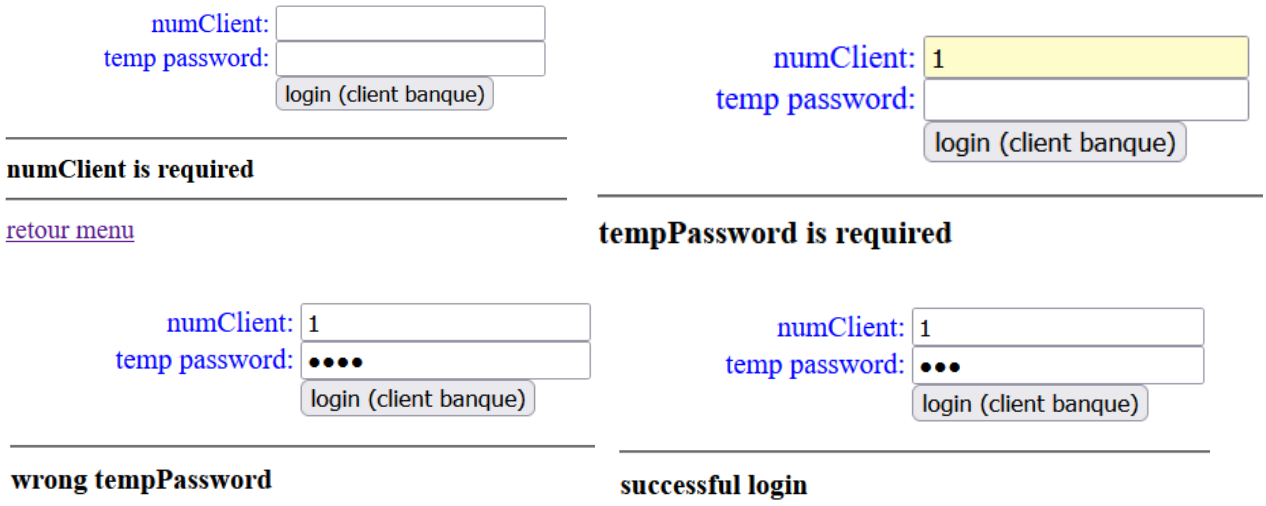
Et tester le nouveau look .

2.3. Préversion du login "client_banque" en version "jsp"

Coder et compléter le nouveau contrôleur suivant :

```
@….
@…..("/site/bank")
public class BankController {
…

   @RequestMapping("/clientLogin")
   public String clientLogin(Model model,
                  @RequestParam(name="numClient", required = false)  Long numClient,
                  @RequestParam(name="tempPassword", required = false)  String tempPassword) {
  …
  return "client_login"; //aiguiller sur la vue "client_login"
   }
}
```

Coder une première version de **client_login.jsp** ayant à peu près le comportement suivant :



NB : pour simplifier le TP , on considérera que la valeur du bon *tempPassword* est "**pwd**" (valeur fixe)

Ajouter un lien hypertexte de ce type dans index-site.html :

```
<a href="site/bank/clientLogin"> login client banque</a> <br/>
```

Tester le bon fonctionnement de tout cet ensemble devant être cohérent .

2.4. Infos "client connecté" en session (version "jsp")

Sans trop tôt améliorer le code de la classe "BankController" , naviguer de "clientLogin" vers index-site.html puis revenir sur "clientLogin". On devrait pour l'instant constater une perte des informations préalablement saisies car elles n'ont pas encore été stockées en session.

Améliorations progressives :

1) ajouter ceci au sein de BankController.java

```
@ModelAttribute("client")
public ClientDto addDefaultClientAttributeInModel() {
        //NB: new ClientDto(numero,  prenom, nom,  email,  adresse)
        return new ClientDto(null,"prenom?", "nom?" ,"ici_ou_la@xyz.com" , "adresse ?");
}
```

et ajouter plein d'affichages de type **${client.prenom}**, ${client.nom} au sein de **client_login.jsp**

Visualiser un premier résultat de ce genre :

Informations "Client"

numero:
prenom: **prenom?**
nom: **nom?**
email: **ici_ou_la@xyz.com**
adresse: **adresse ?**

2) Ajouter ceci dans le haut de BankController

```
@Autowired
private ServiceClientWithDto serviceClient;
```

puis ceci dans la méthode clientLogin(…)

```
if(numClient!=null) {
  ClientDto client = serviceClient.searchById(numClient);
   model.addAttribute("client", client);
 }
```

Visualiser un nouveau comportement de ce genre :

numClient: 1
temp password: ••••
login (client banque)

---

wrong tempPassword

Informations "Client"

numero: **1**
prenom: **alex**
nom: **Therieur**
email: **12 rue Elle 75001 Paris**
adresse: **email1**

3) en naviguant de "clientLogin" vers index-site.html puis en revenant sur "clientLogin" on constate encore pour l'instant une perte de certaines informations préalablement saisies car elles n'ont pas encore été stockées en session.
Ajouter donc ce qu'il manque au sein de la classe BankController pour conserver les informations "client" au de la session utilisateur gérée par springMvc (l'annotation **@SessionAttributes** est une bonne piste).

2.5. Affichage des comptes bancaires d'un client via JSP et JSTL

Au sein d'une nouvelle méthode "***comptesDuClient***" de *BankController* , en s'appuyant sur une référence sur le service "*ServiceCompteWithDto*" (à injecter) , on appellera la méthode **serviceCompte.searchCustomerAccounts(numClient)** de manière à récupérer la liste des comptes bancaires du client (que l'on stockera dans le "model" sous le nom "***listeComptes***") .
NB : le numéro de client (à priori déjà en session) pourra être récupéré via un code du genre ***model.getAttribute("numClient")*** .

Au sein d'une nouvelle page **comptes.jsp** on affichera tous les comptes d'un client dans un tableau HTML (via une boucle **c:forEach** gérée via **JSTL**).

On pourra coder des navigations (liens hypertextes) à partir de index-site.html et client_login.jsp .

Comptes du client numero 2

numero	label	solde
3	compteC3	250.0
4	compteC4	350.0

retour menu

2.6. nouveau compte en mode formulaire JSP (springMvc)

Ajout de compte pour le client 1

label: compteXyz
solde: 50.0
ajouter compte

Comptes du client numero 1

numero	label	solde
1	compteC1	50.0
2	compteC2	150.0
5	compteXyz	50.0

Indications :

Sur ***BankController*** , ajouter les méthodes suivantes :

```
@ModelAttribute("compte")
public CompteDto addDefaultCompteAttributeInModel() {
        return new CompteDto(null,"", 0.0);
}

@RequestMapping("toAddCompte")
public String toAddCompte(Model model) {
        Long numClient=(Long)model.getAttribute("numClient");
        if(numClient==null)
                return "client_login";
return "add_compte";
}


@RequestMapping("doAddCompte")
public String doAddCompte(Model model,
                                @ModelAttribute("compte") CompteDto compte) {
  try {
        Long numClient=(Long)model.getAttribute("numClient");
        if(numClient==null)
                        return "clientLogin";
        compte = serviceCompte.saveNew(compte);
        serviceCompte.fixerProprietaireCompte(compte.getNumero(), numClient);
        } catch (Exception e) {
                e.printStackTrace();
                model.addAttribute("message", e.getMessage());
                return "add_compte";
        }
    return comptesDuClient(model); //réactualiser et afficher nouvelle liste des comptes
   //retourne indirectement "comptes" .
}
```

Coder **add_compte.jsp** en s'appuyant sur **<form:form .../>** et
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>

On pourra coder des navigations vers toAddCompte depuis index-site.html et comptes.jsp et client_login.jsp .

2.7. Validation des informations saisies

Ajouter ceci dans **pom.xml** :

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

Ajouter ceci sur les parties "label" et "solde" de CompteDto :

```
	@NotEmpty(message = "label is required (not empty)")
	private String label;

	@Min(value=-999999)
	@Max(value=999999999)
	private Double solde;
```

Ajouter si besoin ceci dans **styles.css**

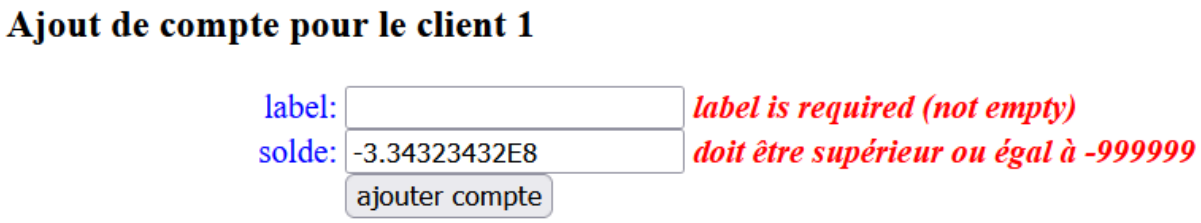
```
.error { color : red; font-weight: bold; font-style: italic; }
```

et ceci (pour label et solde) dans **add_compte.jsp** (à coté de <form:input path="label" />)

```
<form:errors path="label" cssClass="error" />
```

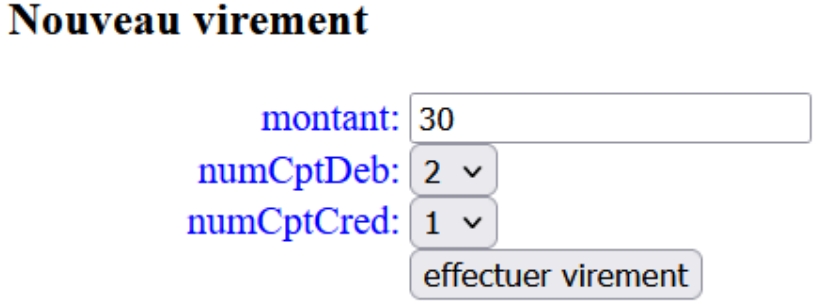
Ajouter **@Valid** et la prise en compte de **BindingResult** au sein de la méthode ***doAddCompte*** de ***BankController*** .

Reconstruire si nécessaire l'application ***appliSpringMvc*** et la redémarrer pour voir le nouveau comportement en cas d'erreurs de saisies :



2.8. Tp facultatif "effectuer virement" en mode "JSP"

Coder facultativement un virement bancaire (BankController.toVirement , BankController.doVirement , virement.jsp , ….)



