API AND JAVA

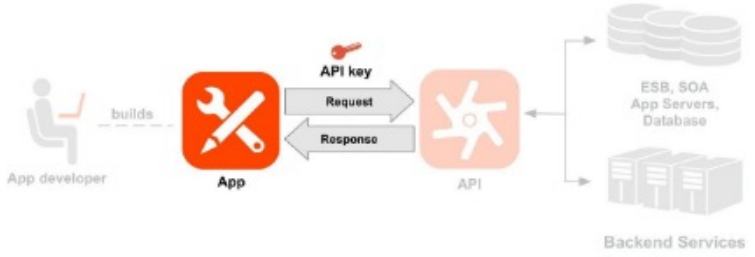
PLAN DE LA FORMATION

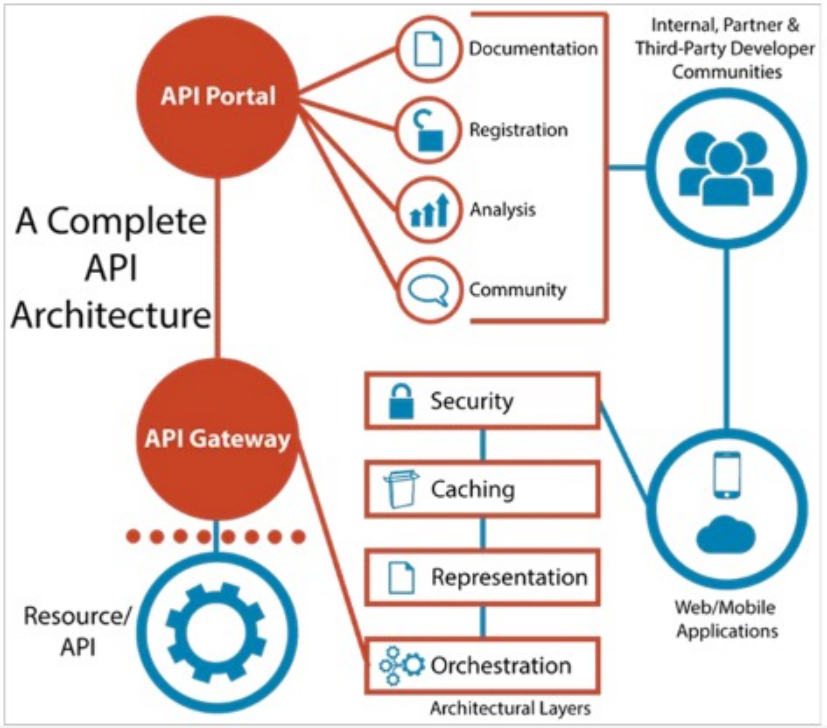
- **Protocole HTTP**
- **API Rest**
- **Le principe HATEOAS**
- **Jersey et Grizzly**
- **Spring boot**
- **Controlleurs**
- **Services**
- **Répositories**
- **Entité**
- **ORM JPA Hibernate**
- **Configuration**
- **Actuator**
- **Documenter un service REST avec Swagger**
- **Tests unitaires**
- **Spring reactive si on a le temps**

PUBLICATION D'API

- La publication des APIs peut se faire en local pour les besoins de communication dans le système d'information comme elle peut se faire à destination de l'extérieur (clients, partenaires) et dans ce cas elle demande une API management solution complète de sécurité, cache, orchestration ...

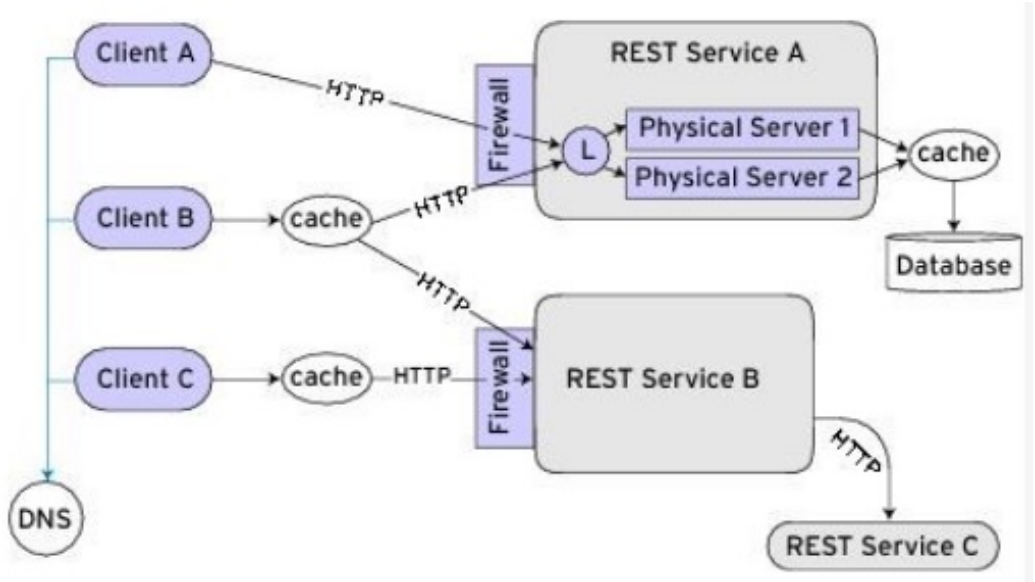
- La publication peut se faire aussi chez les fournisseurs PaaS (voir le slide sur Heroku) en utilisant les outils et couches de ce dernier.





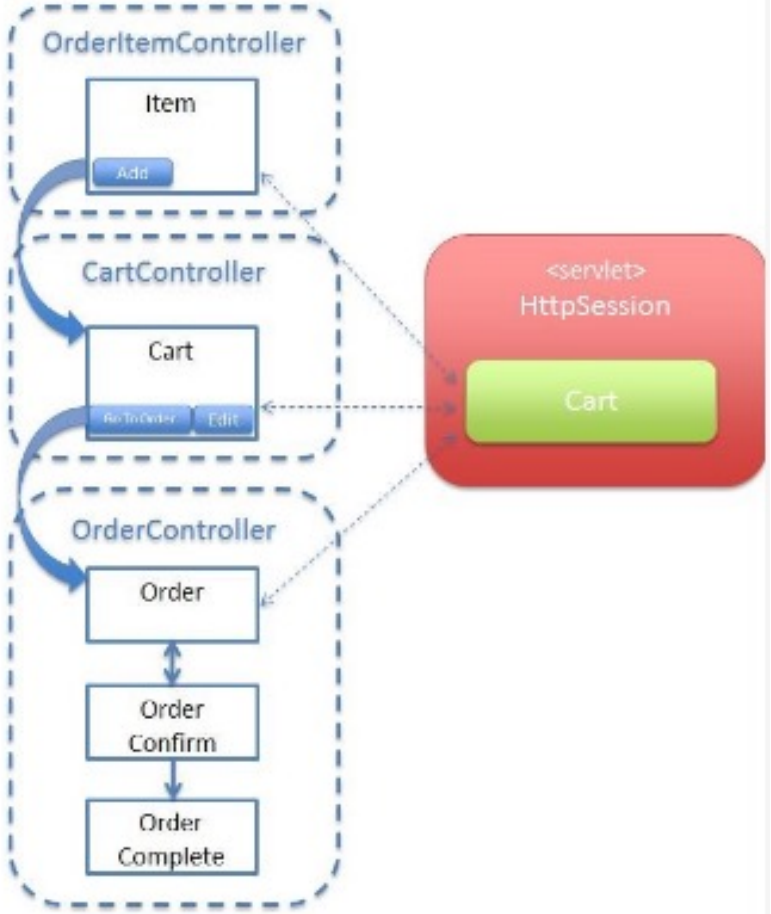
INTRODUCTION À LA TECHNOLOGIE REST (SOURCE WIKIPÉDIA)

