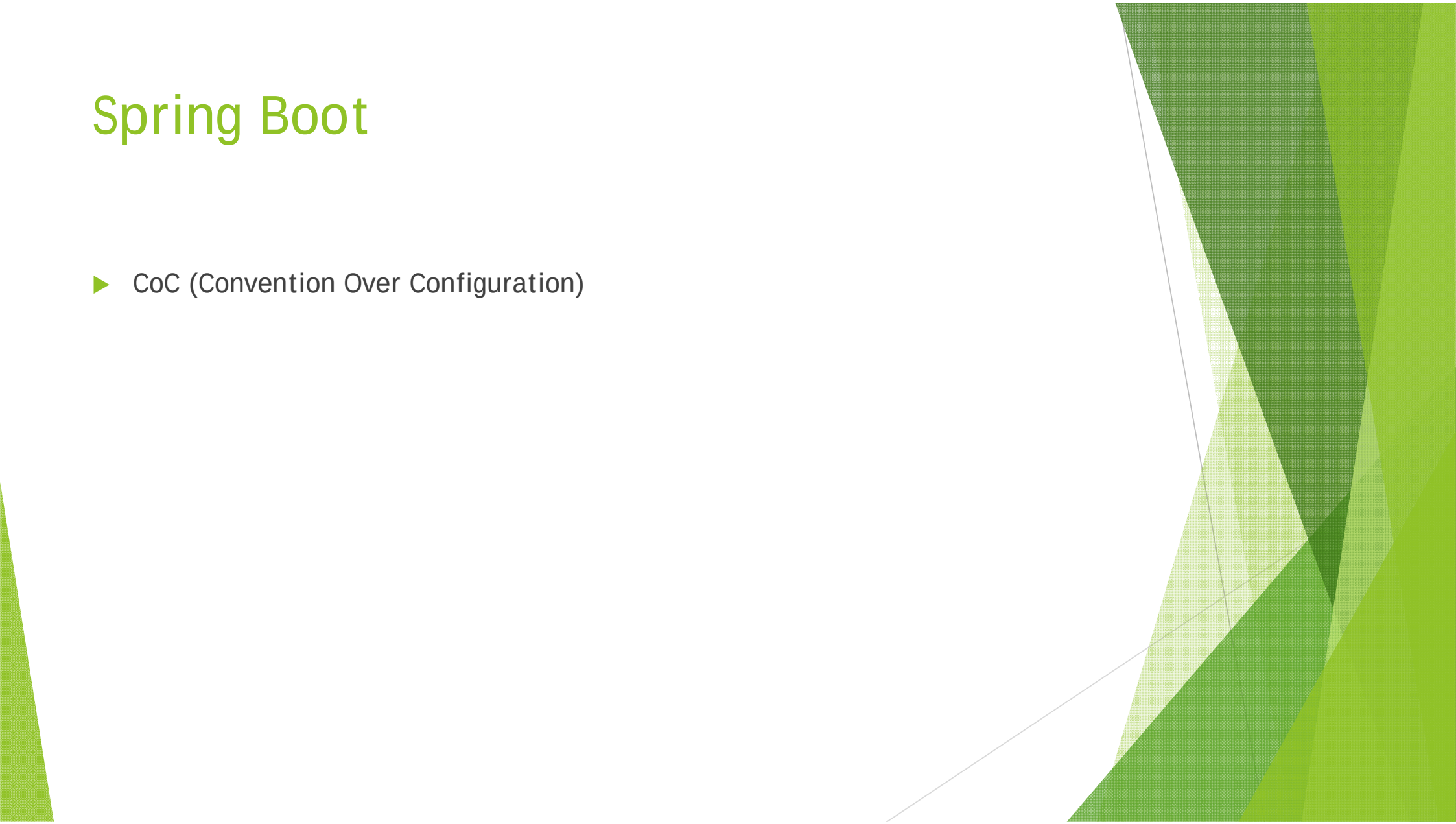