3. TP "springMvc" en version "Thymeleaf"

3.1. Configuration et exemple simple en mode "Thymeleaf"

Effectuer ce **changement de configuration** au sein du fichier **pom.xml** :

```
<!-- selon la phase du Tp , commenter et decommenter une des 2 parties  thymeleaf ou JSP -->

		<!-- partie jsp →
		<!--
		<dependency>
			<groupId>org.apache.tomcat.embed</groupId>
			<artifactId>tomcat-embed-jasper</artifactId>
			<scope>provided</scope>
		</dependency>
		<dependency>
			<groupId>jakarta.servlet.jsp.jstl</groupId>
			<artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
		</dependency>
		<dependency>
			<groupId>org.glassfish.web</groupId>
			<artifactId>jakarta.servlet.jsp.jstl</artifactId>
		</dependency>
		-->
		<!-- partie thymeleaf -->
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-thymeleaf</artifactId>
		</dependency>
		<dependency>
			<groupId>nz.net.ultraq.thymeleaf</groupId>
			<artifactId>thymeleaf-layout-dialect</artifactId>
		</dependency>
```

Revérifier le bon contenu du fichier tp.appliSpringMvc.web.controller.***BasicController.java*** :

```
package tp.appliSpringMvc.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@RequestMapping("/site/basic")
public class BasicController {

	@RequestMapping("helloWorld")
	public String helloWorld(Model model) {
		model.addAttribute("message", "hello world");
		return "displayBasicMessage" ;//.jsp ou .html(thymeleaf)
	}
}
```

Commenter/désactiver ceci au **début** de la méthode ***main***() de ***AppliSpringMvcApplication.java***

```
//activation ou pas au démarrage du profil spring pour vues en version ".jsp"
//System.setProperty("spring.profiles.default", "dev,jsp");
System.setProperty("spring.profiles.default", "dev");
```

Ajouter le fichier displayBasicMessage.html suivant au sein de src/main/resources/templates

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
<title>displayBasicMessage (thymeleaf)</title>
</head>
<body>
  <span th:utext="${message}" ></span>
</body>
</html>
```

Revérifier le bon contenu de ***index-site.html*** à coté de *index.html* :

```
....
   <a href="site/basic/helloWorld"> helloWorld</a> <br/>
```

Effectuer les navigations nécessaires pour **tester le bon fonctionnement de ce premier exemple** en version ***thymeleaf*** (index → index-site → helloWorld).

3.2. layout et template "thymeleaf"

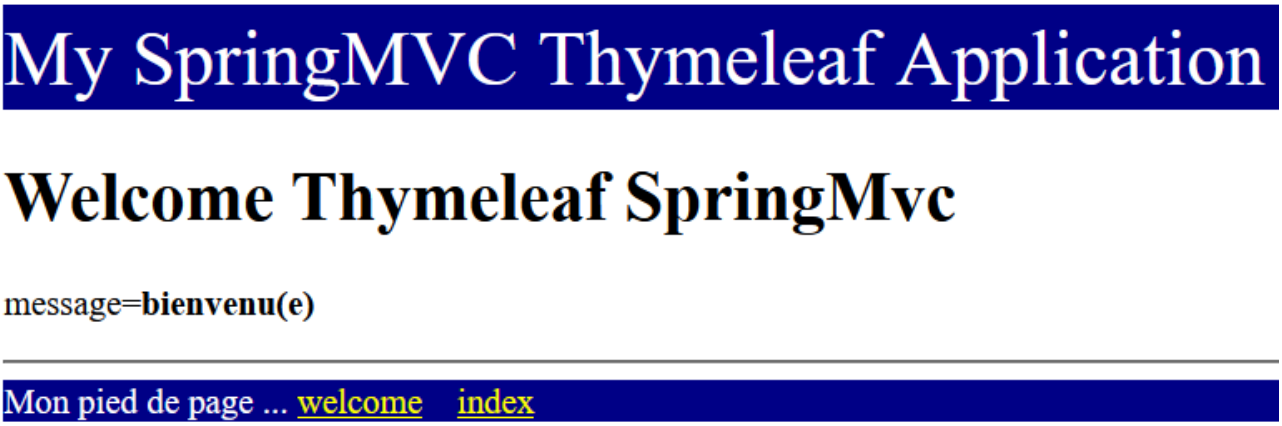
Mettre en place les modèles de mise en page suivants (en s'inspirant du support de cours) :
templates/**_header.html**
templates/**_footer.html**
templates/**_layout.html**

Coder un nouveau contrôleur "**AppCtrl**" comportant ce code initial :

```
...
@...
@RequestMapping("/site/app")
public class AppCtrl {

        @RequestMapping("/toWelcome")
        String toWelcome(Model model) {
                model.addAttribute("message", "bienvenu(e)");
                model.addAttribute("title","welcome");
           return "welcome";
        }
}
```

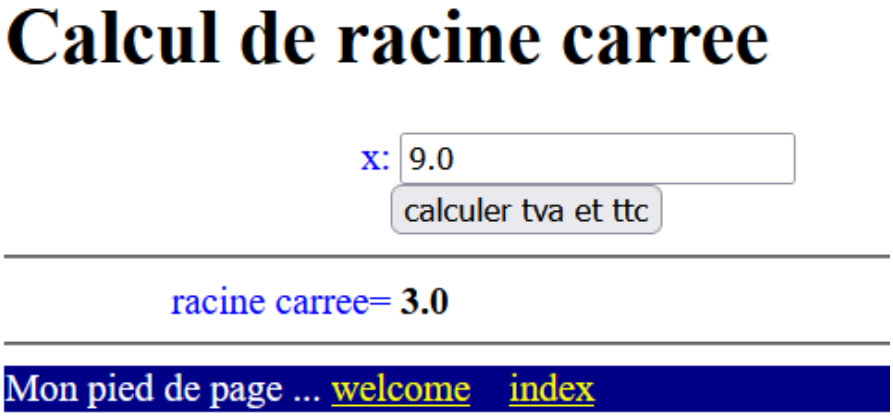
Coder templates/**welcome.html** se basant sur templates/**_layout.html** .
Coder une navigation de **index-site.html** vers **/site/app/toWelcome** .

Type de résultat attendu (selon looks et dispositions choisis):



3.3. Calcul de racine carrée en version "thymeleaf"

Coder un calcul de racine carrée ...



Indications :

- **BasicController.calculRacineCarree(…** model, , … **x**)
  préfixé par *@RequestMapping("calculRacineCarree")*
- paramètre x préfixé par **@RequestParam(name="...",defaultValue = "0")**
- formule de calcul : racine=Math.sqrt(x)
- on stocke tout ("x" , "racine") dans les attributs du model
  et on retourne le besoin d'afficher la vue de nom "calcul_racine"

- dans static/index-site.html , nouveau lien hypertexte
   calcul racine carree

- dans template/welcome.html , nouveau lien hypertexte
  <a th:href="@{/site/basic/calculRacineCarree}" >calculRacineCarree

- dans ***src/main/resources/templates*** ajouter la nouvelle vue **calcul_racine.html**

- structure possible de **calcul_racine.html:**

```
<div xmlns:th="http://www.thymeleaf.org"
     xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
     layout:decorate="~{_layout}"
    layout:fragment="content">
   <h1>Calcul de racine carree</h1>

   <form th:action="@{/site/basic/calculRacineCarree}" method="POST">
                <label class="simpleAlign">x:</label>
                <input type="text" name="x" th:value="${x}" /> <br />

                <label class="simpleAlign"></label>
                <input type="submit" value="calculer racine carree" /> <br />
  </form>
   <hr/>
        <label class="simpleAlign">racine carree=</label>
        <span class="highlight"  th:utext="${racine}"></span><br />

</div>
```

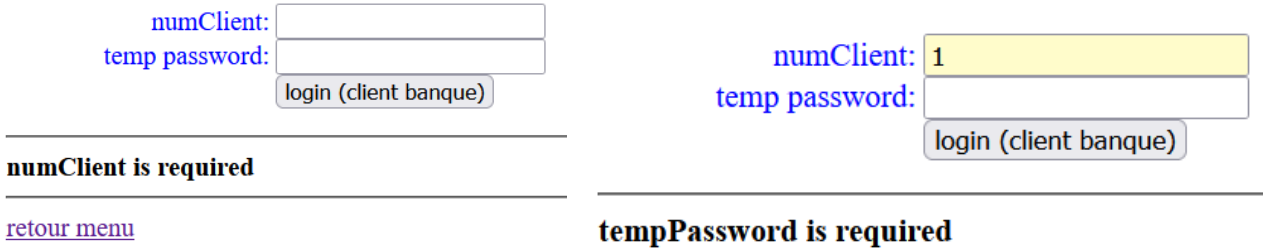
Si Tp tva préalablement effectué en version "jsp" , coder facultativement templates/calcul_tva.html en version thymeleaf .

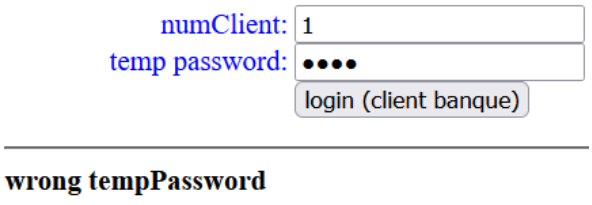
3.4. Préversion du login "client_banque" en version "thymeleaf"

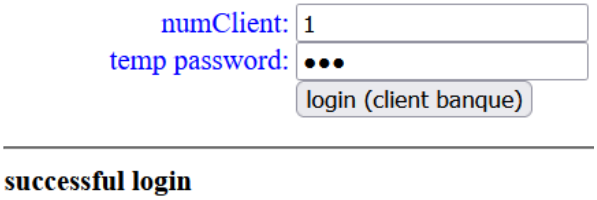
Coder et compléter si besoin (selon Tps antérieurs) le contrôleur suivant :

```
@….
@…..("/site/bank")
public class BankController {
…

   @RequestMapping("/clientLogin")
   public String clientLogin(Model model,
                @RequestParam(name="numClient", required = false)  Long numClient,
                @RequestParam(name="tempPassword", required = false)  String tempPassword) {
  …
  return "client_login"; //aiguiller sur la vue "client_login"
   }
}
```

Coder une première version de **client_login.html** ayant à peu près le comportement suivant :







NB : pour simplifier le TP , on considérera que la valeur du bon *tempPassword* est "**pwd**" (valeur fixe)

Ajouter si besoin un lien hypertexte de ce type dans index-site.html :

```
<a href="site/bank/clientLogin"> login client banque</a> <br/>
```

et ajouter un lien hypertexte au format ***th:href="@{…}"*** au sein de **welcome.html** .

Tester le bon fonctionnement de tout cet ensemble devant être cohérent .

3.5. Infos "client connecté" en session (version "thymeleaf")

Améliorations potentiellement progressives (selon Tps antérieurs):

1) ajouter ceci (si besoin selon Tps antérieurs) au sein de BankController.java