- REST (representational state transfer) est un style d'architecture pour les systèmes distribués. Les contraintes associés cette architecture sont :
  - Client-serveur :la séparation permet l'évolution indépendante.
  - Serveur sans état (session) : plus de visibilité sur l'interaction et de robustesse, moins de contrôle serveur.
  - Avec cache : le serveur marque explicitement le contenu à mettre en cache par le client
  - Interface uniforme :URI, messages auto-descriptifs, hypermédia



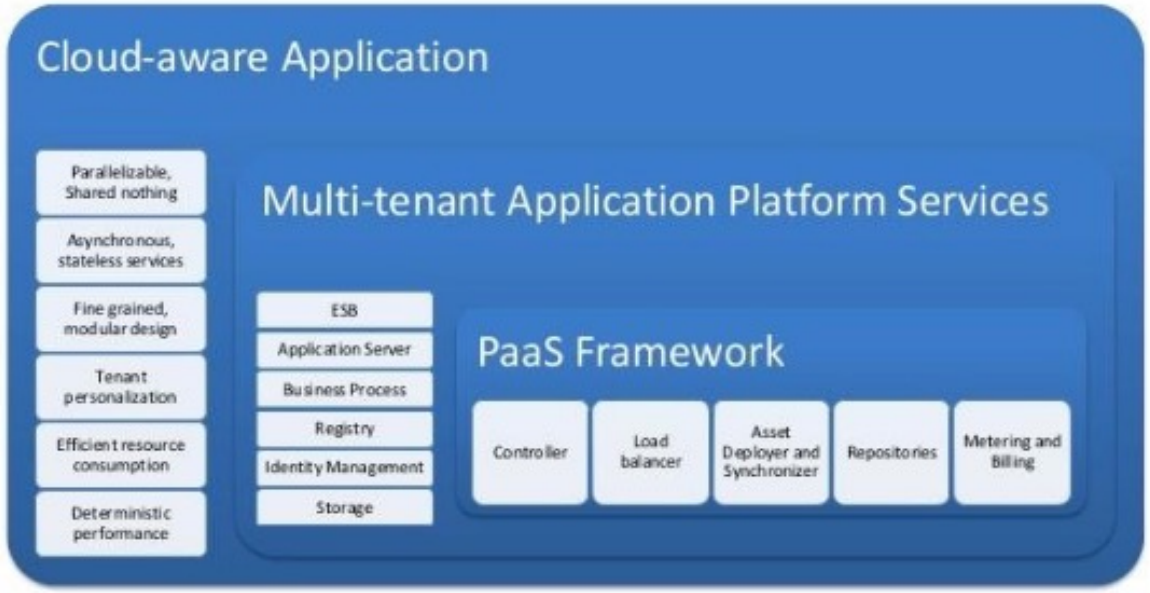
GESTION DE L'ÉTAT :APPLICATIONS STATEFUL

- La gestion de l'état revient à associer (ou non) une session à l'utilisateur connecté, la session garde du côté serveur un ensemble d'attributs ce qui constitue un "état conversationnel".
- Exemple :dans une application de commerce électronique, la session de l'utilisateur contient l'ensemble des articles choisis dans son caddie, les derniers articles visités ...etc
- Le modèle stateful bien que pratique présente un nombre de limitations techniques :
  - Consommation importante de la mémoire due à la gestion des sessions.
  - En cas de distribution de charge il est difficile de passer d'une instance à une autre.
  - En cas de cluster la synchronisation des sessions devient une charge pour les serveurs.
  - En outre un ensemble de bug difficiles à reproduire en environnement de développement apparaissent en production à cause des conditions de concurrence sur les objets affectés à la session.



