SpringMVC ThymeLeaf

1/ Flot d'éxecution

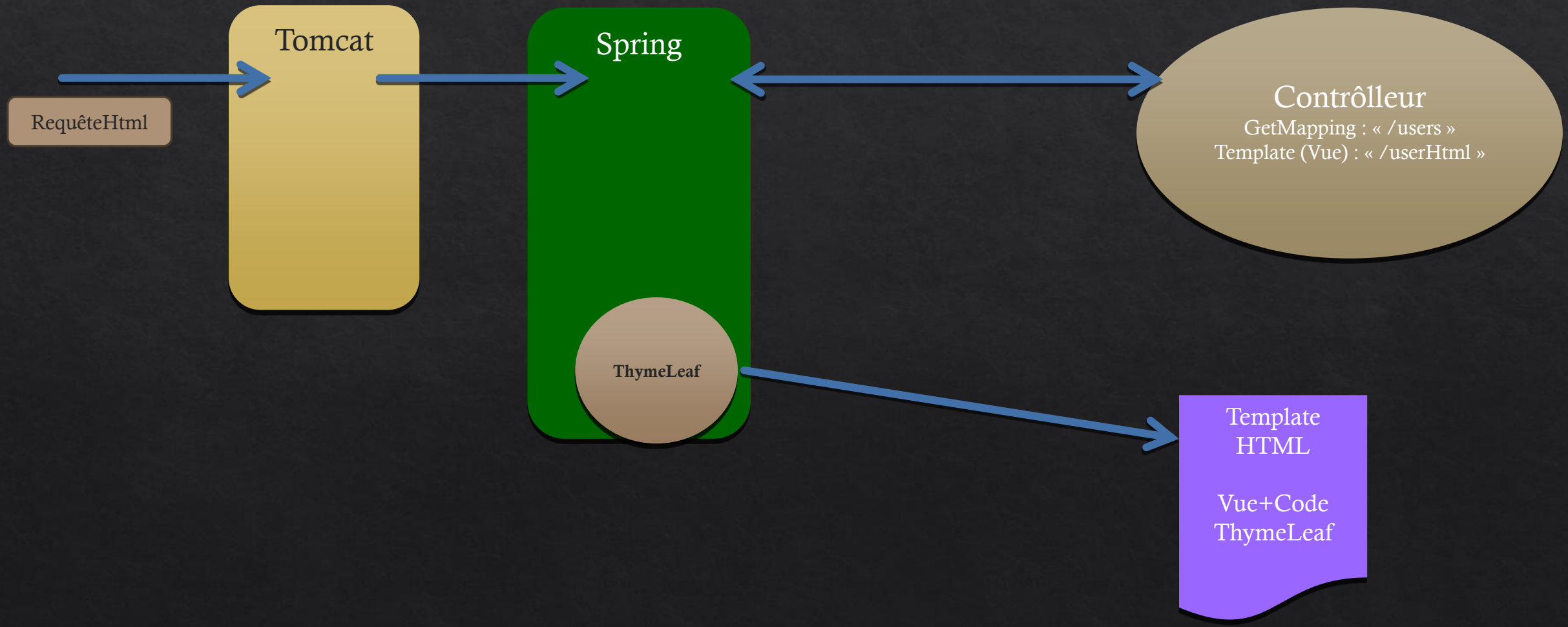
2 / Traiter la requête & choisir le template : @Controller

