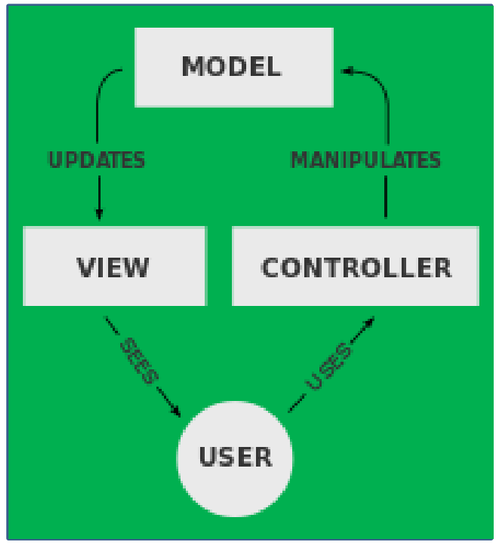
de Spring à Spring Boot

les APIs REST et le WEB MVC avec Spring Web MVC

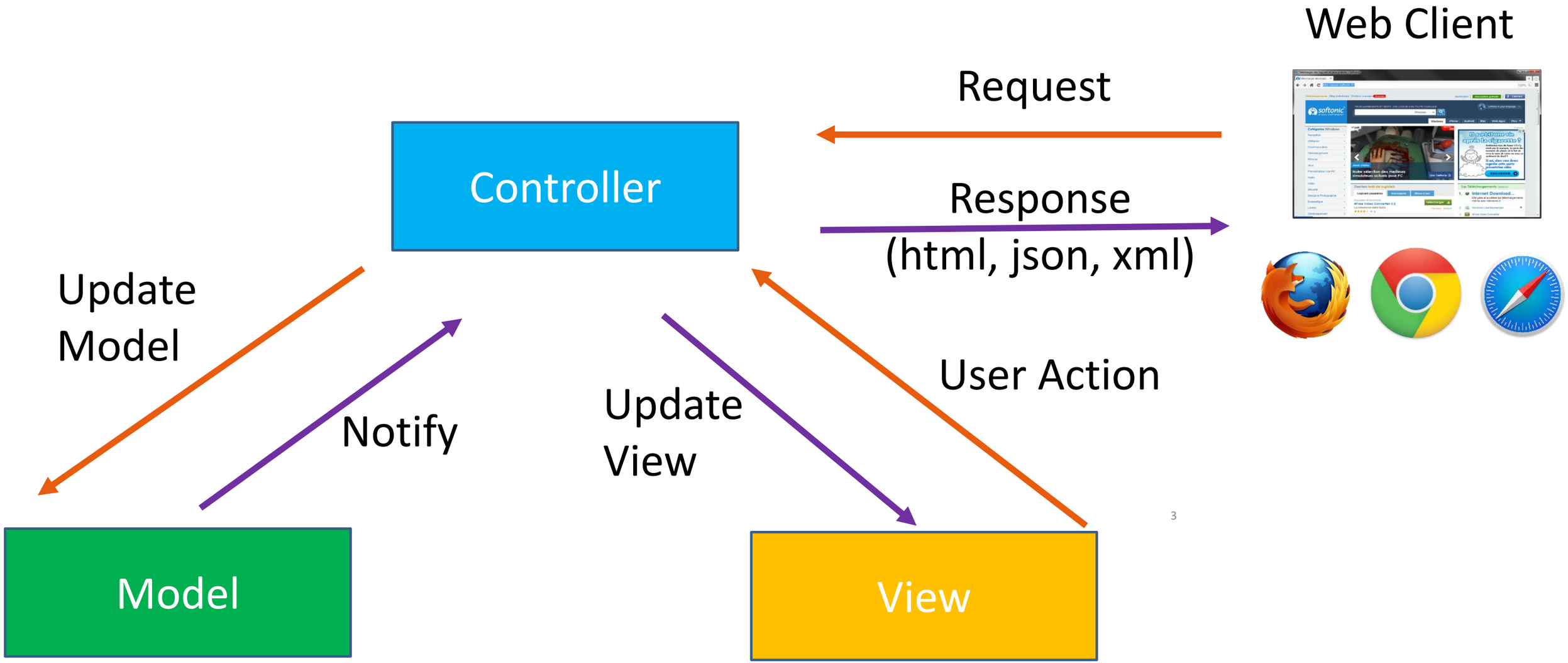
MVC

Modèle de conception avec trois composants indépendants

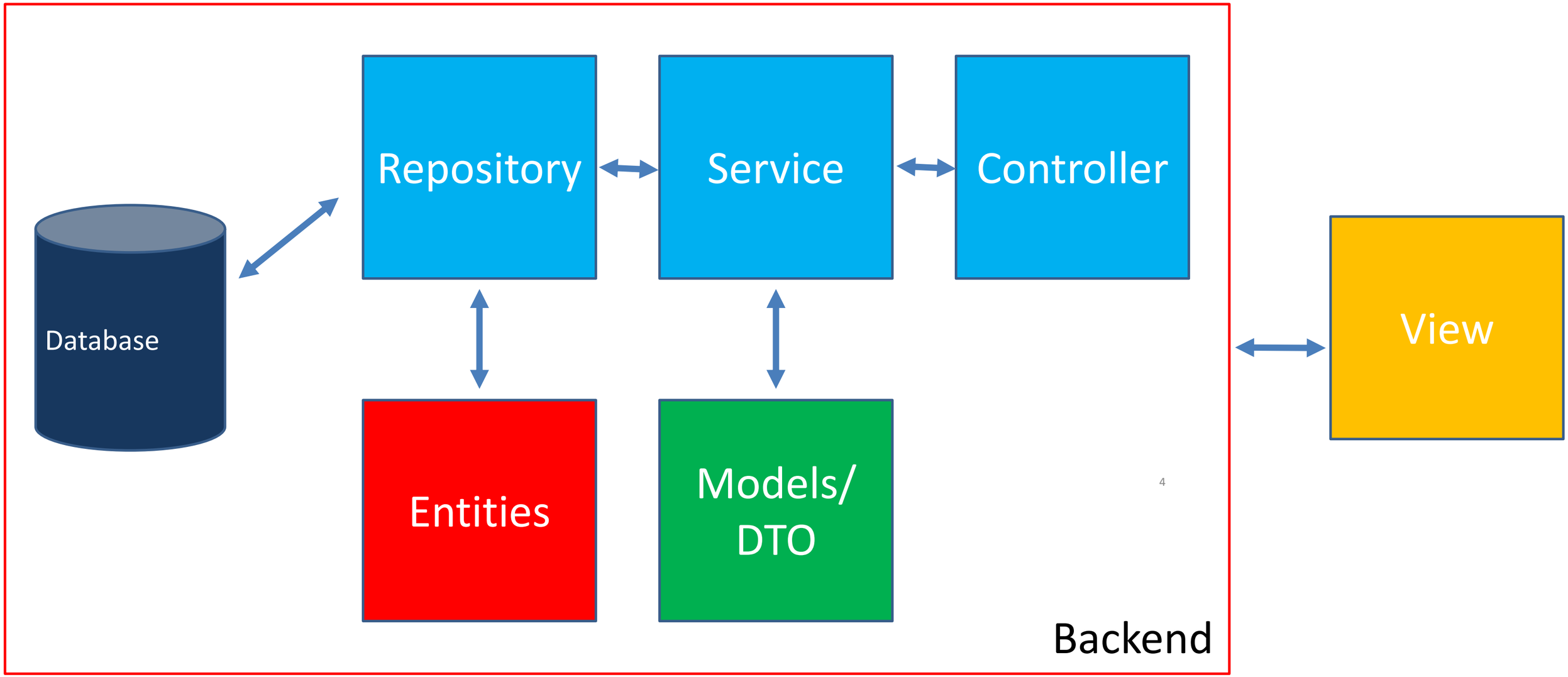
- Modèle – gère les données et la logique applicative
- Vue – couche de présentation
- Contrôleur – convertit l’entrée en commandes et les envoie à la vue ou au modèle