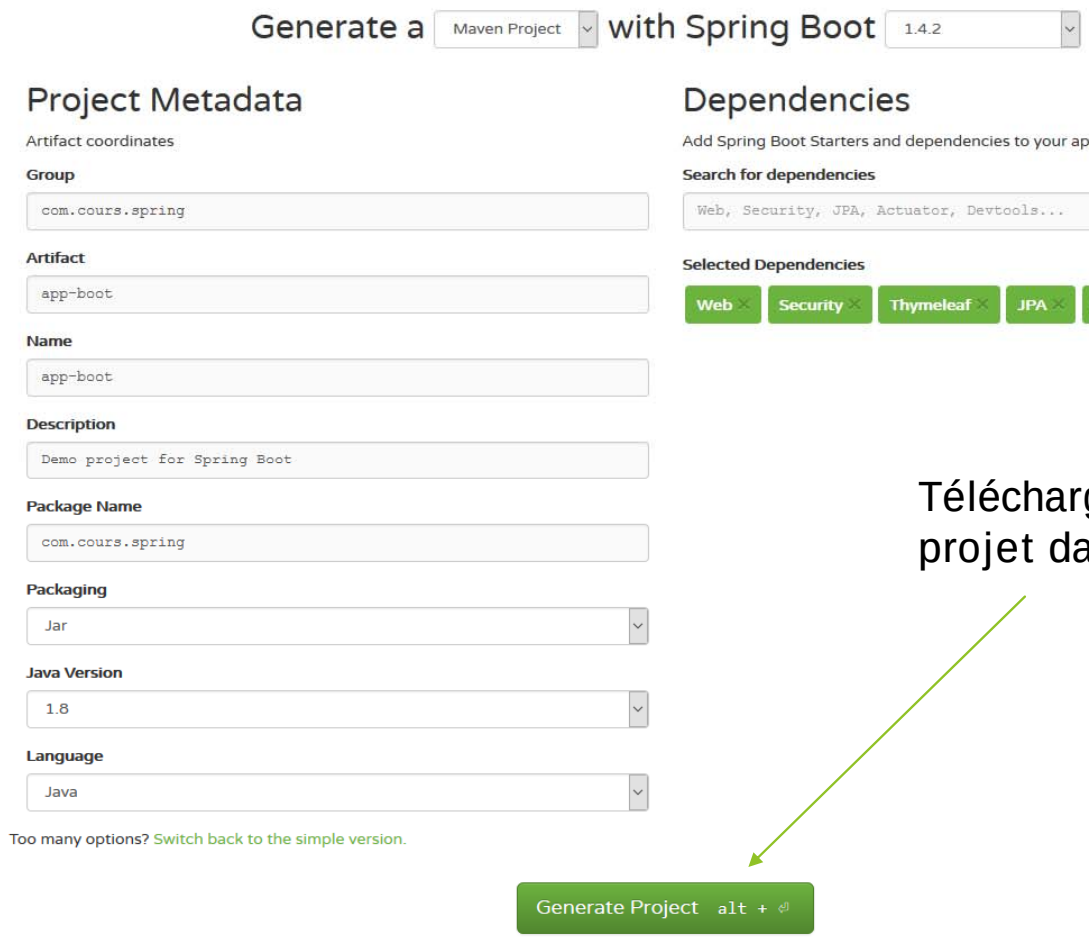
Spring MVC

