Spring, Webservices et REST

GLG 203/Architectures Logicielles Java

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2019–2020

Démonstrations

https://gitlab.cnam.fr/gitlab/glg203_204_demos/10_spring_webservices.git

Communication entre logiciels

- non portable : RPC, RMI ;
- portable : Corba, Web Services, REST ;
- découplée/asynchrone : Message Oriented Middleware (JMS et al.).

Ce cours : Web Service et REST.

Utilisation

- Communication entre applications ;
- Communication entre un front-end Javascript et un back-end Java (ou autre).

Web Services

Web Service

Définition

- Un objet (une façade) dont on peut appeler les méthodes à travers les protocoles du Web ;
- Les clients verront typiquement le web service comme un objet normal, à travers un Proxy sur lequel ils appelleront les méthodes voulues

Intérêt des Web Services

- beaucoup d'autres mécanismes :
- RPC, RMI ;
- EJB proposent un mécanisme sécurisé...

pourquoi les web services ?

- indépendants des langages
- coût d'entrée faible (protocole HTTP)
- les firewalls laissent souvent passer les requêtes HTTP...

Types de Web service

- Web services « lourds » : historiquement, les premiers ;
- Auto-documentés et bien normalisés ;
- Le web service publie un schéma XML qui détaille les appels possibles (fichier WSDL) ;
- Web services REST : se reposent sur la sémantique du protocole HTTP. très souples
- plus ad-hoc que les Web services lourds

Web services SOAP

Principes de SOAP

- Recommandation du W3C (origine : IBM et Microsoft) ;
- Requêtes et réponses passent sur le protocole HTTP(S) ;
- les données sont passée au format XML ;
- le format des données et les méthodes sont décrites par un fichier XML (WSDL) ;
- **ce fichier est lui-même publié par le Web Service ;**
- sécurisation possible : `WS-Security` ;
- mécanisme d'encapsulation des exceptions.

Fichier WSDL

Définition

Web Services Description Language : fichier XML décrivant un web services :

Il contient des zones :

- **types** types de données utilisés (schéma xsd) ;
- **message** paramètres et valeurs de retours utilisés par les fonctions ;
- **portType** méthodes fournies (noms et arguments) ;
- **binding** mode de sérialisation des arguments et autres détails d'appels
- **service** donne l'adresse des ports

WSDL : pourquoi cette modularité ?

- Possibilité de définir des standard (types/messages/portType) et de proposer des implémentations différentes
- Possibilité de définir un standard « métier » partagé, sans préjuger de ses implémentations

WSDL (récapitulation simplifiée)

```
<definitions name="Calculateur">
    <types>    <!-- types utilisés (optionnel) -->
        <xsd:schema>...</xsd:schema>
    </types>
    <!-- définition des données utilisés
         (référence les définitions de la partie "type" -->
    <message name="sommeHexa">…</message>
    <message name="sommeHexaResponse">…</message>…

    <!-- définition des opérations -->
    <portType name="Calculateur">...</portType>
    <!-- format de données et protocole pour chaque "portType" -->
    <!-- en gros, comment est codée, et
         comment est appelée, chaque méthode définie au dessus -->
    <binding name="CalculateurPortBinding" type="tns:Calculateur">
     ...
    </binding>
    <service name="Calculateur">...</service>
</definitions>
```

Fichier WSDL (vision détaillée)

Partie service

```
<wsdl:service name="PublicationsPortService">
  <wsdl:port binding="tns:PublicationsPortSoap11"
             name="PublicationsPortSoap11">
    <soap:address location="http://localhost:8080/ws"/>
  </wsdl:port>
</wsdl:service>
```

- lie le service `PublicationsPortService` à l'URL `http://localhost:8080/ws` ;
- le détail de la communication est décrit par le « binding » `PublicationsPortSoap11`

WSDL : binding

décrit comment sont transportées les informations

```
<wsdl:binding name="PublicationsPortSoap11" type="tns:PublicationsPort">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="listePublications">
      <soap:operation soapAction=""/>
      <wsdl:input name="listePublicationsRequest">
         <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="listePublicationsResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="creerPublication">
       <soap:operation soapAction=""/>
       <wsdl:input name="creerPublicationRequest">
         <soap:body use="literal"/>
       </wsdl:input>
    </wsdl:operation>
</wsdl:binding>
```

WSDL : PortType

Définition effective des méthodes

Lie nom de l'opération, message d'entrée et message de sortie.

```
<wsdl:portType name="PublicationsPort">
  <wsdl:operation name="listePublications">
    <wsdl:input message="tns:listePublicationsRequest"
                name="listePublicationsRequest"></wsdl:input>
    <wsdl:output message="tns:listePublicationsResponse"
                name="listePublicationsResponse"></wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="creerPublication">
    <wsdl:input message="tns:creerPublicationRequest"
                name="creerPublicationRequest"></wsdl:input>
  </wsdl:operation>
</wsdl:portType>
```

WSDL : messages

description des arguments

- Un message est un ensemble d'argument (d'entrée ou de sortie) ;
- le détail des message est décrit par les types.

```
<wsdl:message name="listePublicationsRequest">
  <wsdl:part element="tns:listePublicationsRequest"
        name="listePublicationsRequest"></wsdl:part>
</wsdl:message>
<wsdl:message name="listePublicationsResponse">
  <wsdl:part element="tns:listePublicationsResponse"
        name="listePublicationsResponse"></wsdl:part>
</wsdl:message>
<wsdl:message name="creerPublicationRequest">
  <wsdl:part element="tns:creerPublicationRequest"
        name="creerPublicationRequest"></wsdl:part>
</wsdl:message>
```

WSDL : types

Définition effective des données (schéma xsd)

```
<wsdl:types>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        targetNamespace="http://glg203.cnam.fr/publications">
  <element name="creerPublicationRequest">
    <complexType>
      <sequence>
        <element ref="tns:publication"/>
     </sequence>
    </complexType>
  </element>
...
  <complexType name="publications">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0"
               name="publications" type="tns:publication"/>
    </sequence>
  </complexType>
</schema>
</wsdl:types>
```

Côté serveur, Java EE

- nous allons voir d'abord la définition des Web Services en Java EE
- le code qui suit ne fonctionne pas en Spring !

- En Java EE, définition asssez simple
- déployé dans un war, un EJB, ou défini en Spring sur un serveur applicatif ;
- classe annotée avec @WebService (pas en Spring) ;
- en JEE : par défaut, toutes les méthodes publiques sont accessibles à distance On peut aussi définir une interface qui décrit les méthodes exportées
- Découverte possible des services par Annuaires UDDI ;
- outils pour créer le WSDL à partir des classes Java.