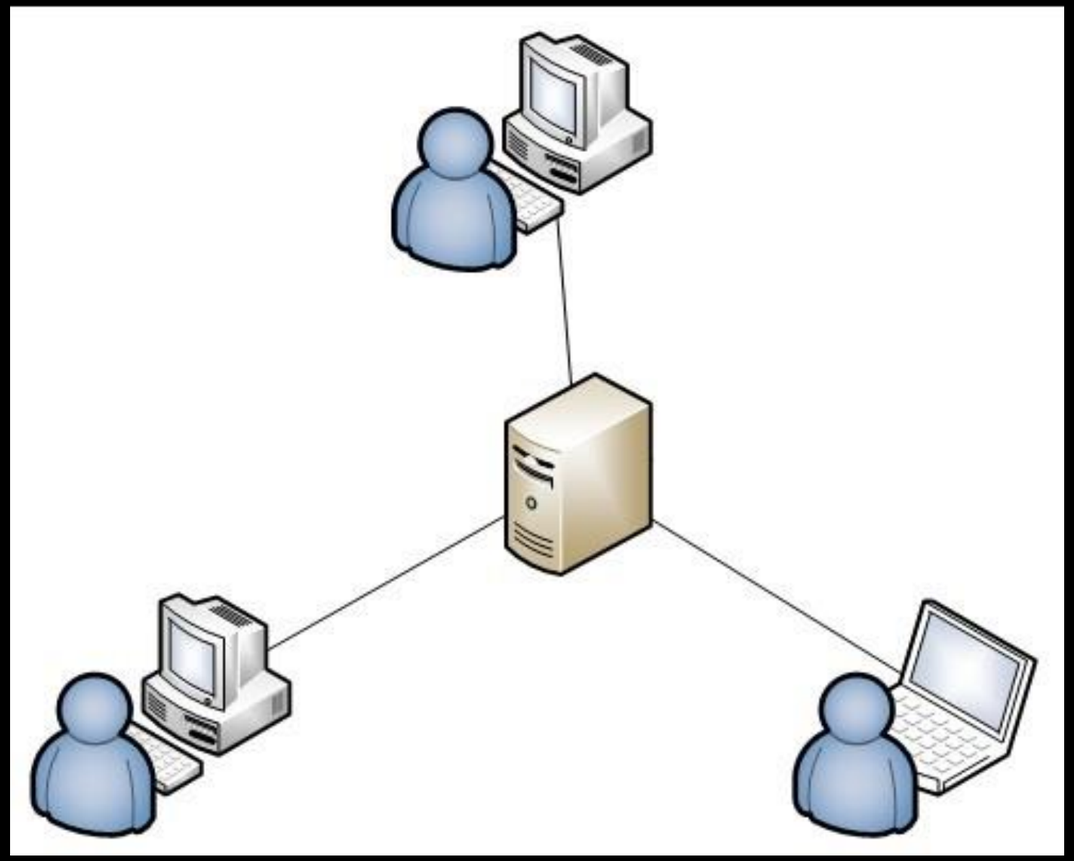
MVC – Control Flow



Architecture



communication client/serveur

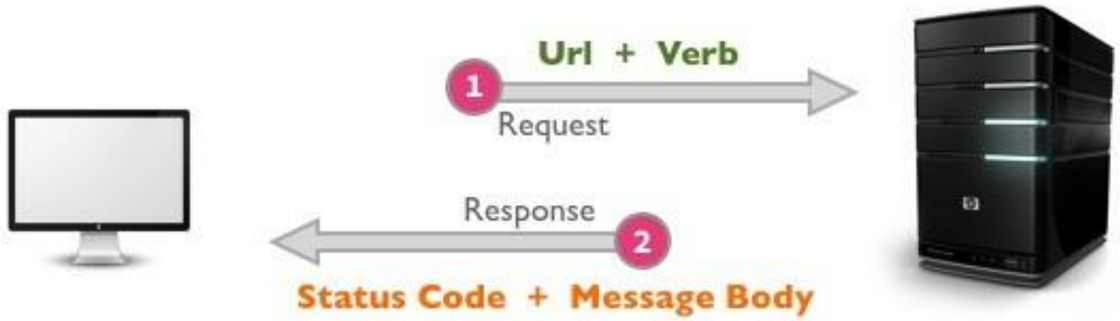


communication client/serveur

HTTP : les concepts

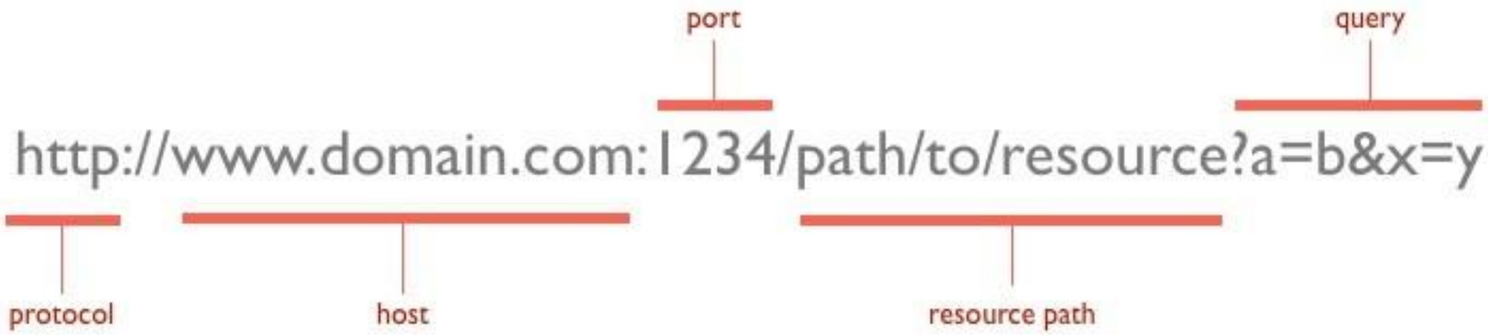
HTTP s'appuie sur 4 concepts fondamentaux :

– le binôme requête/réponse
– les URLs
– les verbes
– les codes de statut.



communication client/serveur

HTTP : les URLs



Les urls sont à la base du fonctionnement de http car elles permettent d'identifier une ressource :

– **protocol** : le protocole utilisé (http, https, ftp, news, ssh...)
– **host** : nom de domaine identifiant le serveur (FQDN)
– **port** : le port utilisé (80 pour http, 443 pour https, 21 pour ftp)
– **ressource path** : identifiant de la ressource sur le serveur
– **query** : paramètres de la requête.

communication client/serveur

HTTP : les verbes

Les **verbes** permettent de manipuler les ressources identifiées par les URLs. Ceux principalement utilisés sont :

– **GET** : le client demande à lire une ressource existante sur le serveur
– **POST** : le client demande la création d'une nouvelle ressource sur le serveur
– **PUT** : le client demande la mise à jour d'une ressource déjà existante sur le serveur
– **DELETE** : le client demande la suppression d'une ressource existante sur le serveur.