Spring Boot

- CoC (Convention Over Configuration)

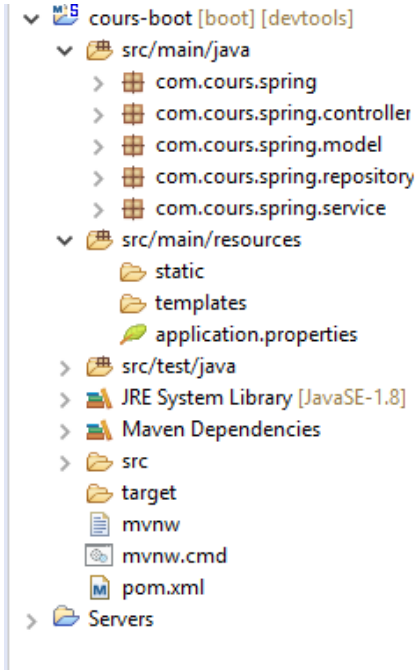


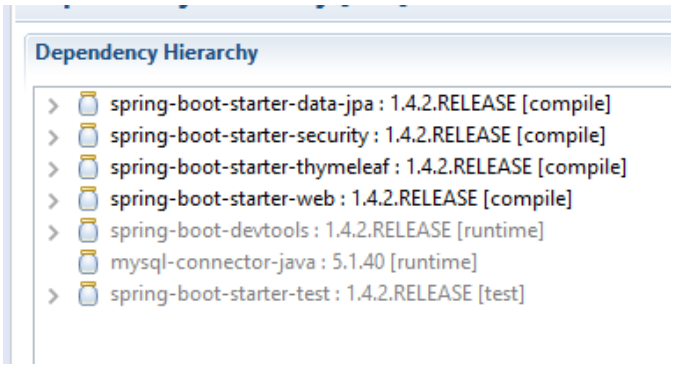
Spring Initializr



Télécharger et importer le projet dans STS

Structure du projet





Les classes et interfaces

model:La classe Ressource

```java
package com.cours.spring.model;

import javax.persistence.Entity;

@Entity
public class Ressource {

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

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }

}
```

Repository: RepositoryRessource

```java
package com.cours.spring.repository;

import org.springframework.data.jpa.repository.JpaRepository;

@Repository("ressourceRepository")
public interface RessourceRepository extends JpaRepository<Ressource,Long> {

}
```

Service: RessourceService

```java
package com.cours.spring.service;

import java.util.List;

public interface RessourceService {

    void ajouterRessource(Ressource r);
    List<Ressource> getRessources();
}
```

Service: RessourceServiceImpl

```java
package com.cours.spring.service;

import java.util.List;

@Service("ressourceService")
public class RessourceServiceImpl implements RessourceService {

    @Autowired
    private RessourceRepository ressourceRepository;

    public void ajouterRessource(Ressource ressource) {

        ressourceRepository.save(ressource);
    }

    @Override
    public List<Ressource> getRessources() {

        return ressourceRepository.findAll();
    }

}
```

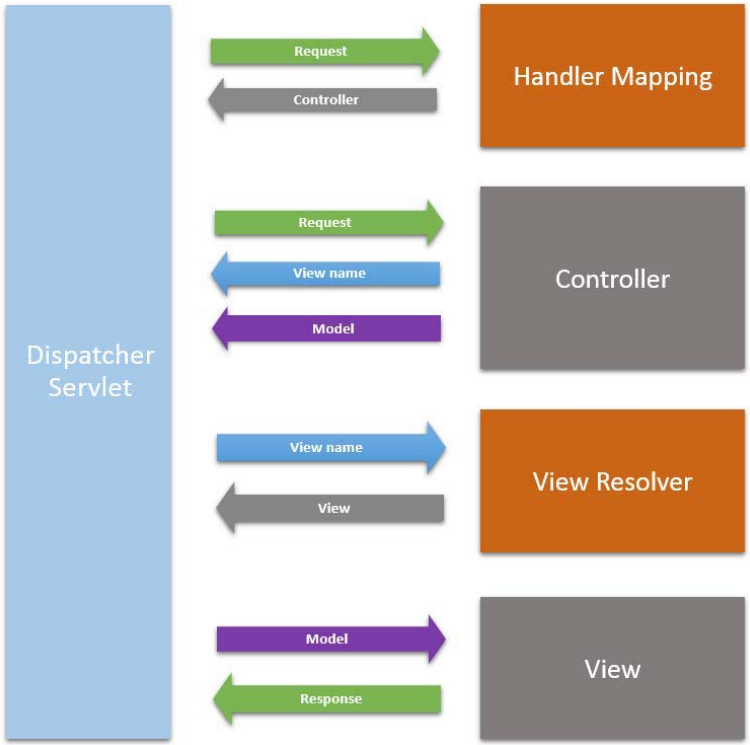
Le contrôleur

```java
package com.cours.spring.controller;

import org.springframework.beans.factory.annotation.Autowired;

@Controller
public class HomeController {
    @Autowired
    private RessourceService ressourceService;

    @RequestMapping(name = "/add", method = RequestMethod.GET)
    @ResponseBody
    public void ajouter(@RequestParam String n) {
        Ressource r = new Ressource();
        r.setNom(n);
        ressourceService.ajouterRessource(r);

    }
}
```

