Architecture des applications

1 Architecture des applications (6 crédits)

Première partie : Archi. JEE avancée (Spring-JPA / REST-JWT / VueJS)

- Responsable : Jean-Luc Massat (`jean-luc.massat@univ-amu.fr`)
- Volume : 30 heures (CM / TP)
- Évaluation : un projet à rendre par groupe de deux étudiants
- Sujet : `http://tinyurl.com/jlmassat/ens/arch-app`
- Pré-requis : Java, JEE 1, SGBDR, HTTP/HTML/CSS/XML
- Compte pour 1/2 de la note finale

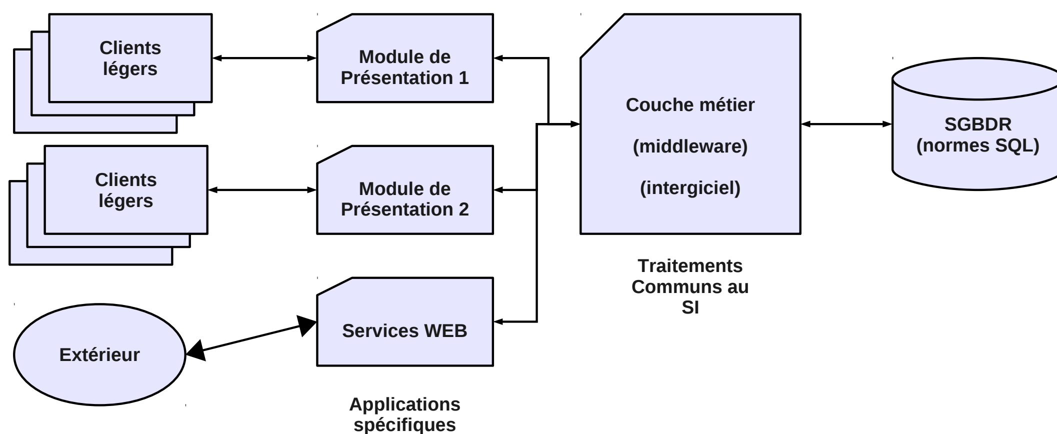
Deuxième partie : Architecture Cloud des applications

- Responsable : Christophe Jullien (capgemini) et d'autres personnes
- Volume : 30 heures (CM / TP)
- Évaluation : à voir avec C. Jullien
- Compte pour 1/2 de la note finale

2 Architecture 3-tiers

Principe : Séparation entre

- la couche de gestion des données (données métier),
- la couche de présentation (logique applicative) et
- la couche métier (actions métier de traitement des données métier).

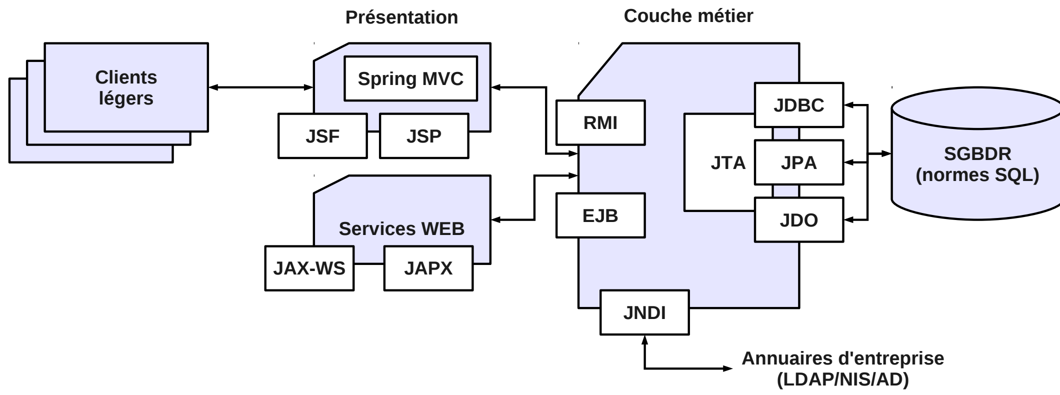


2.1 Architecture JEE

Java Enterprise Edition : JEE 8 (2017), **Jakarta EE 8** (2019), **JEE 9** (2020), **JEE 9.1** (2021), **JEE 10** (préparation)

Cours de M1 : `http://tinyurl.com/jlmassat/ens/jee-pour-M2`

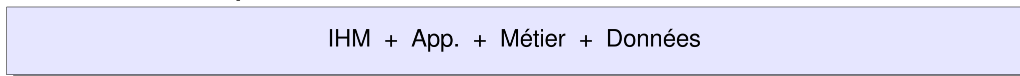
En M1 : **Spring MVC / JSP** - **Spring IoC** - **JDBC/JPA**



