Spring MVC

Arnaud Cogoluègnes

Zenika

Novembre 2015

Plan

Plan

Spring MVC en deux mots

- Framework pour applications web “classiques”
- Support pour web services REST
  - Support serveur et client (`RestTemplate`)
- Dépendances : Spring et API Servlet
- Très flexible (beaucoup de points d’extension)
- Base technique pour d’autres frameworks
  - Spring Web Flow, Grails

DispatcherServlet

- Le coeur de Spring MVC
- Coordonne des composants d'infrastructure
- Appelle les contrôleurs applicatifs

Contrôleur “Hello World”

- Les contrôleurs sont des POJO annotés

```
@RestController // indique à Spring que c'est un contrôleur
public class HelloWorldController {

  @RequestMapping("/hello") // quel URL ?
  public String hello() {
    return "Hello World!";
  }

}
```

Les contrôleurs sont des beans Spring

- Il faut bien déclarer les contrôleurs dans Spring
- Une solution est le component scanning
  - Déclarations en XML ou en Java fonctionnent aussi

Comment démarrer ?

- Utiliser Spring Boot
- Gère les dépendances, la `DispatcherServlet`, etc.

```java
@SpringBootApplication // gère notamment le component scanning
public class SpringMvcOverviewApplication {

  public static void main(String[] args) {
    SpringApplication.run(SpringMvcOverviewApplication.class, args);
  }

}
```

Plan

Beans d'infrastructure

- La `DispatcherServlet` coordonne des beans d'infrastructure
- Valeurs par défaut, tout est surchargeable
- Changer ces beans change le comportement de Spring MVC

Beans d'infrastructure principaux

- Ils sont trois
- `HandlerMapping`, `HandlerAdapter`, `ViewResolver`
- Ils participent au pattern MVC

Beans d'infrastructure principaux

```
// dans la DispatcherServlet (pseudo-code)

// quel contrôleur pour cette requête ?
Object handler = handlerMapping.getHandler(request);
// appeler la bonne méthode du contrôleur, avec les bons paramètres
ModelAndView mav = handlerAdapter.handle(request,response,handler);
// trouver la vue à rendre
View view = viewResolver.resolveViewName(
  mav.getViewName(),request.getLocale()
);
// effectuer le rendu
view.render(mav.getModel(),request,response);
```

Chaîne de beans d'infrastructure

- Les beans d'infrastructure sont généralement organisés en chaîne
  - Le cas pour `HandlerMapping`, `HandlerAdapter`, `ViewResolver`
- Ils sont consultés, le premier qui répond gagne
  - Répondre = retourner autre chose que `null`

Configuration par défaut

- `DispatcherServlet.properties`
  - dans `spring-webmvc.jar`
  - package `org.springframework.web.servlet`

```
org.springframework.web.servlet.LocaleResolver=\
  org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=\
  org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=\
  org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\
  org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=\
  org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\
  org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
  org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

(...)
```

Surcharger un bean d'infrastructure

- Déclarer un bean du type correspondant
- Remplace la configuration par défaut
- Ex. : déclarer un `HandlerAdapter` remplace les 3 par défaut

```
<bean class="o.s.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views" />
  <property name="suffix" value=".jsp" />
</bean>
```

Plan

Spring MVC et les applications web

- 1 modèle de programmation, plusieurs types d'applications
- Application web "classique" : vue générée coté serveur
  - Spring MVC = équivalent de Struts, JSF, Wicket, etc.
- Application web REST : ne gère pas la vue
  - Spring MVC ne retourne que des données
  - Présentation gérée coté client (ex. : JQuery)

Contrôleur application web

- Remplit un `Model` de données
- Indique quelle vue doit être rendue

Contrôleur application web

```java
@Controller
public class ContactController {

  @RequestMapping("/contact")
  public String contact(Model model) { // demande un Model vide
    Contact contact = (...); // chargement depuis la BD
    model.addAttribute(contact); // pour la vue
    return "/WEB-INF/views/contact.jsp"; // la vue à utiliser
  }

}
```