Configuration de la source de données et Hibernate

```
spring.datasource.url = jdbc:mysql://localhost:3306/ressources
spring.datasource.username = root
spring.datasource.password = root
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

SPRING MVC



- DispatcherServlet: Front Controller, c'est le point d'entrée d'une application web MVC.
- HandlerMapping gère les associations entre les urls et les contrôleurs.
- La méthode avec le @RequestMapping qui correspond à l'url est appelée.
- La méthode appelée doit définir le modèle et sélectionner une vue.
- DispatcherServlet Interroge l'interface ViewResolver pour trouver l'mplémentation qui correspond à la vue.
- Par exemple pour thymeleaf, si le nom de la vue est « index », alors DispatcherServlet recherche dans le dossier templates la page index.html.

Contrôleur

```java
import org.springframework.stereotype.Controller;

@Controller
public class RessourceController {

    @RequestMapping(value = "/", method = RequestMethod.GET)
    @ResponseBody
    public String ressource() {
        return "OOO";
    }

}
```

- Arguments supportés par l'action
  - @RequestParam(value="p1« ,defaultvalue="v1",value="v1«)
  - @ModelAttribute: lier un argument à un objet du modèle
  - @CookieValue: récupérer une variable de type Cookie
  - @PathVariable

- Un contrôleur doit être annoté par @Controller
- @RequestMapping: Définit la correspondance entre le contrôleur ou une action du contrôleur et une url.
- Attributs de @RequestParam
  - value:url
  - method: post,get,delete,put,head
  - params: sélection de la méthode selon l'existence ou non de paramètres (valeurs possibles, p1=v1 ou p1!=v1 ou !p1
- Types de retour supportés:
  - ModelAndView: contient le modèle et le nom de la vue à afficher.
  - Model
  - View
  - String : nom de la vue

- Exemple @PathVariable

