Avant Propos

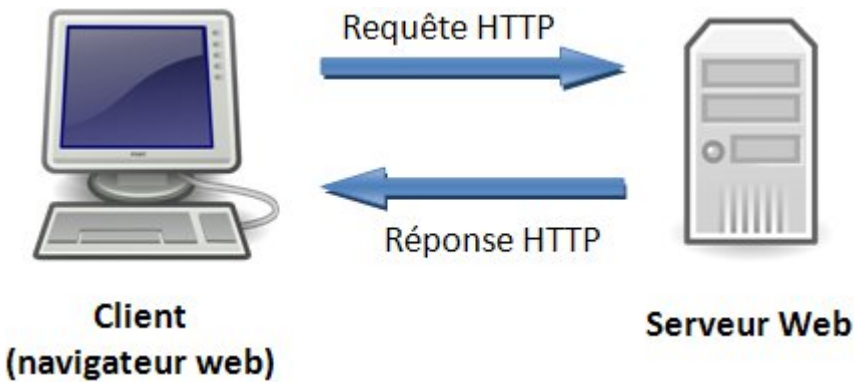
---

semifir

Protocole Réseaux

HTTP est un protocole de communication entre un client et un serveur.

Requête/Response



Requête

Le client envoie une requête au serveur.

Une requête est composée de:

- une méthode: `GET`, `POST`, `PUT`, `DELETE`, ...
- une URL: `http://localhost:8080/api/clients`
- des headers: `Content-Type: application/json`
- un corps: `{"nom": "Dupont", "prenom": "Jean"}`

Exemple de requête

Exemple de requête qui envoie le résultat d'un formulaire au serveur.

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

URI

Une `URI` est une `Uniform Resource Identifier`.

Une `URI` est une chaîne de caractères qui identifie une ressource.

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Host

Le `Host` est l'adresse du serveur.

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

URI + Host = URL

Une `URL` est une `Uniform Resource Locator`.

Une `URL` est une `URI` qui contient le `Host`.

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Méthode

La méthode est la requête que le client veut effectuer.

Ici `POST`:

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Les méthodes

Il existe plusieurs méthodes:

- `GET`: récupérer une ressource
- `POST`: créer une ressource
- `PUT`: modifier une ressource

- `DELETE`: supprimer une ressource
- `PATCH`: modifier une partie d'une ressource

Headers

Les headers sont des informations supplémentaires sur la requête.

Ici `Content-Type: application/json`:

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Les headers

Les headers sont composés d'un nom et d'une valeur.

Ils permettent d'ajouter des informations sur la requête:

- le type de contenu: `Content-Type: application/json`
- la taille du contenu: `Content-Length: 41`
- l'authentification: `Authorization: Bearer 1234567890`
- ...

Vous pouvez créer vos propres headers.

Le corps

Le corps est le contenu de la requête.

Ici `{"nom": "Dupont", "prenom": "Jean"}`:

```
POST /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Les réponses

1. Le client envoie une `requête` au serveur.

2. Le serveur traite la `requête`.

3. Le serveur envoie une `réponse` au client.

Réponse

Une `réponse` est composée de:

- un code status: `200`, `404`, `500`, ...
- des headers: `Content-Type: application/json`
- un corps: `{"nom": "Dupont", "prenom": "Jean"}`

Exemple de réponse

Exemple de réponse qui renvoie le résultat d'une requête au client.

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Code status

Le code status est le code de la réponse.

Ici `200`:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

HttpCats

HttpDog

Code de status

Les Codes commençant par :

- 1xx : Information pour le client
- 2xx : Indique que la demande a été traitée avec succès
- 3xx : Il faut rediriger le client ou la ressource a été déplacée
- 4xx : Erreur du client
- 5xx : Erreur du serveur

Communication front/back

- le front joue le rôle de client
- le back joue le rôle de serveur

API REST

API REST

L'HTTP ne définit pas de standard pour les API.

Exemple:

```
GET /api/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41
```

Exemple:

```
POST /recuperation/des/clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41
```

Pas de standard, pas de convention = problèmes

Sans convention, les API sont difficiles à utiliser.

> Comme dans la vraie vie, il faut des conventions pour que tout le monde comprenne ce que l'autre veut dire.

API REST

Une `API REST` est une API qui respecte les conventions de l'architecture `REST`.

Elle est basée sur le protocole `HTTP`.

Règles d'une API REST

- Utilisation des méthodes HTTP
- Utilisation d'une norme des URI
- Utilisation du JSON pour le corps des requêtes et des réponses

---

Utilisation des méthodes HTTP

- `GET`: récupérer une ressource
- `POST`: créer une ressource
- `PUT`: modifier entièrement une ressource
- `DELETE`: supprimer une ressource
- `PATCH`: modifier une partie d'une ressource

Utilisation d'une norme des URI

Avec la norme `REST`, les URI suivent une convention de nommage.

On parle de ressource pour chaqu'une des données. Par exemple, une ressource `client`.

Utilisation d'une norme des URI

Imaginons une API qui fournit des données sur le client ayant le format suivant:

```
{
    "id": 1,
    "nom": "Dupont",
    "prenom": "Jean",
    "adresse": {
        "rue": "rue de la paix",
        "codePostal": "75000",
        "ville": "Paris"
    }
}
```

Récupération de l'ensemble des clients

L'`URI` commence par le nom de la ressource au pluriel: `/clients` et nous utilisons la méthode `GET`.

```
GET /clients
```

Récupération d'un client

En reste chaque ressource possède un identifiant unique.

Pour récupérer un client, on ajoute l'identifiant à l'URI: `/clients/1` et nous utilisons la méthode `GET`.

```
GET /clients/1
```

Récupération des clients portant le nom "Dupont"

Pour récupérer les clients portant le nom "Dupont", on ajoute le nom à l'URI: `/clients?nom=Dupont` et nous utilisons la méthode `GET`.

```
GET /clients?nom=Dupont
```

Récupération de l'adresse d'un client

Pour récupérer l'adresse d'un client, on ajoute l'identifiant du client ainsi que le nom du champ à l'URI.

```
GET /clients/1/adresse
```

Création d'un client

Pour créer un client, on ajoute le nom de la ressource au pluriel à l'URI et nous utilisons la méthode `POST`.

```
POST /clients HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Modification d'un client

Il existe deux méthodes pour modifier un client:

- `PUT`: on modifie l'ensemble des champs de la ressource.
- `PATCH`: on modifie un ou plusieurs champs de la ressource

Modification de l'ensemble des champs d'un client

Comme pour la création, on ajoute le nom de la ressource au pluriel à l'URI et nous utilisons la méthode `PUT`.

```
PUT /clients/1 HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 41

{"nom": "Dupont", "prenom": "Jean"}
```

Modification partielle d'un client

Comme pour la création, on ajoute le nom de la ressource au pluriel à l'URI et nous utilisons la méthode `PATCH`.

Avec `PATCH`, seul les champs modifiés sont envoyés.

```
PATCH /clients/1 HTTP/1.1
Host: localhost:8080
Content-Type: application/json
Content-Length: 17

{"nom": "Dupont"}
```

Suppression d'un client

Pour supprimer un client, on ajoute l'identifiant à l'URI et nous utilisons la méthode `DELETE`.

`DELETE /clients/1`

Des notions importantes

Spring et IOC

Pour comprendre spring il faut connaître 2 notions:

- `l'injection de dépendance` ou `IOD`
- `l'inversion de contrôle` ou `IOC`
- la notion d'annotation en `java`

Injection de dépendances

L'`injection de dépendances` est une notion de programmation orientée objet. Elle stipule que les classes ne doivent pas dépendre des autres classes.

Exemple

Vous voulez fabriquer une voiture. Vous avez besoin d'un moteur pour la faire fonctionner.