La vue

- Plusieurs technologies supportées : JSP, Freemarker, Velocity
- JSP très couramment utilisé

```
<html>
  <head>
    <title>Spring MVC</title>
  </head>
  <body>
  ${contact.id} <br />
  ${contact.firstname} <br />
  ${contact.lastname} <br />
  </body>
</html>
```

Couplage contrôleur/vue

- Le contrôleur connait le chemin de la vue
- C'est un couplage fort (chemin + technologie)

```
@RequestMapping("/contact")
public String contact(Model model) {
  (...)
  return "/WEB-INF/views/contact.jsp";
}
```

Découplage contrôleur/vue

- Le contrôleur peut utiliser un nom logique
- Couplage lâche
- Il faut déclarer un `ViewResolver`

```
@RequestMapping("/contact")
public String contact(Model model) {
  (...)
  return "contact";
}
```

Découplage contrôleur/vue

- Utiliser `InternalResourceViewResolser`
- Configurer un préfixe et un suffixe
- Ils “décoreront” le nom retourné par le contrôleur

```
<bean class="o.s.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/views/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Plan

REST

- Representational State Transfer
- Un style d'architecture
- Une façon de faire communiquer des applications
- Utiliser HTTP comme un protocole applicatif
  - Pas juste comme un protocole de transport

Principes

- Ressources identifiées
- Interface uniforme
- Sans état
- Représentation
- Hypermedia

Ressources identifiées

- Tout est ressource, chaque ressource a une addresse
- L'adresse est une URI
- Bien : http://somehost.com/zen/contacts/1
- Pas bien : http://somehost.com/zen/contacts?id=1
  - L'identifiant fait partie de l'adresse
  - Il doit être dans l'URL, pas en tant que paramètre

Interface uniforme

- Le client effectue des opérations sur une ressource
- Opérations disponibles :
  - GET : récupération d'une ressource
  - POST : créer une ressource
  - PUT : modifier une ressource
  - DELETE : supprimer une ressource
  - HEAD : GET, mais sans le contenu, juste les entêtes
  - OPTIONS : options de communication de la ressource

Interface uniforme

- Entêtes standardisés
  - type de la requête, type attendu, taille de la réponse, etc.
- Codes réponse standardisés
  - 200 OK
  - 201 Created
  - 404 Not found
  - 409 Conflict
  - 500 Internal server error
  - etc.

Sans état

- Aucun lien entre deux requêtes...
- Même si envoyées par le même client
- Facilite la distribution à grande échelle (“scalability”)
- Si état il y a, il est stocké dans une base de données
- Plus de session !
  - En théorie...
  - On peut choisir de ne pas suivre ce principe

Représentation

- Pas de format imposé pour représenter les ressources
- Formats courants :
  - XML
  - JSON
  - ATOM
- Possibilité d'utiliser des schémas

Hypermedia

- Les ressources ont des liens vers d'autres ressources
- Exactement comme des pages web
- Un client peut suivre les liens d'une ressource à l'autre
  - Un client intelligent...
- Ex. : lien pour avancer dans un workflow ou l'annuler
- Généralement, entente entre client et fournisseur du service

Plan

Spring MVC et REST

- Spring MVC fournit un support REST
- Même modèle de programmation que pour les applications web
- Annotations et mécanismes supplémentaires
- Principale différence : plus de vue
- Spring MVC
  - désérialise le contenu des requêtes
  - sérialise le contenu des réponses

Activer le support REST

- Utiliser le starter web de Spring Boot
- Inclut Spring MVC, sérialisation JSON, etc

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

Récupérer une ressource (HTTP)

- Requête

```
GET /contacts/1 HTTP/1.1
Accept: application/json
Host: localhost:8080
```

- Réponse

```
HTTP/1.1 200 OK
Content-Type: application/json
{"id":1,"firstname":"Joe","lastname":"Dalton","age":37}
```

Récupérer une ressource (contrôleur)

```
@RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
public Contact contact(@PathVariable Long id) {
  return contactRepository.findOne(id);
}
```

Récupérer une ressource (contrôleur)