Exemple simple/Java EE

```
/**
 * Exemple simple de Web service.
 * <ul>
 * <li> adresse:http://localhost:8080/WebService1/Calculateur</li>
 * <li> wsdl:http://localhost:8080/WebService1/Calculateur?wsdl</li>
 * <li> test:http://localhost:8080/WebService1/Calculateur?Tester</li>
 * </ul>
 */
@WebService(serviceName = "Calculateur")
public class Calculateur {

 public int somme(int a, int b) {
   return a+b;
 }

 public String sommeHexa(@WebParam(name = "v1")int a,
                         @WebParam(name="v2")int b) {
   return Integer.toHexString(a+b).toUpperCase();
 }
}
```

Exemple

- Le web service précédent, déployé dans l'application WebService1, sera accessible à l'adresse :
  `http://localhost:8080/WebService1/Calculateur`
- On peut consulter sa définition WSDL à l'adresse
  http ://localhost :8080/WebService1/CalculateurBis ?wsdl
- On peut voir les types utilisés par le WSDL à l'adresse
- http ://localhost :8080/WebService1/CalculateurBisService ?xsd=1
- Enfin, on peut tester interactivement le service à l'adresse
  http ://localhost :8080/WebService1/CalculateurBisService ?Tester

Exemple avec une interface

```
@WebService
public interface CalculateurTerInterface {
    public int somme(int a, int b);
}
```

```
@WebService(
endpointInterface = "glg203.webservice.CalculateurTerInterface")
public class CalculateurTer implements CalculateurTerInterface{
    @Override
    public int somme(int a, int b) {
        return a+b;
    }

    public String sommeHexa(int a, int b) {
        return Integer.toHexString(a+b).toUpperCase();
    }
}
```

- on implémente l'interface et précise « endpointInterface » !!
- seule cette méthode est publiée.

Annotation `@WebService`

- en JEE, déclare la classe ou l'interface comme un WebService ;
- Attributs importants :

  name nom du service. Si pas précisé, NomClasse+ « `Service` »
  endpointInterface interface qui définit les méthodes publiées ;
  targetNameSpace définit le nameSpace XML des fichiers WSDL (généralement pas utile)

Web Services en Spring

- sens : spécifications $\rightarrow$ classes java ;
- on définit au moins la structure des données `xsd` ;
- les classes java peuvent être générées à partir de là.

Java 11 et plus

Suite à la suppression de Jaxb des bibliothèques standards Java, j'ai dû mettre à jour les exemples, qui ne compilaient plus.

Définition d'un serveur SOAP

voir projet webservice

Dans le `build.gradle` :

```
implementation
  'org.springframework.boot:spring-boot-starter-web-services'
implementation "wsdl4j:wsdl4j:1.6.3"
// Pour Jaxb, à partir de Java 11 (supprimé des bibliothèques standard)
implementation "javax.xml.bind:jaxb-api:2.3.1" // l'API
implementation "org.glassfish.jaxb:jaxb-runtime:2.3.1" // implémentation
```

- … plus code pour la génération des classes java ;
- utilise `xjc`, dans une tâche `ant` pour l'instant.

Schéma xsd

- décrit les types utilisés ;
- pour chaque méthode *M* du web service :
  - définir un `element` *MRequest* qui décrit les arguments en entrée de la méthode ;
  - définir un `element` *MResponse* qui décrit la valeur de retour de la méthode.
  - bien utiliser `Request` et `Response` en fin de nom ;
  - bien distinguer ces listes de leur *contenu* - décrit par d'autres éléments.

En-tête du fichier xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://glg203.cnam.fr/publications"
xmlns:tns="http://glg203.cnam.fr/publications"
        elementFormDefault="qualified">
```

targetNamespace défini le namespace cible comme `http://glg203.cnam.fr/publications` ;

elementFormDefault les entités définies appartiendront au namespace cible (sans ça, il faut le préciser pour chaque élément) ;

xmlns :tns le namespace `http://glg203.cnam.fr/publications` aura le préfixe `tns` si nous en avons besoin ;

le fichier est typiquement placé dans `src/resources` ; le nom du fichier xsd sera aussi celui du fichier wsdl produit.

Arguments d'une procédure

```
<element name="creerPublicationRequest">
  <complexType>
    <sequence>
       <element name="maPublication" type="tns:publication"/>
    </sequence>
  </complexType>
</element>
```

- défini comme un `element` dont le nom se termine par `Request` ;
- contenu : typiquement une `sequence` (ici d'un élément !) ;
- les éléments de la séquence sont décrits par la suite (par des déclarations `element`)
- le `name` « `maPublication` » est le nom du champ, pas son type !
- `ref="tns:publication"` : l'élément nommé `publication` dans le namespace `tns` (défini comme le namespace de notre fichier)

Arguments d'une fonction

```
<element name="listePublicationsRequest">
  <complexType/>
</element>
<element name="listePublicationsResponse">
  <complexType>
    <sequence>
      <element name="liste" type="tns:publications"/>
      <!-- <element ref="tns:publications"/> -->
    </sequence>
  </complexType>
</element>
```

- On définit à la fois les arguments et la réponse ;
- dans tous les cas comme un élément ;
- bien comprendre que le type de retour listePublicationsResponse, a priori, a une existence propre — ne pas le confondre avec son contenu.
- deux variantes sont montrées : en donnant le type et le nom de l'élément, ou en utilisant un élément déjà défini
  - ça se traduit dans le xml (éléments xml) ;
  - et dans les classes java (noms des champs).

Types pour les « vraies » classes

```
<complexType name="publication">
 <sequence>
  <element name="titre">
   <simpleType>
    <restriction base="string">
     <minLength value="1"></minLength>
    </restriction>
   </simpleType>
  </element>
  <element name="contenu" type="string"></element>
 </sequence>
</complexType>

<complexType name="publications">
 <sequence>
  <element name="publications"
     type="tns:publication"
      minOccurs="0" maxOccurs="unbounded">
  </element>
 </sequence>
</complexType>
```

Classes java engendrées

- CreerPublicationRequest.java
- ListePublicationsRequest.java
- ListePublicationsResponse.java
- Publication.java
- Publications.java
- ObjectFactory.java : classe utilitaire (Factory)

Configuration du service

```
@EnableWs // Active les Web Services
@Configuration
public class WebServiceConfiguration /*extends WsConfigurerAdapter*/ {

  @Bean
  public ServletRegistrationBean<MessageDispatcherServlet>
         messageDispatcherServlet(ApplicationContext applicationContext){
    MessageDispatcherServlet servlet = new MessageDispatcherServlet();
    servlet.setApplicationContext(applicationContext);
    servlet.setTransformWsdlLocations(true);
    return new ServletRegistrationBean<>(servlet, "/ws/*");
  }

  ...
}
```

- Met en place une servlet qui va gérer les web services ;
- son nom (messageDispatcherServlet) permet de ne pas la confondre avec le front-end de Spring MVC ;
- cette servlet détectera et utilisera tous les beans Spring de type `WsdlDefinition`

```java
public class WebServiceConfiguration /*extends WsConfigurerAdapter*/ {
 ...
 // Définit un schéma à utiliser :
 @Bean
 public XsdSchema publicationsSchema() {
  return new SimpleXsdSchema(new ClassPathResource("publications.xsd"));
 }