3 / Template et utilisation des données du model

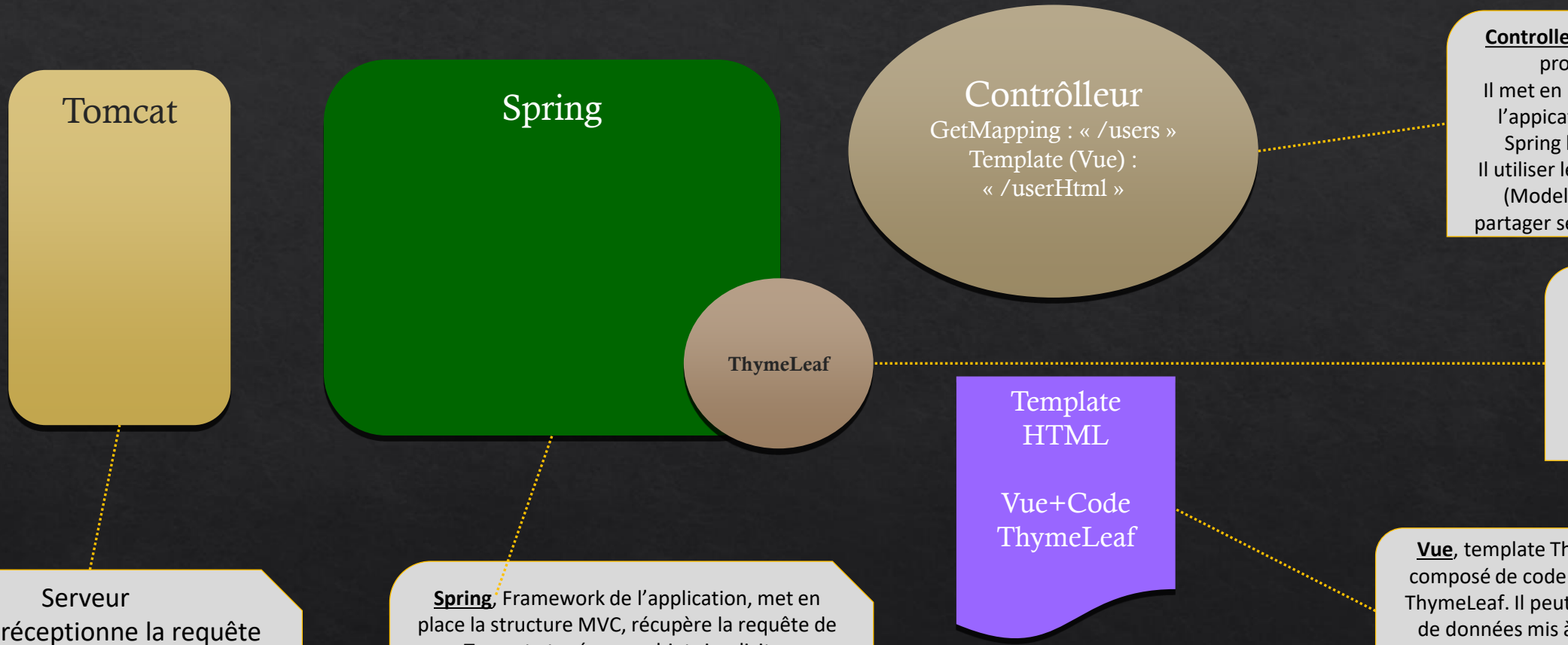
SpringMVC ThymeLeaf

1/ Flot d’éxecution

Mecanique


Tomcat
Spring
RequêteHtml
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »
ThymeLeaf
Template
HTML
Vue+Code
ThymeLeaf

Mecanique : Les acteurs



Controller, Objet que nous programmons.
Il met en place le métier de l'appication et indique à Spring la vue à utiliser.
Il utiliser les objets de Spring (Model, etc) pour pour partager ses donnée à la vue.

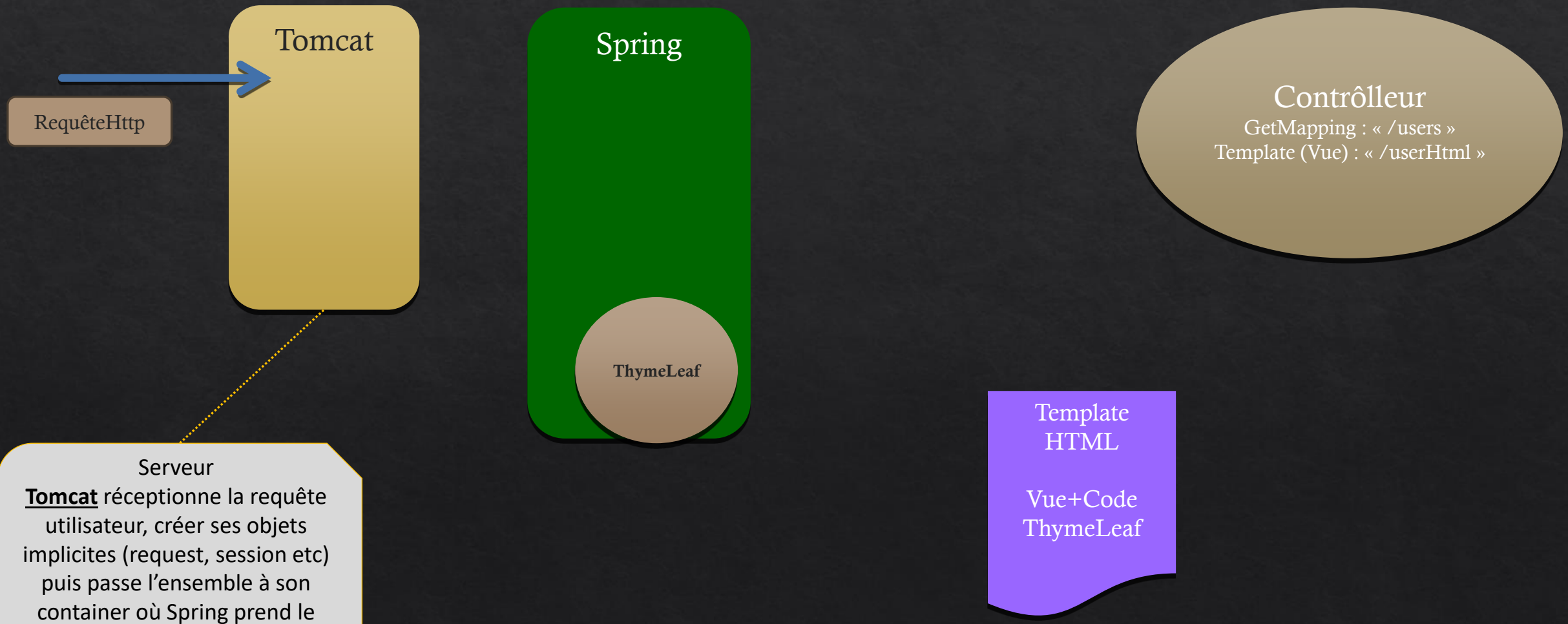
ThymeLeaf, Moteur de template, traduit les balises spécifique du template en code HTML