```
@ModelAttribute("client")
public ClientDto addDefaultClientAttributeInModel() {
        //NB: new ClientDto(numero,  prenom, nom,  email,  adresse)
        return new ClientDto(null,"prenom?", "nom?" ,"ici_ou_la@xyz.com" , "adresse ?");
}
```

et ajouter plein d'affichages de type **${client.prenom}**, ${client.nom} au sein de **client_login.html**

Visualiser un premier résultat de ce genre (ou mieux selon tps antérieurs) :

Informations "Client"

numero:
prenom: **prenom?**
nom: **nom?**
email: **ici_ou_la@xyz.com**
adresse: **adresse ?**

2) Si pas déjà fait au sein d'un Tp antérieur en mode jsp , ajouter ceci dans le haut de BankController

```
@Autowired
private ServiceClientWithDto serviceClient;
```

puis (si besoin) ceci dans la méthode clientLogin(…)

```
if(numClient!=null) {
  ClientDto client = serviceClient.searchById(numClient);
  model.addAttribute("client", client);
 }
```

Visualiser un comportement (peut être amélioré) de ce genre :

numClient: 1
temp password: ••••
login (client banque)

---

wrong tempPassword

Informations "Client"

numero: **1**
prenom: **alex**
nom: **Therieur**
email: **12 rue Elle 75001 Paris**
adresse: **email1**

3) Si en naviguant de "clientLogin" vers index-site.html puis en revenant sur "clientLogin" on constate une perte de certaines informations préalablement saisies c'est que celles-ci n'ont pas encore été stockées en session.
Ajouter donc (si besoin) ce qu'il manque au sein de la classe BankController pour conserver les informations "client" au de la session utilisateur gérée par springMvc (l'annotation **@SessionAttributes** est une bonne piste).

NB : au sein de client_login.html , il faudra peut être remplacer th:value="${numClient}" par th:value="${**#ctx.session.**numClient}" . Idem pour tempPassword .

Pour bien tester le login on pourra coder et déclencher (depuis welcome) un **logout** que l'on peut par exemple coder comme cela :

```
@RequestMapping("/logout")
        public String clientLogout(Model model,
                HttpSession httpSession,  SessionStatus sessionStatus) {
     httpSession.invalidate();  sessionStatus.setComplete();
      model.addAttribute("message", "session terminée");
      model.addAttribute("title","welcome");
    return "welcome";
}
```

3.6. Affichage des comptes bancaires d'un client via Thymeleaf

Au sein d'une méthode "***comptesDuClient***" de *BankController* , en s'appuyant sur une référence sur le service "*ServiceCompteWithDto*" (à injecter) , on appellera si besoin (selon Tps antérieurs) la méthode **serviceCompte.searchCustomerAccounts(numClient)** de manière à récupérer la liste des comptes bancaires du client (que l'on stockera dans le "model" sous le nom "***listeComptes***") .
NB : le numéro de client (à priori déjà en session) pourrait être récupéré via un code du genre ***model.getAttribute("numClient")*** .

Au sein d'une nouvelle page **comptes.html** on affichera tous les comptes d'un client dans un tableau HTML (via une boucle **th:each** gérée via **Thymeleaf**).
Indication : <tr th:each="elementDeLaListe : ${liste à parcourir}">

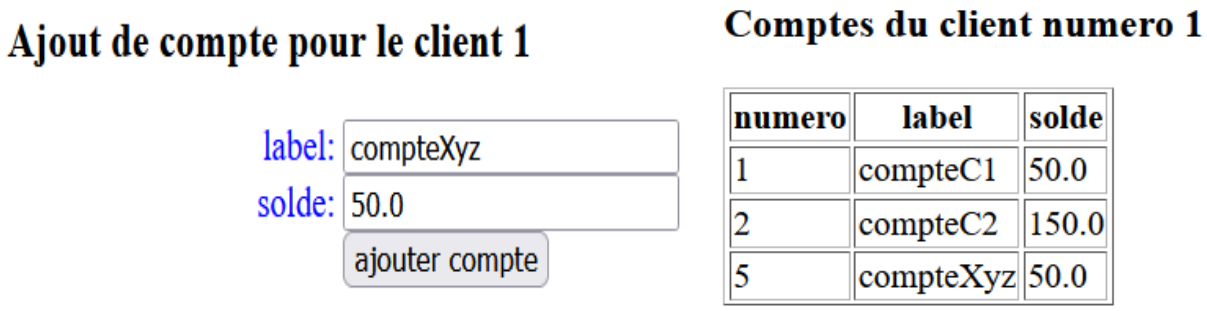
On pourra coder des navigations (liens hypertextes) à partir de index-site.html , welcome.html et client_login.html .

Comptes du client numero 2

numero	label	solde
3	compteC3	250.0
4	compteC4	350.0

retour menu

3.7. nouveau compte en mode formulaire Thymeleaf



Indications :
Sur ***BankController*** , ajouter les méthodes suivantes (si pas déjà fait dans TPs antérieurs):

```
@ModelAttribute("compte")
public CompteDto addDefaultCompteAttributeInModel() {
        return new CompteDto(null,"", 0.0);
}

@RequestMapping("toAddCompte")
public String toAddCompte(Model model) {
        Long numClient=(Long)model.getAttribute("numClient");
        if(numClient==null)
                return "client_login";
return "add_compte";
}

@RequestMapping("doAddCompte")
public String doAddCompte(Model model,
                        @ModelAttribute("compte") CompteDto compte) {
  try {
        Long numClient=(Long)model.getAttribute("numClient");
        if(numClient==null)
                        return "clientLogin";
        compte = serviceCompte.saveNew(compte);
        serviceCompte.fixerProprietaireCompte(compte.getNumero(), numClient);
        } catch (Exception e) {
                e.printStackTrace();
                model.addAttribute("message", e.getMessage());
                return "add_compte";
        }
   return comptesDuClient(model); //réactualiser et afficher nouvelle liste des comptes
  //retourne indirectement "comptes" .
}
```

Coder **add_compte.html** en s'appuyant sur **<form th:object="${...}" />**
et **<input th:field="*{…}">**

On pourra coder des navigations vers toAddCompte depuis index-site.html et comptes.html et client_login.html .

3.8. Validation des informations saisies

Si besoin (selon tp antérieurs) , ajouter ceci dans **pom.xml** :

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

Si besoin (selon tp antérieurs) , ajouter ceci sur les parties "label" et "solde" de CompteDto :

```
	@NotEmpty(message = "label is required (not empty)")
	private String label;

	@Min(value=-999999)
	@Max(value=999999999)
	private Double solde;
```

Si besoin (selon tp antérieurs) , ajouter si besoin ceci dans **styles.css**

```
.error { color : red; font-weight: bold; font-style: italic; }
```

et ceci (pour label et solde) dans **add_compte.html** (à coté de <form:input path="label" />)

```
<span th:if="${#fields.hasErrors('label')}" th:errorclass="error" th:errors="*{label}" ></span>
```

Ajouter **@Valid** et la prise en compte de **BindingResult** au sein de la méthode ***doAddCompte*** de ***BankController*** .

Reconstruire si nécessaire l'application ***appliSpringMvc*** et la redémarrer pour voir le nouveau comportement en cas d'erreurs de saisies :

Ajout de compte pour le client 1

label: *label is required (not empty)*
solde: -3.34323432E8 *doit être supérieur ou égal à -999999*
ajouter compte

3.9. Tp "effectuer virement" en mode "Thymeleaf"

Coder un virement bancaire (BankController.toVirement , BankController.doVirement , virement.html , ….)

Nouveau virement

montant: 30
numCptDeb: 2
numCptCred: 1
effectuer virement

```
<select th:field="*{numCptDeb}" >
	<option th:each ="cpt : ${listeComptes}" th:value="${cpt.numero}"
		th:utext="${cpt.numero} + ' - ' + ${cpt.label} + ' : ' + ${cpt.solde}"></option>
</select>
```

3.10. Tp sur syntaxes de thymeleaf (checkbox, radio, ...)

Ajouter la nouvelle classe **InscriptionForm** au sein du package **tp.appliSpringMvc.web.form** en s'inspirant du code suivant :

```
...
import java.time.LocalDate;
...
class Sport{
	private String nom;
	private Integer nbHeuresParSemaine;

	//+get/set et .toString()
}

public class InscriptionForm {
    public enum Situation { CELIBATAIRE , EN_COUPLE };
    private String nom; //input text
    private String pays; //select/option
    private LocalDate dateDebut=LocalDate.now(); //input de type date
    private Situation situation; //radio button
    private Boolean sportif=Boolean.FALSE; //checkbox
    private Sport sportPrincipal=new Sport(); //may be null

   public List<String> getListePays() {
        return Arrays.asList("France" , "Alemagne" , "Italie" , "Espagne");
   }

   //+get/set et .toString()
}
```

Ajouter ceci au sein de BasicController :

```
...
	@ModelAttribute("inscriptionF")
	public InscriptionForm addDefaultInscriptionAttributeInModel() {
		return new InscriptionForm();
	}

	@RequestMapping("toInscription")
	public String toInscription(Model model) {
		return "inscription";
	 }

	@RequestMapping("doInscription")
	public String doInscription(Model model,
			@ModelAttribute InscriptionForm inscriptionForm) {
		if(inscriptionForm.getSportif()==false)
			inscriptionForm.setSportPrincipal(null);
		model.addAttribute("inscriptionF" , inscriptionForm );
		return "recapInscription";
	 }
```

Coder les templates ***inscription.html*** et ***recapInscription.html*** (via Thymeleaf) puis encoder de nouveaux liens hypertextes de manière à obtenir le résultat/comportement suivant :

Inscription

nom: Didier
pays: France
dateDebut : 13/06/2024 (ex: 2024-12-25 ou 25/12/2024)
situation: CELIBATAIRE EN_COUPLE
sportif:

sportPrincipal

sport principal: vélo
nbHeuresParSemaine: 2

effectuer inscription

Recapitulatif inscription

InscriptionForm [nom=Didier, pays=France, dateDebut=2024-06-13, situation=EN_COUPLE, sportif=true, sportPrincipal=Sport [nom=vélo, nbHeuresParSemaine=2]]

4. Sécurité pour springMvc (jsp ou thymeleaf)

4.1. Comportement par défaut et paramétrages attendus

Ajouter simplement ceci dans **pom.xml**

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

et redémarrer l'application.

La console affiche un message de ce genre :

```
Using generated security password: 656a4d96-2f13-46fd-b4b9-2e2c90b3fbb6

This generated password is for development use only. Your security configuration must be updated before running your application in production.
```

Ce mot de passe (régénéré à chaque démarrage) est pour le username "**user**"

Après avoir saisi "user" et le mot de passe attendu par copier/coller dans la boîte de dialogue , on peut accéder à la page d'accueil principale (index.html) et naviguer vers d'autres pages .

4.2. Gestion générique des erreurs "SpringMvc & thymeleaf"