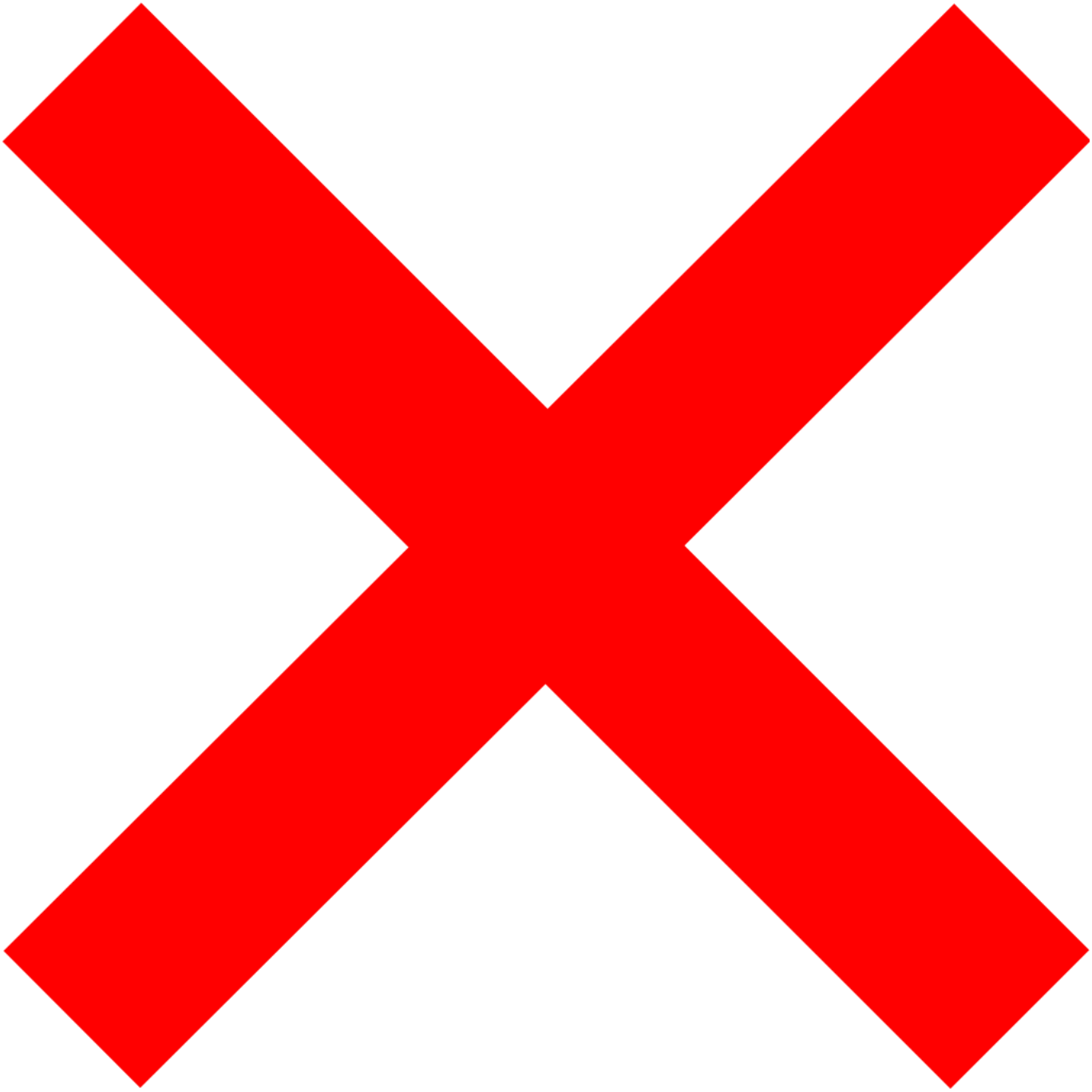
Quand vous créer une voiture, vous pouvez créer le moteur directement dans la classe voiture:

```
public class Voiture {
    private Moteur moteur = new Moteur();

    public void rouler() {
        moteur.demarrer();
    }
}
```

Exemple

Dans ce cas, la classe voiture dépend de la classe moteur. Si vous voulez changer de moteur, vous devez changer la classe voiture.



```
public class Voiture {
    private Moteur moteur = new MoteurElectrique();

    public void rouler() {
        moteur.demarrer();
    }
}
```

Dépendance forte

On parle ici de `dépendance forte`. La classe voiture dépend de la classe moteur. Si vous voulez changer de moteur, vous devez changer la classe voiture.

Les `dépendances fortes` sont mauvaises car elles rendent le code difficilement maintenable.

Solution

La solution est d'injecter le moteur dans la classe voiture en passant par le constructeur:

```
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

Solution

Dans ce cas, la classe voiture n'a plus besoin de connaître la classe moteur. Elle peut donc être utilisée avec n'importe quel moteur.