Ils sont invisibles pour l'utilisateur mais sont envoyés lors des échanges réseaux. Chaque requête est accompagnée d'un verbe pour indiquer l'action à effectuer sur la ressource ciblée.

```
GET http://welcome.com.intraorange/
GET http://www.monsite.fr/index.php
POST http://api.orange.com/monappli/users/
```

communication client/serveur

HTTP : les codes de statut

Chaque requête de la part d'un client reçoit une réponse de la part du serveur, comportant un **code de statut**, pour informer le client du bon déroulement ou non du traitement demandé.

Ces codes de statut sont rangés par **plages numériques** :

– **1xx** : message d'information provisoire
– **2xx** : requête reçue, interprétée, acceptée et traitée avec succès
– **3xx** : message indiquant qu'une action complémentaire de la part du client est nécessaire (exemple : redirection vers une autre url)
–**4xx** : erreur du serveur du fait des données en entrée envoyées par le client (exemple : authentification, autorisations, paramètres d'entrée)
– **5xx** : erreur du serveur du fait d'un motif interne au serveur (exemple : indisponibilité d'un composant du serveur, erreur inattendue).

Spring Controllers

- Defined with the **@Controller** annotation:

```
@Controller
public class HomeController {
…
}
```

- Controllers can contain multiple actions on different routes.

Get Mapping

- Easier way to create route for a GET request:

```
@GetMapping("/home")
public String home() {
	return "home-view";
}
```

- This is alias for **RequestMapping** with method GET

Post Mapping

- Similar to the **GetMapping** there is also an alias for **RequestMapping** with method POST:

```
@PostMapping("/register")
public String register() {
        …
}
```

- Similar annotations exist for all other types of request methods

Controller Routing

- You can set all actions in a controller to start with a given routing:

```
@Controller
@RequestMapping("/admin")
public class AdminController {
  @GetMapping("/")
  public String adminPanel() {
        …
  }
}
```

- Calling the **adminPanel()** will be done on route "**/admin/**"

Controller Actions

- Annotated with with **@RequestMapping(…)**

```
@RequestMapping("/home")
public String home() {
    return "home-view";
}
```

- Or

```
@RequestMapping("/home")
public ModelAndView home(ModelAndView mav) {
    mav.setViewName("home-view");
    return mav;
}
```

Request Mapping

- Problem when using **@RequestMapping** is that it accepts all types of request methods (get, post, put, delete, head, patch…)

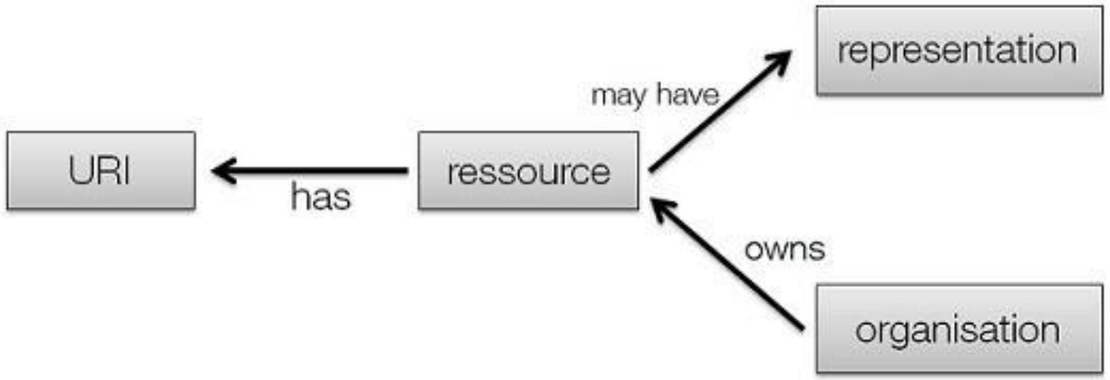
- Execute only on GET requests :

```
@RequestMapping(value="/home", method=RequestMethod.GET)
public String home() {
    return "home-view";
}
```

qu'est-ce qu'une API REST ?

qu'est-ce qu'une API REST ?

Une **API REST (REpresentational State Transfer)** permet à une application d'exposer les services qu'elle offre aux autres applications (pourvues d'une IHM ou pas).



REST s'articule autour de la notion de **ressource** :

– une ressource représente n'importe quel concept (une commande, un client, un message...)
– une représentation est un document qui capture l'état actuel d'une ressource (au format Json, XML, pdf...)
– une ressource appartient à une organisation (une entreprise, un service public...)
– une ressource est accessible via une URI.

HTTP GET

Utilisé pour récupérer des tableaux de données

GET /items

Response

```
[
{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
},
...
]
```

Server

HTTP GET

Utilisé pour récupérer des entités de données uniques

Web Client

GET /items/1

Response

{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}

Server

HTTP POST

Utilisé pour enregistrer des données

Web Client

```
{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}
```

POST /items

Response

Server

HTTP PUT

Utilisé pour mettre à jour les données.

Web Client

```
{
'id': 32,
'name': 'Read News',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}
```

PUT /items/1

Response

Server

HTTP DELETE

Utilisé pour supprimer des données.

Web Client

DELETE /items/delete/1

Response

OK Response

Server

exemple d'API : ToDoList

exemple d'API : ToDoList

1/ identifier les ressources qui constitueront l'API

La plupart du temps on retrouve :

– des ressources **entités** : concepts manipulés par l'API
– des ressources **composites** : agrégation de plusieurs ressources entités en une seule
– des ressources **collections** d'entités ou de composites.

Dans le cas d'une API de **ToDoList**, on peut imaginer avoir les ressources suivantes :

– entités : **ToDoList** et **ToDoItem**
– collections : **ToDoLists** et **ToDoItems**.