 // Définit le web service publications
 @Bean(name = "publications")
 public DefaultWsdl11Definition publications(XsdSchema publicationSchema)
  DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition()
  wsdl11Definition.setPortTypeName("PublicationsPort");
  wsdl11Definition.setLocationUri("/ws");
  wsdl11Definition.setTargetNamespace(
                              "http://glg203.cnam.fr/publications");
  wsdl11Definition.setSchema(publicationSchema);
  return wsdl11Definition;
 }
}
```

- Définit le service publication ;
- le fichier `wsdl` sera disponible à `http://localhost/ws/publications.wsdl`

Implémentation

```
@Endpoint
public class PublicationFacade {
 static final String NS = "http://glg203.cnam.fr/publications";

 @Autowired
 PublicationService service;

 @PayloadRoot(namespace = NS, localPart = "creerPublicationRequest")
 public void ajouterPublication(
    @RequestPayload CreerPublicationRequest creerPublicationRequest) {
   service.add(creerPublicationRequest.getMaPublication());
 }
}
```

- classe annotée par `@Endpoint` ;
- `@PayloadRoot` définit une méthode du web service ;
- notez que les types des arguments sont les types `...Request` (et `...Response`). Ici, `CreerPublicationRequest` ;
- l'argument est annoté par `@RequestPayload.` ;
- le nom de l'argument et celui de la méthode coïncident.

Implémentation

```
@PayloadRoot(namespace = NS, localPart = "listePublicationsRequest")
public @ResponsePayload ListePublicationsResponse listePublications() {
  Publications publications = new Publications();
  publications.getPublications().addAll(service.getPublications());
  ListePublicationsResponse response = new ListePublicationsResponse();
  response.setListe(publications);
  return response;
}
```

- Pour une fonction, la réponse est annotée par `@ResponsePayload` ;
- les données (requête et réponse) seront sérialisées en XML avec JAXB.

Client de Web Service en Spring

Voir projet `websoapclient`

- il faut créer les classes DTO à l'aide du fichier wsdl ;
- ... et correctement paramétrer l'application.

Dans build.gradle :

```
dependencies {
 // On supprime la partie serveur des web-services (pas de serveur)
 implementation (
   'org.springframework.boot:spring-boot-starter-web-services') {
   exclude group: 'org.springframework.boot',
           module: 'spring-boot-starter-tomcat'
 }
 implementation 'org.springframework.ws:spring-ws-core'
 jaxb("org.glassfish.jaxb:jaxb-xjc:2.3.2") // Génération des classes
 compile(files(genJaxb.classesDir).builtBy(genJaxb))
}
```

(génération jaxb par une tâche ant)

Appel des web services

Globalement : on crée l'objet Request correspondant et on l'envoie.

```
public class PublicationsClient extends WebServiceGatewaySupport {

 public ListePublicationsResponse listePublications() {
  ListePublicationsRequest req = new ListePublicationsRequest();
  ListePublicationsResponse resp = (ListePublicationsResponse)(
           getWebServiceTemplate().
                marshalSendAndReceive("http://localhost:8080/ws", req,
                                      new SoapActionCallback("")));
  // SoapActionCallback : header SOAPAction:
  // "used to indicate the intent of the SOAP HTTP request." Obligatoire.
  return resp;
 }

 public void creerPublication(String titre, String contenu) {
  CreerPublicationRequest req= new CreerPublicationRequest();
  Publication publication = new Publication();
  publication.setTitre(titre);
  publication.setContenu(contenu);
  req.setMaPublication(publication);
  getWebServiceTemplate().
    marshalSendAndReceive("http://localhost:8080/ws", req,
    new SoapActionCallback(""));
 }
```

Conclusion

- Web services plus transparents en JEE qu'en Spring ;
- Spécifications riches (ex. sécurité WS-Security) ;
- auto-documenté (dans une certaine mesure) ;
- Noter que les classes des arguments sont essentiellement des DTO (même en JEE, au moins côté client).

REST

Web Services REST

Définition

REST (REpresentational State Transfer) : architecture définie dans la thèse de Roy Fielding

- plus un « style » qu'un protocole ;
- Idée : les données sont vues comme des ressources, auxquelles sont associées des URL
- exemple d'un carnet d'adresse :

  http ://localhost :8080/RestExemple/rest/personne toutes les personnes dans le carnet

  http ://localhost :8080/RestExemple/rest/personne/1 la personne numéro 1 dans le carnet
- le terme est souvent utilisé de manière très large.

Richardson Maturity Model

Niveau 0 utilisation de HTTP comme mécanisme générique de « remote procedure call » - généralement avec JSON ou XML ;

Niveau 1 ressources : une URL est maintenant associée à une ressource ; *les actions sur cette ressources sont toutes dirigée sur cette URL* ;

Niveau 2 utilisation des verbes HTTP : GET, POST, DELETE...sont utilisés sur les URL précédentes de manière logique ;

Niveau 3 « Hypermedia Controls « : les données renvoyées contiennent des liens qui explicitent les actions possibles (HATEOAS)

- Le « vrai » REST est le niveau 3 ;
- Spring fournit des outils variés pour tous les niveaux.

(cf. site de Martin Fowler)

Web Services REST

Les méthodes du protocole HTTP (GET, POST, PUSH...) se voient associer une sémantique :

- GET sur une URL : récupère les données associées à cette URL, dans un format qui peut être XML ou JSON
- POST : crée une nouvelle ressource (on renvoie généralement la nouvelle ressource ou son id/son url)
- PUT : met à jour une ressource existante (ou crée une nouvelle ressource, mais à une adresse connue)
- DELETE : détruit une ressource

Exemple

Soit l'URL `http://localhost:8080/RestExemple/api/contact`
La ressource : les contacts.

- `GET http://localhost:8080/RestExemple/api/contact` : récupère tous les contacts (éventuellement paginés) ;
- `GET http://localhost:8080/RestExemple/api/contact/3` : récupère les données du contact d'id 3 ;
- `POST http://localhost:8080/RestExemple/api/contact` : crée un nouveau contact (ses données sont expédiées en JSON comme corps de la requête)
- `PUT http://localhost:8080/RestExemple/api/contact/3` : met à jour le contact numéro 3
- `DELETE http://localhost:8080/RestExemple/api/contact/3` : détruit le contact numéro 3.

Exemple "à la main" avec GET

```
telnet localhost 8080
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GET http://localhost:8080/RestExemple/api/contact
Accept: application/json

HTTP/1.1 200 OK
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server)
Server: GlassFish Server Open Source Edition  4.0
Content-Length: 490