En M2 : **Rest-API / VueJS** - **Spring** - **JPA / redis**

2.2 Évolution des architectures

Architecture monolithique :

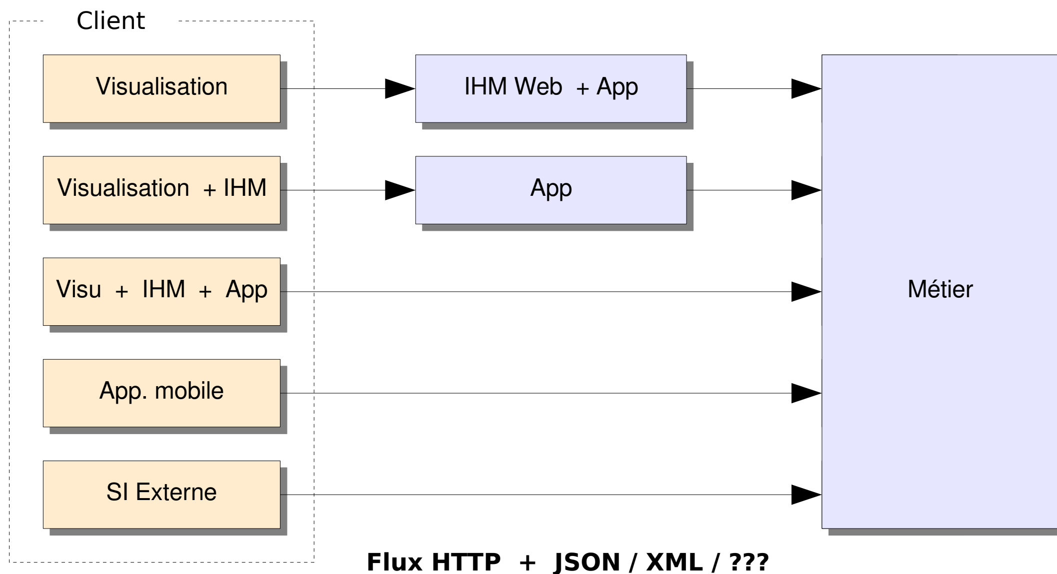


Serveurs de données :

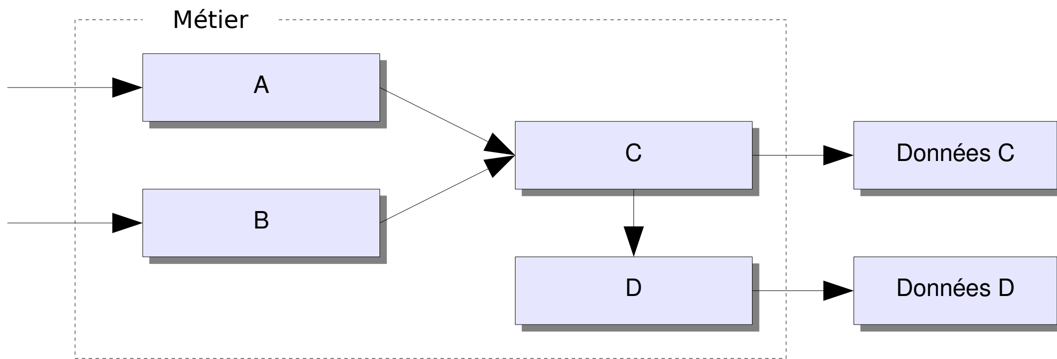
Architecture client-serveur :

Architecture 3-tiers :

2.3 Architectures orientées services



2.4 Architectures micro-services



- Chaque micro-service possède une **cohérence interne forte**.
- Les MS sont déployés dans une **machine virtuelle** ou dans un **conteneur** via docker.
- L'ensemble est organisé via un **système d'orchestration** (kubernetes).
- La conception, la réalisation et le déploiement doit être rapide et souple.

3 API-Rest : Principes

le protocole **REST** (REpresentational State Transfer), proposé en 2000, est une solution simple à la mise en place de (micro-)services WEB.

Caractéristiques :

- Indépendance des consommateurs et des producteurs.
- Protocole sans état.
- Les ressources manipulées sont identifiées par des noms (`URI`).
- Les actions sont limitées (en lien avec le protocole sous-jacent).
- Pas d'auto-description (contrairement à SOAP et WSDL).
- Le protocole ne nécessite pas d'encodage sophistiqué (pas d'enveloppe).
- Un cache est envisageable.
- L'utilisation de la bande passante est limitée.

Utilisation : basé la plupart du temps sur `HTTP` et `JSON/XML` , il est largement utilisé dans les applications WEB et mobiles.