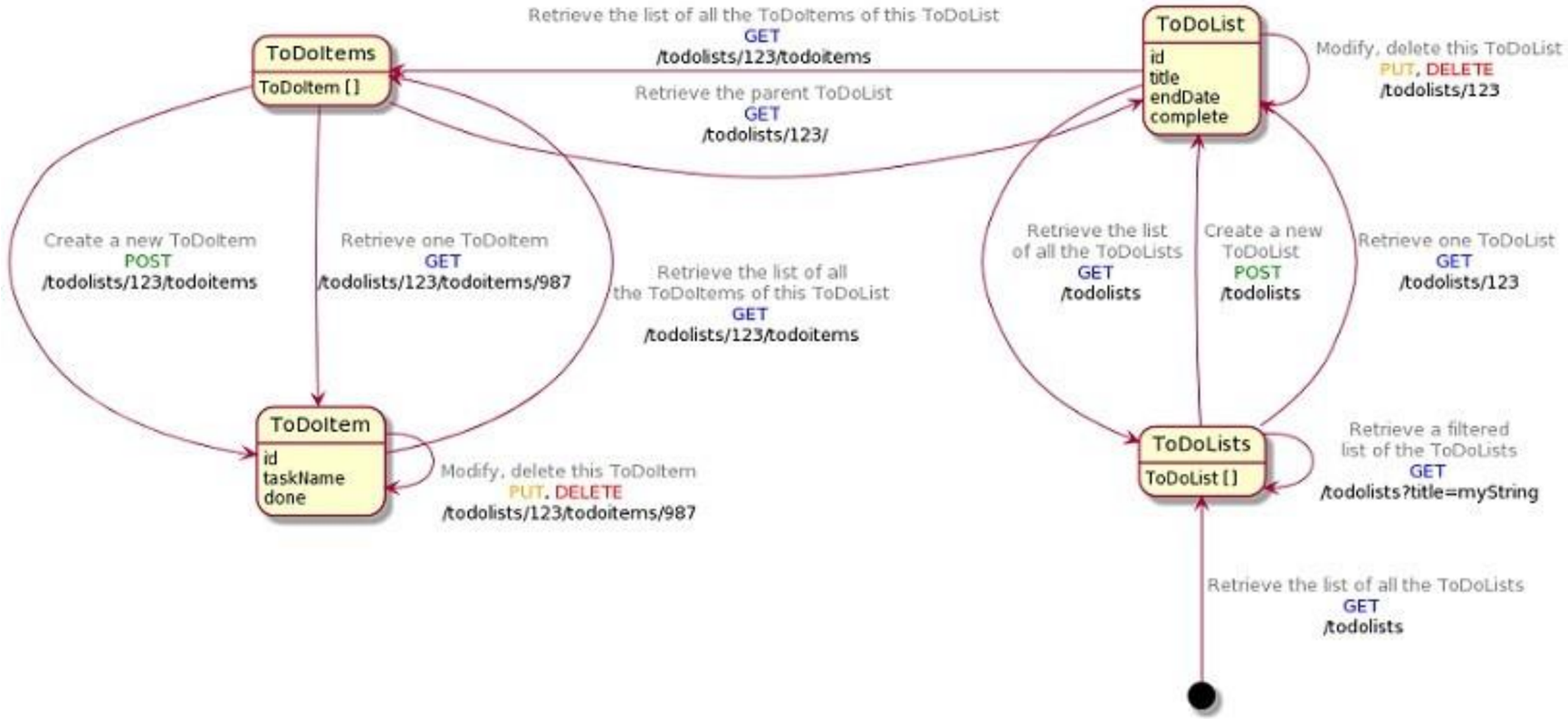
2/ déterminer les URLs de ces ressources et les liens activables

Pour chaque ressource, il faut déterminer :

– comment y accéder (leur URL)
– les actions autorisées :
    - changement d'état de la ressource
    - navigation vers une autre ressource.

exemple d'API : ToDoList

Pour représenter plus clairement les ressources et leur *cinématique*, on peut utiliser un **diagramme d'interactions** :



Response Body On MVC Controller

Returning plain-text in MVC controller:

```
@GetMapping('/info/{id}')
@ResponseBody
public String getInfo(@PathVariable Long id){

  ...
  return 'Plane text';
}
```

Setting the correct Response Code

```
@GetMapping('{id}/info')
@ResponseStatus(HttpStatus.OK)
public GameInfoView getInfo(@PathVariable Long id){

  GameInfoView gameInfo = this.gameService.getInfoById(id);

  return new Gson().toJson(gameInfo);
}
```

REST Controllers

@RestController est equivalent à
@Controller + @ResponseBody

```
@RestController
public class OrderController {

  @GetMapping('{id}/info')
  public ResponseEntity<Game>(@PathVariable Long id){
    ...
  }
}
```

Controlling the entire response object

The **`ResponseEntity<>`** object allows you **to change the response body**, response headers and response code

```
@GetMapping('{id}/title')
public ResponseEntity<Game> getTitle(...){
  ...
  return new ResponseEntity(gameService.getGame(id));
}
```

Consommer une API REST

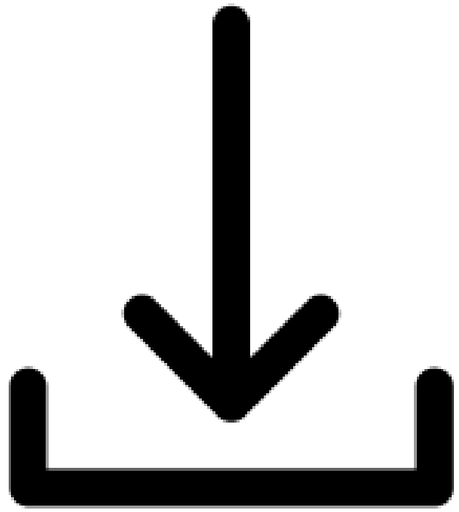
jQuery - Fetch API (Demo)

```
@GetMapping('/')
public ModelAndView index(ModelAndView modelAndView) {
    modelAndView.setViewName('index');
    return modelAndView;
}

@GetMapping(value = '/liste', produces = 'application/json')
@ResponseBody
public Object fetchData() {
    return new ArrayList<Product>() {{
        add(new Product(){{
            setName('Chewing Gum');
            setPrice(new BigDecimal(1.00));
            setBarcode('133242556222');
        }});
        ...
    }};
}
```

HomeController.java

```
public class Product {
    private String name;
    private BigDecimal price;
    private String barcode;

    // Getters & Setters
    ...
}
```

Product.java



jQuery - Fetch API (Demo)

Now let's head to the view

There is no need for a separate .js file for one-time use

index.html

```
...
<div class='container-fluid'>
    <h1 class='text-center mt-5 display-1'>Data Fetch</h1>
    <div class='data-container mt-5'></div>
    <div class='button-holder mt-5'>
        <button id='fetch-button' class='btn btn-info'>Fetch Data</button>
        <button id='clear-button' class='btn btn-secondary'>Clear Data</button>
    </div>
</div>
<script>
    // jQuery Event handlers
    $('#fetch-button').click(() => {...}); // Fetch and render the data
    $('#clear-button').click(() => $('.data-container').empty()); // Clear the data
</script>
```

jQuery - Fetch API (Demo)

```
$('#fetch-button').click(() => {
    fetch('http://localhost:8000/liste') // Fetch the data (GET request)
        .then((response) => response.json()) // Extract the JSON from the Response
        .then((json) => json.forEach((x, y) => { // Render the JSON data to the HTML
                if (y % 4 === 0) {
                    $('.data-container').append('<div class='row d-flex justify-content-around mt-4'>');
                }

                let divColumn =
                    '<div class='col-md-3'>' +
                    '<h3 class='text-center font-weight-bold'>' + x.name + '</h3>' +
                    '<h4 class='text-center'>Price: $' + x.price + '</h4>' +
                    '<h4 class='text-center'>Barcode: $' + x.barcode + '</h4>' +
                    '</div>';

                $('.data-container .row:last-child').append(divColumn);
            }))
            .catch((error)=> {console.log(error);});
});
```

jQuery - ajax (Demo)

```
$("# fetch-button").click(function() {
            // Make an AJAX request
            $.ajax({
                url: 'http://localhost:8000/fetch', // API endpoint
                method: 'GET',
                dataType: 'json',
                success: function(data) {
                // Update the content on success
                 $('.data-container').text('Title: ' + data.title);
                },
                error: function(error) {
                // Handle errors
                console.error('Error:', error);
                }
            });
        });
```

REST Client vs WebClient vs RestTemplate

Choisir le bon client HTTP dans Spring Boot

Lorsqu'ils utilisent des API externes dans Spring Boot, les développeurs ont plusieurs choix pour effectuer des requêtes HTTP.