[{"adresse":"Oxford","id":1,"nom":"Turing","prenom":"Alan",
  "telephones":[{"numero":"0656345367","typeTelephone":"MOBILE"}]}
 {"adresse":"Oxford","id":51,"nom":"Toto","prenom":"TOTO",
  "telephones":[]},
]
```

Note importante

Attention !

Ça n'est pas parce qu'une API n'est pas « vraiment » REST qu'elle est mauvaise.
Il est parfaitement légitime d'utiliser REST comme on utilise SOAP.

HATEOAS n'est donc utile que si on veut profiter des bénéfices d'un REST complet :

- auto-documentation, exploration possible des actions autorisées par le client ;
- éventuel modèle stateless - potentiellement plus efficace.
- au prix d'un peu plus de travail (mais voir Spring data HATEOAS)

HATEOAS

Définition

Hypermedia As The Engine of Application State : les réponses sont accompagnées de la liste des actions possibles.
Elles représentent donc l'état de l'application.

- En théorie, le REST de niveau 3 est *sans état* côté serveur.
- pas de formalisation universelle de HATEOAS ;
- souvent `hal` (`application/hal+json`) pour la structure des liens ;
- et ALPS pour décrire la structure des données.

Requête sur /contact


```
{"elements" :
  [{"nom" : "n1",
    "_links" : {
      "self" : {"href" : "http://localhost:8080/contacts/5"},
      "coordonnees" :
         {"href" : "http://localhost:8080/contacts/5/coordonnees"}
      }
   },
   {"nom" : "n2"...}],
 "_links" : {
    "self" : {
      "href" : "http://localhost:8080/contacts{?page,size,sort,projection}
      "templated" : true
    },
    "search" : {
      "href" : "http://localhost:8080/contacts/search"
    }
    "next" : {
      "href" : "http://localhost:8080/contacts?page=1&size=10"
    },
  },
  "page" : {"size" : 10, "totalElements" : 2,
            "totalPages" : 1, "number" : 0}
}
```

HAL : Hypertext Application Language

Propose une structure pour les ressources retournées par un REST/Hateoas :

- des propriétés simples ;
- la propriété `"_link"` : des liens vers d'autres ressources ;
- `"_self"` : lien vers la ressource elle-même ;
- `"_embedded"` : permet de manière optionnelle de donner le contenu (éventuellement partiel) d'autres ressources.

Voir `https://tools.ietf.org/html/draft-kelly-json-hal-08`

REST et Spring, niveau 0 et 1

- Spring Web MVC fournit les outils nécessaires grâce à `@ResponseBody` ;
- Quand une méthode d'un contrôleur est marquée par `@ResponseBody`, l'objet retourné est converti en objet JSON ou XML (si c'est possible) et expédié ;
- l'annotation `@RestController` équivaut à `@Controller` et à `@ResponseBody`

Exemple simple

```
@RestController
@RequestMapping("/api/contact")
public class ContactController {
    @Autowired ContactService service;
    @GetMapping
    public ArrayList<Contact> findAll() {
        return service.findAll();
    }

    @GetMapping("/{id}")
    public Contact get(@PathVariable int id) {
        return service.getContact(id);
    }

    @PostMapping
    @ResponseBody
    public Contact post(@RequestBody Contact contact) {
        service.addContact(contact);
        return contact;
    }
}
```

ResponseEntity

Le type générique `ResponseEntity` permet de retourner un objet java, comme `@ResponseBody`, mais aussi de fournir, si besoin un code de retour et de paramétrer les headers http.
Exemple : retour d'une erreur 404 :

```
@GetMapping("/{id}")
public ResponseEntity<Contact> get(@PathVariable int id) {
  return service.getContact(id)
            .map(contact -> ResponseEntity.ok(contact))
            .orElse(ResponseEntity.notFound().build());
   }
```

`ResponseEntity` fournit un **builder** pour construire les réponses, en fixant les codes d'erreur si nécessaire.

Interaction

Liste des contacts :

```
curl http://localhost:8080/api/contact
```

Création d'un nouveau contact

```
TYP="Content-type: application/json"
D='{"nom" : "toto", "telephone" : "truc"}'
curl -X POST -H "$TYP"  -d "$D" http://localhost:8080/api/contact
```

Annotations et types utiles

- `@ResponseStatus` : annote une méthode ou une exception, et permet de fixer le code d'erreur HTTP associé (`OK`, `NOT_FOUND` par exemple) ;
-

Client Rest en Spring

Voir projet `05_restclient`

- On interroge le serveur en passant par l'une des classes `RestTemplate` ou `WebClient` ;
- difficulté principale liée à la sérialisation (et surtout la désérialisation) JSON

Dans l'exemple, notre client n'est pas un serveur Web (il pourrait l'être). On désactive donc tomcat :

```
dependencies {
 // on a besoin de spring-boot-starter-web, mais sans serveur !
 implementation ('org.springframework.boot:spring-boot-starter-web') {
    exclude group: 'org.springframework.boot',
            module: 'spring-boot-starter-tomcat'
 }
}
```

RestTemplate

- On injecte un objet `RestTemplate` dans l'application ;
- il sert à envoyer des requêtes (GET, POST, etc...) ;
- les données sont matérialisées par des classes sérialisées par Jackson ;
- il peut donc être utile d'avoir les mêmes classes côté client et côté serveur ;
- le RestTemplate est créé par un `RestTemplateBuilder` ;
- adaptations : voir référence Spring boot, § 35 :
  - on adapte localement le `RestTemplateBuilder` injecté ; c'est foncièrement un objet immutable : les méthodes sont fonctionnelles et créent de nouveaux `RestTemplateBuilder`
  - on peut déclarer un bean `RestTemplateCustomizer` qui sera automatiquent utilisé par `RestTemplateBuilder`.
- permet de configurer la sérialisation, la connexion par `basic authentication`, etc.

Usage simple de RestTemplate

```
String url = "http://localhost:8080/api/contact/0";
Contact contact = restTemplate.getForObject(url, Contact.class);
```

Autres méthodes :

- `delete`
- `getForEntity`
- `postForObject`
- `postForEntity`
- `put`

...forObject : récupère un objet (dont on passe la classe) à partir d'une sérialisation ;

...forEntity : récupère un `ResponseEntity<T>` qui contient un objet (le body), et toutes les données du header http : status, content-type...

Récupération de types génériques

- En fait, plus un problème Jackson qu'un problème de REST ;
- le problème : l'effacement des génériques ;
- la solution : « figer » le paramètre en définissant une nouvelle classe (anonyme).
- ParameterizedTypeReference est une classe anonyme et générique ;
- en en créant une sous-classe (même anonyme), pour un paramètre générique donné, on résoud le problème de l'effacement.