```
@RequestMapping(
  value="/contacts/{id}",      // URL
  method=RequestMethod.GET     // opération
)
public Contact contact(
    @PathVariable Long id) {  // à extraire de l'URL
  return contactRepository.findOne(id);
}
```

Récupérer une ressource (client)

```
RestTemplate tpl = new RestTemplate();
Contact contact = tpl.getForObject(
  "http://localhost:8080/contacts/{id}",
  Contact.class,
  id
);
```

Récupérer des ressources (HTTP)

- Requête

```
GET /contacts HTTP/1.1
Accept: application/json
Host: localhost:8080
```

- Réponse

```
HTTP/1.1 200 OK
Content-Type: application/json
[
 {"id":1,"firstname":"Joe","lastname":"Dalton","age":37},
 {"id":2,"firstname":"William","lastname":"Dalton","age":35},
 {"id":3,"firstname":"Jack","lastname":"Dalton","age":33},
 {"id":4,"firstname":"Averell","lastname":"Dalton","age":31}
]
```

Récupérer des ressources (contrôleur)

```
@RequestMapping(value="/contacts",method=RequestMethod.GET)
public List<Contact> contacts() {
  return contactRepository.findAll();
}
```

Récupérer des ressources (client)

```
RestTemplate tpl = new RestTemplate();
Contact [] contacts = tpl.getForObject(
  "http://localhost:8080/contacts",
  Contact[].class
);
```

Créer une ressource (HTTP)

- Requête

```
POST /contacts HTTP/1.1
Content-Type: application/json;charset=UTF-8
Host: localhost:8080
{"id":null,"firstname":"Oncle","lastname":"Picsou","age":100}
```

- Réponse

```
HTTP/1.1 201 Created
Location: http://localhost:8080/contacts/130
Content-Length: 0
```

Créer une ressource (contrôleur)

```
@RequestMapping(value="/contacts",method=RequestMethod.POST)
public ResponseEntity<Void> create(@RequestBody Contact contact,
                                   UriComponentsBuilder uriComponentsBuilder) {
  contactRepository.save(contact);
  URI location = uriComponentsBuilder
    .pathSegment("contacts","{id}")
    .build()
    .expand(contact.getId())
    .encode()
    .toUri();
  return ResponseEntity.created(location).build();
}
```

Créer une ressource (contrôleur)

```
@RequestMapping(value="/contacts",method=RequestMethod.POST)
public ResponseEntity<Void> create(
      @RequestBody Contact contact,  // extraire du corps de requête
      UriComponentsBuilder uriComponentsBuilder) {
  (...)
}
```

Créer une ressource (contrôleur)

```
@RequestMapping(value="/contacts",method=RequestMethod.POST)
public ResponseEntity<Void> create(@RequestBody Contact contact,
                                   UriComponentsBuilder uriComponentsBuilder) {
  (...)
  return ResponseEntity.created(      // code réponse
    location                          // URL de la ressource créée
  ).build();
}
```

Créer une ressource (client)

```
Contact contact = new Contact();
contact.setFirstname("Oncle");
contact.setLastname("Picsou");
contact.setAge(100);
RestTemplate tpl = new RestTemplate();
URI location = tpl.postForLocation(
  "http://localhost:8080/contacts",
  contact
);
```

Modifier une ressource (HTTP)

- Requête

```
PUT /contacts/130 HTTP/1.1
Content-Type: application/json;charset=UTF-8
Host: localhost:8080
{"id":130,"firstname":"Oncle","lastname":"Picsou","age":90}
```

- Réponse

```
HTTP/1.1 204 Not Content
Content-Length: 0
```

Modifier une ressource (contrôleur)

```
@RequestMapping(value="/contacts/{id}",method=RequestMethod.PUT)
@ResponseStatus(HttpStatus.NO_CONTENT)  // code réponse
public void update(@RequestBody Contact contact) {
  contactRepository.save(contact);
}
```

Modifier une ressource (client)

```
RestTemplate tpl = new RestTemplate();
// création donne l'URI
URI location = tpl.postForLocation(...);
// modification
contact.setAge(90);
tpl.put(location,contact);
```

Supprimer une ressource (HTTP)

- Requête