Serveur
Tomcat réceptionne la requête utilisateur, créer ses objets implicites (request, session etc) puis passe l'ensemble à son container où Spring prend le relai

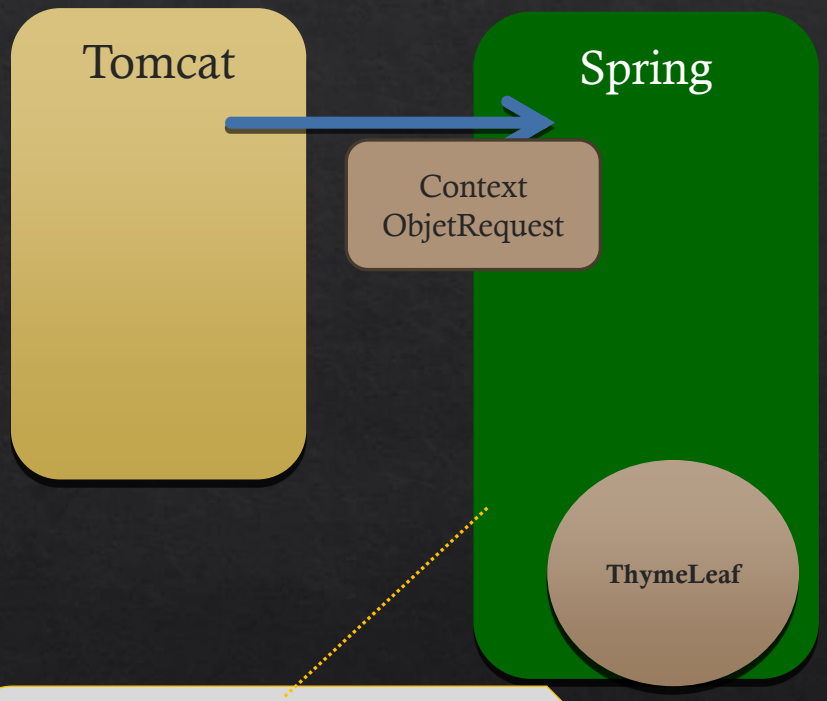
Spring, Framework de l'application, met en place la structure MVC, récupère la requête de Tomcat et créer ses objets implicites (Model,RequestParameters etc)
Il passe la requête au bon contoller en fonction du Mapping puis en fonction de la vue renvoyée par celui-ci, il utilisera ThymeLeaf pour la créer et la renvoyer au client

Vue, template ThymeLeaf, fichier composé de code HTML et de code ThymeLeaf. Il peut utiliser les objets de données mis à diposition pour crée le fichier html renvoyé à l'utlisateur