```
ParameterizedTypeReference<ArrayList<Contact>> responseType;
// Attention à la ligne suivante !!
// Création d'une classe anonyme (notez les  !)
responseType = new ParameterizedTypeReference<ArrayList<Contact>>(){};

// Exchange est une rest template à tout faire.
String url = "http://localhost:8080/api/contact/";
ResponseEntity<ArrayList<Contact>> res = restTemplate.exchange(
     url, HttpMethod.GET, null, responseType);
for (Contact c: res.getBody()) {System.out.println(c);}
```

WebClient

- La classe `WebClient` permet d'exécuter des requêtes de manière bloquante ou asynchrone ;
- elle utilise le pattern **builder** pour construire et exécuter les requêtes.
- documentation complète dans la documentation de Spring Web Reactive ;
- on peut créer un `WebClient`, le paramétrer, et le réutiliser...

Requête simple

```
WebClient client = WebClient.create("http://localhost:8080");

// Usage simple.
Contact contact = client
    .method(HttpMethod.GET) // (ou simplement .get())
    .uri("/api/contact/{id}", 0)
    .accept(MediaType.APPLICATION_JSON) // fin config requête
    .retrieve() // expédition...
    .bodyToMono(Contact.class) // type attendu
    .block(); // opération bloquante...
```

Noter le passage de paramètres dans l'URI

Récupération de collections

Le problème est le même qu'avec `RestTemplate` :

```
ParameterizedTypeReference<ArrayList<Contact>> listContactType;
listContactType =
    new ParameterizedTypeReference<ArrayList<Contact>>() {};

ArrayList<Contact> liste =  client
  .get()
  .uri("/api/contact")
  .accept(MediaType.APPLICATION_JSON) // fin config requête
  .retrieve() // expédition...
  .bodyToMono(listContactType) // type attendu
  .block(); // opération bloquante...
```

Requêtes POST

Les données sont typiquement passées en JSON, selon l'usage en REST :

```
Contact nouveau = new Contact("test", "0123456");
client
  .post()
  .uri("/api/contact")
  .contentType(MediaType.APPLICATION_JSON) // optionnel.
  .bodyValue(nouveau)
  .retrieve() // expédition...
  .bodyToMono(Void.class) // type attendu
  .block(); // opération bloquante...
```

Pour un `POST` non REST, à destination d'un formulaire, utiliser comme valeur une `MultiValueMap<String, String>`, sans préciser de `ContentType`

HATEOAS et Spring

projet exemple : resthateoas

spring-boot-starter-hateoas

Permet d'ajouter des liens aux objets qu'on renvoie

Principe :

- soit l'objet DTO renvoyé étend `RepresentationModel` ;
- soit on encapsule le DTO dans `EntityModel` ;
- avant de renvoyer l'objet, on lui ajoute les liens souhaités ;
- utilitaires pour construire les « bonnes » URL des liens.

Exemple simple

Une application de gestion de contacts.

- liste des contacts paginée ;
- chaque contact a un lien vers lui-même ;
- plus un lien qui permet de savoir si la personne en question est disponible.

Construction de lien (méthode simpliste)

```
@RestController @RequestMapping("/api/contact")
public class ContactController {
    @Autowired ContactService service;
    @GetMapping("/{id}")
    public ContactDTO get(@PathVariable final Long id) {
      Contact c = service.getContact(id);
      ContactDTO res = new ContactDTO(c);
      // Ajout des liens
      res.add(new Link("http://localhost:8080/api/contact/"
                       +c.getId(), "self"));
      res.add(new Link("http://localhost:8080/api/contact/"+
                        c.getId()+ "available", "available"));
      return res;
    }
```

- code très fragile : URL en dur ;
- dépendent du serveur...
- et des URL des méthodes.

Construction des liens (méthode avancée)

Spring sait extraire les url par *introspection* grâce à **WebMvcLinkBuilder**

```
@GetMapping("/{id}")
public ContactDTO get(@PathVariable final Long id) {
  Contact c = service.getContact(id);
  ContactDTO res = new ContactDTO(c);
  res.add(WebMvcLinkBuilder.linkTo(ContactController.class)
            .slash(c.getId())
            .withSelfRel());
  ...
```

Avec un import static :

```
  res.add(linkTo(ContactController.class)
            .slash(c.getId())
            .withSelfRel());
  ...
```

WebMvcLinkBuilder

Fonctionnement :

1. on précise le contrôleur qui définit l'adresse avec `linkTo` ;
2. on précise ensuite la *méthode* du contrôleur à utiliser **ou** le chemin sur le contrôleur.
3. s'appuyer sur une méthode (et non un chemin) : résiste mieux aux modifications du code ;
4. on termine en précisant le nom de la relation à utiliser ("self", "next", etc.) ;
5. certains liens sont plus ou moins standardisés. On trouve leurs noms dans la classe `IanaLinkRelations`.

Construction avec une méthode

```
@GetMapping("/{id}")
public ContactDTO get(@PathVariable final Long id) {
 try {
  Contact c = service.getContact(id);
  ContactDTO res = new ContactDTO(c);
  // Méthode "isAvailable" qui prend un "Long" comme argument :
  Method availableMethod = ContactController.class.getMethod(
                                "isAvailable", Long.class);
  res.add(WebMvcLinkBuilder.linkTo(availableMethod, c.getId())
                           .withRel("available"));
  ...
  return res;
 } catch (NoSuchMethodException | SecurityException e) {
   throw new RuntimeException(e);
 }
```

- Sûr, mais un peu lourd à écrire ;
- exception déclarée (absurde) à gérer.

Variante avec proxy

```
Link nextLink = linkTo(methodOn(ContactController.class)
                            .listContacts(page+1))
                       .withRel(IanaLinkRelations.NEXT);
```

- linkTo et methodOn méthodes statiques de `WebMvcLinkBuilder` (import statique ici) ;
- `methodOn` retourne un proxy vers `ContactController` ;
- on peut appeler (fictivement) la méthode qui nous intéresse (ici, `listContact`, méthode de `ContactController`)
- ne fonctionne que s'il est possible de créer un proxy *sur la valeur de retour* du contrôleur visé ;
- ça ne fonctionnait pas avec la méthode `isAvailable`, qui retournait un **Boolean.**

Résultats

GET http ://localhost :8080/api/contact/9


```
{ "id":9,
  "nom":"n8",
  "telephone":"tel8",
  "_links":{
     "self":{"href":"http://localhost:8080/api/contact/9"},
     "available":{
      "href":"http://localhost:8080/api/contact/9/available"}
  }
}
```

Pagination

Les listes sont découpée en page avec un nombre maximum (par exemple 10) d'enregistrements. On retourne un `PagedModel` qui donne :

- la liste des éléments ;
- un objet `PagedModel.PageMetadata` qui indique :
  - la taille maximale de la page ;
  - le numéro de la page ;
  - le nombre total d'éléments dans la collection ;
- les liens portant sur la page.

Pour cela :

- On calcule le nombre de page et le contenu d'une page ;
- on crée les liens `next` et `previous` en fonction de la page (pas de next pour la dernière page ni de `previous` pour la première)
- $\rightarrow$ le client n'aura pas à faire de calcul

Exemple de pagination