```
package tp.appliSpringMvc.web.controller;
...
@ControllerAdvice
public class ErrorController {

   private static Logger logger = LoggerFactory.getLogger(ErrorController.class);

   @ExceptionHandler(Throwable.class)
   @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
   public String exception(final Throwable throwable, final Model model) {
      logger.error("Exception during execution of SpringSecurity application", throwable);
      String errorMessage = (throwable != null ? throwable.getMessage() : "Unknown error");
      model.addAttribute("errorMessage", errorMessage);
      return "error";
   }
}
```

templates/error.html

```
<div xmlns:th="http://www.thymeleaf.org"
     xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
     layout:decorate="~{_layout}"
    layout:fragment="content">
   <h2 class="error">Error Page (navigation invalide ou autre)</h2>
   <hr/>
   <h3>Generic Error - <span th:text="${status}">Status</span></h3>
         <ul>
           <li>Timestamp: <span th:text="${timestamp}">Timestamp</span></li>
```

```
            <li>Path: <span th:text="${path}">Path</span></li>
            <li>Error: <span th:text="${error}">Error</span></li>
          </ul>
    <hr/>
    <div>
       <p th:utext="${errorMessage}"></p>
       <a href="index.html" th:href="@{/index.html}">Back to main index</a> <br/>
       <a href="index.html" th:href="@{/index-site.html}">Back to index-site</a> <br/>
       <a href="index.html" th:href="@{/site/app/toWelcome}">Back to welcome</a> <br/>
    </div>
</div>
```

Ceci permettra ultérieurement d'afficher (de manière un peu générique) des erreurs à destination de l'utilisateur :

Error Page (navigation invalide ou autre)

Generic Error - 500

- Timestamp: Wed Jun 19 15:33:45 CEST 2024
- Path: /appliSpringMvc/site/bank/comptesDuClient
- Error: Internal Server Error

4.3. Paramétrages explicites de la sécurité

Au sein du package principal ***tp.appliSpringMvc***, ajouter les 3 classes suivantes :

WithoutSecurityConfig.java

```
...
@Configuration
@Profile("!withSecurity")
public class WithoutSecurityConfig {

    @Bean
   protected SecurityFilterChain withoutSecurityFilterChain(HttpSecurity http) throws Exception {
                return http.securityMatcher("/**")
                    .authorizeHttpRequests(
                                auth -> auth.requestMatchers("/**").permitAll()
                                )
                    .cors( Customizer.withDefaults() )
                    .csrf( csrf -> csrf.disable() )
                    .build();
        }
}
```

MyUserDetailsService.java

```
package tp.appliSpringMvc;

@Profile("withSecurity")
@Service
public class MyUserDetailsService implements UserDetailsService {
	Logger logger = LoggerFactory.getLogger(MyUserDetailsService.class);
	@Autowired
	private PasswordEncoder passwordEncoder;
	@Autowired
	private ServiceClientWithDto serviceCustomer;

	@Override
	public UserDetails loadUserByUsername(String username)
					throws UsernameNotFoundException {
		UserDetails userDetails = null;
		List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
		String password = null;
		if (username.equals("admin1")) {
			password = passwordEncoder.encode("pwd1");// simulation password ici
			authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
			authorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
			userDetails = new User(username, password, authorities);
		} else if (username.equals("user1")) {
			password = passwordEncoder.encode("pwd1");// simulation password ici
			authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
			userDetails = new User(username, password, authorities);
		}
		else {
			//NB le username considéré comme potentiellement égal à "client_" + numClient
		    try {
			Long numClient = Long.parseLong(username.substring(7));
			ClientDto customer = serviceCustomer.searchById(numClient);
			authorities.add(new SimpleGrantedAuthority("ROLE_USER"));
			authorities.add(new SimpleGrantedAuthority("ROLE_CUSTOMER"));
			 password = passwordEncoder.encode("pwd");
					//en dur (pas recherché en base dans cette version "cas ecole")
			userDetails = new User(username, password, authorities);
			} catch (Exception e) {
				//e.printStackTrace();
			}
		    }
		    if (userDetails == null) {
			throw new UsernameNotFoundException(username + " not found");
		     }
			return userDetails;
			//NB: en retournant userDetails = new User(username, password, authorities);
			//on retourne comme information une association entre usernameRecherché et
			//(bonMotDePasseCrypté + liste des rôles)
			//Le bonMotDePasseCrypté servira simplement à effectuer une comparaison avec le mot
			//de passe qui sera saisi ultérieurement par l'utilisateur
			//(via l'aide de passwordEncoder.matches())
		}
}
```

SecurityConfig.java

```
...
@Configuration
@Profile("withSecurity")
public class WithoutSecurityConfig {
	//future autre @Bean @Order(1) pour future partie API-REST

	@Bean
	@Order(2)
	protected SecurityFilterChain siteFilterChain(HttpSecurity http) throws Exception {
		return http.securityMatcher("/site/**")
		   .authorizeHttpRequests(
		        auth -> auth.requestMatchers("/site/app/**").permitAll()
		                    .requestMatchers("/site/basic/**").permitAll()
		                   /*.requestMatchers("/site/bank/**").authenticated()*/
		                    .requestMatchers("/site/bank/**").hasRole("CUSTOMER")
		                )
		    .csrf( Customizer.withDefaults() )
		    //.formLogin( formLogin -> formLogin.permitAll() )
		    .formLogin( formLogin -> formLogin.loginPage("/site/app/login")
		                                      .failureUrl("/site/app/login-error")
		                                      .defaultSuccessUrl("/site/app/toWelcome", false)
		                                      .permitAll())

		    .build();
		   //NB: /site/app/login et /site/app/login-error redigirent tous les deux vers templates/login.html
	}

	@Bean
	@Order(3)
	protected SecurityFilterChain otherFilterChain(HttpSecurity http,
	                                PasswordEncoder passwordEncoder) throws Exception {
		return http.securityMatcher("/**")
		    .authorizeHttpRequests(
		                // pour image , html , css , js
		                auth -> auth.requestMatchers("/**").permitAll())
		    .build();
	}

	@Bean
	public PasswordEncoder passwordEncoder() {
		return new BCryptPasswordEncoder();
	}
}
```

NB :
Au sein de la méthode **main**() de la classe principale ***AppliSpringMvcApplication***, on activera ou pas la sécurité en activant ou pas le profile "withSecurity' :

```
//System.setProperty("spring.profiles.default", "dev");
System.setProperty("spring.profiles.default", "dev,withSecurity");
```

Ajout spécifique "spring-security6" + "thymeleaf" dans **pom.xml** :

```
<dependency>
        <groupId>org.thymeleaf.extras</groupId>
        <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

Ceci permet d'ajouter ceci dans **templates/_footer.html** ou ailleurs :

```
<footer xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
      th:fragment="_footer">
 <div … >...
    <span sec:authorize="isAuthenticated()">
        authenticated user: <span sec:authentication="name">Unknown</span>  
        roles: <span sec:authentication="principal.authorities">[]</span>
  </span>
</div>
</footer>
```

De manière à afficher ultérieurement

De façon cohérente à la partie **formLogin** de *SecurityConfig.java*, on ajouter ces nouvelles méthodes au sein de la classe **AppCtrl.java**

```
...
@Controller
@RequestMapping("/site/app")
public class AppCtrl {
...

  // Login form
  @RequestMapping("/login")
   public String login() {
          return "login";
         }

  // Login form with error
  @RequestMapping("/login-error")
    public String loginError(Model model) {
          model.addAttribute("loginError", true);
          return "login";
         }

  //version avec spring_security + spring_mvc
  @RequestMapping("/logout")
       public String logout(Model model,
                    HttpServletRequest request, HttpServletResponse response,
                    SessionStatus sessionStatus) {
       Authentication auth = SecurityContextHolder.getContext().getAuthentication();
```

```
        if (auth != null){
             new SecurityContextLogoutHandler().logout(request, response, auth);
            //httpSession.invalidate() is indirectly called
           }
            sessionStatus.setComplete();
            model.addAttribute("message", "session terminée");
            model.addAttribute("title","welcome");
            return "welcome";
       }
…
}
```

et logiquement avec **templates/login.html** :

```
<div xmlns:th="http://www.thymeleaf.org"
     xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
     layout:decorate="~{_layout}"
    layout:fragment="content">
   <h1>login (springSecurity)</h1>
   <p th:if="${loginError}" class="error">Wrong user or password</p>
   <form th:action="@{/site/app/login}" method="post">
     <label for="username">Username</label>:
     <input type="text" id="username" name="username" autofocus="autofocus" /> <br />
     <label for="password">Password</label>:
     <input type="password" id="password" name="password" /> <br />
     <input type="submit" value="Log in" />
     <p> user1/pwd1 , admin1/pwd1 , client_n/pwd</p>
   </form>
</div>
```

Nouvelle méthode **clientLoginWithSecurity()** à ajouter dans ***BankController.java*** (à coté/en plus de clientLogin(…)) :

```
@RequestMapping("/clientLoginWithSecurity")
public String clientLoginWithSecurity(Model model) {
        //avec/après "navigation hook" géré automatiquement par spring-security
        // (redirection automatique vers login.html , ...)

       //on récupère le username de l'utilisateur loggé avec spring security
       Authentication auth = SecurityContextHolder.getContext().getAuthentication();
          if (auth != null){
            String username =auth.getName();
            System.out.println("clientLoginWithSecurity , username="+username);
            //on considère que username vaut (par convention dans ce Tp) "client_" + numClient
            //et on extrait donc le numero du client authentifié:
            Long numClient= Long.parseLong(username.substring(7));
            System.out.println("clientLoginWithSecurity , numClient="+numClient);
            if(numClient!=null) {
                    ClientDto client = serviceClient.searchById(numClient);
                    model.addAttribute("client", client);
                    model.addAttribute("tempPassword", "pwd");//cas d'école (tp)
                    model.addAttribute("message", "successful login");
                    model.addAttribute("numClient", numClient);
```