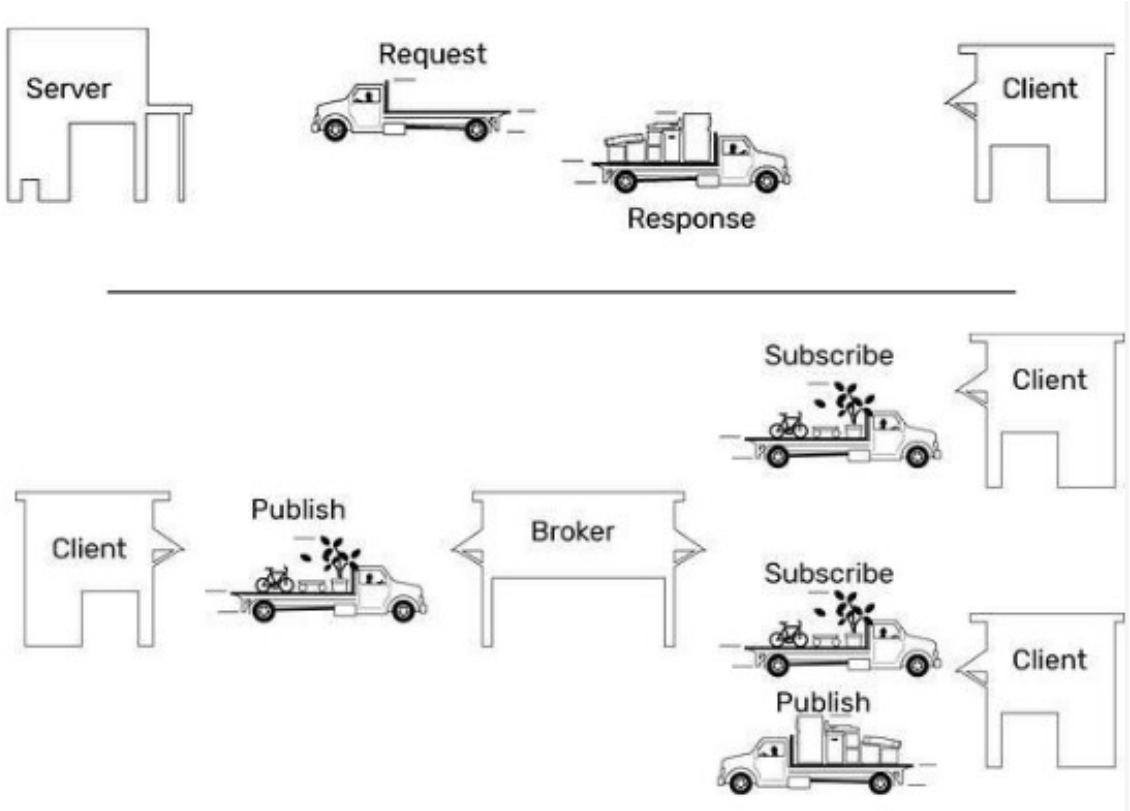
GESTION DE L'ÉTAT : APPLICATIONS STATELESS

- Les applications stateless répondent à l'image d'une fonction mathématique selon les paramètres qui leurs sont passés, elles sont par conséquent plus optimisées et indépendantes du client ou de la plateforme de déploiement.
- Les applications microservices (et REST) en général sont stateless par définition



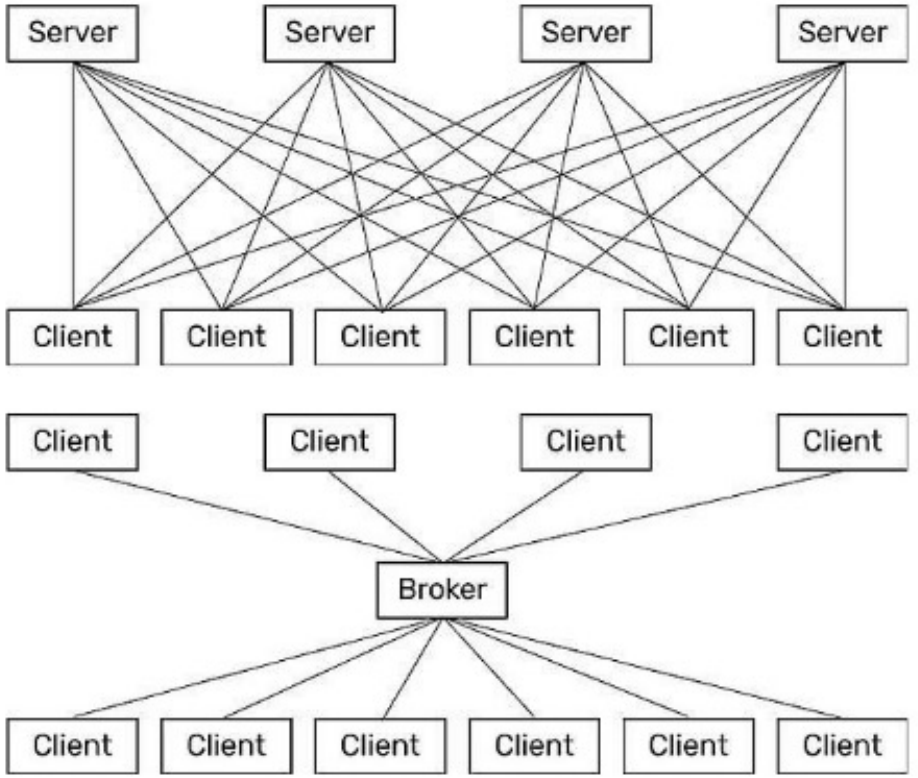
REST VS PUBLISH-SUBSCRIBE

- Une communication REST s'inscrit dans le modèle typique request-response, un modèle synchrone dans lequel le serveur renvoie une réponse adéquate à une requête qualifiée.
- L'autre modèle de communication est Publish-subscribe (pub-sub) ayant un composant central (broker) qui reçoit et distribue toutes les données (messages). Les clients peuvent publier des message ou s'inscrire pour recevoir des messages sur un sujet (topic)