Les **requêtes** sont basées sur

- le transport via le protocole HTTP (TCP/IP),
- une action codée par la méthode HTTP (`GET` , `POST` , `PUT` , `DELETE` , ...),
- une ressource identifiée par l' `URI` ,
- des données codées par
  - des paramètres dans l'URI,
  - une donnée XML/JSON placée dans le corps,
  - des en-têtes.

```
GET /calculator/show HTTP/1.0
connection: keep-alive
cache-control: no-cache

100
```

Les **réponses** sont basées sur

- le résultat HTTP (`OK` , `Created` , `Accepted` ,...)
- les données codées par
  - ▷ une donnée texte/XML/JSON placée dans le corps,
  - ▷ des en-têtes.

```
HTTP/1.0 200 OK
cache-control: no-cache, no-store, max-age=0, must-revalidate
pragma: no-cache
expires: 0
content-type: application/json
date: Wed, 06 Oct 2020 16:01:43 GMT

[100,200,300]
```

3.1 API-Rest : Bonnes pratiques

Pour les actions, **utilisez les méthodes HTTP** :

- `GET` : Lire une information
- `POST` : Ajouter une information
- `PUT` : Mettre à jour une information
- `DELETE` : Supprimer une information
- ...

Pour les ressources, **utilisez des noms en anglais et au pluriel** (pas de verbe) :

- `GET /movies` : Lire tous les films
- `POST /comments` : Ajouter un commentaire

Pour les réponses, **utilisez le résultat HTTP** :

- `200 OK` : ressource trouvée
- `404 Not Found` : ressource non trouvée
- `403 Forbidden` : opération interdite
- `201 Created` : ressource créée
- `204 No Content` : ressource détruite
- ...

Pour identifier une donnée, **utilisez les sous-ressources** :

- `GET /users/100` : Lire l'utilisateur identifié par `100`
- `PUT /movies/AZ401` : Modifier le film `AZ401`
- `DELETE /comments/200` : Supprimer le commentaire `200`

Les clefs primaires doivent être simples. Cela favorise la mise en place d'ID **auto-générés** et l'utilisation de clefs naturels n'est pas souhaitable.

Pour gérer les relations, **utilisez également les sous-ressources** :

Modèle de données

```
User  --> OneToMany --> Post
Post  --> OneToMany --> Comment
```

- `GET /users/100/posts` :
  Lire les publications de l'utilisateur `100`.

- `POST /posts/200/comments/300` :
  Ajouter le commentaire `300` à la publication `200`.

- `DELETE /posts/400/comments/500` :
  Délier le commentaire `500` à la publication `400` (nous pouvons aussi faire `DELETE /comments/500`).

Utilisez de préférence un **encodage** `UTF-8` et typez correctement le résultat de vos requêtes : `application/json` (ou XML).

Enrichissez vos réponses de **liens vers les autres actions possibles** (principe HATEOAS : Hypermedia As The Engine of Application State). Un exemple : la requete `GET /users/100` va vous renvoyer

```
{
    "name": "User 100",
    "id": 100,
    "_links" : {
        "self": "http://localhost:8080/users/100",
        "list": "http://localhost:8080/users"
        "xxxx": "http://localhost:8080/users/100/xxxx"
    }
}
```

Ce principe est très difficile à respecter à moins d'utiliser un outil automatique.

Prévoyez d'offrir des fonctions **de filtrage**, de **tri** et de **pagination** :

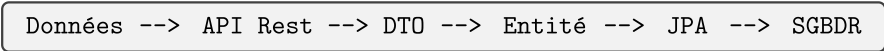
- `GET /users?name=fred` : lister les utilisateurs qui s'appellent « fred ».
- `GET /users?sort=-name,+id` : lister les utilisateurs triés de manière décroissante sur le nom et de manière croissante sur l'ID.
- `GET /users?limit=10&offset=100` : lister au plus 10 utilisateurs à partir de la position 100.

Prévoyez un mécanisme de **versionnage** de votre API.

- `GET /api/users` : les utilisateurs (version par défaut).
- `GET /api3/users` : les utilisateurs dans la version 3.
- `GET /api/users?version=4` : les utilisateurs dans la version 4 (les versions croissantes doivent être rétro-compatibles).

3.2 Le problème des relations

Il est intéressant de gérer les relations de manière plus directe en utilisant des versions adaptées des entités (DTO : Data Transfert Object).