Les trois options les plus populaires sont

- RestTemplate
- WebClient
- le nouveau RestClient introduit dans Spring 6.0 et Spring Boot 3.0.

```
RestTemplate restTemplate = new RestTemplate();
String response = restTemplate.getForObject("https://api.example.com/data", String.class);
System.out.println(response);
```

```
WebClient webClient = WebClient.create("https://api.example.com");

String response = webClient.get()
    .uri("/data")
    .retrieve()
    .bodyToMono(String.class)
    .block();

System.out.println(response);
```

```
RestClient restClient = RestClient.create("https://api.example.com");

String response = restClient.get()
    .uri("/data")
    .retrieve()
    .body(String.class);

System.out.println(response);
```

REST Client vs WebClient vs RestTemplate

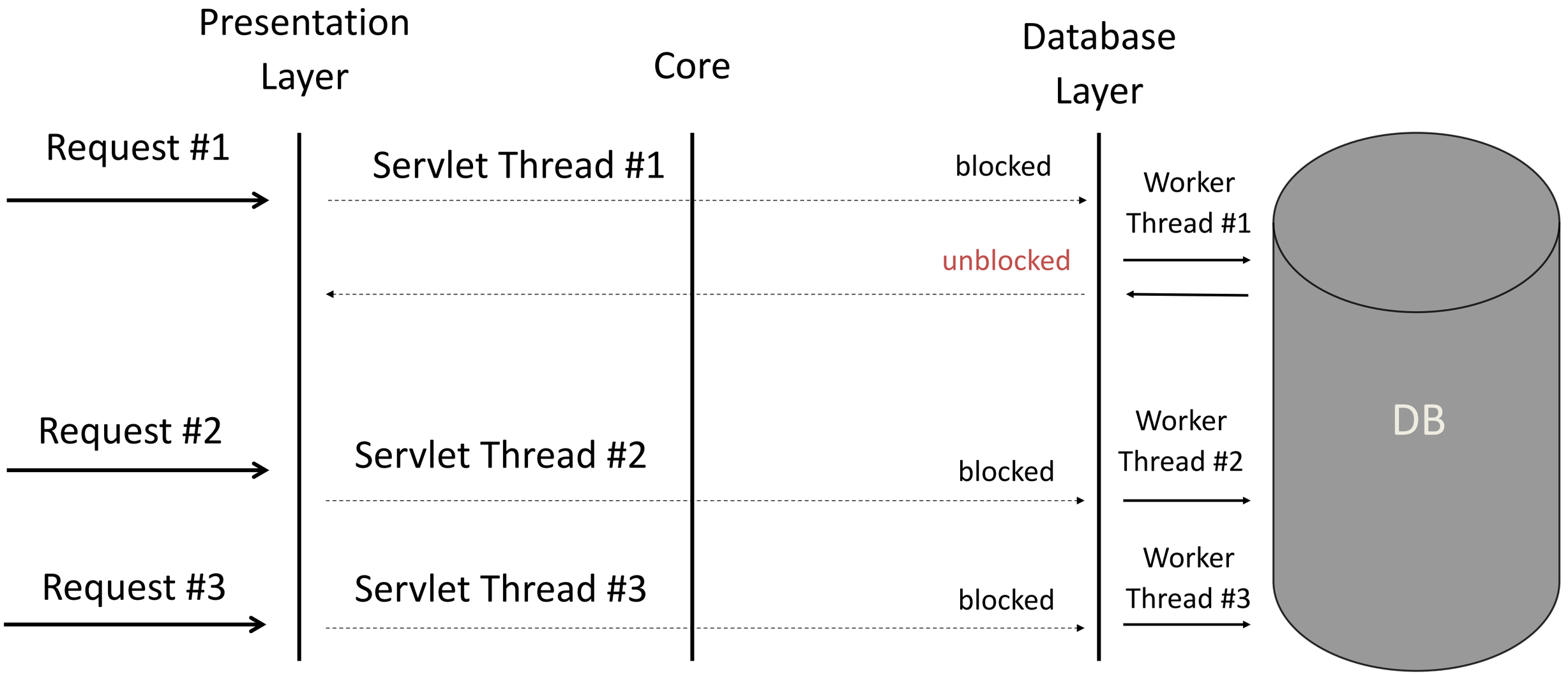
Choisir le bon client HTTP dans Spring Boot

Feature	RestTemplate	WebClient	RestClient
Blocking/Synchronous	✅ Yes	✅ (Optional)	✅ (Optional)
Non-blocking/Reactive	❌ No	✅ Yes	✅ Yes
Ease of Use	✅ Simple	❌ Complex	✅ Simplified
Spring 6+ Recommended	❌ No	✅ Yes	✅ Yes
Suitable for WebFlux	❌ No	✅ Yes	✅ Yes
Future Proof	❌ No (Deprecated)	✅ Yes	✅ Yes

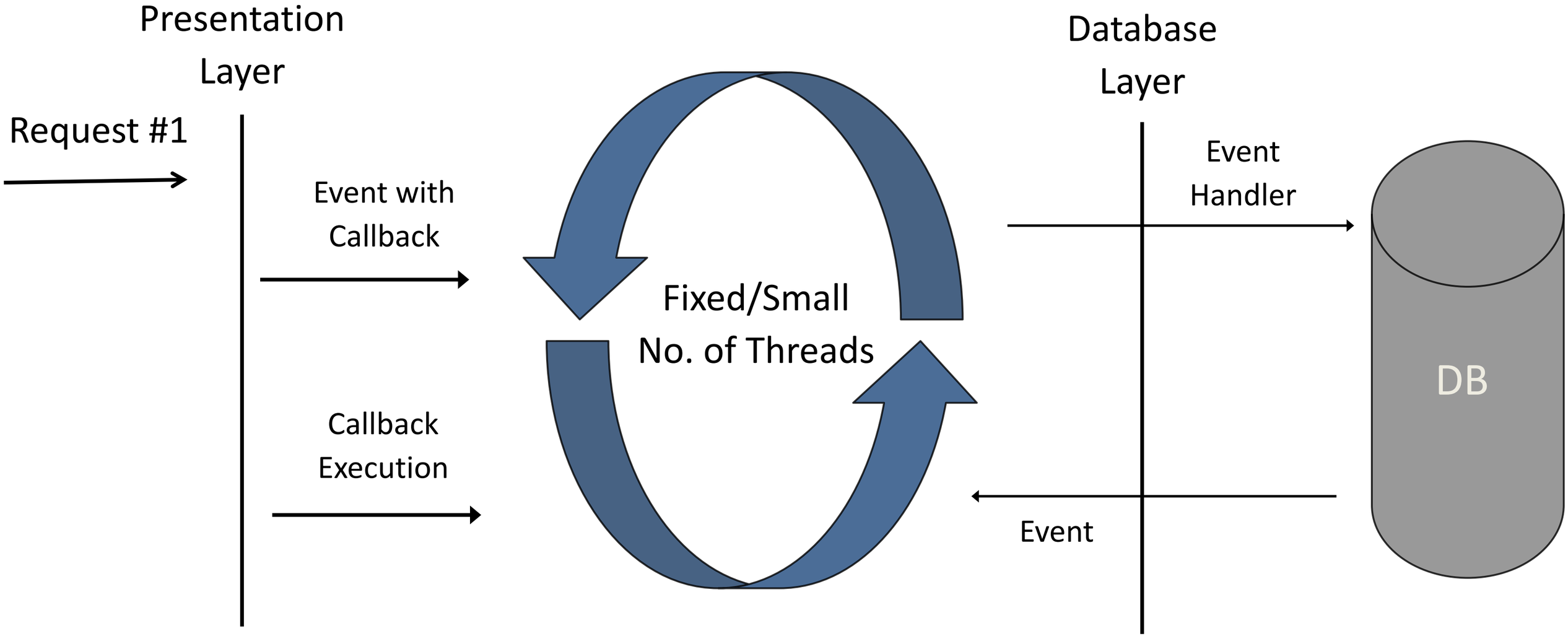
- Utilisez RestClient pour les nouvelles applications Spring MVC qui ne nécessitent pas une programmation entièrement réactive.
- Utilisez WebClient pour les applications Spring WebFlux et les cas où une forte concurrence et un comportement non bloquant sont nécessaires.
- Évitez RestTemplate, sauf si vous gérez une application existante.