Mecanique : Flux



Mecanique : Flux



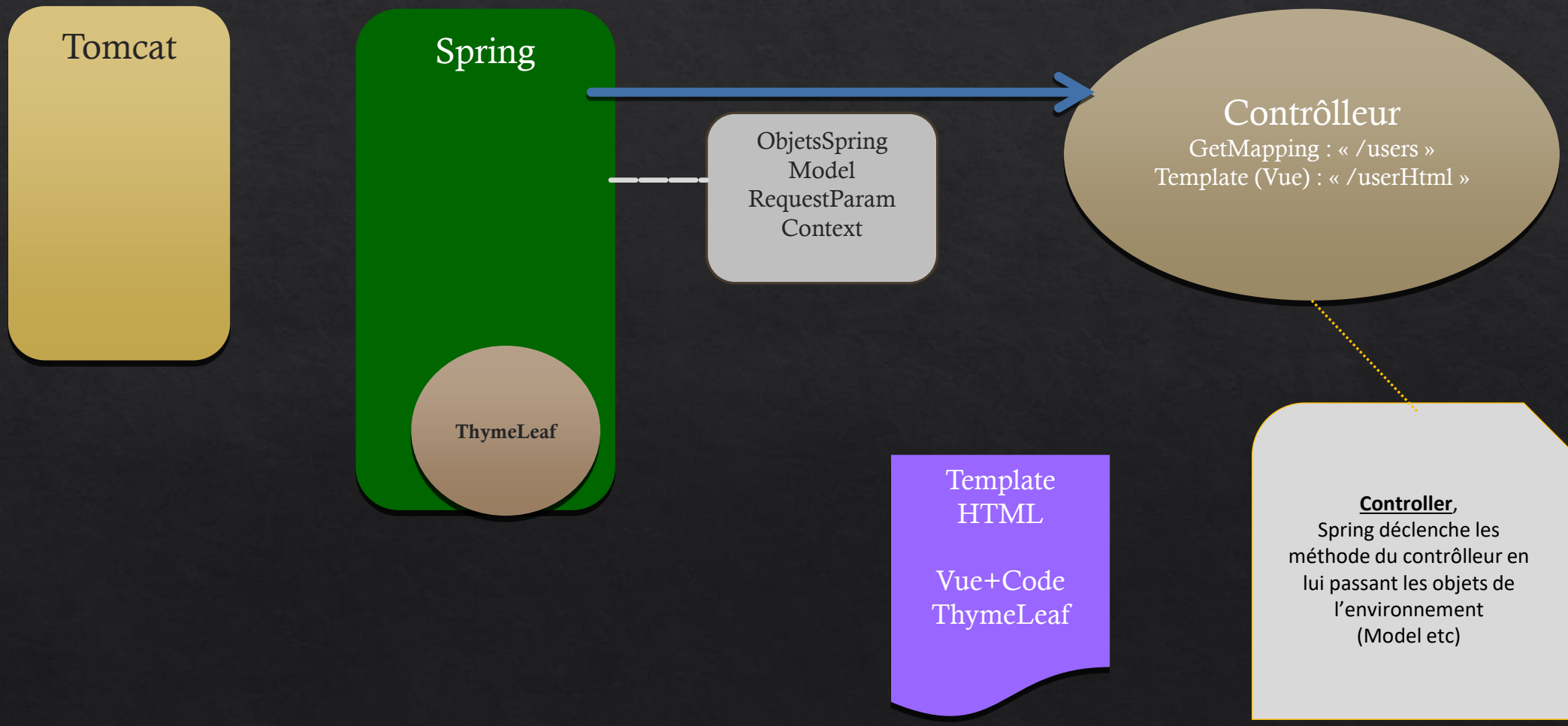
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »

Spring, Framework de l'application, met en place la structure MVC, récupère la requête de Tomcat et créer ses objets implicites (Model,RequestParameters etc)
Il passe la requête au bon contoller en fonction du Mapping puis en fonction de la vue renvoyée par celui-ci, il utilisera ThymeLeaf pour la créer et la renvoyer au client