```
@GetMapping
public PagedModel<ContactDTO> listContacts(
     @RequestParam(value = "page", defaultValue = "0") int page) {
   List<ContactDTO> all =
      service.findAll().stream().map(this::contact2ContactDTO)
                                .collect(Collectors.toList());
  // Pagination...
  List<ContactDTO> subList;
  int startIndex = page * 10;
  int endIndex = Math.min(page * 10 + 10, all.size());
  if (startIndex < endIndex)
   subList = all.subList(startIndex, endIndex);
  else
   subList = Collections.emptyList();
  // Les liens ...
  List<Link> links = new ArrayList<>();
  ...
```

Exemple de pagination (suite)

```
 // Les liens ...
 List<Link> links = new ArrayList<>();

 if (endIndex < all.size()) {
  Link nextLink = linkTo(methodOn(ContactController.class)
                              .listContacts(page+1))
                      .withRel(IanaLinkRelations.NEXT);
  links.add(nextLink);
 }

 if (startIndex > 0) {
  Link prevLink = linkTo(methodOn(ContactController.class)
                              .listContacts(page-1))
     .withRel(IanaLinkRelations.PREVIOUS);
  links.add(prevLink);
 }
 PagedModel<ContactDTO> pagedModel = new PagedModel<>(subList,
     new PagedModel.PageMetadata(10, page, all.size()), links);
 return pagedModel;
}
```

Pagination (résultat)


```
{
 "_embedded": {
  "contactDTOList": [
   {"id": 1, "nom": "n0", "telephone": "tel0",
    "_links": {
     "self": {"href":"http://localhost:8080/api/contact/1"},
     "available": {"href":"http://localhost:8080/api/contact/1/available"}
    }
   },
   {"id": 2, "nom": "n1", "telephone": "tel1",
    "_links": {
     "self": {"href":"http://localhost:8080/api/contact/2" },
     "available": {"href":"http://localhost:8080/api/contact/2/available"}
    }
   },...
  ]
 },
 "_links": {
  "next": { "href": "http://localhost:8080/api/contact?page=1"}
 },
 "page": {"size": 10, "totalElements": 25,
          "totalPages": 3, "number": 0
 }
}
```

Pagination (résultat)

self	lien réflexif ;
_embedded	éléments optionnels « pré-chargés » - on pourrait avoir des liens à la place ;
_links	liens liés à une ressource : propriétés complexes, autres ressources
next et previous	url de la page suivante/de la plage précédente.

Spring data Rest

voir projet restData

Publie automatiquement les méthodes de `Repository` à travers une interface REST HATEOAS.

- inclure `spring-boot-starter-data-rest` dans les dépendances ;
- publie les ressources (i.e. les repositories JPA) ;
- publie des métadonnées (URL `profile`) ;
- publie une table des matières ;
- comportement par défaut très permissif ;
- publie les méthodes `CRUD` des `repositories` annotés comme `@RestResource` ;
- on peut filtrer ce qui est disponible ;
- on peut modifier ce qui est retourné.

Configuration

En implémentant `RepositoryRestConfigurer` :

```
@Component
public class MyRestConf implements RepositoryRestConfigurer {
  @Override
  public void configureRepositoryRestConfiguration(
                          RepositoryRestConfiguration config) {
    config.setDefaultPageSize(10);
  }
}
```

Interdiction d'une opération

Se fait sur le repository :

```
@RestResource
public interface TypeEntreeRepository
                    extends JpaRepository<TypeEntree, Long> {

    // Empêche la création et la modification de TypeEntree par REST.
    @Override
    @RestResource(exported = false)
    <S extends TypeEntree> S save(S entity);
}
```

Publication des méthodes de recherche

À partir de

```
public interface ContactRepository extends JpaRepository<Contact,
 List<Contact> findByNom(@Param("nom") String nom);
}
```

On obtient sur l'URL : `http://localhost:8080/contacts/search`

```
{
  "_links" : {
    "findByNom" : {
      "href" : "http://localhost:8080/contacts/search/findByNom{?nom}",
      "templated" : true
    },
    "self" : {
      "href" : "http://localhost:8080/contacts/search"
    }
  }
}
```

Ce qui permet :

`http://localhost:8080/contacts/search/findByNom?nom=Graffion`

Projection

- Permet de visualiser une sélection d'attributs pour un objet
- c'est récursif : une projection peut faire référence à d'autres projections.

Déclaration :

- **dans le même package que les entités** ;
- ce sont des interfaces.

```
@Projection(name = "nomP", types = {Contact.class})
public interface NomProjection {
    String getNom();
    String getPrenom();
}
```

Utilisation des projections

- par défaut : définie sur le repository

```
@RepositoryRestResource(excerptProjection
                                = ContactProjection.class)
public interface ContactRepository
                 extends JpaRepository<Contact, Long>{
```

- dans ce cas, utilisée pour les valeurs *embedded* (les listes) ;
- utilisation explicite :
  `http://localhost:8080/contacts/5?projection=contactFull`

Cascade de projections

Une projection peut faire référence à d'autres projections :

```
@Projection(name = "contactFull", types = {Contact.class})
public interface ContactProjection {
    String getNom();
    String getPrenom();
    List<EntreeP> getEntrees();
}
```

```
@Projection(name = "entreeP", types = {Entree.class})
public interface EntreeP {
     TypeEntree getTypeEntree();
     String getValeur();
}
```

Au lieu de référencer les typeEntree avec des liens, cette projection place directement leur valeur dans l'objet.

Cascade de projection

Sans projection

http ://localhost :8080/contacts/5


```
{
  "nom" : "Graffion",
  "prenom" : "Pascal",
  "entrees" : [ {
    "valeur" : "3845",
    "_links" : {
     "typeEntree" : {"href" : "http://localhost:8080/typeEntrees/1"
    }}}, {
    "valeur" : "pascal@cnam.net",
    "_links" : {
     "typeEntree" : {"href" : "http://localhost:8080/typeEntrees/3"
     }}}],
  "_links" : {...}
}
```

Cascade de projection

Avec projection

http ://localhost :8080/contacts/5 ?projection=contactFull