Reactive Programming - WebFlux

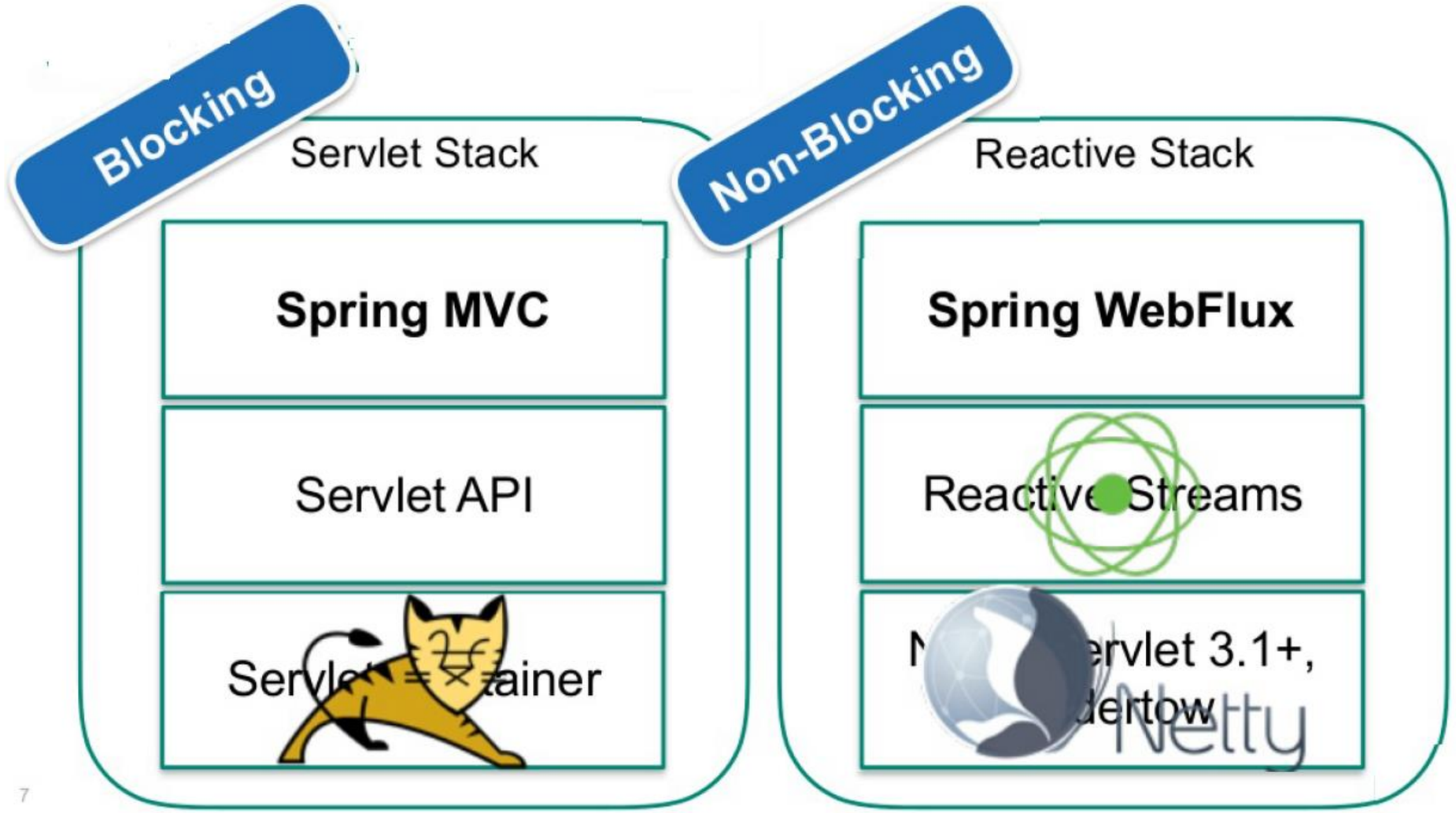
Blocking vs Non-blocking(MVC Traditionnel)



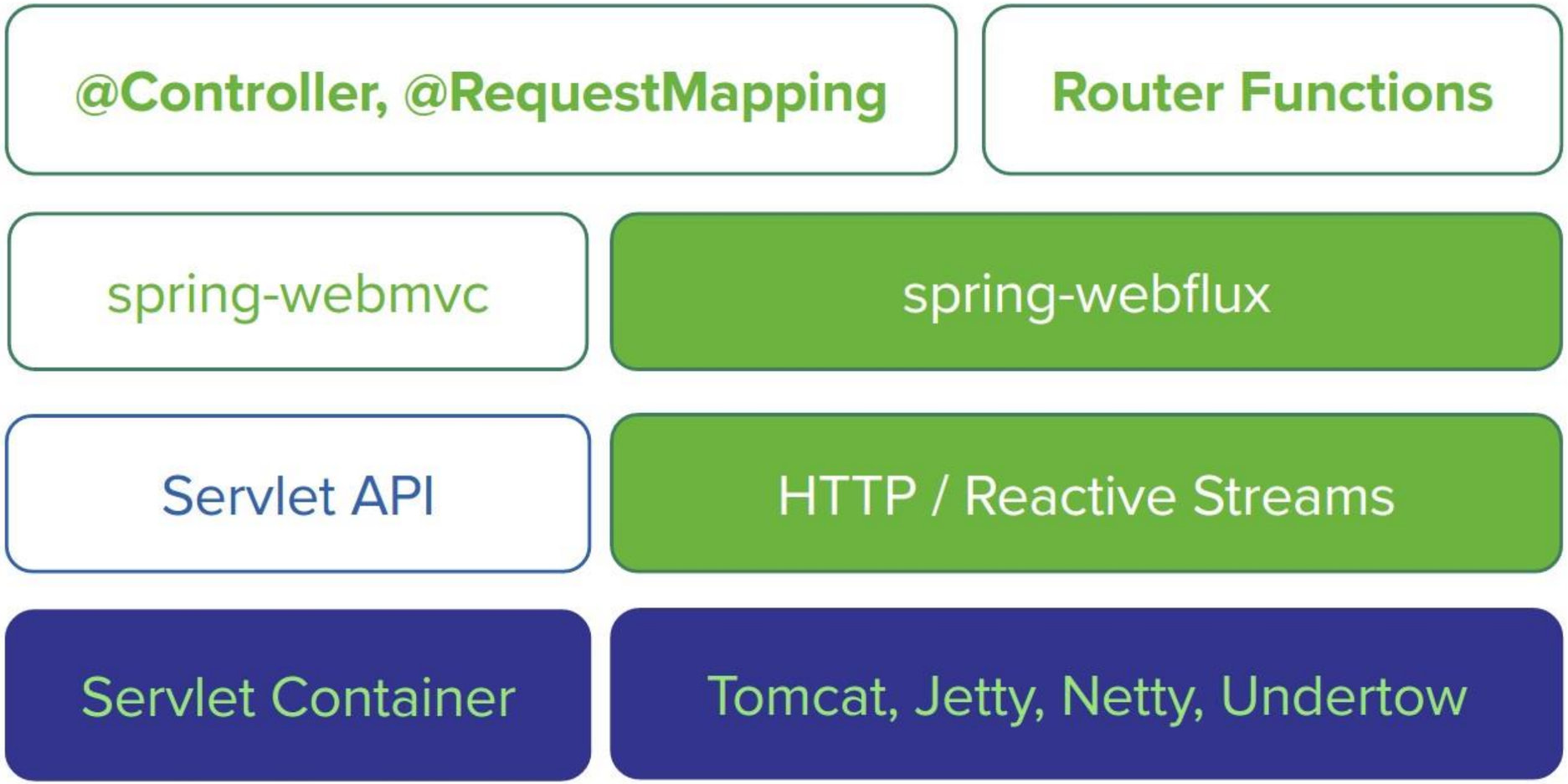
Blocking vs Non-blocking(2)