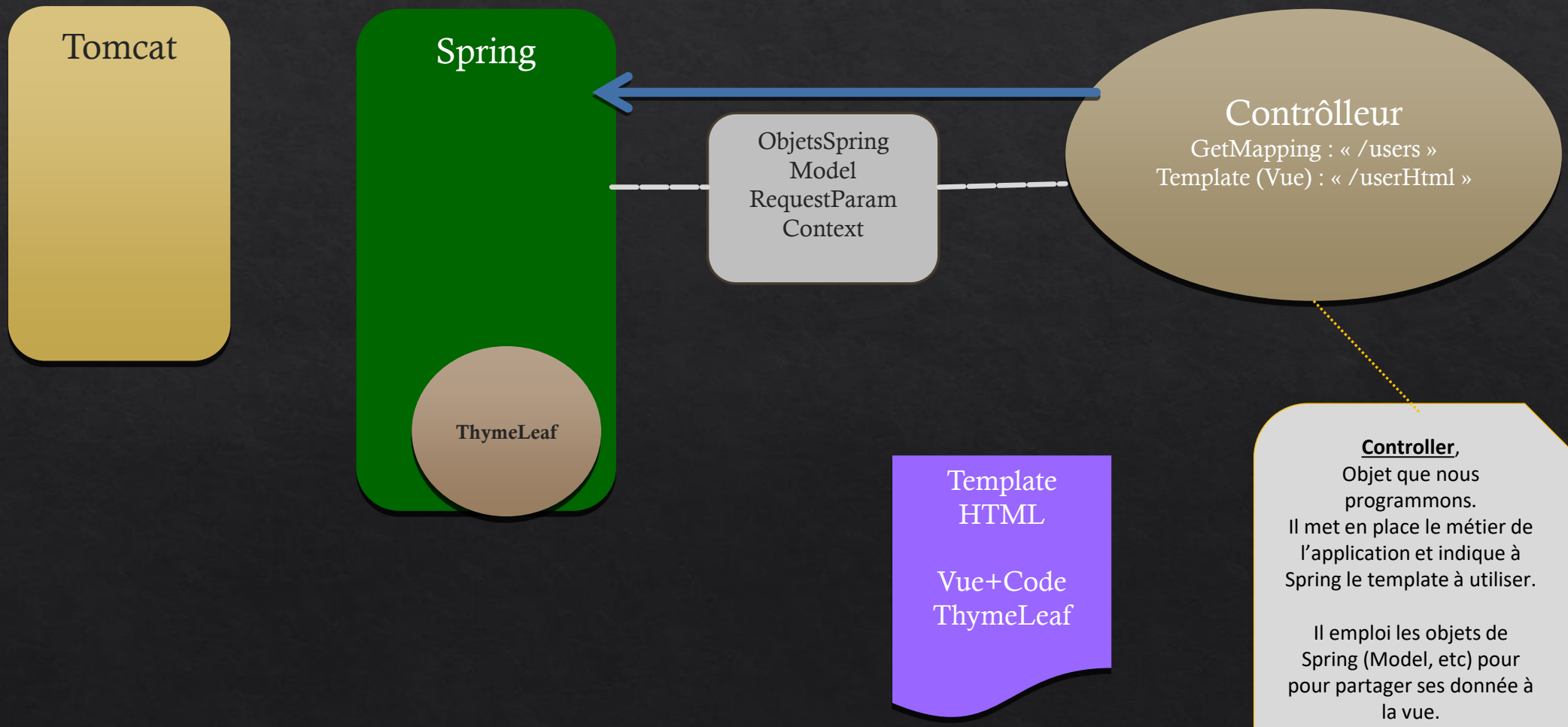
Template
HTML

Vue+Code
ThymeLeaf

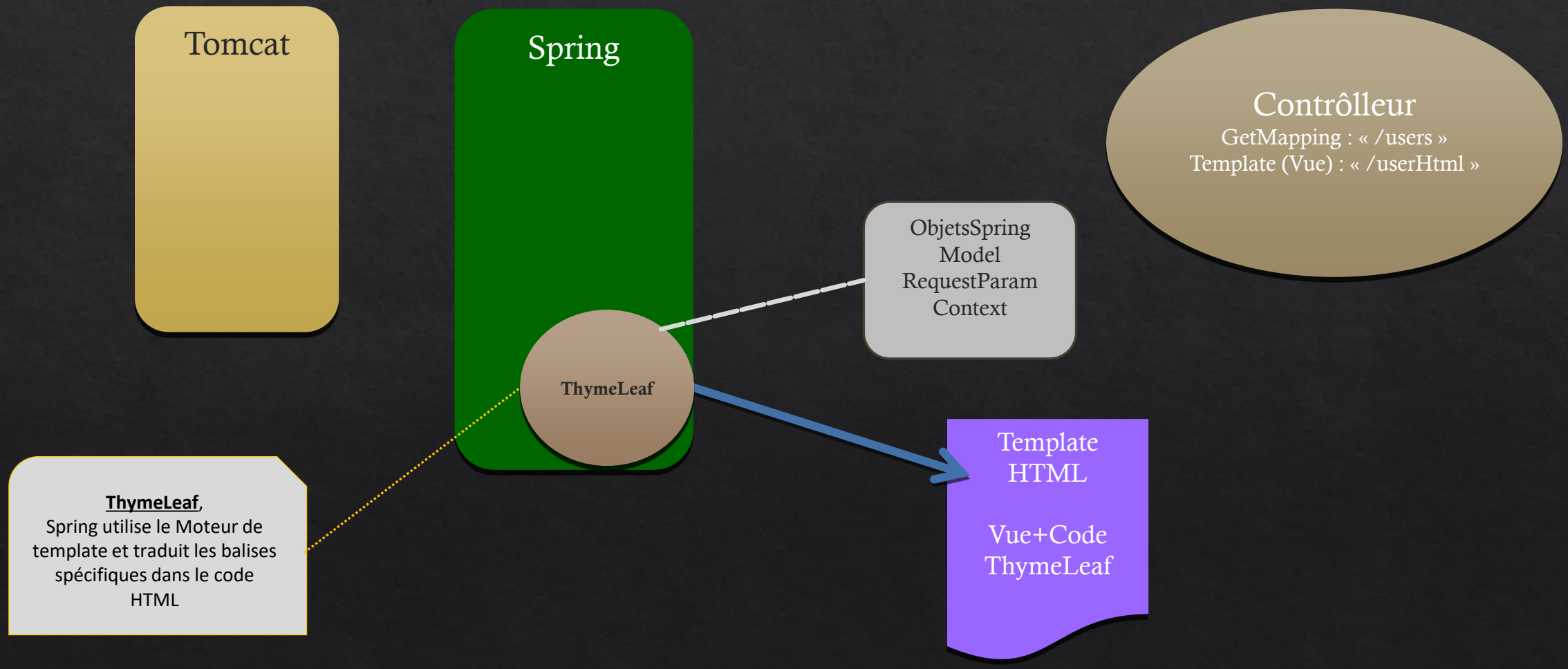
Mecanique


Tomcat
Spring
ThymeLeaf
ObjetsSpring
Model
RequestParam
Context
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »
Template
HTML
Vue+Code
ThymeLeaf
Controller,
Spring déclenche les
méthode du contrôlleur en
lui passant les objets de
l'environnement
(Model etc)