```
DELETE /contacts/130 HTTP/1.1
Host: localhost:8080
```

- Réponse

```
HTTP/1.1 204 Not Content
Content-Length: 0
```

Supprimer une ressource (contrôleur)

```
@RequestMapping(value="/contacts/{id}",method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void delete(@PathVariable Long id) {
  contactRepository.delete(id);
}
```

Supprimer une ressource (client)

```
RestTemplate tpl = new RestTemplate();
// création donne l'URI
URI location = tpl.postForLocation(...);
// suppression
tpl.delete(location);
```

Gestion des erreurs

- Ex. : demande d'une ressource qui n'existe pas
- Le serveur doit retourner une erreur 404
- Comment faire avec Spring MVC, cotés serveur et client ?

Gestion des erreurs, coté serveur, solution 1

```
@RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
public Contact contact(@PathVariable Long id) {
  Contact contact = contactRepository.findOne(id);
  if(contact == null) {
    throw new EmptyResultDataAccessException(1);
  }
  return contact;
}

@ExceptionHandler(EmptyResultDataAccessException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public void notFound() { }
```

Gestion des erreurs, coté serveur, solution 2

- Retourner une `@ResponseEntity`

```
@RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
public ResponseEntity<Contact> contact(@PathVariable Long id) {
  Contact contact = contactRepository.findOne(id);
  ResponseEntity<Contact> response = new ResponseEntity<Contact>(
    contact,
    contact == null ? HttpStatus.NOT_FOUND : HttpStatus.OK
  );
  return response;
}
```

Gestion des erreurs, coté client

```
try {
  Contact contact = tpl.getForObject(location, Contact.class);
} catch (HttpStatusCodeException e) {
  // e.getStatusCode() == HttpStatus.NOT_FOUND
}
```

- Stratégie de gestion des exceptions configurable
- Propriété `errorHandler` du `RestTemplate`

Plan

Principes

- Tester toute la stack Spring MVC
  - validation, (dé)sérialisation, codes réponse, etc.
- Sans utiliser un conteneur web
  - Pas de requête HTTP

Pourquoi ?

- Tester toute la couche web
  - Cas de base et cas limites
- Utiliser des mock pour les services métiers
  - Plus facile pour simuler tous les cas

Configuration du test

- Ajouter `@WebAppConfiguration`
- Injecter le contexte Spring

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class ContactControllerTest {

  @Autowired
  WebApplicationContext ctx;

  ...
}
```

Configuration dans le test

- Classe interne statique, détectée automatiquement

```
public class ContactControllerTest {

  ...

  @Configuration
  @EnableWebMvc
  @ComponentScan(basePackageClasses = ContactController.class)
  public static class TestConfiguration {

      @Bean
      public ContactRepository contactRepository() {
          return mock(ContactRepository.class);
      }

  }
}
```

Initialisation de MockMvc

```
public class ContactControllerTest {

    @Autowired
    WebApplicationContext ctx; // le contexte Spring

    @Autowired ContactRepository repo; // une dépendance mockée

    MockMvc mockMvc; // Spring MVC, version mock

    @Before public void setUp() {
        this.mockMvc = webAppContextSetup(ctx) // méthode statique
          .build();                            // Spring MVC test
        reset(repo); // méthode statique Mockito
    }
    ...
}
```

Fluent API, le prix à payer

```
import static o.m.Mockito.*;
import static o.s.test.web.servlet.request.MockMvcRequestBuilders.*;
import static o.s.test.web.servlet.result.MockMvcResultMatchers.*;
import static o.s.test.web.servlet.setup.MockMvcBuilders.*;
import static o.s.test.web.servlet.result.MockMvcResultHandlers.*;
```

Configuration du test

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class ContactControllerTest {

    @Autowired WebApplicationContext ctx;

    @Autowired ContactRepository repo;

    MockMvc mockMvc;

    @Before public void setUp() {
        this.mockMvc = webAppContextSetup(ctx).build();
        reset(repo);
    }
    ...
    @Configuration
    @EnableWebMvc
    @ComponentScan(basePackageClasses = ContactController.class)
    public static class TestConfiguration {

        @Bean
        public ContactRepository contactRepository() {
            return mock(ContactRepository.class);
        }