```
@RequestMapping(value = "/chemin/{v1}")
public String action(@PathVariable("v1") long var) {
```

- ModelAndView

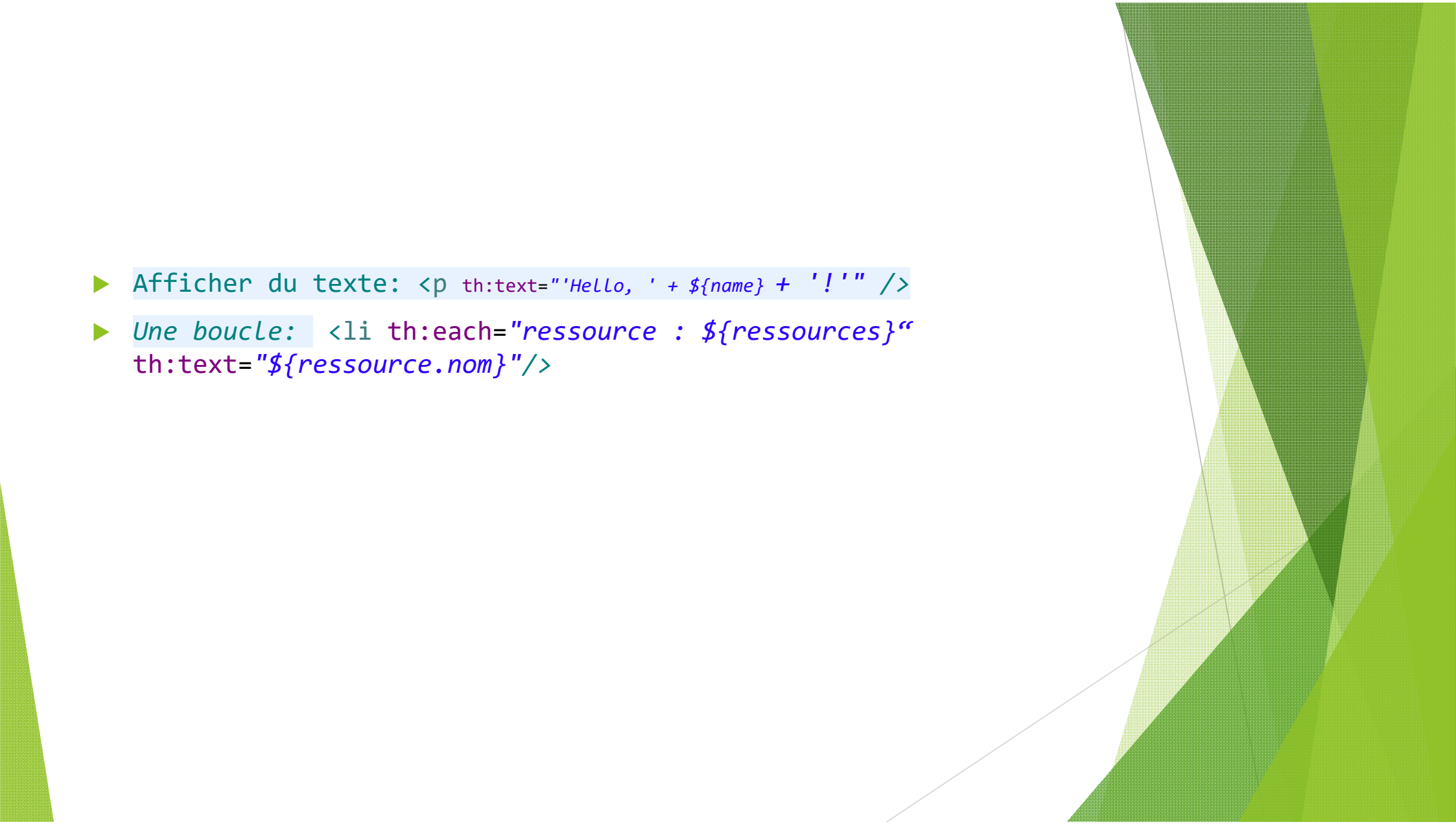
```
public ModelAndView action() {
    //On associe la vue à modelAndView
    ModelAndView modelAndView = new ModelAndView("redirect:/ressources/list");
    //On associe le modàle
    modelAndView.addObject("nomObjet", ressourceService.getRessources());
    return modelAndView;
}
```

- Action(@ModelAttribute Ressource ressource): si ressource existe dans le modèle, alors il sera récupéré sinon il sera instancié et ajouté dans le modèle.

Thymeleaf

- Moteur de vue réalisé en 2014
- Écrit en Java et supporte XML/XHTML/HTML5

- Afficher du texte: `<p th:text="'Hello, ' + ${name} + '!'" />`
- *Une boucle:* `<li th:each="ressource : ${ressources}“ th:text="${ressource.nom}"/>`



La sécurité

Spring Spring Security supporte deux modes d'authentification:

- Authentification HTTP
- Authentification par formulaire
- LDAP
- JAAS
- Java open SSO
- Authentification OpenID
- OAuth

Intégration de la sécurité dans l'application

1. Ajouter la dépendance

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

2. Configuration de la sécurité par formulaire

Ajouter une classe de configuration dans un sous package config

```
@Configuration
@EnableWebSecurity
public class SecuriteConfig extends WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {

http.authorizeRequests()
.anyRequest()
.authenticated() //toutes les requêtes doivent être authentifiées
.and()
.formLogin() //authentification  avec formulaire
.permitAll(); // le formulaire de login doit être accessible à tout le monde


}
```

On peut aussi configure les autorisations par url et par méthode http, exemple:
on peut ajouter les lignes suivantes juste après .auhtoriseRequests():

```
.antMatchers(HttpMethod.GET,"/","images/**","/css/**","/Webjars/**").permitAll()
.antMatchers(HttpMethod.POST,"/images").hasRole("USER")
```

On peut configurer dans la même méthode l'url de destination après déconnexion

En ajoutant:

```
.logout()
    .logoutSuccessUrl("/");
```

3. Comptes utilisateurs

- Dans la mémoire:

```
@Autowired
public void configureInMemoryUsers(AuthenticationManagerBuilder auth) throws Exception{
auth.inMemoryAuthentication()
.withUser("u1").password("p1").roles("ADMIN","USER")
.and()
.withUser("u2").password("p2").roles("USER")
.and()
.withUser("u3").password("p3").roles("USER").disabled(true)
.and()
.withUser("u4").password("p4").roles("USER").accountLocked(true);
}
```

- Ou bien dans la base de données, mais au préalable, il faut déjà créer les classes et interfaces User UserRepository,UserService et UserServiceImpl