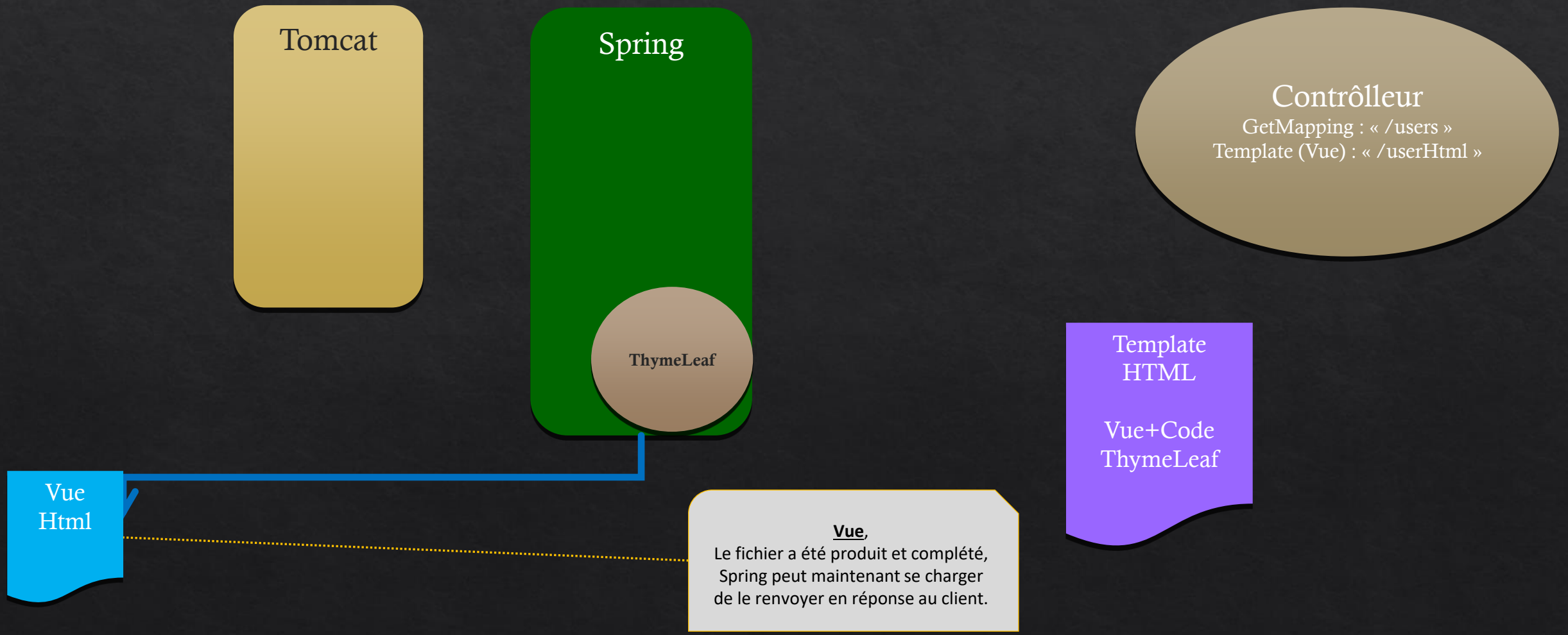
Mecanique


Tomcat
Spring
ThymeLeaf
ObjetsSpring
Model
RequestParam
Context
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »
Template
HTML
Vue+Code
ThymeLeaf
Controller,
Objet que nous programmons.
Il met en place le métier de l'application et indique à Spring le template à utiliser.
Il emploi les objets de Spring (Model, etc) pour pour partager ses donnée à la vue.