    }
}
```

Méthode à tester

```
@RestController
public class ContactController {

  @Autowired ContactRepository contactRepository;

  @RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
  public Contact contact(@PathVariable Long id) {
    Contact contact = contactRepository.findOne(id);
    if(contact == null) {
      throw new EmptyResultDataAccessException(1);
    }
    return contact;
  }

}
```

Lancer une requête

```
@Test
public void contactExists() throws Exception {
    Long id = 1L;
    when(repo.findOne(id)).thenReturn(new Contact(id,"John","Doe",33));
    mockMvc.perform(get("/contacts/{id}", id))
        .andDo(print()); // affichage requête/réponse dans la console
}
```

Voir ce qui se passe

```
mockMvc.perform(get("/contacts/{id}", id)).andDo(print());
```

```
...
MockHttpServletResponse:
           Status = 200
    Error message = null
          Headers = {Content-Type=[application/json;charset=UTF-8]}
     Content type = application/json;charset=UTF-8
             Body = {"id":1,"firstname":"John","lastname":"Doe","age":33}
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
...
```

Tester un GET

```
@Test
public void contactExists() throws Exception {
    Long id = 1L;
    when(repo.findOne(id)).thenReturn(new Contact(id,"John","Doe",33));
    mockMvc.perform(get("/contacts/{id}", id))
          .andExpect(status().isOk())
          .andExpect(jsonPath("id").value(1))
          .andExpect(jsonPath("firstname").value("John"))
          .andExpect(jsonPath("lastname").value("Doe"))
          .andExpect(jsonPath("age").value(33));
}
```

Si la ressource n'existe pas...

```
@RestController
public class ContactController {

  @Autowired ContactRepository contactRepository;

  @RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
  public Contact contact(@PathVariable Long id) {
    Contact contact = contactRepository.findOne(id);
    if(contact == null) {
      throw new EmptyResultDataAccessException(1);
    }
    return contact;
  }

  @ExceptionHandler(EmptyResultDataAccessException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  public void notFound() { }

}
```

Tester un GET qui retourne 404

```
@Test
public void contactDoesNotExists() throws Exception {
    Long id = 1L;
    when(repo.findOne(1L)).thenReturn(null);
    mockMvc.perform(get("/contacts/{id}", id))
            .andExpect(status().isNotFound());
}
```

Un GET qui retourne un tableau

```
[
  {"id":1,"firstname":"John","lastname":"Doe","age":33},
  {"id":2,"firstname":"Jane","lastname":"Doe","age":30}
]
```

```
@Test public void contacts() throws Exception {
    when(repo.findAll()).thenReturn(Arrays.asList(
            new Contact(1L,"John","Doe",33),
            new Contact(2L,"Jane","Doe",30)
    ));
    mockMvc.perform(get("/contacts"))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$[0].id").value(1))
        .andExpect(jsonPath("$[0].firstname").value("John"))
        .andExpect(jsonPath("$[0].lastname").value("Doe"))
        .andExpect(jsonPath("$[1].id").value(2))
        .andExpect(jsonPath("$[1].firstname").value("Jane"))
        .andExpect(jsonPath("$[1].lastname").value("Doe"));
}
```

Une méthode POST avec un corps

```
@RequestMapping(value="/contacts",method=RequestMethod.POST)
public ResponseEntity<Void> create(@RequestBody Contact contact,
                                   UriComponentsBuilder uriComponentsBuilder) {
 contact = contactRepository.save(contact);
 URI location = uriComponentsBuilder
   .pathSegment("contacts","{id}")
   .build()
   .expand(contact.getId())
   .encode()
   .toUri();
  return ResponseEntity.created(location).build();
}
```

Tester une méthode POST avec un corps

```
@Test public void create() throws Exception {
    Contact toBeCreated = new Contact(1L,"John","Doe",33);

    when(repo.save(any(Contact.class))).thenReturn(toBeCreated);
    mockMvc.perform(post("/contacts")
        .content(
          "{\"firstname\":\"John\",\"lastname\":\"Doe\",\"age\":33}"
        )
        .contentType(MediaType.APPLICATION_JSON))
      .andExpect(status().isCreated())
      .andExpect(header().string(
         "Location","http://localhost/contacts/1")
      );