Nous remplaçons (pour les publications) l'entité par l'objet DTO :

```
public class Post {
    String title;
    String description;
    User user;
    ...
}
```

```
{ "title":"Un titre",
  "description":"Un texte",
  "user": { "name":"User1" } }
```

```
public class PostDTO {
    String title;
    String description;
    int userId;
    ...
}
```

```
{ "title":"Un titre",
  "description":"Un texte",
  "userId":100 }
```

3.3 Utiliser JEE (API JAX-RS)

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("/api")
public class HelloJEERestController {

    @POST() @GET() @Path("hello")
    public String hello() {
        return "Hello";
    }

    @GET() @Path("hello/{message}")
    public String message(@PathParam("message") @DefaultValue("Salut") String m) {
        return "Hello␣" + m;
    }

}
```

L'API **JAXB** (Java API For XML Binding) est utilisée pour les transformations d'instances java en données JSON/XML et vice-versa.

3.4 Utiliser Spring

La partie API-Rest de **Spring** est une extension de **Spring MVC** :

```
import java.util.Date;

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

@RestController
@RequestMapping("/api")
public class HelloRestController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello␣" + (new Date());
    }

}
```

Le framework **Jackson** est utilisé pour les transformations d'instances java en données JSON/XML et vice-versa.

3.5 Utiliser Spring Data Rest

- **Constat** : Les échanges de données en entrée et en sortie représentent une grande partie des API-Rest.
- Des actions métier sont également proposées, mais elles sont minoritaires.
- **Spring Data Rest** se propose d'automatiser la construction d'API-Rest à partir des *repositories* de **Spring data**.

```
GET /users/100/posts
```

3.6 Authentification

Les API-Rest sont sans état (comme HTTP). Il faut donc ajouter un système de **jeton** afin d'identifier un client et assurer la sécurité des échanges.

La solution standard repose sur l'utilisation d'un `JWT` (Json Web Token). Ce dernier est

- composé de trois parties codées en base64 et séparées, par un point :

```
<HEADER.PAYLOAD.SIGNATURE>
```

- L'en-tête est composée de la spécification de l'algorithme de signature et du type de jeton :

```
{ "alg": "HS256", "typ": "JWT" }
```

- La charge utile (Payload) est un ensemble de couples clé/valeur (appelés des claims) :