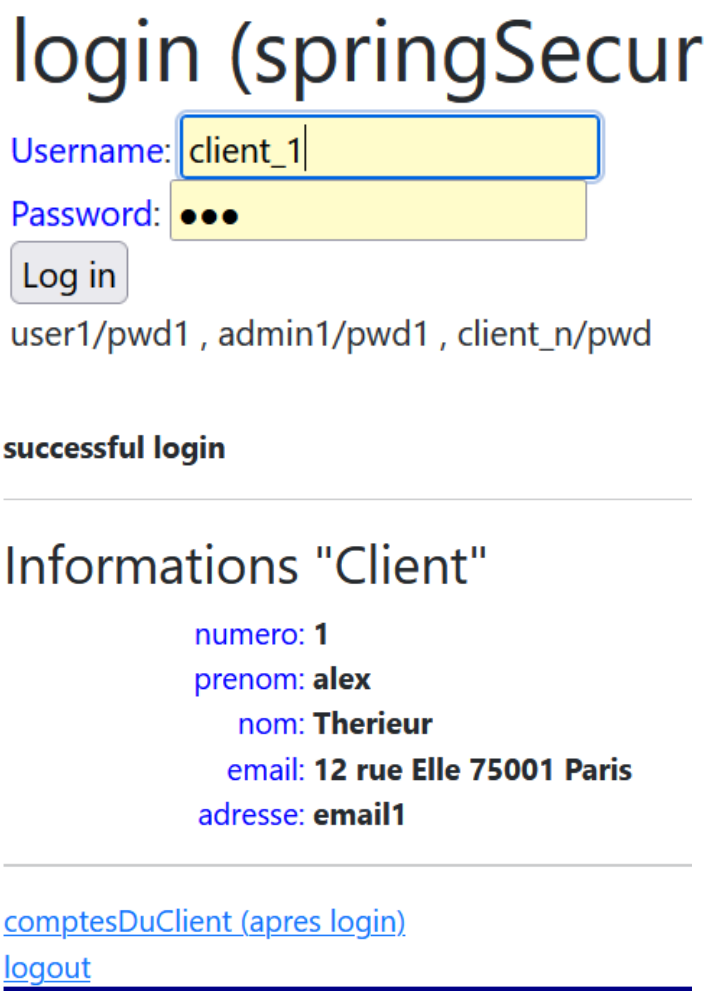
```
                    }
                return "client_login";
            }
            return "welcome"; /* else , if no numClient */
        }
...
```

Et **enfin**, après quelques petits ajustements à effectuer au sein de templates/**welcome.html** et static/index-site.html

```
<!-- <a th:href="@{/site/bank/clientLogin}" >clientLogin (sans securite)</a> <br/> -->
  <a th:href="@{/site/bank/clientLoginWithSecurity}" >clientLoginWithSecurity</a> <br/>
  <a th:href="@{/site/app/login}" >login (spring security) (user,admin,client)</a> <br/>

<!-- <a th:href="@{/site/bank/logout}"> logout (sans spring_security) </a> <br/> -->
  <a th:href="@{/site/app/logout}"> logout (avec spring security)</a> <br/>
```

On devrait pouvoir redémarrer l'application en obtenir un comportement de ce genre :

- l**es parties "calculTva" , "inscription" , "calculRacineCarree" sont directement accessibles (sans login préalable)** car **.permitAll()** sur parties "/site/app" et "/site/basic" au sein de SecurityConfig.java
- Toutes les parties de ***/site/bank*** ne sont accessibles qu'après un login réussi en tant que ***client_n*** (de manière à avoir le rôle "CUSTOMER") :



NB : En mode JSP/HTML (conforme à la sécurité JEE standard), si l'on demande à naviguer vers une zone protégée telle que "/site/app/clientLoginWithSecurity" et que formLogin (de SecurityConfig.java) a été configuré via formLogin.loginPage("/site/app/login")… on se retrouve alors automatiquement redirigé vers /site/app/login et login.html et si succès on est ensuite redirigé vers ce que l'on avait demandé ("clientLoginWithSecurity" ou autre).

Si l'on se connecte avec le compte *user1/pwd1* (qui n'est pas associé au rôle "CUSTOMER") on pourra éventuellement obtenir un message d'erreur de ce genre :

Attention, Spring5 et SpringBoot2 géraient très bien la sécurité "_csrf" .
Depuis Spring6 et SpringBoot3 , la sécurité "_csrf" a été chamboulée (pas systématique , selon le mode HTTP : POST,PUT,DELETE mais pas GET , …) .

En d'autres termes, sur un ancien projet "SpringMvc + JSP ou Thymeleaf" , il vaut mieux rester sur du Spring5 et SpringBoot2 et sur un projet récent "SpringMVC + Api REST" , Spring6 et SpringBoot3 sont plus appropriés (un peu plus en phase) .

5. TP "springMvc" en version "Api REST"

Ouf , enfin des choses du monde moderne (des années 2020) et pas des années 2010 !!!

NB : dans un premier temps , on va désactiver la sécurité et on va la réactiver ultérieurement lorsque celle ci aura été réadaptée aux besoins des web services "REST" .

```
System.setProperty("spring.profiles.default", "dev");
//System.setProperty("spring.profiles.default", "dev,withSecurity");
```

5.1. Début d'api rest en mode GET

Créer un nouveau package tp.appliSpringMvc.**rest** .
Au sein de ce package , ajouter le début de classe suivante :

CompteRestCtrl.java

```
…

@RestController
@RequestMapping(value="/rest/api-bank/compte" , headers="Accept=application/json")
public class CompteRestCtrl {

	@Autowired
	private ServiceCompteWithDto serviceCompte;

	@GetMapping("/{id}")
	public CompteDto getCompteById(@PathVariable("id") long numeroCompte) {
			return serviceCompte.searchById( numeroCompte);
	}
}
```

Compléter **index-rest.html**

```
<html>
<head>
<title>index</title>
</head>
<body>
   <h1>index/menu de la partie "API REST / JSON"</h1>
   <hr/>
   <a href="index.html"> retour index/menu principal</a> <br/>
   <a href="rest/api-bank/compte/1"> compte 1 en JSON</a> <br/>
</body>
</html>
```

en démarrant l'application et en navigant de ***index.html*** vers ***index-rest.html*** on devrait pourvoir déclencher l'appel en GET précédent et obtenir ce résultat :

```
{"numero":1,"label":"compteC1","solde":50.0}
```

En exercice, coder maintenant une méthode ***getComptesByCriteria***() que l'on pourra déclencher via ces URLs

```
<a href="rest/api-bank/compte"> tous les comptes en JSON</a> <br/>
<a href="rest/api-bank/compte?soldeMini=200"> comptes avec solde >= 200.0 </a> <br/>
```

de manière à obtenir un résultat de ce genre :

```
[ {"numero":3,"label":"compteC3","solde":250.0},
  {"numero":4,"label":"compteC4","solde":350.0}  ]
```

5.2. De l'importance des Dto

Ajouter la nouvelle classe suivante au sein du package ***tp.appliSpringMvc.rest***

BadRestCtrl.java

```
@RestController
@RequestMapping(value="/rest/api-bank/bad-compte" , headers="Accept=application/json")
public class BadRestCtrl {

	@Autowired
	private DaoCompte daoCompte;

	@GetMapping("/{id}")
	public Compte getCompteEntityById(@PathVariable("id") long numeroCompte) {
			return daoCompte.findById( numeroCompte).get();
	}
}
```

On pourra également ajouter le déclenchement suivant au sein de index-rest.html :

```
<a href="rest/api-bank/bad-compte/1"> compte 1 en JSON sans DTO (pas bien)</a> <br/>
```

pour obtenir un bug (boucle infinie) :

{"clients":[{"numero":1,"prenom":"alex","nom":"Therieur","adresse":"email1","email":"12 rue Elle 75001 Paris"}],"numero":1,"label":"compteC1","solde":50.0,"operations":[{"numOp":1,"label":"achat bonbons","montant":-4.67,"dateOp":"2024-06-19","compte":{"clients":[{"numero":1,"prenom":"alex","nom":"Therieur","adresse":"email1","email":"12 rue Elle 75001 Paris"}],"numero":1,"label":"compteC1","solde":50.0,"operations":[{"numOp":1,"label":"achat bonbons","montant":-4.67,"dateOp":"2024-06-19","compte":{"clients":[{"numero":1,"prenom":"alex","nom":"Therieur","adresse":"email1","email":"12 rue Elle 75001 Paris"}],"numero":1,"label":"compteC1","solde":50.0,"operations":[{"numOp":1,"label":"achat bonbons","montant":-4.67,"dateOp":"2024-06-19","compte":{"clients":[{"numero":1,"prenom":"alex","nom":"Therieur","adresse":"email1","email":"12 rue Elle 75001 Paris"}],"numero":1,"label":"compteC1","solde":50.0,"operations":[{"numOp":1,"label":"achat bonbons","montant":-4.67,"dateOp":"2024-06-19","compte":{"clients":[{"numero":1,"prenom":"alex","nom":"Therieur","adresse":"email1","email":"12 rue Elle 75001 Paris"}],"numero":1,"label":"compteC1","solde":50.0,"operations":[{"numOp":1,"label":"achat bonbons","montant":-4.67,"dateOp":"2024-06-19","compte":{"clients":[{"numero":1,".....

D'où vient le problème ????

La technologie **Spring-Mvc** utilise en interne la technologie **jackson-databind** pour convertir automatiquement des objets java en JSON et vice-versa .

La technologie jackson-databind suit par défaut tous les liens qui existent entre les objets java.

La version BadRestCtrl utilise des DAO/repository qui remontent directement des objets persistants (souvent gérés par JPA/Hibernate et comportant quelquefois des liens bidirectionnels) .

Dans notre projet , la classe entity.Compte est reliée à une liste d'objets entity.Operation et chaque objet entity.Operation est lui même relié à un objet entity.Compte .

C'est ce lien bi-directionnel qui est trop suivi et qui provoque une boucle infinie.

En ajoutant **@JsonIgnore** a coté de @OneToMany(mappedBy = "compte") sur la entity.Operation.comptes , le bug disparaît mais ceci n'est faisable que sur un tout petit projet et considéré comme une mauvaise pratique sur la plupart des projets sérieux.

En conclusion :

- Il est très déconseillé de manipuler des entités persistantes au niveau d'une api REST.
- Une bonne api REST doit généralement s'appuyer sur une service métier qui expose des DTO (Data transfert Object) déjà convertis à partir de parties d'entités persistantes.
- Au sein du projet appliSpringMvc , les conversions entity<--→DTO sont effectuées au niveau des "ServiceXxxxWithDto" en s'appuyant sur les classes utilitaires du package converter . On peut faire mieux (mais en plus complexe) en s'appuyant par exemple sur mapStruct.

Dans la suite de cette série de Tps , la classe ~~BadRestCtrl~~ ne sera plus utilisée et on continuera à améliorer la bonne version **CompteRestCtrl.java** .