```
{
  "nom" : "Graffion",
  "prenom" : "Pascal",
  "entrees" : [ {
    "typeEntree" : { "label" : "bureau" },
    "valeur" : "3845"
   }, {
    "typeEntree" : { "label" : "mail"  },
     "valeur" : "pascal@cnam.net"
  } ],
  "_links" : {...}
}
```

Conclusion

- outil pratique et puissant pour définir des API REST ;
- « tape » peut être un peu à bas niveau ;
- attention à garantir côté serveur les problèmes de droit et d'intégrité des données.

Authentification et services REST

Authentification et services REST

- sécurité possible par Spring Security ;
- pour une *single page application*, le login doit être réalisé en AJAX ;
- possibilités :
  - basic : login et mot de passe expédiés à chaque requête ;
    pas trop gênant si on utilise https (de toute manière, on utilise https) ;
    passés dans le header http `Authorization` en BASE64 ;
  - connexion par ID de session ;
    stateful ; demande une session ; assez usuel ;
  - JWT : Jason Web Token
    en théorie, stateless ; signature cryptographique ; utilisé par OAuth2.

Principes de Jason Web Token

- connexion au serveur d'authentification avec login et mot de passe ;
- celui-ci produit un objet JSON :
  - contenant un certain nombre de champs (login, droits...) : le *payload* ;
  - **signant cryptographiquement** (clef publique/privée ou privée uniquement) le token.
  - le client n'a plus qu'à transmettre le token pour prouver son identité ;
  - la validité du token est prouvée par la validité de la signature ;
  - on peut invalider le token en y plaçant une date d'expiration (risque de vol de token) ;
- problèmes potentiels : où conserver le token dans une application Javascript (pas de problème pour un client lourd) ;
- un exemple : démonstration simple (Alexey Ponomaruev) ;

la question de l'utilisation de cookies ou d'un autre mécanisme demande réflexion, et dépend du contexte.

Pages sur JWT, REST et Spring

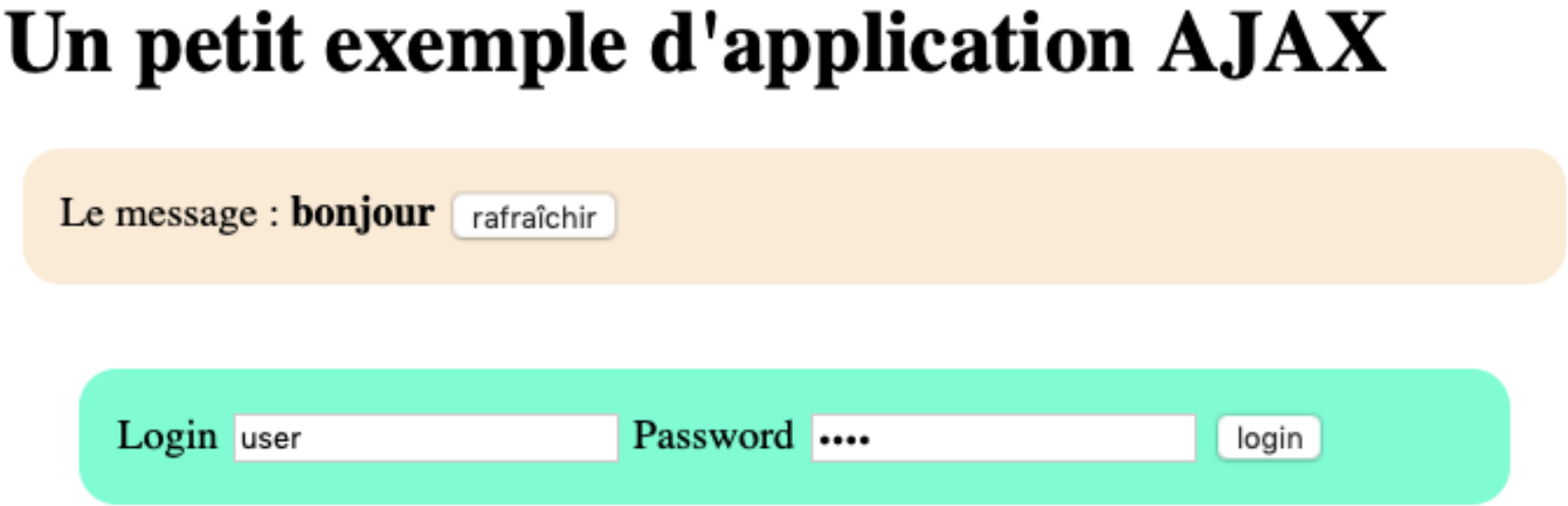
- https://github.com/alexatiks/spring-security-jwt-csrf ;
- https://medium.com/@xoor/jwt-authentication-service-44658409e12c
- https://medium.com/emblatech/secure-your-spring-restful-apis-with-jwt-a-real-world-example-bfdd267
- https://www.toptal.com/java/rest-security-with-jwt-spring-security-and-java
- https://spring.io/guides/tutorials/spring-security-and-angular-js/
- https://blog.marcosbarbero.com/centralized-authorization-jwt-spring-boot2
- https://projects.spring.io/spring-security-oauth/docs/oauth2.html
- https://www.baeldung.com/spring-security-oauth-2-verify-claims
- https://www.baeldung.com/spring-security-oauth-jwt
- https://www.tutorialspoint.com/spring_boot/spring_boot_oauth2_with_jwt.htm
- https://wesleyhill.co.uk/p/alternatives-to-jwt-tokens/ ; https://jolicode.com/blog/why-you-dont-need-jwt à propos de PASETO

REST, Spring Security et session

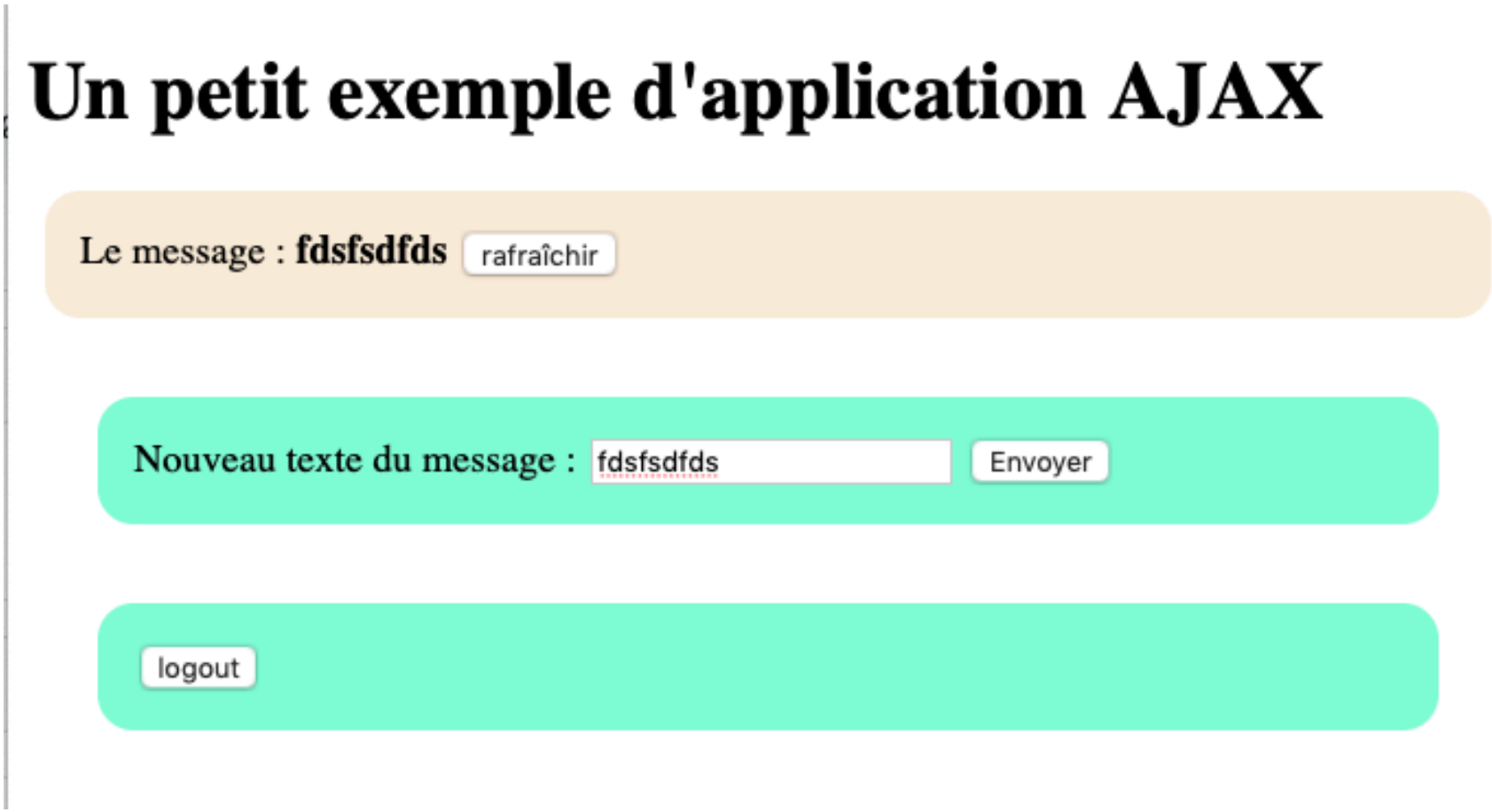
voir projet json-csrf

- problème principal (mais pas trop complexe) : les requêtes POST utilisent un token pour éviter le **CSRF** ;
- il faut transmettre ce token avec les requêtes ;
- problèmes annexe : gérer le login et le logout en Ajax ;
- éviter les redirections, etc... proposées par Spring Security :
  on gère la navigation côté client, on ne veut pas être renvoyé vers une page de login... qu'on ne verra pas.

Exemple


Un petit exemple d'application AJAX
Le message : bonjour
rafraîchir
Login
user
Password
••••
login

Exemple


Un petit exemple d'application AJAX
Le message : fdsfsdfds
rafraîchir
Nouveau texte du message :
fdsfsdfds
Envoyer
logout

SpringSecurity

Au départ, du classique :

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Bean // publie pour injection dans LoginController
    public AuthenticationManager authenticationManagerBean()
                                                    throws Exception {
        return super.authenticationManagerBean();
    }
    protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
        PasswordEncoder encoder =
                    PasswordEncoderFactories
                            .createDelegatingPasswordEncoder();
        auth.inMemoryAuthentication().withUser("user")
                                    .password(encoder.encode("user"))
                                    .roles("USER");
    }
    public void configure(WebSecurity web) throws Exception {
        web.ignoring()
           .antMatchers("/js/**")
           .antMatchers("/webjars/**")
           .antMatchers("/index.html");
    }
```