```
{ "id": "1234", "name": "pierre", "exp": 30 }
```

- Les claims peuvent être
  - ▷ réservées : leur objet est défini dans une norme (expiration par exemple)
  - ▷ publiques : définies librement (mais pas trop longues). Stocker des données est sans doute une mauvaise idée.
  - ▷ privées : informations spécifiques à une application.
- La signature est fabriquée avec algorithme spécifié, l'en-tête, la charge utile et une clé secrète connue du serveur. Il existe trois niveaux de sécurité :
  - ▷ aucune : le JWT n'est pas signé.
  - ▷ signé : la signature permet au serveur de valider un JWT qu'il a lui même fabriqué. Il n'y a pas de vérification de l'identité de l'émetteur
  - ▷ crypté : une clé privée est utilisée pour crypter la charge utile avant sa signature. La clef publique est utilisée pour le décodage. Il est donc possible de vérifier l'émetteur.

3.7 Génération d'un JWT

Le jeton est

- fabriqué sur le serveur par une requête d'authentification (`GET /api/login`),
- placé dans les en-têtes de la réponse pour transmission au client,
- récupéré par le client et replacé dans les en-têtes des requêtes suivantes,
- vérifié par le serveur pour les entrées qui nécessitent une authentification,

Attention :

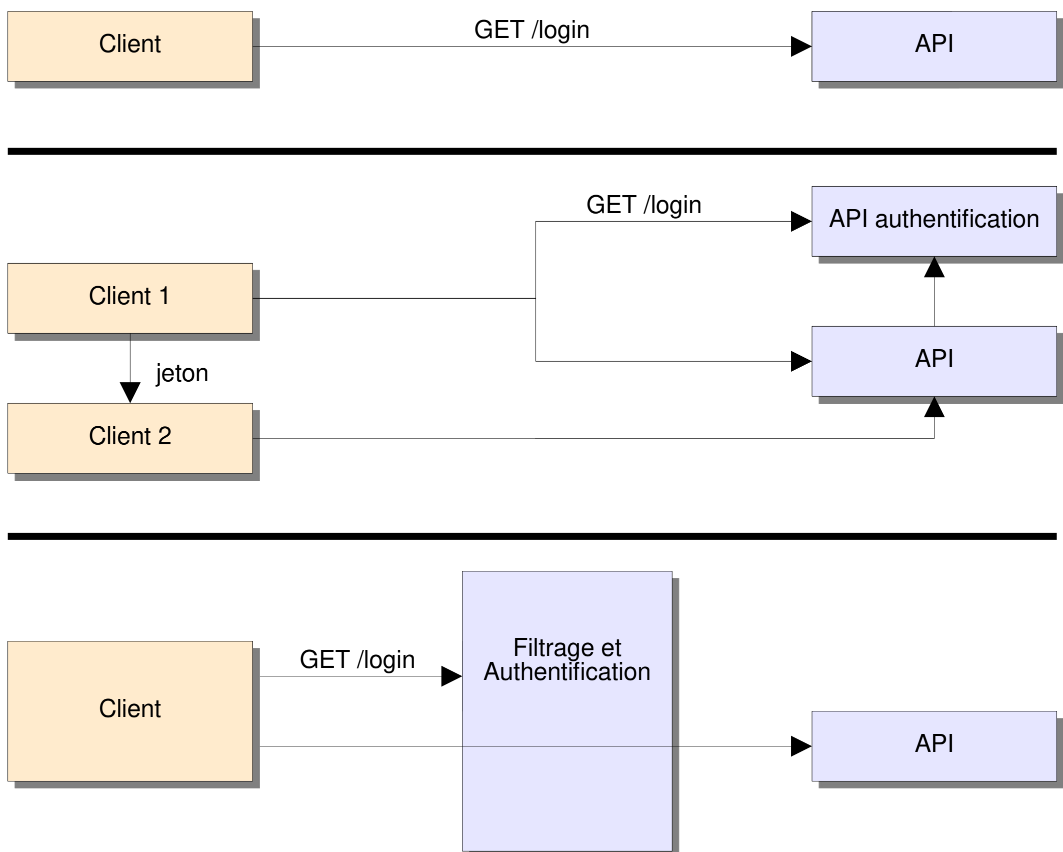
- Un JWT est sans état, il ne peut donc pas être désactivé.
- La déconnexion doit entraîner l'oubli du jeton par le client, mais ce dernier est toujours valide (jusqu'à expiration).
- Pour renforcer la sécurité, il faut maintenir, coté serveur une liste noire des jetons désactivés.
- Cette liste noire doit périodiquement être nettoyée des jetons expirés.

Nous allons utiliser le framework **jjwt** pour utiliser les JWT.

L'utilisation de **HTTPS** est conseillé pour la sécurité et la confidentialité des échanges entre client et serveur.

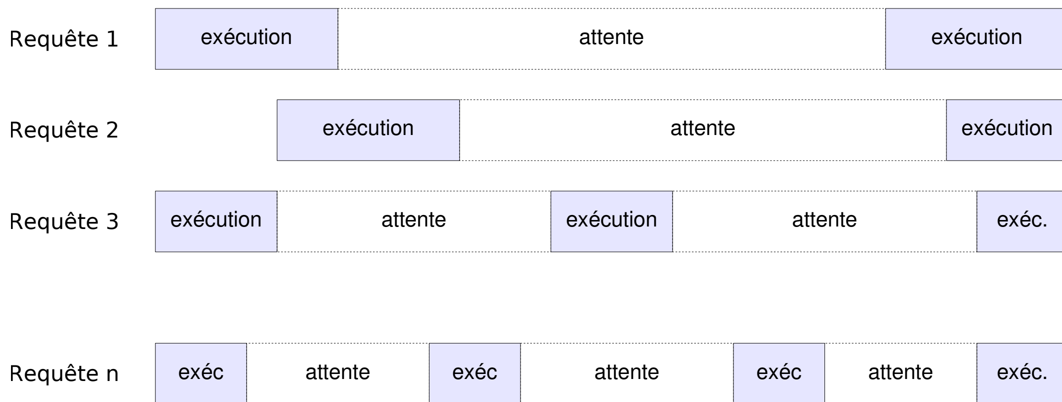
```java
public String createToken(String username) {

  Claims claims = Jwts.claims().setSubject(username);
  claims.put("auth", "USER,ADMIN");

  Date now = new Date();
  Date validity = new Date(now.getTime() + validityInMilliseconds);

  return Jwts.builder()//
      .setClaims(claims)//
      .setIssuedAt(now)//
      .setExpiration(validity)//
      .signWith(SignatureAlgorithm.HS256, secretKey)//
      .compact();
}
```

3.8 Architectures JWT



4 Services à haut-débit

Le serveur est continuellement occupé par le traitement des requêtes **synchrones** et la gestion des **threads bloqués** :



Cette situation entraîne une **surcharge système** et une mauvaise utilisation des ressources.

4.1 Programmation asynchrone

L'idée est de remplacer la programmation synchrone par une programmation **asynchrone** :

```
public void aa() { ... }
public void bb() { ... }
public void cc() { ... }

public void job() {
    aa();
    bb();
    cc();
}
```

```
public void aa(Runnable next) { ... }
public void bb(Runnable next) { ... }
public void cc() { ... }

public void job() {
    aa(() -> {
        bb(() -> {
            cc();
        });
    });
}
```

La distribution de la CPU est effectuée par une boucle d'événements.

Si nous avons des **données** qui circulent :