5.3. Gestion explicite des codes de retour via ResponseEntity<>

Si l'on ne fait aucune attention aux cas d'erreur possible , notre code n'est pas fiable.
Par exemple si la base de données ne comporte que 4 comptes et que l'on demande le compte 99 via une URL de ce genre

http://localhost:8080/appliSpringMvc/rest/api-bank/compte/**99**

on obtient une erreur pas très explicitée :

Error Page (navigation invalide ou autre)

Generic Error - 500

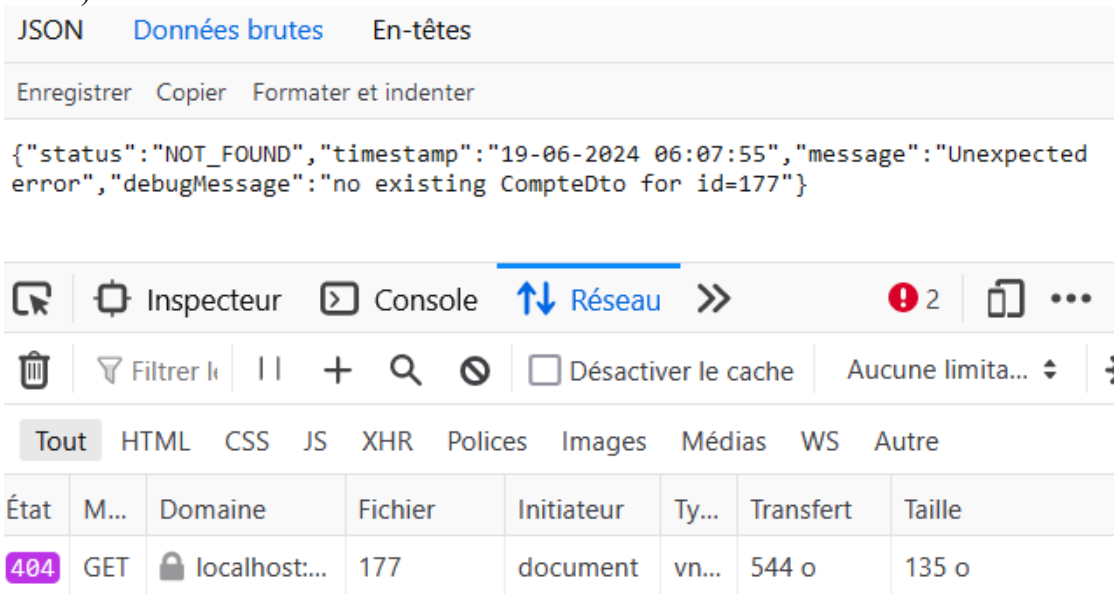
- Timestamp: Wed Jun 19 17:49:32 CEST 2024
- Path: /appliSpringMvc/rest/api-bank/compte/99
- Error: Internal Server Error

(si Tps effectués antérieurement avec thymeleaf) ou bien (par défaut avec aucune configuration) :
Une **page blanche** ou bien "**white label error**" avec un code 500 (Internal Server Error) que l'on ne voit qu'avec la console web du navigateur internet (firefox ou chrome ou autre).

En **exercice** on pourra :
- commenter tout le bloc de code actuel de la méthode getCompteById de CompteRestCtrl
de manière à restaurer ce code plus tard
- coder une nouvelle variante de getCompteById() retournant ResponseEntity<CompteDto> ou bien ResponseEntity<ApiEror> avec un code 404/NOT_FOUND selon que la méthode serviceCompte.searchById remonte ou pas une exception de type MyNotFoundException .

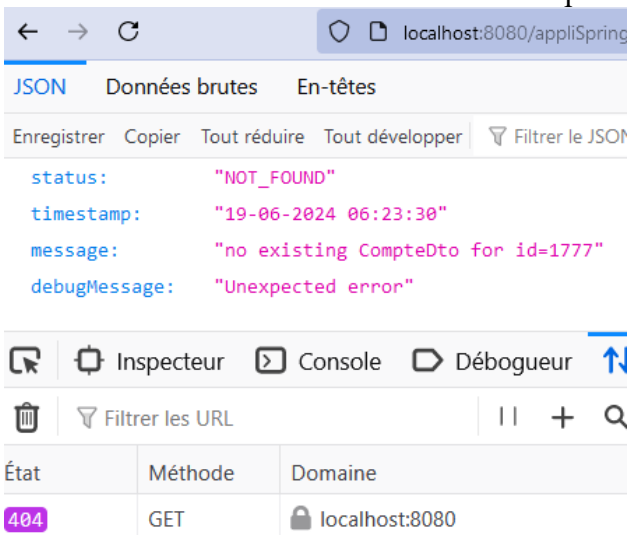
NB : ResponseEntity<?> est un type compatible avec ResponseEntity<CompteDto> et ResponseEntity<ApiEror>

Le retour en cas d'erreur est maintenant beaucoup plus précis (code 404 , message précis au format JSON) :



5.4. Gestion automatique des exceptions

Ajouter ou bien activer une classe gérant automatiquement les exceptions (***@ControllerAdvice*** implémentant l'interface ***ResponseEntityExceptionHandler***)
Restaurer l'ancienne version (simple , sans ResponseEntity< ?>) pour voir si l'automatisme amène au même résultat mais avec un code plus léger et plus explicite du coté valeur de retour .



NB: Pour faire simple en Tp , on désactivera l'annotation //@ControllerAdvice en la mettant en commentaire au dessus de la classe **tp.appliSpringMvc.web.controller.ErrorController** pour éviter un conflit entre les 2 versions (pour thymeleaf ou bien pour api-rest).

En pratique , il est déconseillé de mixer "thymeleaf" et "api-rest" sur un même projet springMvc d'entreprise car c'est assez compliqué de bien faire cohabiter ces 2 aspects sans faille de sécurité.

5.5. intégration de swagger/openApidoc

Ajouter ceci au sein du fichier **pom.xml**

```
<!-- springdoc-openapi-ui for spring5/springBoot2 ,
     springdoc-openapi-starter-webmvc-ui for spring6/springBoot3 -->
        <dependency>
                        <groupId>org.springdoc</groupId>
                        <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
                        <version>2.3.0</version>
        </dependency>
```

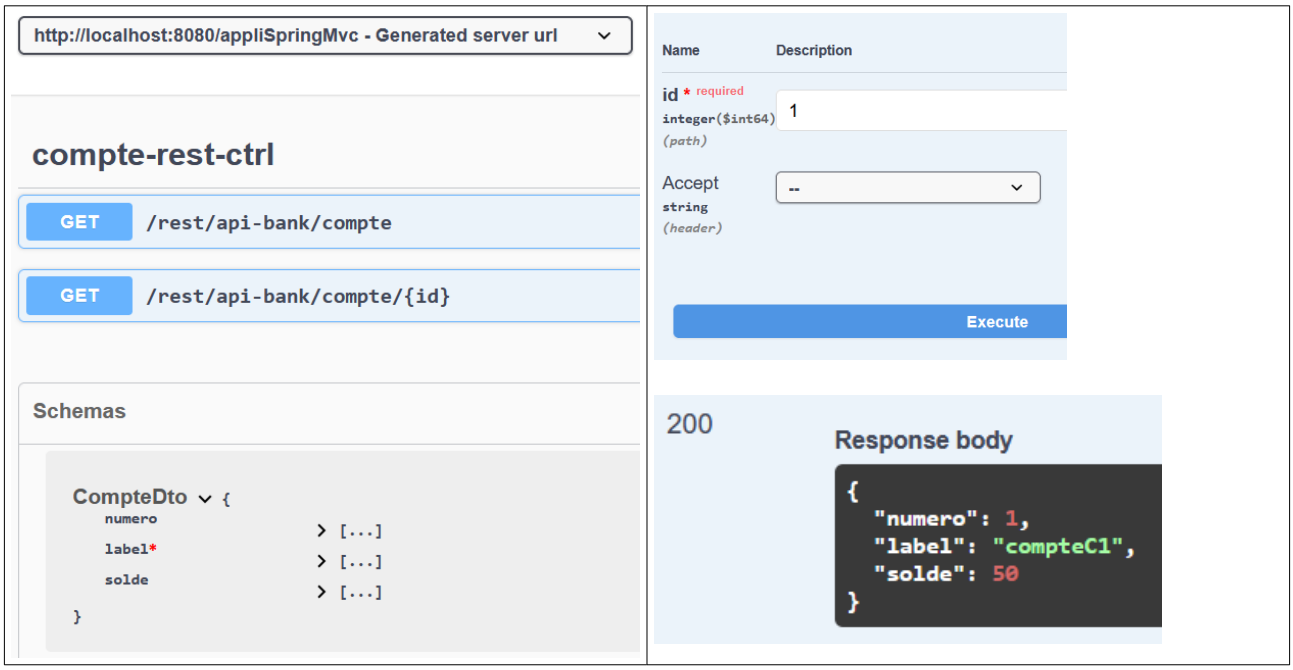
et ceci au sein de **application**-dev**.properties**

```
springdoc.swagger-ui.path=/doc-swagger.html
```

et encore ceci au sein de **static/index-rest.html**

```
<a href="doc-swagger.html">documentation swagger3/openapi</a>
```

En redémarrant l'application et en suivant les liens hypertextes ajoutés on accède à **une documentation** de l'api REST générée automatiquement qui est carrément testable/exécutable via "**try it out**" et "**execute**" :



5.6. Gestion des modes POST, PUT et DELETE

Coder des accès en mode **POST, PUT et DELETE** au sein de la classe **CompteRestCtrl** .
NB : On pourra effectuer des tests avec **postMan** ou bien avec **swagger/openApiDoc** .

Indications :

- on va s'appuyer au maximum sur la gestion automatique des exceptions (via de "exceptionHandler" mis en place) et l'on n'utilisera directement/explicitement ResponseEntity<> que pour une logique supplémentaire aux cas d'exceptions "MyNotFoundException" , "….Exception" .
- De manière à ce que notre api REST puisse être facilement appelée par du code javascript déjà préparé on essaiera de respecter les formats et comportements suivants :

Mode	*URL*	*(e)ntrées et (r)etour*
POST	.../api-bank/compte	**e:**{"numero": null ,"label":"ccc","solde" :50.0} r:{"numero": 786 ,"label":"ccc","solde" :50.0}
PUT	.../api-bank/compte/1 ou 2 ou n ou bien .../api-bank/compte	**e:**{"numero": 786 ,"label":"CCC","solde" :60.0} r:{"numero": 786 ,"label":"CCC","solde" :60.0}
DELETE	.../api-bank/compte/1 ou 2 ou n	r: NO_CONTENT(204) ou NOT-FOUND(404)

NB: @PutMapping({"","/{id}"}) permet d'accepter plusieurs variantes dans les URL (avec @PathVariable(name="id",**required = false**))

NB : La mise en place d'un mini frontEnd javascript du Tp ci-après permettra indirectement de tester le fonctionnement le l'api REST dans tous les modes (GET,POST,PUT et DELETE).