Piles Web dans Spring

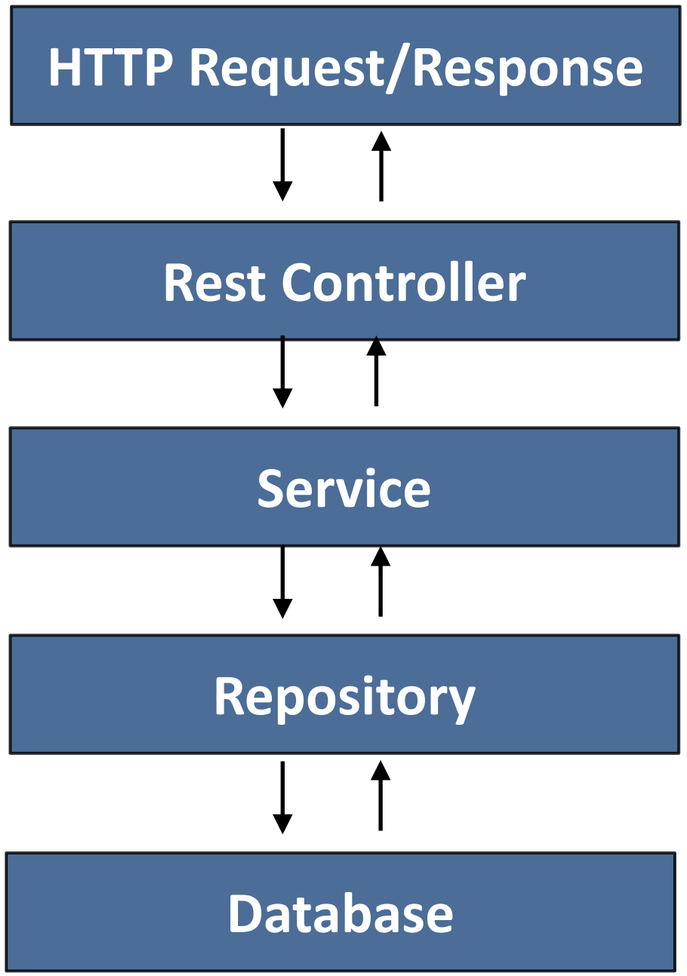


Spring Reactive Building Blocks

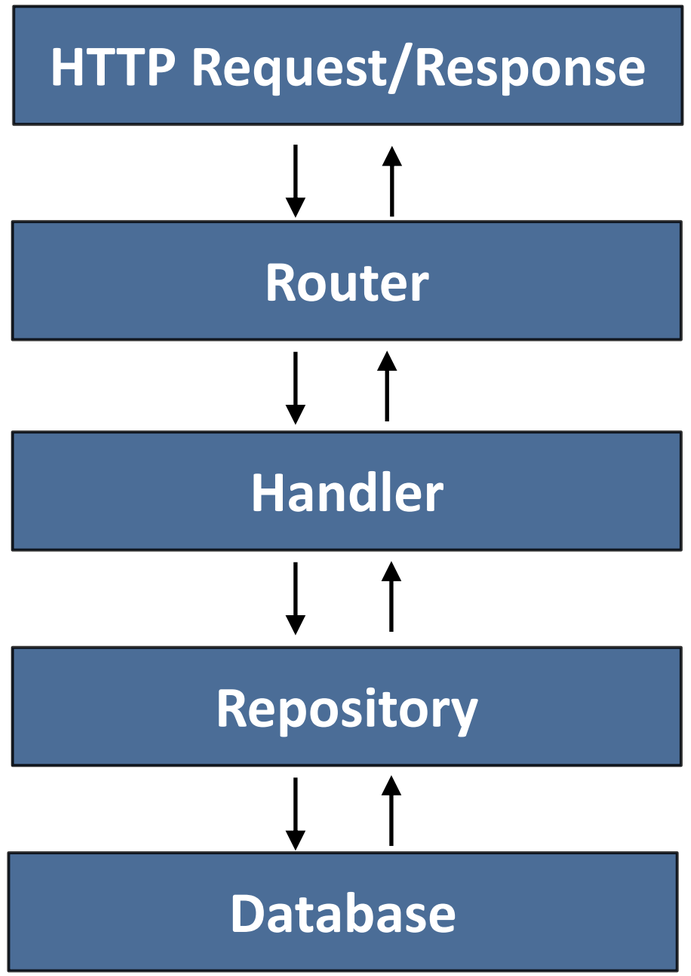


Annotated Controllers VS Func Endpoints

Annotated Controllers

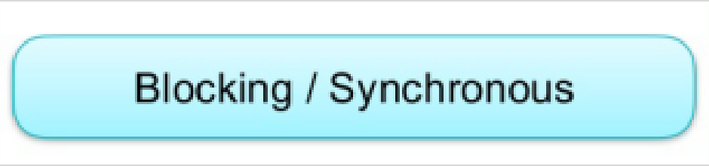


Functional Endpoints



Controller Spring MVC

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    @GetMapping("/person")
    Collection<Person> list() {
    return this.repository.findAll();
}

   @GetMapping("/person/{id}")
    Person findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```



Controller Spring WebFlux

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }

    @GetMapping("/person")
    Flux<Person> list() {
    return this.repository.findAll();
    }

   @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```

Non-blocking / Asynchronous

Functional Endpoints Example – Handler

Une requête HTTP est gérée avec une fonction **HandlerFunction** qui prend ServerRequest et renvoie une ServerResponse retardée - **Mono<ServerResponse>**

```
public class StudentHandler {

//...

public Mono<ServerResponse> getStudent(ServerRequest request) {

        int studentId = Integer.valueOf(request.pathVariable("id"));

        return repository.getStudent(studentId)

        .flatMap(student ->
                ok().contentType(APPLICATION_JSON).bodyValue(student))

        .switchIfEmpty(ServerResponse.notFound().build());

} //... }
```

Functional Endpoints Example – Router

Les requêtes entrantes sont acheminées vers une fonction de gestionnaire avec une **RouterFunction**
prend ServerRequest et renvoie une HandlerFunction retardée - **Mono<HandlerFunction>**

```
//... Inject PersonHandler in constructor

RouterFunction<ServerResponse> router = router()

    .GET("/student/{id}", accept(APPLICATION_JSON), handler::getStudent)

    .GET("/student ", accept(APPLICATION_JSON), handler::listStudents)

    .POST("/student ", handler::createStudent)

    .build();
```

Annexe - tests d'API REST (pour votre lecture)

tests unitaires d'API REST

Spring MVC Test : dépendance Maven

Comme pour chaque composant, il faut pouvoir tester l'API REST unitairement, en bouchonnant les éléments dont elle dépend.