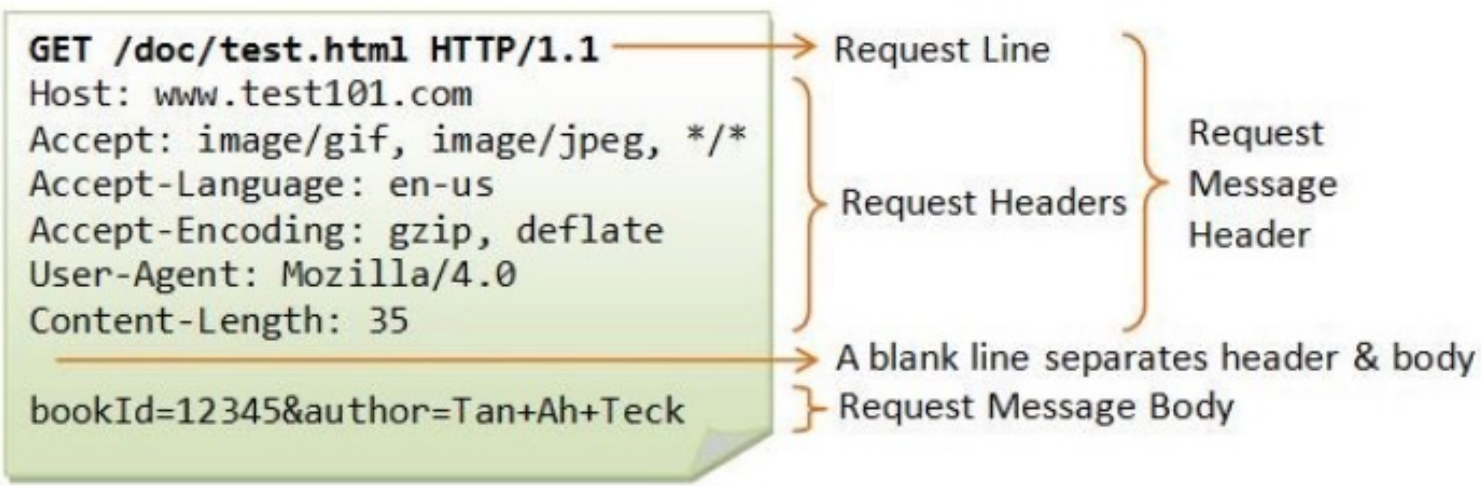
REST VS PUBLISH-SUBSCRIBE

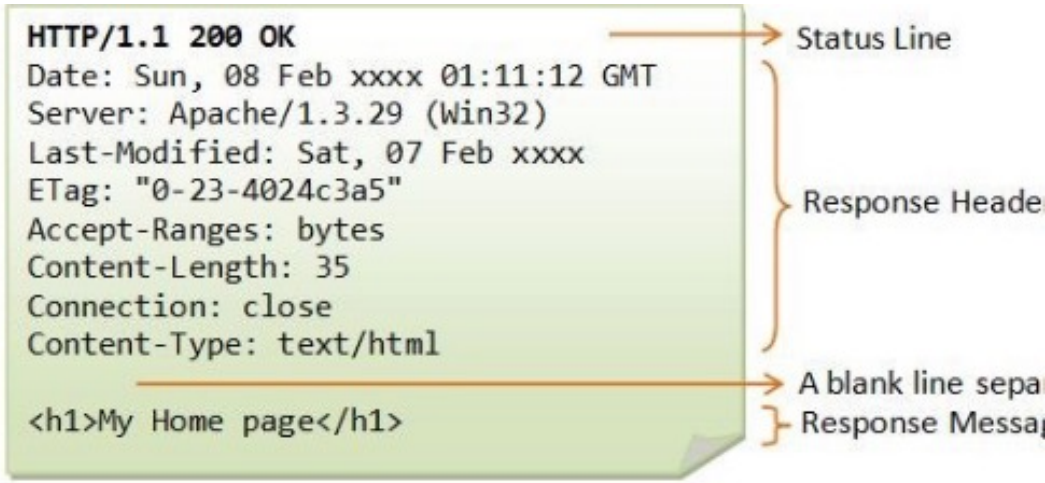
- Le communication en request response est un standard fiable et reconnu, c'est le choix par défaut sauf dans les cas suivants :
  - Multiplication des canaux (serveurs et clients connectés): pub-sub centralise le publication et distribution.
  - Réponses redondantes sur certains canaux: pub-sub fonctionne en report by exception (au changement).
  - Quand les échanges sont en mode asynchrones.
  - Quand le réseau n'est pas fiable ou que les transfert sont lents.



RAPPELS SUR LE PROTOCOLE HTTP

- L'hypertexte transfer protocol est un protocole reposant sur un unique échange initié par le client.
- La version actuelle du protocole est la 2.0 publiée en 2015 elle adoptée sur 17% des serveurs web actuellement.
- Le client formule une requête HTTP ayant une entête et un corps, la réponse serveur est aussi constituée d'une entête et d'un corps.





RAPPELS SUR LE PROTOCOLE HTTP : LA REQUÊTE

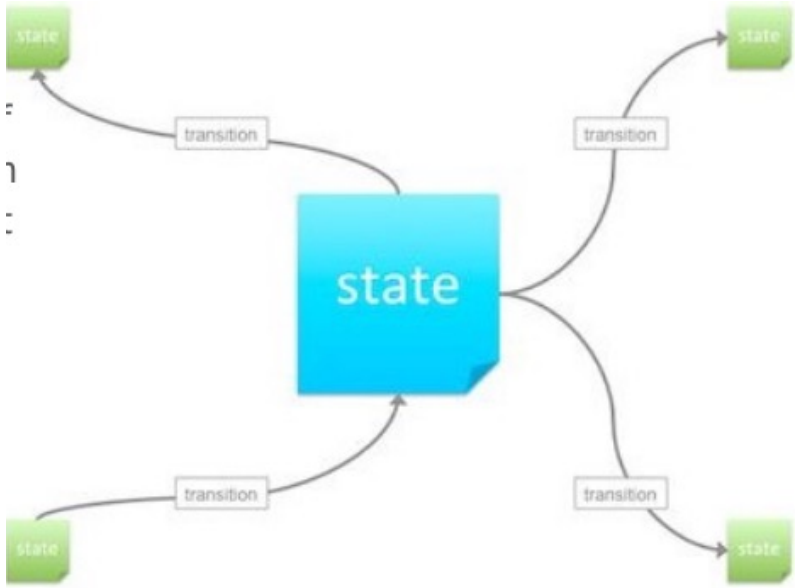
- Les méthodes utilisées (verbes) pour la requête HTTP sont : GET, HEAD, POST, PUT, DELETE, OPTIONS, TRACE, CONNECT.
- L'entête de la requête spécifie les éléments suivants :
  - Host
  - User-agent
  - Cookie
  - Content-type et Content-length
  - D'autres entêtes :(cache, langue, format préféré)
    Le corps de la requête est seulement utilisé dans la méthode POST.

RAPPELS SUR LE PROTOCOLE HTTP : LA RÉPONSE

- La réponse du serveur spécifie un ensemble de données dans son entête :
  - Code de retour
  - Date, serveur (logiciel)
  - Type et taille du contenu, Date d'expiration (pour le cache)
  - Cookie
- Les codes de retour les plus utilisés sont :
  - 2xx Opération déroulée correctement (contenu existant, absent ou partiel).
  - 3xx Le contenu n'est plus disponible (redirection temporaire ou permanente).
  - 4xx La requête a un problème de format, de ressource ou de sécurité
  - 5xx La requête n'a pas pu être traitée (surcharge, exception, non implémentée)

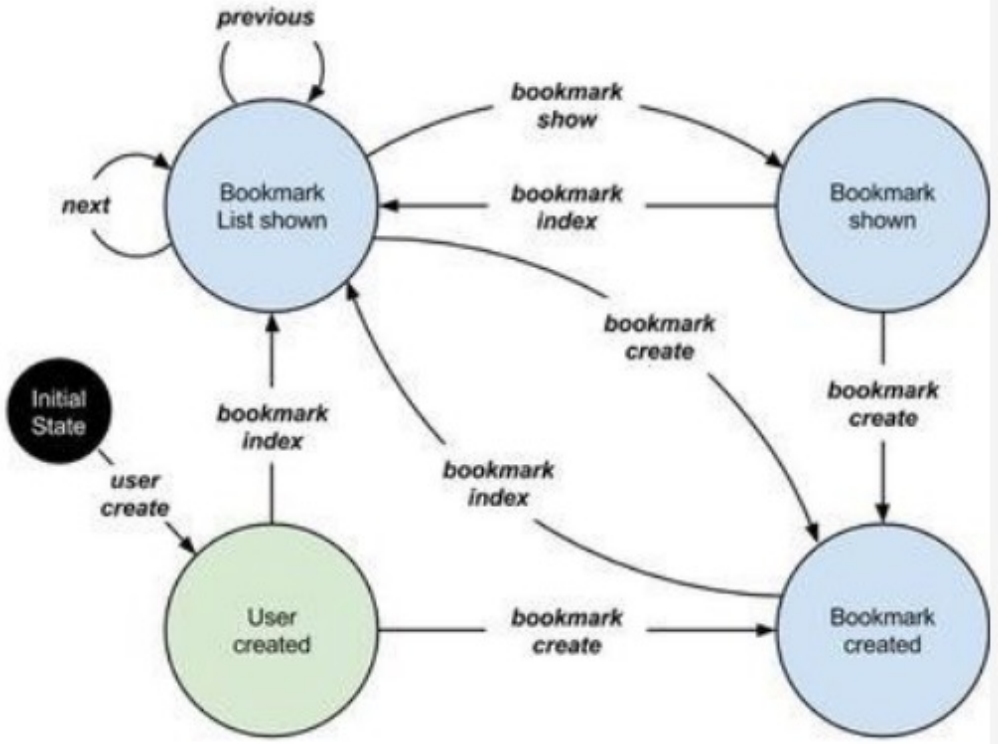
HYPERMEDIA AS THE ENGINE OF APPLICATION STATE

- " The name 'Representational State Transfer' is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use." ~Roy Fielding,
- Une application web bien conçue rend un état demandé selon la sémantique et le contenu accepté par l'utilisateur ainsi que les transitions possibles après chaque état atteint.