5.7. Exemple d'invocation depuis javascript/html (ajax)

Recopier du projet "debutAppliSpringMvc" vers votre projet la partie "appel_ajax" :

src/main/resources/static
- **compteAjax.html**
- **virementAjax.html**
- **loginAjax.html**
- **js/compteAjax.js**
- **js/virementAjax.js**
- **js/loginAjax.js**
- **js/my_crud_util.js**
- **js/my_ajax_util.js**

On ajoutera un lien hypertexte de ce genre au sein de **index-rest.html**

```
<a href="compteAjax.html">compteAjax (mini frontend html/js appelant api rest)</a>
<hr/>
```

NB : les fichiers réutilisables **js/my_ajax_util.js** et **js/my_crud_util.js** sont codés en pur "javascript" sans aucun framework additionnel (vanilla-js) .

Le fichier essentiel **js/my_ajax_util.js** comporte des fonctions utilitaires permettant d'effectuer des appels http/ajax vers une api REST (avec une logique de callbacks en s'appuyant simplement sur XMLHttpRequest et en n'utilisant pas l'api fetch.

Le fichier **js/my_crud_util.js** comporte une logique comportementale "**crud**" qu'il faut ajuster par un fichier spécifique à la page html tel que js/compteAjax.js .

Le fichier **compteAjax.html** comporte un formulaire en mode CRUD et un tableau affichant tous les comptes recherchés (ayant un certain solde minimum) :

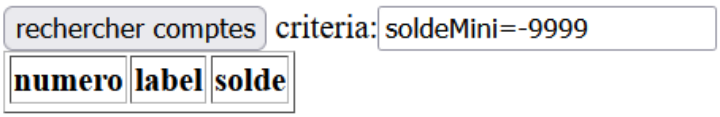
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
<title>compteAjax</title>
<script src="js/my_ajax_util.js"></script>
<script src="js/my_crud_util.js"></script>
<script src="js/compteAjax.js"></script>
</head>
<body>
        numero(id): <input id="inputId" /> <br/>
        label: <input id="inputLabel" /> <br/>
        soldeInitial: <input id="inputSoldeInitial" /> <br/>
        <button id="btnReset">reset</button>  
        <button id="btnAdd">add</button>  
        <button id="btnUpdate">update</button> 
        <button id="btnDelete">delete</button>
        <hr/>
   <span id="spanMsg"></span>
   <hr/>
        <button id="btnRefreshAll">rechercher comptes</button>
        criteria:<input id="inputCriteria" value="soldeMini=-9999" /> <br/>
   <table border="1">
          <thead>
                <tr><th>numero</th><th>label</th><th>solde</th></tr>
          </thead>
          <tbody id="table_body">
                <!-- <tr><td>1</td><td>compteA</td><td>50.0</td></tr> -->
          </tbody>
        </table>
        <hr/>
        <a href="index-rest.html">retour vers index-rest.html</a> <br/>
</body>
</html>
```

Cette partie (récupérée par copier/coller de fichier) a été prévue pour être compatible avec les URLs et le format de notre Api REST .

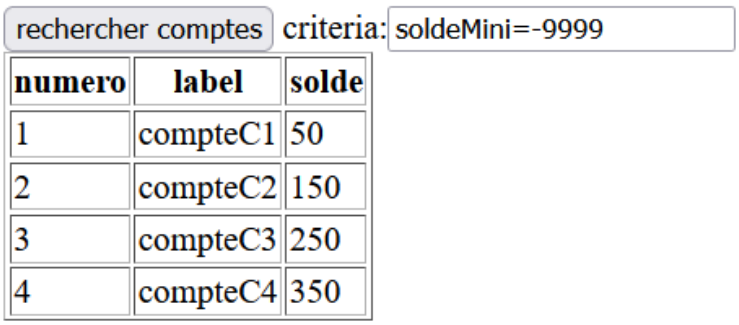
js/compteAjax.js comporte entre autre la fonction suivante :

```
function getWsBaseUrl(){
        return "./rest/api-bank/compte";
}
```

En navigant de **index.html** vers **index-rest.html** puis vers **compteAjax.html** on devrait normalement pouvoir faire fonctionner ce mini frontEnd incorporé à notre application SpringMvc .

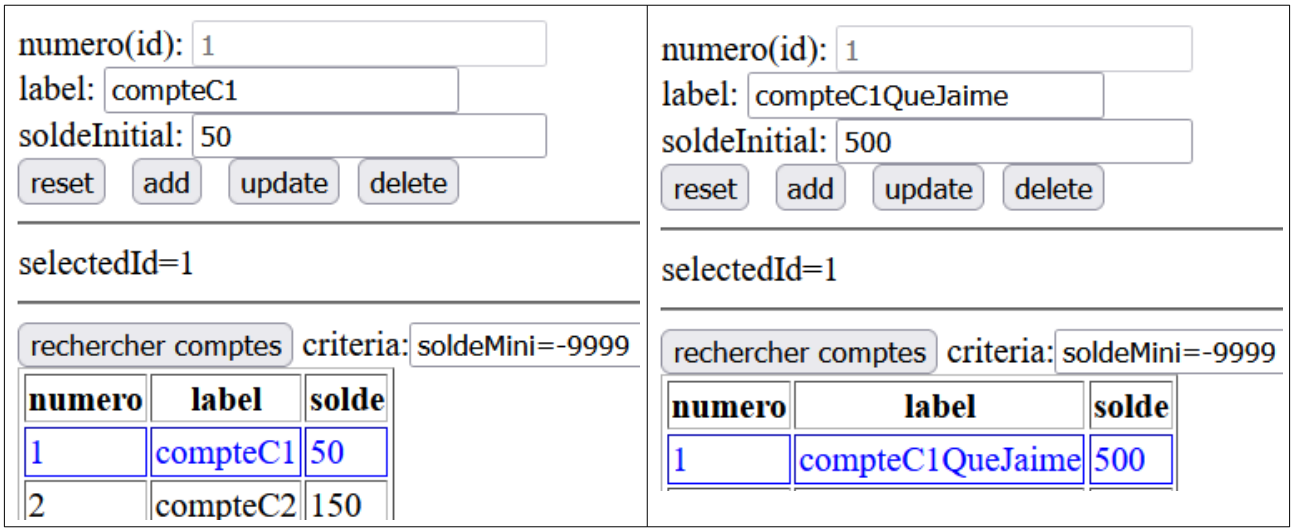


Un click sur le bouton "rechercher comptes" déclenche un appel en mode GET avec "?soldeMini=-9999" :



En cliquant sur une des lignes du tableau , les valeurs de celle-ci s'affichent normalement dans le formulaire du haut de la page.

On peut alors effectuer quelques modifications puis cliquer sur le bouton update qui va déclencher un appel en mode PUT puis un appel en mode GET pour actualiser le tableau.



En cliquant sur le bouton "**reset**" , on peut réinitialiser à vide tous les champs du formulaire de manière à saisir les valeurs d'un nouveau compte que l'on pourra ensuite sauvegarder avec le bouton "**add**" déclenchant un appel en **POST** suivi d'un appel en GET pour réactualiser le tableau .

On pourra sélectionner un nouveau compte (préalablement ajouté dans le tableau) puis déclencher un click sur le bouton "**delete**" pour déclencher un appel en mode **DELETE** suivi d'un appel en mode GET pour réactualiser le tableau .

Attention : certains comptes ne peuvent pas être supprimés car ils sont rattachés à des clients .

5.8. Virement via Api REST (tp facultatif)

Améliorer la méthode **getComptesByCriteria()** de **CompteRestCtrl** de manière à ce que l'on puisse récupérer les comptes d'un client précis via le paramètre optionnel **?customerId=1** en fin d'url

Programmer la nouvelle classe **ViremmentRestCtrl** avec une méthode de ce type

```
public VirementResponse postVirement(@RequestBody VirementRequest virementRequest) {…}
```

Cette méthode sera invoquée avec une url se terminant par **/rest/api-bank/virement**

Son code interne reposera sur un appel à

```
serviceCompte.transfert(virementRequest.getMontant(),
                        virementRequest.getNumCompteDebit(),
                        virementRequest.getNumCompteCredit());
```

Structure conseillée pour **dto.VirementRequest** :

```
public class VirementRequest {
        private Long numCompteDebit;
        private Long numCompteCredit;
        private Double montant;
        //+get/set
}
```

Structure conseillée pour **dto.VirementResponse** :

```
public class VirementResponse {
        private Long numCompteDebit;
        private Long numCompteCredit;
        private Double montant;
        private Boolean status;//true si ok , false si echec
        private String message;//"virement bien effectué" ou "echec"
        //+get/set
}
```

Comportements attendus :

200

Response body

```
{
  "numCompteDebit": 1,
  "numCompteCredit": 2,
  "montant": 3,
  "status": true,
  "message": "virement bien effectué"
}
```

200

Response body

```
{
  "numCompteDebit": 1,
  "numCompteCredit": 2,
  "montant": 3000,
  "status": false,
  "message": "echec virement solde insuffisant sur compte 1"
}
```

Via appels html/ajax :

customerId(id): 1

comptes du client

numero	label	solde
1	compteC1	41
2	compteC2	159

---

virement bien effectué [montant=3 numCompteDebit=1 numCompteCredit=2]

Virement

montant: 3

compte_a_debiter: {"numero":1,"label":"compteC1","solde":41}

compte_a_crediter: {"numero":2,"label":"compteC2","solde":159}

effectuer virement

---

retour vers index-rest.html

5.9. Eventuel démonstration de l'utilité d'une autorisation "CORS"

Selon le temps disponible , le formateur pourra (ou pas) effectuer une petite démonstration sur l'utilité (variable , selon le contexte) de placer

```
@CrossOrigin(origins = "*" , methods = { RequestMethod.GET , RequestMethod.POST})
```

au dessus des classes CompteRestCtrl et VirementRestCtrl

5.10. Petit exemple d'invocation en java moderne (tp facultatif)

En utilisant l'api **httpClient** de java ≥ 11

5.11. Petit exemple de test de web service rest (tp facultatif)

En utilisant les choses proposées par l'écosystème spring

6. TP "Spring-Security" pour "Api REST"

Selon que la série de Tp a été ou pas commencée sur la partie "Spring-mvc avec thymeleaf ou jsp" , la partie "4. Sécurité pour springMvc (jsp ou thymeleaf)" a soit déjà été traitée ou pas.

- Si la sécurité à déjà été mise en place avec thymeleaf, il suffira de la compléter sur la partie "api rest" .
- Si la sécurité n'a encore jamais été mise en place , il faudra (si besoin) effectuer les quelques tâches suivantes rédigées dans la partie "4. Sécurité pour springMvc (jsp ou thymeleaf)" :
  - ajouter **spring-boot-starter-security** dans **pom.xml**
  - ajouter **WithoutSecurityConfig.java** et **SecurityConfig.java** et **MyUserDetailsService.java**
  - pas besoin des autres parties spécifiques à thymeleaf

NB :
Au sein de la méthode **main**() de la classe principale ***AppliSpringMvcApplication***, on activera ou pas la sécurité en activant ou pas le profile "withSecurity' :