Mecanique


Tomcat
Spring
ThymeLeaf
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »
ObjetsSpring
Model
RequestParam
Context
Template
HTML
Vue+Code
ThymeLeaf
ThymeLeaf,
Spring utilise le Moteur de template et traduit les balises spécifiques dans le code HTML

Mecanique


Tomcat
Spring
ThymeLeaf
Contrôlleur
GetMapping : « /users »
Template (Vue) : « /userHtml »
Template
HTML
Vue+Code
ThymeLeaf
Vue
Html
Vue,
Le fichier a été produit et complété,
Spring peut maintenant se charger
de le renvoyer en réponse au client.

SpringMVC ThymeLeaf

2 / Traiter la requête & choisir le template : @Controller

Spring Controller

```
@Controller
public class PostController {

   @GetMapping("/posts")
   public String onePost()
   {
            return "onePost";
   }
}
```

Le plus simple possible.

@Contoller pour indiquer à spring que cet objet répondra à une requête client dans sa mécanique MVC

@GetMapping pour indiquer à quelle url il répond

return « onePost »

Renvoi à Spring le nom du template à utiliser pour renvoyer la réponse client.

Ici il utilisera onePoste.html placé dans le répertoire template

Spring Controller

```
@Controller
public class PostController {


   @GetMapping("/posts/{id}")
   public String onePostRest(@PathVariable int id)
   {

      return "onePost";
   }
```

@GetMapping("/posts/{id}")
Un mapping plus évolué indiquant une variable à utiliser et issue de l'url de la request.

@PathVariable int id, indique à Spring de nous injecter la valeur de cet id depuis le path dans un variable utilisable dans notre contôleur.

Spring Controller

```
@GetMapping("/post")
 public String onePost( @RequestParam int id)
{   //Pour une requête /post?id=3

    return "onePost";
}
```

@GetMapping("/post")
Un mapping pour une requête de type /post?id=3.

@RequestParam int id, indique à Spring de nous injecter la valeur de cet id depuis les paramètres de requête, dans un variable utilisable dans notre contrôleur.

Spring Propriétés

```
@Controller
public class PostController {

   @Autowired
   private Environment env;

   @Value("${quizz.url}")
   String quizzUrl;
```

Dans le fichier application.properties :

```
quizz.url=https://opentdb.com/api.php?amount=10&category=17
```

```
@Autowired
   private Environment env;
```

Spring va injecter l'objet environnement. Il met à disposition des ressources comme par exemple les paramètres indiqués dans le fichier application.properties

Dans le code du contrôlleur ensuite :

```
env.getProperty("quizz.url");
```

```
@Value
```

Même comportement mais plus light, spécifie directement la clef de la propriété.

Attention cela ne fonctionne que si Spring prend en charge la classe parente. Ici c'est le cas puisque l'on est dans une contrôlleur

Spring et Les Listeners

- Les Listeners comme ServletContextListener ou HttpSessionListener ne sont pas des objets « spring » pris en charge naturellement par le framwork dans le serveur embarqué de spring-boot. (On peut aussi remarquer l'import de l'annotation @WebListener qui vient de jakartaa et non pas de springframework)

Pour pouvoir utiliser ses objets utilitaires avec spring-boot, ainsi que les filtres il faut que spring les enregistre. Si le déploiement se fait ensuite sur un serveur standard le problème ne se pose pas.

On utilise pour le cas l'annotation @ServletComponentScan sur la classe de l'application

Où l'on trouve déjà @SpringBootApplication

Spring et Les Listeners

L'enregistrement « à la main » peut aussi se faire si le problème se pose.

```
@Bean
public ServletListenerRegistrationBean<ServletContextListener> customListenerBean() {
   ServletListenerRegistrationBean<ServletContextListener> bean = new
ServletListenerRegistrationBean();
   bean.setListener(new ApplListener());
   return bean;

}
@Bean
public ServletListenerRegistrationBean<HttpSessionListener> customSessionListenerBean() {
   ServletListenerRegistrationBean<HttpSessionListener> bean = new ServletListenerRegistrationBean();
   bean.setListener(new Initialisation());
   return bean;
}
```

SpringMVC ThymeLeaf

2 / Préparer les données pour le template

Spring Model

Objet implicite de Spring, on peut l'utiliser en argument des méthodes de nos contrôleurs tout comme l'a fait pour nos paramètres des requêtes d'url.