HYPERMEDIA AS THE ENGINE OF APPLICATION STATE

- Hypermedia as the Engine of Application State (HATEOAS) offre les informations de navigation d'une interface REST en incluant les liens hypermedia avec les réponses.
- HATEOAS pour REST est différent de ce qu'est WSDL pour les WS
  - WSDL peut être situé dans un autre site, voir être partagé par email, alors que HATEOS inclut les méta-données dans la réponse.
  - HATEOAS décrit l'état et les transitions alors que WSDL décrit l'interface



HATEOAS : EXEMPLES

- Selon le principe HATEOAS la réponse doit inclure au moins une transition d'état

```
class Customer {
    String name;
}
```

A simple JSON presentation is traditionally rendered as:

```
{
    "name" : "Alice"
}
```

A HATEOAS-based response would look like this:

```
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

HATEOAS : EXEMPLES

- Dans un exemple plus complet on trouve un choix de liens ou transitions avec les attributs :
  - href =“...” URL complète
  - rel =“...”Qualification du type de relation (self, client, ...)


```
{
    "content": [ {
        "price": 499.00,
        "description": "Apple tablet device",
        "name": "iPad",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/1"
        } ],
        "attributes": {
            "connector": "socket"
        }
    }, {
        "price": 49.00,
        "description": "Dock for iPhone/iPad",
        "name": "Dock",
        "links": [ {
            "rel": "self",
            "href": "http://localhost:8080/product/3"
        } ],
        "attributes": {
            "connector": "plug"
        }
    } ],
    "links": [ {
        "rel": "product.search",
        "href": "http://localhost:8080/product/search'
    } ]
}
```

MAVEN

```
<!-- https://mvnrepository.com/artifact/org.reflections/reflections -->
<dependency>
    <groupId>org.reflections</groupId>
    <artifactId>reflections</artifactId>
    <version>0.9.10</version>
</dependency>

<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-jaxrs</artifactId>
    <version>1.5.20</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.swagger/swagger-annotations -->
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-annotations</artifactId>
    <version>1.5.20</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.swagger/swagger-core -->
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-core</artifactId>
    <version>1.5.20</version>
</dependency>

<!-- https://mvnrepository.com/artifact/io.swagger/swagger-models -->
<dependency>
    <groupId>io.swagger</groupId>
    <artifactId>swagger-models</artifactId>
    <version>1.5.20</version>
</dependency>
```

RAPPELS JERSEY

- @PathParam est l'annotation que nous avons vue dans notre premier exemple. Elle permet d'associer un morceau de l'URL de requête à un champ ou un paramètre.
- @QueryParam et @FormParam permettent d'associer un paramètre de la requête à un champ ou un paramètre d'une méthode de notre classe.
- @CookieParam permet d'associer une valeur stockée dans un *cookie* ou l'instance de Cookie elle-même.
- @HeaderParam permet d'associer une valeur d'un champ HTTP à un champ ou un paramètre d'une méthode de notre classe.
- @MatrixParam permet d'associer un paramètre HTTP matriciel à un champ ou un paramètre de méthode de notre classe. Le problème est que les URL portant des paramètres matriciels ne sont pas clairement supportées par les navigateurs et les serveurs web. Cette annotation est donc à utiliser avec précautions.

DOC D'UN SERVICE

```java
@Path("/users")
@Api(value = "User Service", description = "REST Endpoints for User Service")
public class UserService {

    @GET
    @Path("/getAllUsers")
    @Produces(MediaType.APPLICATION_JSON)
    public List<User> getAllUsers(){
        List<User> users = new ArrayList<>();
        User user1 = new User(1, "Tunde", "Samson");
        User user2 = new User(2, "Rondo", "James");
        User user3 = new User(3, "Samuel", "Kate");

        users.add(user1);
        users.add(user2);
        users.add(user3);

        return users;
    }
```

INTRODUCTION

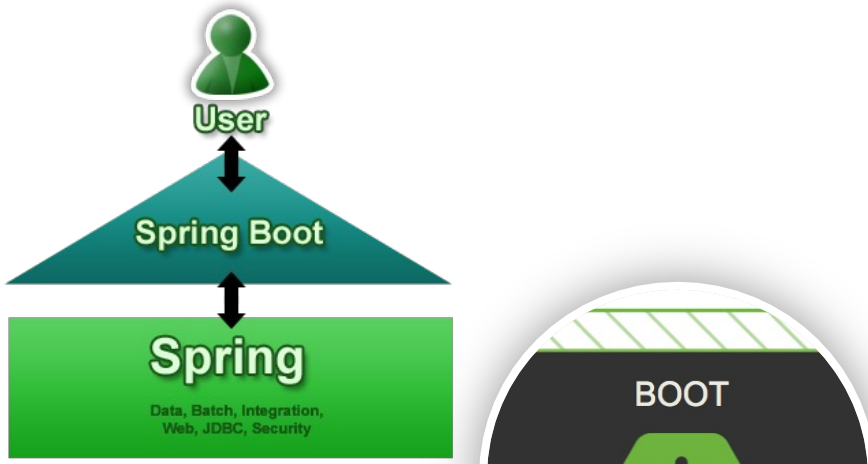
- En java, l'implémentation des microservices peut se faire de plusieurs manière, vu qu'un microservice c'est d'abord un ensemble de principes plutôt qu'une technologie. Parmi les choix disponibles :
  - Springboot
  - Dropwizard
  - Grails
  - Play +Akka
  - ....
- Les frameworks permettent de lancer rapidement les projets en se basant souvent sur les principes convention over configuration.

DROPWIZARD VS SPRING BOOT

	DropWizard	Springboot
Dependency	Single	Multiple
Implementation	Jersey, jetty, jackson, freemarker, mustache	Spring MVC, tomcat, jackson, thymleaf
Market share	limited	important
Bootstarping	easy	medium
Extensibility	External	Withing framework

SPRING BOOT

- Framework Spring basé sur les servlets qui
- Intègre un serveur Tomcat dans une application JSE
- => Simplification du développement
- => Architecture micro-services simplifiée également
- Avatanges principaux :
  - Autoconfiguration
  - Starter



SPRING BOOT CONFIGURATION

- **Maven**

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

- **Graddle**