```
Voiture voitureElectrique = new Voiture(new MoteurElectrique());

Voiture voitureEssence = new Voiture(new MoteurEssence());

Voiture voitureHybride = new Voiture(new MoteurHybride());
```

Injection de dépendances

C'est le principe de l'`injection de dépendances`. On injecte les dépendances dans les objets.

`Spring` est un framework qui permet de faire de l'injection de dépendances!

Il y a 2 types d'injection de dépendances

- L'injection par constructeur

- L'injection par setter

```
public class Voiture {
    private Moteur moteur;

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void setMoteur(Moteur moteur) {
        this.moteur = moteur;
    }

    ...
}
```

Injection de dépendances

La solution par `constructeur` est la plus utilisée car:

- Plus facile à tester
- Plus facile à maintenir
- Plus facile à réutiliser

Injection de dépendances

- Plus facile à tester car on peut injecter des `mocks` dans les tests.

Injection de dépendances

- Plus facile à maintenir car on peut changer les dépendances sans changer le code source.

Injection de dépendances

- Plus facile à réutiliser car on peut réutiliser les classes avec n'importe quelles dépendances.

Inversion de contrôle

Définition

> L'`inversion de contrôle` (inversion of control, IoC) est un patron d'architecture commun à tous les `frameworks` (ou cadre de développement et d'exécution).

> Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

Exemple(Sans inversion de contrôle): Un virement

```
public void virement(int c1, int c2, double montants) {
    // Création d'une transaction
    EntityTransaction transaction = entityManager.getTransaction();

    // Démarrer la transaction
    transaction.begin();
    try{
        retirer(c1, montant);
        ajouter(c2, montant);

        // Valider la transaction
        transaction.comit();
    } catch(Exception e){
        // Annuler la transaction en cas d'exception
        transaction.rollback();
    }
}
```

Mauvaise pratique:

- Modification du code source
- Changement du code lourd pour de futures implémentations

Avec l'inversion de contrôle:

```
@Transactional // Annotation pour délégué la transaction à spring
public void virement(int c1, int c2, double montants) {
    retirer(c1, montant);
    ajouter(c2, montant);
}
```

Permet:

- De ne plus modifier le code source
- D'avoir un code plus léger et souple Note: Un deuxième exemple est de parler de la sécurité et l'annotation @Secured(role="admin")

Les annotations Java

Qu'est ce qu'une `@annotation`

Une annotation est:

- Un marqueur sur un élément de code.
- N'execute pas de code.
- Permet de retrouver des éléments
- Permet une abstraction

Equivalence

Les annotations en java sont comparables aux `décorateurs` en python ou typescript.

Mais elles n'ont pas réellement le même fonctionnement.

Créer son annotation

Créer une annotation est très simple. Il suffit de créer une interface avec le mot clé `@interface`:

```
public @interface MonAnnotation {
}
```

Utilisation de l'annotation

En java il est possible d'utiliser `les @annotations` sur:

- Une classe ou une interface
- Une méthode ou un controller
- Un paramètre
- Un attribut.

Sur une classe

```

@MonAnnotation
class MaClass {
}
```

Sur une méthode

```
class MaClass {
    @MonAnnotation
    public void maMethode() {
    }
}
```

Sur un paramètre

```
class MaClass {
    public void maMethode(@MonAnnotation String param) {
    }
}
```

Sur un attribut

```
class MaClass {
    @MonAnnotation
    private String attribut;
}
```

[Avancé] récupération

Il est possible de savoir si une annotation est présente sur un élément de code.

```
public class MaClass {
    @MonAnnotation
    private String attribut;

    public static void main(String[] args) {
        if (MaClass.class.isAnnotationPresent(MonAnnotation.class)) {
            System.out.println("L'annotation est présente");
        }
    }
}
```

[Avancé] introspection

Avec la librairie `Reflections` il est possible de retrouver l'ensemble des éléments annotés.

```
public class MaClass {
    @MonAnnotation
    private String attribut;

    public static void main(String[] args) {
        Reflections reflections = new Reflections("com.example");
        Set<Class<?>> annotated =
reflections.getTypesAnnotatedWith(MonAnnotation.class);
    }
}
```

Annotation et Spring

Depuis la version 5 de Spring, nous utilisons les annotations pour le développement de nos applications.

Annotation et Spring

Le but sera donc de bien comprendre ce que font les annotations pour pouvoir les utiliser correctement.

Et gagner en productivité.