SpringSecurity

On va désactiver les mécanismes de login et de logout par formulaire :

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  ...
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic().disable()
            .logout().disable()
            .formLogin().disable()
            .authorizeRequests()
                // On autorise le put uniquement aux utilisateurs connectés
                .mvcMatchers(HttpMethod.PUT, "/**").authenticated()
                // Le reste est autorisé à tout le monde.
                .mvcMatchers("/**").permitAll();
    }
}
}
```

Contrôleur de login

```
@Controller
public class LoginController {
    @Autowired
    AuthenticationManager manager;

    @PostMapping("/login")
    public void login(String username, String password,
                      HttpServletResponse response,
                      HttpServletRequest request)
                        throws IOException, ServletException {
        request.login(username, password); // suffit !
        response.getWriter().append("ok");
        response.setStatus(200);
    }

    @PostMapping("/logout")
    @ResponseBody
    public String logout(HttpServletRequest request)
                            throws ServletException {
        // request.logout(); ne fonctionne pas.
        SecurityContextHolder.getContext().setAuthentication(null);
        return "logout";
    }
}
```

CSRF

- On a besoin du jeton CSRF ;
- on peut le placer dans un en-tête de la page html si elle est dynamique ;
- si la page est du HTML statique accompagné de Javascript : on publie le jeton en mode GET
- sa valeur est (normalement) protégée : voir **CORS**

```
@RestController
public class MessageController {

 @GetMapping("/csrf")
 public CsrfToken csrf(CsrfToken token) {
   return token;
 }
}
```

La valeur du token est injectée par Spring.

En Javascript

```
var user = null
async function csrfPromise() {
 // retourne par exemple {"token":"d8573ebd",
 //           "parameterName":"_csrf","headerName":"X-CSRF-TOKEN"}
 return axios.get("/csrf")
}

async function doLogin() {
 clearErreur()
 var csrf = (await csrfPromise()).data;
 let headers = {}
 §headers[csrf.headerName] = csrf.token§
 let login = byId("login").value
 let password = byId("password").value
 axios({
   method: 'post', url: "/login",
   params:
     { "username": login, "password": password },
   headers: headers
 }).then(function () { user = login; mettreAJourVisibilites()}
        ).catch(function (erreur) {
           byId("erreur").textContent = "Connexion refusée"
         })
}
```

Commentaire

- notez l'utilisation de `async` et `await` pour simplifier l'utilisation de code asynchrone ;
- quand on a reçu le code `csrf`, on peut expédier la requête POST en plaçant celui-ci comme header ;
- on ajoute le header :

  ```
  X-CSRF-TOKEN: d8573ebd
  ```

- une fois connecté, tout est transparent : c'est géré par le cookie de connexion, et il est transmis automatiquement au serveur.

Que choisir ?

- Étude : http ://www2008.org/papers/pdf/p805-pautassoA.pdf
- Coûts/avantage des deux systèmes

Conclusions

- pas très éloignés en fait.
- REST plus facile à appréhender « from scratch »,
- mais les outils des Web Services SOAP les rendent facile d'emploi
- pour du web service interne, application entreprise : plutôt Web services lourds (couche d'authentification, etc...)
- pour de la publication externe (en particulier consommable par du javascript) : plutôt du REST

(note : ces conclusions ont probablement vieilli !)

Guide des exemples

- **01_j2ee** Web services en J2EE (non maintenus) ;
- **02_webservice** web service en spring ;
- **03_websoapclient** client web service en spring ;
- **04_restserveur** serveur REST simple ;
- **05_restclient** client REST en Spring (avec RESTTemplate) ;
- **06_restWebClient** client REST en Spring (avec WebClient) ;
- **07_resthateoas** démo d'utilisation de hateoas en Spring ;
- **08_restData** démo de spring-data-rest ;
- **09_json-csrf** login Ajax sur application Spring ;
- **10_jwt** connexion simple avec Jason Web Token.

Dans chaque projet, le fichier `README.md` donne des indications sur l'utilisation/l'architecture du projet.