```
repositories {
    mavenCentral()
    maven { url 'https://repo.spring.io/milestone'
}
    maven { url 'https://repo.spring.io/snapshot' }
}
dependencies {
    implementation
'org.springframework.boot:spring-boot-starter-web'
    testImplementation
'org.springframework.boot:spring-boot-starter-test'
}
```

UNE APPLICATION DESKTOP POUR LANCER UNE APPLICATION WEB

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

SPRING ET HATEOAS

- Le support de HATEOAS est nouveau pour Spring, il est possible de servir un contenu conforme en utilisant JSON ou JAXB.
- Les mécanismes du ServletDispatcher avec un contrôleur REST (ou méthode marquée par @ResponseBody) sont à la base du contenu HATEOAS
- Toute classe peut être une ressource (avec une représentation JSON ou XML), les liens sont construits avec l'aide de la classe Link.
- Le contenu est rendu à l'aide de la classe ResponseEntity<Ressource> qu'on construit sur la base d'une ressource et des liens associés

Spring HATEOAS

UN CONTROLLEUR REST API

```
package com.example.demo.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseBody;


@Controller
class ThisWillActuallyRun
{
    @RequestMapping("/")
    @ResponseBody
    String home() { return "Hello World!" }
}
```

ANNOTATIONS

- L'annotation **@Value** est utilisable sur un attribut ou un paramètre pour un type primitif ou une chaîne de caractères. Elle donne la valeur par défaut à injecter.
- **@Resource** peut se substituer à l'annotation **@Autowired** sur les attributs et les méthodes *setter*. Le Spring Framework réalise une injection de dépendance basée sur le type attendu. Si l'annotation spécifie un nom grâce à son attribut name alors l'injection de dépendance se fait en cherchant un *bean* du même nom. Ex: @Resource(name = "tache")
- **@PostConstruct** s'utilise sur une méthode publique sans paramètre afin de signaler que cette méthode doit être appelée par le conteneur IoC après l'initialisation du *bean*. Il s'agit d'une alternative à la déclaration de la méthode d'initialisation.
- **@PreDestroy** s'utilise sur une méthode publique sans paramètre afin de signaler que cette méthode doit être appelée juste avant la fermeture du contexte d'application. Il s'agit d'une alternative à la déclaration de la méthode de destruction.

APPLICATION.PROPERTIES

- Ce fichier est présent dans le chemin de classe (*classpath*) de l'application.
- Pour un projet géré avec Maven, cela signifie que le fichier application.properties se trouve dans src/main/resources.
- Ce fichier est utilisé pour paramétrer le comportement par défaut de l'application.
- En fonction des dépendances déclarées dans notre projet et en fonction de la valeur des propriétés présentes dans ce fichier, Spring Boot va adapter la création du contexte d'application.

VALUES DÉCLARÉES DANS LE FICHIER DE PROPRIÉTÉ

database.uri = jdbc:mariadb://localhost:3306/db

database.login = root

database.password = r00t

API REST

- Services Web
- Echange de données en format Json
- Standard du web

EN SPRING

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class GreetingController {
    private static final String template = "Hello, %s!";
    @GetMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name", defaultValue =
"World") String name) {
        return new Greeting(counter.incrementAndGet(), String.format(template,
name));
    }
}
```

@CONTROLLER

- Cette annotation est utilisée pour créer une classe en tant que contrôleur Web, qui peut gérer les demandes des clients et renvoyer une réponse au client.
- Il s'agit d'une annotation au niveau de la classe, qui est placée au-dessus de votre classe de contrôleur.
- Semblable à @Service et @Repository, il s'agit également d'une annotation stéréotypée.

@REQUESTMAPPING

- La classe Controller contient plusieurs méthodes de gestionnaire pour gérer différentes requêtes HTTP, mais comment Spring mappe-t-il une requête particulière à une méthode de gestionnaire particulière ?
- A l'aide de l'annotation @RequestMapping.
  - C'est une annotation au niveau de la méthode qui est spécifiée sur une méthode de gestionnaire.
  - Il fournit le mappage entre le chemin de la demande et la méthode du gestionnaire.
  - Il prend également en charge certaines options avancées qui peuvent être utilisées pour spécifier des méthodes de gestion distinctes pour différents types de requêtes sur le même URI, comme vous pouvez spécifier une méthode pour gérer la requête GET et une autre pour gérer la requête POST sur le même URI.

@REQUESTMAPPING

- En termes simples, @RequestMapping marque les méthodes du gestionnaire de requêtes à l'intérieur des classes @Controller ; il peut être configuré en utilisant :
  - path ou ses alias, nom et valeur : l'URL à laquelle la méthode est mappée
  - method : méthodes HTTP compatibles
  - params : filtre les requêtes en fonction de la présence, de l'absence ou de la valeur des paramètres HTTP
  - headres : filtre les requêtes en fonction de la présence, de l'absence ou de la valeur des en-têtes HTTP
  - *consumes* : quels types de médias la méthode peut consommer dans le corps de la requête HTTP
  - *produces* : quels types de médias la méthode peut produire dans le corps de la réponse HTTP

  - De plus, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping et @PatchMapping sont des variantes différentes de @RequestMapping avec la méthode HTTP déjà définie sur GET, POST, PUT, DELETE et PATCH respectivement.

@REQUESTPARAM

- Il s'agit d'une autre annotation Spring MVC utile qui est utilisée pour lier les paramètres HTTP aux arguments de méthode des méthodes de gestionnaire.
- Par exemple, si vous envoyez des paramètres de requête avec URLlikie pour la pagination ou simplement pour fournir des données clés, vous pouvez les obtenir en tant qu'arguments de méthode dans vos méthodes de gestionnaire.

@PATHVARIABLE

- Il s'agit d'une autre annotation utilisée pour récupérer des données à partir de l'URL.
- Contrairement à l'annotation @RequestParam qui est utilisée pour extraire les paramètres de requête, cette annotation permet au contrôleur de gérer une demande d'URL paramétrées comme les URL qui ont une entrée variable dans le cadre de leur chemin comme :
  - http://localhost:8080/books/900083838

```
@RequestMapping(value="/books/{ISBN}", method= RequestMethod.GET)
public String showBookDetails(@PathVariable("ISBN") String id,
Model model){
    model.addAttribute("ISBN", id); return "bookDetails";
}
```

@REQUESTBODY

- Cette annotation peut convertir les données HTTP entrantes en objets Java transmis à la méthode de gestion du contrôleur.
- Tout comme @ResponseBody indique au Spring MVC d'utiliser un convertisseur de message lors de l'envoi d'une réponse au client, les annotations @RequestBody indiquent au Spring de trouver un convertisseur de message approprié pour convertir une représentation de ressource provenant d'un client en un objet.

```
@RequestMapping(method=RequestMethod.POST, consumers=
"application/json")

public @ResponseBody Course saveCourse(@RequestBody Course
aCourse){

    return courseRepository.save(aCourse);

}
```