Spring MVC Test :

– permet de simplifier l'écriture de tests des **@RestController**

–intègre **JUnit 5** pour écrire les tests unitaires, **Hamcrest** pour faire des comparaisons de données (**equalTo()**, **hasItems()**, **containsString()**...) ou encore **Mockito** (et **BDDMockito**) pour bouchonner les composants sous-jacents.

Les dépendances nécessaires sont toutes importées par Spring Boot dans le **spring-boot-starter-test** :

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
   <exclusions>
      <exclusion>
         <groupId>org.junit.vintage</groupId>
         <artifactId>junit-vintage-engine</artifactId>
      </exclusion>
   </exclusions>
</dependency>
```

tests unitaires d'API REST

Spring MVC Test : initialisation

Utilisation de **@WebMvcTest(...)** plutôt que **@SpringBootTest** :

– c'est + performant car seul le **RestController** est chargé, au lieu de toute l'application
– cela oblige à fournir des mocks pour les dépendances du **RestController**.

```
@WebMvcTest(ToDoList_API.class)
public class ToDoList_API_Test {

    @Inject
    private MockMvc mvc;

    @MockBean
    private ToDoList_Service tdlService;

    @MockBean
    private ToDoItem_Service tdiService;

    //all tests
    ...

}
```

tests unitaires d'API REST

Spring MVC Test : requête GET

> *Exemple de test d'une requête **GET /api/v1/todolists/123** :*

```
@DisplayName("returns a todolist")
@Test
public void shouldReturnRelevantToDoList() throws Exception {

   given(tdlService.getToDoList(123)).willReturn(new ToDoList(...));
   mvc.perform(
      get("/api/v1/todolists/{listId}", 123).contentType(MediaType.APPLICATION_JSON_VALUE)).
      andDo(print()).
      andExpect(status().is2xxSuccessful()).
      andExpect(content().contentType(MediaType.APPLICATION_JSON_VALUE)).
      andExpect(jsonPath("title").value("vacances")).
      andExpect(jsonPath("complete").value(false)).
      andExpect(jsonPath("todoitems").exists()).
      andExpect(jsonPath("todoitems").hasSize(2))).
      andExpect(jsonPath("todoitems[*].id", contains(987, 361))).
      andExpect(jsonPath("links").isNotEmpty()).
      andExpect(jsonPath("links[0].rel").exists()).
      andExpect(jsonPath("links[0].rel").value("self")).
      andExpect(jsonPath("links[0].href").value("http://localhost/api/v1/todolists/123"));
}
```

tests unitaires d'API REST

Spring MVC Test : requête GET

Côté Mockito, on dit au mock de **tdlService.getToDoList(listId)** de retourner la **ToDoList** initialisée pour le bouchon, si **listId** passé en paramètre = **123**.

Grâce à **andDo(print())**, on peut afficher sur la console ce qui se passe :

– requête : verbe, URL, paramètres, headers...
– controller : méthode appelée
– réponse : code de statut, headers, body de la réponse...

On vérifie dans la réponse que :

– le code de statut est de la famille **2xx** (**200 (OK)**, **201 (Created)**...)
– **title="vacances"**
– **complete=false**
– il existe un structure todoitems ayant 2 éléments
– parmi les todoitems, on a bien les **id** 987 et 361
– que la structure **links** :
    - n'est pas vide
    - contient une sous-struture **self** ayant pour **href="**http://localhost/api/v1/todolists/123**"**.

tests unitaires d'API REST

Spring MVC Test : requête GET

> *Exemple de test d'une requête **GET /api/v1/todolists** :*

```
@DisplayName("returns all todolists")
@Test
public void shouldReturnAllToDoLists() throws Exception {
    List<ToDoList> todolists = new ArrayList<ToDoList>();
    todolists.add(new ToDoList(...));
    ...

    given(tdlService.getAllToDoLists(ArgumentMatchers.anyString())).willReturn(todolists);

    mvc.perform(
        get("/api/v1/todolists").contentType(MediaType.APPLICATION_JSON_VALUE)).
        andExpect(status().isOK()).
        andExpect(jsonPath("$", hasSize(2))).
        andExpect(jsonPath("$[*].id", contains(123, 45))).
        andExpect(jsonPath("$[0].id", is(123)));
}
```

tests unitaires d'API REST

Spring MVC Test : requête GET

Côté Mockito, on dit au mock de **tdlService.getAllToDoLists(String)** de retourner la liste de **ToDoList** initialisée au préalable, quelque soit la **String** passée en paramètre.

On vérifie dans la réponse que :

– **code de statut=200 (OK)**
– il y a 2 éléments à la racine **$**
– parmi ces éléments, on a bien les **id 123** et **45**
– que le 1er élément à la racine a pour **id=123**.

tests unitaires d'API REST

Spring MVC Test : requête GET

> *Exemple de test d'une requête **GET /api/v1/todolists/999** :*

```
@DisplayName("if todolist id is not exist, returns 404")
@Test
public void shouldReturnToDoListNotFoundStatus() throws Exception {
    given(tdlService.getToDoList(999)).willThrow(new IllegalArgumentException());

    mvc.perform(
        get("/api/v1/todolists/{listId}", 999).contentType(MediaType.APPLICATION_JSON_VALUE)).
        andExpect(status().isNotFound());
}
```

Côté Mockito, on dit au mock de **tdlService.getToDoList(listId)** de retourner une **IllegalArgumentException**, si **listId** passé en paramètre = **999**.

Dans la requête, on passe l'objet **ToDoList** transformé en **JSON**, puis on vérifie dans la réponse que :

– **code de statut=404 (Not Found)**.

tests unitaires d'API REST

Spring MVC Test : requête POST

On lance une requête **POST /api/v1/todolists** :

```
@DisplayName("creates a todolist with its id")
@Test
public void shouldReturnRelevantToDoListId() throws Exception {
    given(tdlService.createToDoList(Matchers.any(ToDoList.class))).willReturn(new ToDoList(...));

    mvc.perform(
        post("/api/v1/todolists")
            .contentType(MediaType.APPLICATION_JSON_VALUE).
            .content(object2Json(new ToDoList(...))).
        andExpect(status().isCreated()).
        andExpect(jsonPath("id").value("1"));
}
```

Côté Mockito, on dit au mock de **tdlService.createToDoList(ToDoList)** de retourner une **ToDoList** bouchonnée, quelque soit la **ToDoList** passée en paramètre.

Dans la requête, on passe l'objet **ToDoList** transformé en **JSON**, puis on vérifie dans la réponse que :

- **code de statut=201 (Created)**
- **id=1**.

tests unitaires d'API REST

Spring MVC Test : requête DELETE

On lance une requête **DELETE /api/v1/todolists/123** :

```
@DisplayName("delete a todolist")
@Test
public void shouldReturnOKStatusWhenDeleteToDoList() throws Exception {
    doNothing().when(tdlService).deleteToDoList(Matchers.any(Long.class));

    mvc.perform(
        delete("/api/v1/todolists/{listId}", 123).contentType(MediaType.APPLICATION_JSON_VALUE)).
        andExpect(status().isOk());
}
```

Côté Mockito, on dit au mock de **tdlService.deleteToDoList(Long)** de ne rien faire, quelque soit l'identifiant passé en paramètre (ne s'applique que pour bouchonner les méthodes **void**).

On vérifie dans la réponse que :

- **code de statut=200 (OK)**

tests d'intégration d'API REST