```
// seulement producteur
public void aa(Consumer<Data> next) { ... }

// consommateur et producteur
public void bb(Data d, Consumer<Data> next) { ... }

// seulement consommateur
public void cc(Data d) { ... }

public void job() {
    aa((data) -> {
        bb(data, (dataNext) -> {
            cc(dataNext);
        });
    });
}
```

Cette approche est appelée **programmation réactive** : l'appel à `bb` est une réaction à la publication de données par `aa`.

4.2 Programmation Réactive

Nous allons utiliser le framework `reactor` pour nous aider dans la mise en place de cette programme réactive :

```
public Data aa(Data input) {
    // consume and produce
    return new Data(...);
}

public void job() {
    Data d1 = new Data(10);
    Data d2 = new Data(20);
    Flux.just(d1, d2)//
            .map(this::aa)//
            .map(this::bb)//
            .map(this::cc)//
            .doOnNext(System.out::println)//
            .doAfterTerminate(() -> { System.out.println("Done"); })//
            .subscribe() ;
}
```

Les flux peuvent être **(a)synchrones**, **fusionnés**, **filtrés**, **transformés**.

4.3 Utiliser Spring Web-flux

Le framework **Spring-flux** se propose d'appliquer la programmation réactive à la construite d'API-Web (notamment d'API-REST) :

Forme classique

```
@GetMapping("/users")
private Flux<User> getAllUsers() {
    return userRepository.findAllUsers();
}
```

Forme fonctionnelle

```
@Bean
RouterFunction<ServerResponse> userRoute() {
    return route(GET("/users"),
        req -> ok().body(
            userRepository.findAllUsers(), User.class))

    .and(route(GET("/users/{id}"),
        req -> ok().body(
            userRepository.findUser(req.pathVariable("id")), User.class)));
}
```

Code du client

```
WebClient client = WebClient.create("http://localhost:8080");
Flux<User> userFlux = client.get()
  .uri("/users")
  .retrieve()
  .bodyToFlux(User.class);

employeeFlux.subscribe(System.out::println);
```

Remarques :

- La forme fonctionnelle est adaptée à la construction de micro-service simple et facile à concevoir.

- L'utilisation d'une API réactive nécessite une couche de stockage asynchrone. Ce n'est pas le cas de JDBC ou de JPA.
- Il faut éventuellement se tourner vers des systèmes NoSQL qui offrent ce service.

5 Architecture JEE classique

Dans l'approche **JEE traditionnelle**, la pile de spécifications prévues est la suivante :

- `CDI` (Context and Dependency Injection) : Injection de dépendances.
- `JPA` : Accès aux données.
- `EJB` (Enterprise Java Bean) : Construction des services métiers.
- `JSF` (Java Server faces) : Construction des applications WEB par une approche composants et événements.
- `JAX-RS` : Construction d'API REST.

5.1 CDI : Context and Dependency Injection

Définition d'un POJO avec un nom et une portée :

```java
package myapp.cdisample;

import java.io.Serializable;
import javax.enterprise.context.RequestScoped;
import javax.inject.Named;

@RequestScoped
@Named("normal")
public class Home implements Serializable {

    private static final long serialVersionUID = 1L;

    String place;

    // getter / setter
}
```

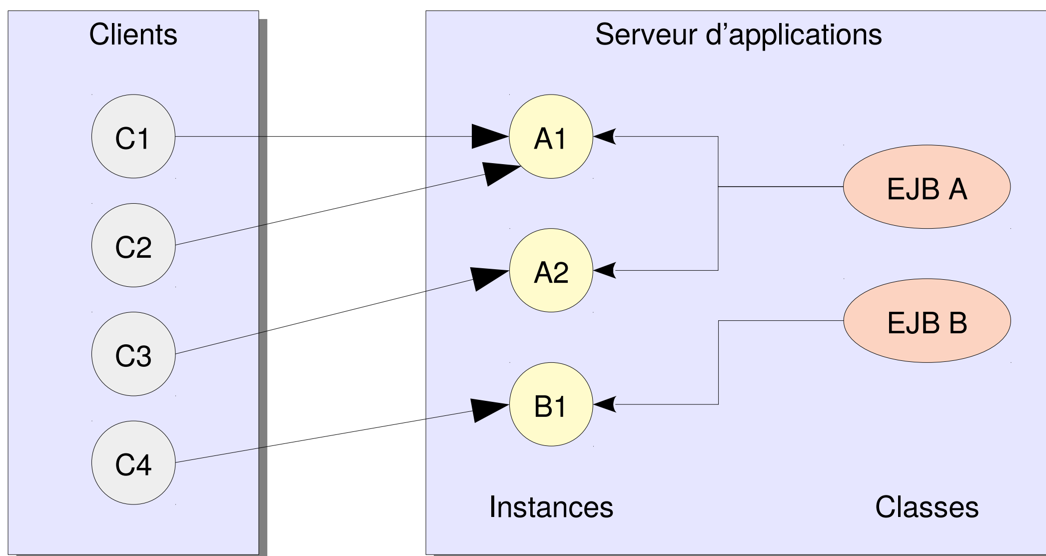
Injection via les annotations :