    (...)
}
```

Tester une méthode POST avec un corps (suite)

```
@Test public void create() throws Exception {
    Contact toBeCreated = new Contact(1L,"John","Doe",33);

    (...)
    // JSON => Contact OK ?
    ArgumentCaptor<Contact> contactCaptor = ArgumentCaptor.forClass(
      Contact.class
    );
    verify(repo).save(contactCaptor.capture());
    Contact captured = contactCaptor.getValue();
    Assert.assertEquals(toBeCreated.getFirstname(),
      captured.getFirstname());
    Assert.assertEquals(toBeCreated.getLastname(),
      captured.getLastname());
    Assert.assertEquals(toBeCreated.getAge(),
      captured.getAge());
}
```

Plan

Représentation

- En REST, aucun format n'est imposé
  - En SOAP, XML est imposé
- Le client et le serveur négocie le format
- Tout se passe avec deux entêtes
  - `Accept` : dans la requête, ce que le client comprend
  - `Content-Type` : dans la réponse, ce que le serveur renvoie

Accept et Content-Type

- Requête

```
GET /contacts/1 HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Host: localhost:8080
```

- Réponse

```
HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<contact>
  <age>37</age>
  <firstname>Joe</firstname>
  <id>1</id>
  <lastname>Dalton</lastname>
</contact>
```

Dans Spring MVC

- Spring MVC gère de façon transverse
  - La négociation de contenu
  - La désérialisation/sérialisation de la requête/réponse
- Conséquences pour le développeur :
  - Un même contrôleur peut retourner plusieurs types de contenu
  - Travail sur des objets de domaine

Désérialisation/sérialisation

```
// désérialiser le corps de la requête pour obtenir un Contact
@RequestMapping(value="/contacts",method=RequestMethod.POST)
@ResponseStatus(HttpStatus.CREATED)
public void create(@RequestBody Contact contact,
                   HttpServletRequest request,
                   HttpServletResponse response) {
  (...)
}

// sérialiser le Contact retourné
// puis le mettre dans le corps de la réponse
@RequestMapping(value="/contacts/{id}",method=RequestMethod.GET)
public Contact contact(@PathVariable Long id) {
  return contactRepository.findOne(id);
}
```

Désérialisation/sérialisation

- Idem avec le `RestTemplate`

```
// désérialiser le corps de la réponse pour obtenir un Contact
Contact contact = tpl.getForObject(
  "http://localhost:8080/contacts/1",
  Contact.class,
  id
);

// sérialiser le Contact à créer
// puis le mettre dans le corps de la requête
Contact contact = new Contact();
URI location = tpl.postForLocation(
  "http://localhost:8080/contacts",
  contact
);
```

Désérialisation/sérialisation, qui ?

- Des `HttpMessageConverters`
- Autour du contrôleur (dans le `HandlerAdapter`)
- Dans le `RestTemplate`

HttpMessageConverter

```
public interface HttpMessageConverter<T> {

  boolean canRead(Class<?> clazz, MediaType mediaType);

  boolean canWrite(Class<?> clazz, MediaType mediaType);

  List<MediaType> getSupportedMediaTypes();

  // désérialisation (représentation vers objet)
  T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
      throws IOException, HttpMessageNotReadableException;

  // sérialisation (objet vers représentation)
  void write(T t, MediaType contentType,
             HttpOutputMessage outputMessage)
      throws IOException, HttpMessageNotWritableException;

}
```

`HttpMessageConverter` dans Spring MVC

- Spring MVC connait
  - Le format (`Accept` ou `Content-Type`)
  - Le type Java attendu (en paramètre ou en retour)
- Spring MVC consulte ses `HttpMessageConverters`
- Il utilise celui qui peut faire la conversion

`HttpMessageConverters` disponibles

- JAXB2 (XML), Jackson (JSON), Atom, RSS, Spring OXM (XML)
- Formulaire HTML, `byte[]`, etc.
- Automatiquement enregistrés (si librairie tierce présente)

Déclarer un HttpMessageConverters coté serveur