@RESPONSEBODY

- L'annotation @ResponseBody est l'une des annotations les plus utiles pour développer un service Web RESTful à l'aide de Spring MVC.
- Cette annotation est utilisée pour transformer un objet Java renvoyé par un contrôleur en une représentation de ressource demandée par un client REST.

```
@RequestMapping(method=RequestMethod.POST,consumers= "application/json")
public @ResponseBody Course saveCourse(@RequestBody Course aCourse){
    return courseRepository.save(aCourse);
}
```

@RESPONSESTATUS

- Cette annotation peut être utilisée pour remplacer le code de réponse HTTP pour une réponse.
- Vous pouvez utiliser cette annotation pour la gestion des erreurs lors du développement d'une application Web ou d'un service Web RESTful à l'aide de Spring.

```
@Exceptionhandler(BookNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error bookNotFound(BookNotFoundException bnfe){ l
    ong ISBN = bnfe.getISBN();
    return new Error(4, "Book [" + ISBN + "] not found");
}
```

RESPONSE ENTITY

- La classe ResponseEntity est utilisée lorsque nous spécifions par programme tous les aspects d'une réponse HTTP.
- Cela inclut les en-têtes, le corps et, bien sûr, le code d'état.
- Cette méthode est la manière la plus détaillée de renvoyer une réponse HTTP dans Spring Boot, mais aussi la plus personnalisable.
- Beaucoup préfèrent utiliser l'annotation @ResponseBody couplée à @ResponseStatus car elles sont plus simples.

```
@GetMapping("/response_entity")
public ResponseEntity<String> withResponseEntity() {
    return
ResponseEntity.status(HttpStatus.CREATED).body("HTTP Status
will be CREATED (CODE 201)\n");
}
```

CONTROLLER EXEMPLE

```java
@RestController
@RequestMapping("/api/v1")
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/users")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @GetMapping("/users/{id}")
    public ResponseEntity<User> getUserById(
    @PathVariable(value = "id") Long userId) throws ResourceNotFoundException {
        User user = userRepository.findById(userId)
        .orElseThrow(() -> new ResourceNotFoundException("User not found on :: "+ userId));
        return ResponseEntity.ok().body(user);
    }

    @PostMapping("/users")
    public User createUser(@Valid @RequestBody User user) {
        return userRepository.save(user);
```

SERVICE ET REPOSITORY

- Le contrôleur doit faire appel au Service pour accéder aux données.
- Le Service lui fait appel au Repository.
- La présence du Repository nous permet également d'avoir une persistance de la donnée, par exemple via une base de données SQL.

```
public interface BookService {
  Book getBookById(Integer bookId);
  void addBook(Book book);
  List<Book> getAllBooks();
}
public interface BookRepository {
  Book getBookById(Integer bookId);
  void addBook(Book book);
  List<Book> getAllBooks();
}
```

INJECTION DANS LE CONTROLLEUR

```
private BookService bookService;


@Autowired
public BookController(final BookService bookService){
  this.bookService = bookService;
}
```

REPOSITORY

- Notez que nous avons annoté l'interface avec l'annotation @Repository. Cela indique à Spring d'amorcer le référentiel lors d'une analyse de composant.
- L'interface UserRepository étend JpaRepository qui fournit les méthodes ci-dessous pour gérer les opérations de base de données :

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.companyname.springbootcrudrest.model.User;

@Repository
public interface UserRepository extends JpaRepository<User, Long>{

}
```

MÉTHODES DU REPO

Keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1

MÉTHODES DU REPO

Keyword	Sample	JPQL snippet
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)

MÉTHODES DU REPO

Keyword	Sample	JPQL snippet
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

SPRING ACTUATOR : MÉTRICS AVEC SPRING

- **Spring Boot Actuator** est un sous projet (sub-project) du projet **Spring Boot**.
- Il est créé pour collecter, superviser les informations de l'application.
- Maven

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-actuator -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
    <version>2.6.1</version>
</dependency>
```

ACTUATOR

- En substance, Actuator apporte à notre application des fonctionnalités prêtes pour la production.
- La surveillance de notre application, la collecte de métriques, la compréhension du trafic ou l'état de notre base de données deviennent triviales avec cette dépendance.
- Le principal avantage de cette bibliothèque est que nous pouvons obtenir des outils de production sans avoir à implémenter ces fonctionnalités nous-mêmes.
- Actuator est principalement utilisé pour exposer des informations opérationnelles sur l'application en cours d'exécution - santé, métriques, informations, dump, env, etc. Il utilise des points de terminaison HTTP ou des beans JMX pour nous permettre d'interagir avec elle.
- Une fois que cette dépendance est sur le chemin de classe, plusieurs points de terminaison sont disponibles pour nous hors de la boîte.
- Comme avec la plupart des modules Spring, nous pouvons facilement le configurer ou l'étendre de plusieurs manières.

ENDPOINTS

- Dans une application Spring Boot, nous exposons un point de terminaison d'API REST en utilisant l'annotation @RequestMapping dans la classe du contrôleur.
- Actuator permet de récupérer des informations sur ces endpoints.

- Application.properties

management.endpoints.web.exposure.include=*

management.endpoint.xxx.cache.time-to-live=10s //*temps de réponse en cache*

management.endpoints.web.discovery.enabled=false

// CORS

management.endpoints.web.cors.allowed-origins=https://example.com management.endpoints.web.cors.allowed-methods=GET,POST

- Informations : http://localhost:9001/actuator.
- Deux endpoints sont publics par défaut : /health et /info.
  - { "status": "UP"} ou { "status": »DOWN"}

ENDPOINTS PRÉDÉFINI

- */beans*
  - envoie tous les beans disponibles dans notre BeanFactory
- */conditions*
  - anciennement connu sous le nom de /autoconfig, crée un rapport des conditions autour de la configuration automatique.
- */configprops*
  - nous permet de récupérer tous les beans @ConfigurationProperties.
- */env*
  - renvoie les propriétés actuelles de l'environnement. De plus, nous pouvons récupérer des propriétés individuelles.
- */health*
  - résume l'état de santé de notre application
- */heapdump*
  - construit et renvoie un vidage de tas à partir de la JVM utilisée par notre application.

ENDPOINTS PRÉDÉFINI

- */info*
  - renvoie des informations générales. Il peut s'agir de données personnalisées, d'informations de build ou de détails sur le dernier commit.
- */logfile*
  - renvoie les journaux d'application ordinaires.
- */loggers*
  - nous permet d'interroger et de modifier le niveau de journalisation de notre application.
- */metrics*
  - détaille les métriques de notre application. Cela peut inclure des métriques génériques ainsi que des métriques personnalisées.
- */scheduledtasks*
  - fournit des détails sur chaque tâche planifiée dans notre application.
- */sessions*
  - répertorie les sessions HTTP étant donné que nous utilisons Spring Session.
- */shutdown*
  - effectue un arrêt progressif de l'application.
- */threaddump*
  - vide les informations de thread de la JVM sous-jacente.