```
//System.setProperty("spring.profiles.default", "dev");
System.setProperty("spring.profiles.default", "dev,withSecurity");
```

6.1. Sécurisation via jeton JWT gérés par l'application Spring

Ajouter les dépendances suivantes dans **pom.xml** :

```
        <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt-impl</artifactId>
                <version>0.11.4</version> <!-- et indirectement jjwt-api -->
        </dependency>
        <dependency>
                <groupId>io.jsonwebtoken</groupId>
                <artifactId>jjwt-jackson</artifactId>
                <version>0.11.4</version>
    </dependency>
```

Recopier du projet "***debutAppliSpringMvc***" vers votre projet la partie "code technique pour jetons JWT" : tous les packages qui commencent par "**org.mycontrib.mysecurity**" avec leur contenus (classes et interfaces) .

Parties importantes	***utilités***
...jwt.util.**JwtUtil**	Classe utilitaire pour construire et vérifier des jetons JWT (on s'appuie sur l'api jjwt)
...jwt.util.**JwtTokenProvider**	Comme JwtUtil mais en tant que composant spring injectable et avec des paramétrages ajustables via application.properties et des valeurs par défaut
...jwt.util.**JwtAuthenticationFilter**	Filtre pour vérifier la présence d'un jeton JWT valide au sein d'une requête HTTP entrante
...mysecurity.standalone.rest.**LoginRestCtrl**	WS Rest pour la phase de login (vérif password et génération de jeton JWT retourné dans réponse

	JSON)

NB : toutes ces parties sont réutilisables et l'on pourraient les placer dans un sous projet (librairie).

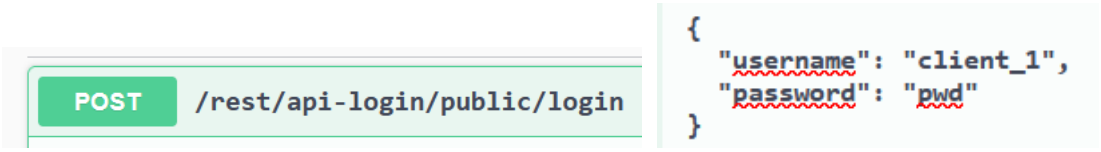
Ajouter la configuration fondamentale suivante au sein de **tp.appliSpringMvc.SecurityConfig**

```
...
@Bean
@Order(1)
protected SecurityFilterChain restFilterChain(
        HttpSecurity http,JwtAuthenticationFilter jwtAuthenticationFilter) throws Exception {
   return http.securityMatcher("/rest/**")
       .authorizeHttpRequests(
           auth -> auth.requestMatchers("/rest/api-login/public/login").permitAll()
                       .requestMatchers(HttpMethod.GET,"/rest/api-bank/compte/**").permitAll()
                       .requestMatchers("/rest/**").authenticated()
                                   )
        //enable CORS (avec @CrossOrigin sur class @RestController)
        .cors( Customizer.withDefaults())

        .csrf( csrf -> csrf.disable() )

        // If the user is not authenticated, returns 401
        .exceptionHandling(eh -> eh.authenticationEntryPoint(getRestAuthenticationEntryPoint()))

        // This is a stateless application, disable sessions
        sessionManagement(
              sM -> sM.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

        // Custom filter for authenticating users using tokens
        .addFilterBefore(jwtAuthenticationFilter,  UsernamePasswordAuthenticationFilter.class)
         .build();
}

private AuthenticationEntryPoint getRestAuthenticationEntryPoint() {
        return new HttpStatusEntryPoint(HttpStatus.UNAUTHORIZED);
}

//explicit config of AuthenticationManager from ProviderManager and list of AuthenticationProvider
@Bean
public AuthenticationManager authenticationManager(
    List<AuthenticationProvider> authenticationProviders) {
           return new ProviderManager(authenticationProviders);
}

//explicit config of (Dao)AuthenticationManager from UserDetailsService (in MyUserDetailsService)
@Bean
public DaoAuthenticationProvider authenticationProvider(UserDetailsService userDetailsService,
PasswordEncoder encoder) {
           DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
           authProvider.setUserDetailsService(userDetailsService);
           authProvider.setPasswordEncoder(encoder);
           return authProvider;
}
```

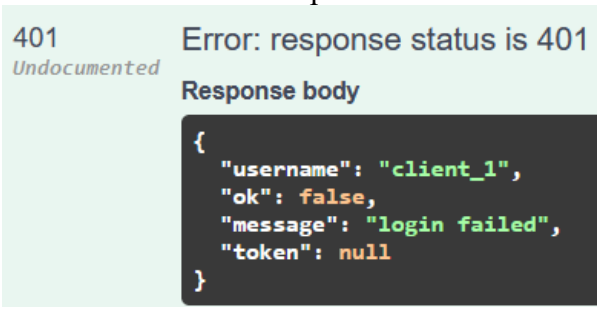
Ajouter ces annotations au dessus de la classe SecurityConfig :

```
@ComponentScan(basePackages = {"org.mycontrib.mysecurity"})
@EnableMethodSecurity()//pour futur interprétation de @PreAuthorize("hasRole('ADMIN')")
public class SecurityConfig {...}
```

En redémarrant l'application (avec les profils "dev" et "**withSecurity**") ,
on devrait normalement pouvoir faire fonctionner le WS de login qui produit des jetons JWT :



et avec mauvais mot de passe :



Selon le temps disponible , le formateur pourra (ou pas) effectuer une petite démonstration de l'utilité de récupérer et retransmettre un jeton JWT en utilisant **postMan** pour tester l'invocation du webService de login et du web service de modification d'un compte (en mode post).

En intégrant la partie loginAjax dans **index-rest.html**

```
<a href="loginAjax.html">loginAjax (avec spring-security et JWT)</a>  <br/>
```

On peut tester la sécurité via le frontEnd javascript/ajax :

login (post) pour obtenir token

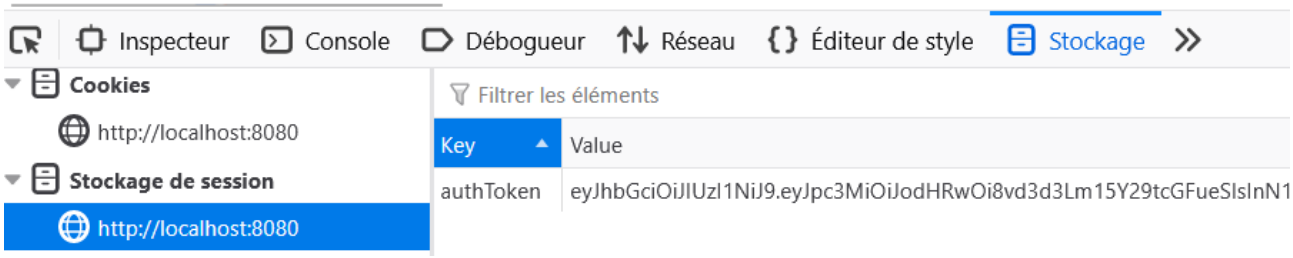
username: client_1
ex: user1/pwd1 with ROLE_USER
admin1/pwd1 with ROLE_ADMIN
client_n/pwd with ROLE_CUSTOMER (bank)
password: pwd (ex: pwd,pwd1,pwd2,...)
login logout
message : **reponse login={"username":"client_1","ok":true,"message":"successful login","token":"eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vd3d3Lm15Y29tcGFueSIsInN**

---

current token : **{"iss":"http://www.mycompany","sub":"client_1","iat":1718891153,"authorities":"[ROLE_CUSTOMER, ROLE_USER]","exp":1718892953}**

avec **sessionStorage.setItem("authToken",jwtToken);** au sein de **loginAjax.js** .

Ceci a pour effet de stoker le jeton jwt dans le "sessionStorage" du navigateur

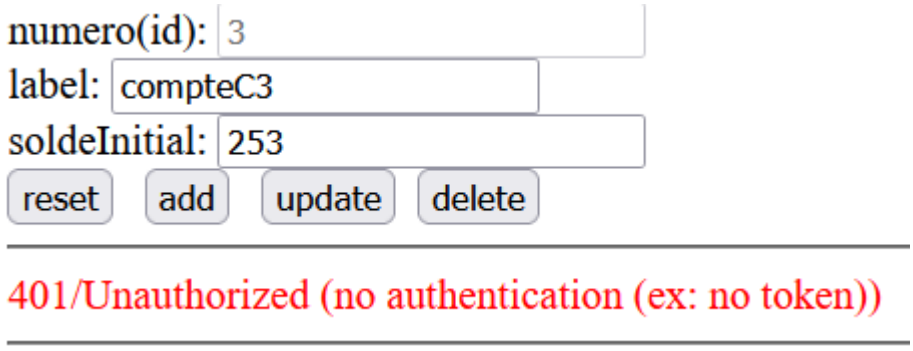


La sous fonction suivante de **my_ajax_util.js** récupère et retransmet ce jeton au sein de toutes les requêtes HTTP/ajax :

```
function setTokenInRequestHeader(xhr){
        let authToken = sessionStorage.getItem("authToken");
        if(authToken!=null && authToken!="")
          xhr.setRequestHeader("Authorization", "Bearer " + authToken);
}
```

Par la suite , lorsque le backend appliSpringMvc reçoit une requête avec un début d'url commençant par "/rest/…" le filtre technique **JwtAuthenticationFilter** va récupérer et vérifier la valididé de ce jeton et l'on obtiendra le comportement suivant :

- 200/OK si tout est ok
- 401/UNAUTHORIZED si l'on a oublié la phase de login (pas de jeton)
- 403/FORBIDDEN si le jeton ne comporte pas un rôle suffisant (ex : manque ROLE_CUSTOMER).



dans compteAjax.html apres logout + login en échec (mauvais mot de passe).

Il faut vérifier la présence de **@EnableMethodSecurity()** ou d'un équivalent sur la classe ***SecurityConfig***

et il faut ajouter

```
@PreAuthorize("hasRole('ROLE_CUSTOMER') || hasRole('ROLE_ADMIN')")
```

au dessus de certaines méthodes telles que ***CompteRestCtrl.putCompte()***

pour obtenir quelquefois le résultat suivant : **403/Forbidden** .