```
@Autowired
UserService userService;
```

```
@Autowired
public void configureJpaBasedUsers(AuthenticationManagerBuilder auth) throws Exception{

    auth.userDetailsService(userService);

}
```

Intégration thymeleaf avec spring security

4. Ajouter la dépendance

```
<dependency>
<groupId>org.thymeleaf.extras</groupId>
<artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
```

5. Intégration de la sécurité dans les pages thymeleaf

- Ajouter le namespace

```
xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity4">
```

- exemples d'attributs:

```
Utilisateur:<span sec:authentication="name"/>:
<span sec:auhtentication="authorities"/>
<form method="post" th:action="@{logout}">
<input type="submit" value="log off"/>
</form>
```

6. la classe User

```java
@Entity
public class User {
    @Id @GeneratedValue
    private long id;
    private String username;
    private String password;
    private String[] roles;
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String[] getRoles() {
        return roles;
    }
    public void setRoles(String[] roles) {
        this.roles = roles;
    }

    private User(){

    }

    public User(String username, String password, String... roles) {

        this.username = username;
        this.password = password;
        this.roles = roles;
    }
}
```

7. L'interface UserRepository

```java
@Repository("userRepository")
public interface UserRepository extends CrudRepository<User, Long> {

    User findByUsername(String username);
}
```

8. L'interface UserService

```java
public interface UserService  extends UserDetailsService{

    UserDetails loadUserByUsername(String username) ;
}
```

9. La classe UserServiceImpl

```java
@Service("userService")
public class UserServiceImpl  implements UserService {

   @Autowired
    private  UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user=userRepository.findByUsername(username);
        return new org.springframework.security.core.userdetails.User(username,user.getPassword(),
                Stream.of(user.getRoles()) //Retourne un objet stream de String
                .map(SimpleGrantedAuthority::new) //Transforme le flux de type SimpleGranredAuthority
                .collect(Collectors.toList()));  //retourne une collection de type List
    }

}
```

`.map(SimpleGrantedAuthority::new)` **est éqauivalente à** `.map(role-> new SimpleGrantedAuthority(role))`

L’interfaction Authentication

- L’interfaction Authentication contient les informations de l’utilisateur courant.
- Pour obtenir l’objet Authentication dans le code:

```
SecurityContextHolder.getContext().getAuthentication();

String username=SecurityContextHolder.getContext().getAuthentication().getName();
```

CommandRunner

```
@Bean
@Autowired
CommandLineRunner setUp(UserRepository userRepository){
return (args)->{
if (userRepository.count()<1)
userRepository.save(new User("u1", "p1", "ADMIN","USER"));
userRepository.save(new User("u2", "p2", "USER"));
userRepository.save(new User("u3", "p3", "USER"));
userRepository.save(new User("u4", "p4", "USER"));
};
}
}
```

Stream

Un stream (java.util.stream.Stream) supporte deux types d'opérations: les opérations intermédiaires et les opérations terminales

les opérations intermédiaires s'effectuent de façon lazy et renvoient un nouveau stream,(ex map et filter) , tant qu'aucune opération terminale n'aura été appelée sur un stream pipeline, les opérations intermédiaires ne seront pas réellement effectuées

quand une opération terminale sera appelée (Stream.reduce ou Stream.collect), alors toutes les opérations intermédiaires seront effectuées, puis l'opération terminale ajoutée (les streams seront dits consommés, ils seront détruits et ne pourront plus être utilisés)

Création d'un stream

appel de la méthode stream ou parallelStream sur une collection, mais un nombre de méthodes ont été ajoutées aux classes existantes: String.chars() renvoie un IntStream, BufferedReader.lines(), Random.ints()

Il existe aussi des méthodes statiques dans la classe Stream:

- Stream.iterate(1,x->x*2) construit une suite de puissance de 2 à partir de 1.

-Stream.of

Les opérations intermédiaires

Elles peuvent être statefull ou stateless

- Les opérations stateless s'effectuent sur les éléments du stream un à un sans prendre en compte les autres éléments du flux

- Les opérations statefull ont besoin de connaitre généralement l'ensemble du stream pour donner un résultat (distinct, sorted)

Les opérations terminales

Il existe deux types de réductions dans l'API les réductions simples(sum, max,count) et les réductions mutables