```java
package myapp.cdisample;

import java.io.Serializable;
import javax.inject.Inject;
import javax.inject.Named;

@Named("person")
public class Person implements Serializable {

    private static final long serialVersionUID = 1L;

    @Inject @Named("normal")
    private Home home;

    // getter / setter
}
```

Autres possibilités :

- Qualification des instances,
- producteurs et recycleurs d'instances,
- abonnement à des événements,

5.2 EJB : Enterprise java Bean Stateless



Déroulement (exploitation des architectures multi-processeurs-coeurs) :

```
thread 1 : <--- C1 sur A1 ---><--- C2 sur A1 --->
thread 2 : <------- C3 sur A2 ------->
thread 3 : <---- C4 sur B1 ---->
```

Avec une troisième instances de A :

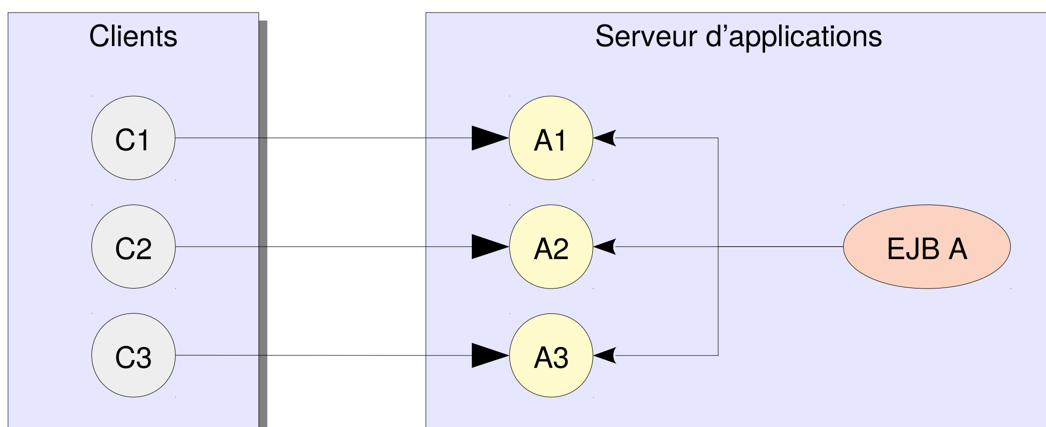
```
thread 1 : <--- C1 sur A1 --->
thread 4 : <--- C2 sur A3 --->
thread 2 : <------- C3 sur A2 ------->
thread 3 : <---- C4 sur B1 ---->
```

La **montée en charge** implique une augmentation du nombre d'instances.

Gestion automatique (mais paramétrée) par le serveur d'applications.

5.2.1 EJB Statefull

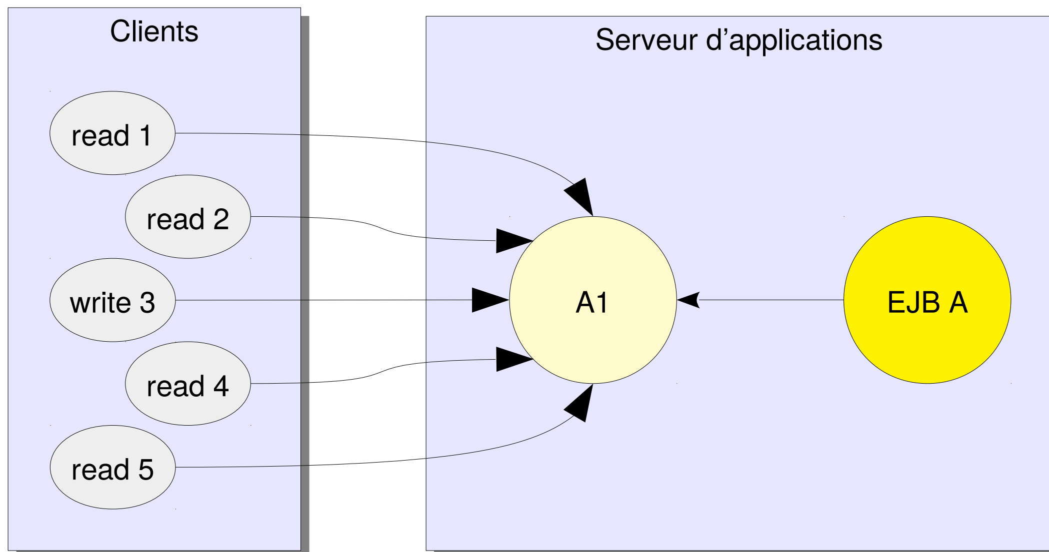
EJB avec état utilisé par un seul client (**non simultanément**).



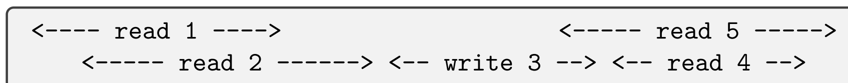
Utilité : représentation d'un utilisateur et fonctions associées.

5.2.2 EJB Singleton

EJB a instance unique (avec état) utilisé par plusieurs clients.



Déroulement (parallélisme des *read* et mise en série des *write*) :