CUSTOM INDICATEUR

- Exemple : health
- Hériter le controlleur de HealthIndicator et redéfinir la éthode health()

```
@RestController
public class ProductController implements HealthIndicator {
    @Override
    public Health health() {
        …
        if(problem){
            return Health.down().build();
        }
        return Health.up().build();
    }
```

INFO

- Application.propreties

```
info.app.version=1.0-Beta
```

- Résultat

```
{
  "app": {
      "version": "1.0-Beta"
  }
}
```

CUSTOM ENDPOINT

- Annotations @Endpoint, @WebEndpoint ou @EndpointWebExtension
- Path par défaut : /actuator/id_actuator

Operation	HTTP method
@ReadOperation	GET
@WriteOperation	POST
@DeleteOperation	DELETE

EXEMPLE

- management.endpoints.web.exposure.include=helloworld
- http://localhost:8080/actuator/helloworld
- http://localhost:8080/actuator/helloworld/name=toto
- http://localhost:8080/actuator/helloworld/toto

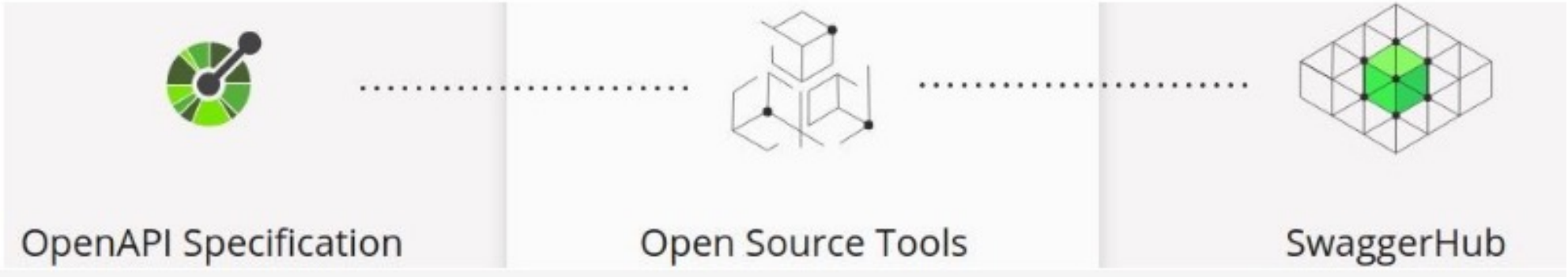
```
@Endpoint(id = "helloworld")
public class HelloWorldEndpoint {

	@ReadOperation
		public String helloWorld() {
		return "Hello World";
	}

	@ReadOperation
	public String helloName(String name) {
		return "Hello " + name;
	}

	@ReadOperation
	public String helloNameSelector(@Selector String name) {
		return "Hello " + name;
	}
}

@Configuration
public class CustomActuatorConfiguration {

	@Bean
	public HelloWorldEndpoint helloWorldEndpoint() {
		return new HelloWorldEndpoint();
	}

}
```

EXEMPLE POST & DELETE

```
@WriteOperation
public String helloNameBody(String name) {
    return "Hello " + name;
}

@DeleteOperation
public String goodbyeNameParam(String name) {
    return "Goodbye " + name;
}

@DeleteOperation
public String goodbyeNameSelector(@Selector String name) {
    return "Goodbye " + name;

}
```

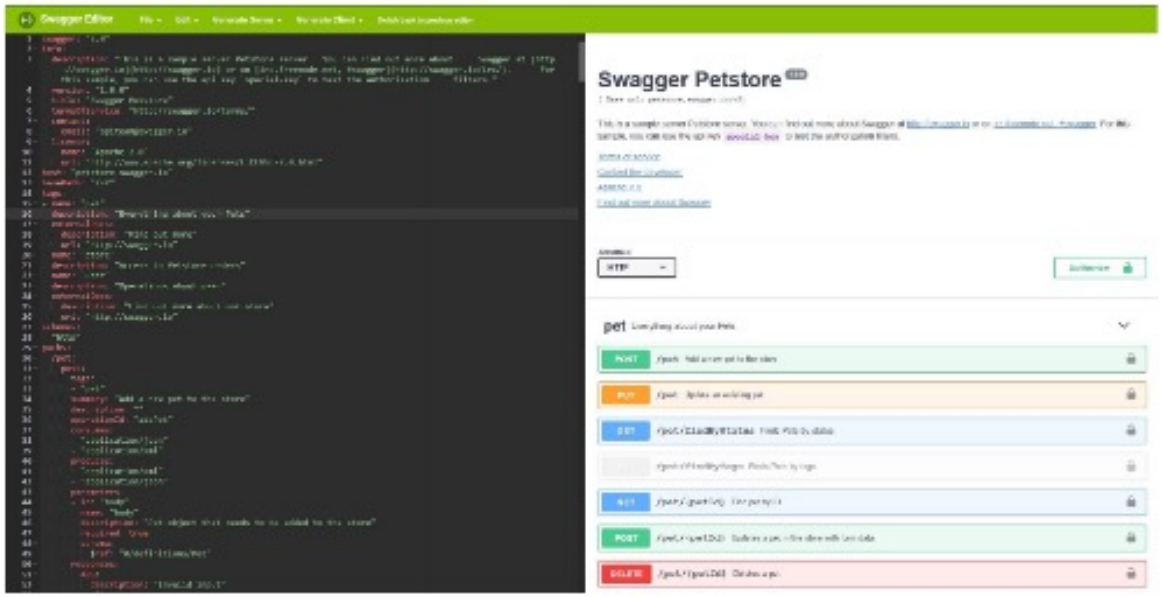
SWAGGER

- Swagger réfère à un framework opensource basé sur une famille d'outils qui permettent de conceptualiser, construire, documenter et consommer des service web REST.
  - L'open API : actuellement en version 3 renforce l'utilisation d'un format, structure, datatype et schéma standards
  - Outils :Notamment un éditeur, un générateur de code et une IHM
  - Swagger hub :Centralisation pour l'hébergement, collaboration et publication des APIs standardisées



SWAGGER

- Swagger est devenu un standard largement accepté, il permet d'aborder les microservices par le design d'abord. Smart bear met à disposition ses outils openApi codegen et swagger hub pour concevoir et collaborer sur les microservices avant de passer à l'implémentation sur un framework de choix.
- Pour définir un service en openApi sur codegen on utilise le langage Yaml pour :
  - Donner les informations générales de l'API,
  - Décrire les méthodes REST, leurs paramètres et retours.
  - Définir les ressources ou structures des objets échangés.



DES QUESTIONS?

MERCI POUR VOTRE ATTENTION.