```
@Configuration
public static class ContentNegociationConfiguration
              extends WebMvcConfigurerAdapter {

  @Override
  public void configureMessageConverters(
                    List<HttpMessageConverter<?>> converters)
    converters.add(new CsvHttpMessageConverter());
  }

}
```

Déclarer un HttpMessageConverters coté client

```
RestTemplate tpl = new RestTemplate();
List<HttpMessageConverter<?>> convs =
  new ArrayList<HttpMessageConverter<?>>();
convs.add(new MappingJacksonHttpMessageConverter());
tpl.setMessageConverters(convs);
```

Intercepteurs

- Spring MVC propose des intercepteurs, cotés client et serveur
- Pratique pour des traitements transverses ou systématiques

Intercepteur coté serveur

```
public interface HandlerInterceptor {

  boolean preHandle(HttpServletRequest request,
                    HttpServletResponse response,
                    Object handler) throws Exception;

  void postHandle(HttpServletRequest request,
                  HttpServletResponse response,
                  Object handler, ModelAndView modelAndView)
              throws Exception;

  void afterCompletion(HttpServletRequest request,
                       HttpServletResponse response,
                       Object handler, Exception ex)
              throws Exception;

}
```

Déclarer un intercepteur coté serveur

```
@Configuration
public static class ContentNegociationConfiguration
              extends WebMvcConfigurerAdapter {

  @Override
  public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new LogHandlerInterceptor());
  }

}
```

Intercepteur coté client

```
public interface ClientHttpRequestInterceptor {

  ClientHttpResponse intercept(HttpRequest request, byte[] body,
                               ClientHttpRequestExecution execution)
                        throws IOException;

}
```

Déclarer un intercepteur coté client

```
RestTemplate tpl = new RestTemplate();
List<ClientHttpRequestInterceptor> interceptors =
  new ArrayList<ClientHttpRequestInterceptor>();
interceptors.add(new LogClientHttpRequestInterceptor());
tpl.setInterceptors(interceptors);
```

Plan

HATEOAS

- Hypermedia as the engine of application state
- Permet de découvrir les actions futures/possibles
  - Naviguer vers une autre ressource
  - Avoir le détail d'une ressource
  - Récupérer d'autres représentations d'une ressource

HATEOAS : exemple

```
[
  {
    "firstname":"Joe",
    "lastname":"Dalton",
    "links":[
      {
        "rel":"self",
        "href":"http://localhost:8080/hateoas/zen-contact/contacts/1"
      }
    ]
  }, ...
]
```

Spring HATEOAS

- Une librairie proposant un support HATEOAS
- S'intégre avec Spring MVC
- En cours de développement !

Link

```
Link link = new Link(
  "http://localhost:8080/hateoas/zen-contact/contacts/1",
  Link.REL_SELF
);
```

- Links ajoutés aux ressources...
- ... puis sérialisés en JSON, XML...

Ressource avec des liens

- Rajouter un support pour `Link` dans ses ressources...
- ... ou utiliser `ResourceSupport`

```
public class ShortContact extends ResourceSupport {

  private String firstname,lastname;
  (...) // getters and setters
}
...
ShortContact resource = new ShortContact();
resource.add(new Link("http://localhost/contacts/1"));
```

Intégration avec Spring MVC

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;

@Controller
@RequestMapping("/contacts")
public class ContactController {
  ...

  // dans une méthode
  // lien à partir de l'URL du contrôleur, ajout d'un identifiant
  Link detail = linkTo(ContactController.class)
    .slash(contact.getId())
    .withSelfRel();

}
```

Lien sur une méthode de contrôleur

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;

@RequestMapping(method=RequestMethod.GET)
@ResponseBody
public List<ShortContact> contacts() {

  // fait référence à la méthode du contrôleur
  Link detail = linkTo(
    methodOn(ContactController.class).contact(contact.getId())
  ).withSelfRel();

}

@RequestMapping(value="/{id}",method=RequestMethod.GET)
public ResponseEntity<Contact> contact(@PathVariable Long id) { }
```