```
<---- read 1 ---->                      <----- read 5 ----->
   <----- read 2 ------> <-- write 3 --> <-- read 4 -->
```

5.2.3 EJB : Mise en oeuvre de JPA

```
@PersistenceContext(unitName = "myBase")
EntityManager em;

public List<User> findAllUsers() {
    TypedQuery<Person> q = em.createNamedQuery("findAllUsers", User.class);
    return q.getResultList();
}
```

Le serveur d'applications gère (via les proxys) :

- l'ouverture de la transaction
- la validation de la transaction
- les erreurs d'exécution

Approche transactionnelle (très utile pour les opérations complexes).

5.2.4 EJB : Autres fonctions

- définition de traitements périodiques (similaire à `cron` d'UNIX)
- mise en place d'intercepteur (programmation par aspects)
- définition de traitements asynchrones (pour les traitements longs)
- EJB **orientés message** : traitement d'un file de messages (requêtes)

5.3 JSF : Java Server Faces

Principes : Framework WEB basé sur l'architecture **MVC** :

- **View** : facelet (document XML doté d'un langage d'expression)
- **Controler** : *Managed beans* instanciés par le framework
- **Model** : *POJO et Managed beans*

Caractéristiques :

- Utilisation d'une approche composant
- Mise en correspondance automatique entre POJO et Facelet
- Framework Javascript de gestion des pages (AJAX)
- Processus de validation
- Présence de nombreuses librairies externes (PrimeFaces, ...)

5.3.1 JSF : Un premier exemple

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html" >
    <h:body>
        <h1><h:outputText value="#{helloControler.hello}" /></h1>
    </h:body>
</html>
```

```
package myapp;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "helloControler") @SessionScoped
public class MyControler {

    private String hello = "Hello, world";

    public String getHello() { return hello; }
    public void setHello(String hello) { this.hello = hello; }
}
```

5.3.2 JSF : Utilisation d'un formulaire

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html" >

<h:body>
    <h1><h:outputText value="#{helloControler.hello}" /></h1>

    <h:form>
        <h:inputText id="text" value="#{helloControler.hello}"
            required="true" requiredMessage="Le texte est obligatoire">
            <f:validateLength minimum="3" maximum="15" />
        </h:inputText>
        <h:message errorClass="error" for="text" />
        <br/>

        <h:commandButton value="Submit" action="#{helloControler.sayHello}" />
    </h:form>
</h:body>
</html>
```

Amélioration du contrôleur :

```
package myapp;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "helloControler") @SessionScoped
public class MyControler {

    private String hello = " Hello, world";

    // getter / setter
    public String getHello() { return hello; }
    public void setHello(String hello) { this.hello = hello; }

    public String sayHello() {
        hello = hello.toUpperCase();
        return null;
    }

}
```

5.3.3 JSF : Autres caractéristiques

- Grande facilité de mise en place d'AJAX
- Possibilités de navigation avancées
- Cinq portés disponibles :
  - ▷ application
  - ▷ session
  - ▷ conversation (notion de sous-session)
  - ▷ vue (on reste sur la même page JSF)
  - ▷ requête
- Traitement des événement HTML (éventuellement en AJAX)

- Possibilité de couplage avec bootstrap ou Angular

5.4 Intégration

Dans une application, l'API CDI va

- Injecter les EntityManager dans les EJB.
- Injecter les EJB dans d'autres EJB et dans les contrôleurs JSF
- Injecter les beans de portée vue, requêtes, session dans les contrôleurs JSF