Spring se chargera de créer l'objet et de nous le mettre à disposition.

Le scope de l'objet est la requête et il nous permet d'y stocker sous forme de clefs/valeurs tout ce dont on a besoin pour alimenter le template associé.

Il n'est pas le seul objet auquel le template aura accès mais il est pratique d'utilisation.

Même si techniquement ce n'est pas exactement le cas, on peut le considérer comme un proxy sur l'objet Request utilisé dans les servlets.

Spring Model

```
@Controller
public class PostController {

   @Autowired
   private Environment env;

   @GetMapping("/posts/{id}")
   public String onePostRest(Model model, @PathVariable int id)
   {
       String petitMessage = « Salut Thymeleaf ! »;
      model.addAttribute("message", petitMessage);
      return "onePost";
   }
}
```

Ici l'objet model nous permet de stocker un petit message.

Une fois le model chargé, on retourne le nom du template et Spring à maintenant tout pour fabriquer la vue pour notre client.

Il utilisera pour cela le templateEngine de ThymeLeaf, pour compléter le HTML.

${message}récupèrera la valeur

th:text=« ${message} » par exemple que l'on verra plus tard, indiquera l'emplacement où l'afficher dans la vue

Spring Model

```
@Controller
public class PostController {

   @Autowired
   private Environment env;

   @GetMapping("/posts/{id}")
   public String onePostRest(Model model, @PathVariable int id)
   {
      RestClient restClient = RestClient.builder()
                  .baseUrl(env.getProperty("post.placehorlder.url"))
                  .defaultHeader("Content-Type", "application/json")
                  .build();

      Post onePost = restClient.get()
            .uri("/{id}",id)
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .body(Post.class);

      model.addAttribute("onePost", onePost);
      return "onePost";
   }
}
```

Exemple plus intéressant :

Ici l'objet model nous permet de stocker notre objet métier Post.

Celui-ci contenant les données issues de la requête sur l'api rest.

Une fois le model chargé, on retourne le nom du template et Spring à maintenant tout pour fabriquer la vue pour notre client.

Il utilisera pour cela le templateEngine de ThymeLeaf,

SpringmMVC : Documentation

Documentation SpringMVC :

https://docs.spring.io/spring-framework/reference/web/webmvc.html

La flexibilité du framework provient des nombreuses formes que peuvent prendre les signatures des méthodes dans les contrôleurs.

Les arguments :

https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/arguments.html

Les valeurs retournées :

https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-methods/return-types.html

SpringMVC ThymeLeaf

3 / Template et utilisation des données du model

Thymeleaf : Template HTML

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
lang="en">
<head>
   <meta charset="UTF-8">
   <title>Liste des Post</title>
</head>
<body>

            <div  th:text="${message}"> MyFakeMessage</div><br>

</body>
</html>
```

Les template de Thymeleaf sont des fichiers HTML.

Le système d'attributs permet d'accéder aux objets stockés dans notre model associé et de les manipulers.

- Affichages
- Boucles
- Conditionnelles (if…)
- Etc etc à découvrir

Pour l'exemple l'attibut th:text remplacera le contenu de la balise ou il est par la valeur retrouvée depuis le modèle "${message}"

Thymeleaf : Template HTML

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="http://www.thymeleaf.org" lang="en">
<head>
   <meta charset="UTF-8">
   <title>UnPost</title>
</head>
<body>
<div>
   Affiche le post n° : <span  th:text="${onePost.getId()}">MyFakePost</span><br>
   <div  th:text="${onePost.getTitle()}"> MyFakePostTitle</div><br>
   <div  th:text="${onePost.getBody()}"> MyFakePostBody</div><br>
</div>
</body>
</html>
```

Ici c'est un objet métier que l'on retrouve et dont on affiche les propriétés.

Thymeleaf : Template HTML

```
<li th:each="item : ${listePost}" th:text="${item.body}">Item</li>
```

Il existe des dizaines d'attributs dans Thymeleaf, nous ne verrons que les bases.

Utiliser la doc officielle pour retrouver les outils appropriés à vos besoins spécifiques.

Thymeleaf : Documentations

Thymeleaf est un outils de templating complexe qui s'intègre à différents types d'applications.
Pour aller à l'essentiel voici des liens vers les contextes qui nous intéresseront :

Thymeleaf intégration avec Spring : (Attention Spring n'est pas Spring-boot)

https://www.thymeleaf.org/doc/tutorials/3.1/thymeleafspring.html

Thymeleaf dans une application sans Spring

https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.html

Site Officiel :

https://www.thymeleaf.org/index.html