

ARCHITECTURE

BACKEND



COMMENT EST STRUCTURÉ UN PROJET WEB?

ARCHITECTURE MONOLITHIQUE



ARCHITECTURE MONOLITHIQUE

Tout ce fait dans une seule application.

- l'affichage des pages
- le traitement des données
- la gestion de la persistance

SPRING BOOT ET L'ARCHITECTURE MONOLITHIQUE

Spring boot permet de faire des architectures Monolithique.

SPRING BOOT ET L'ARCHITECTURE MONOLITHIQUE

Spring boot permet de faire des architectures Monolithique.

Mais...

5.1

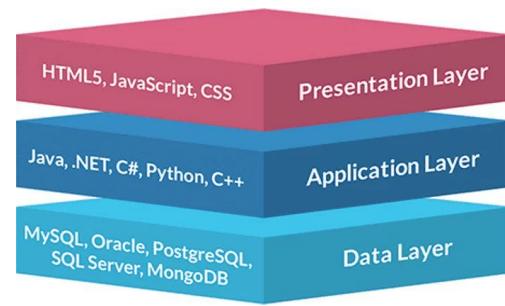
SPRING BOOT ET L'ARCHITECTURE MONOLITHIQUE

Spring boot permet de faire des architectures Monolithique.

Mais...

Ce n'est plus ce que nous faisons

ARCHITECTURE 3 TIERS



ARCHITECTURE 3 TIERS

Nous utilisons trois applications au lieu d'une:

- **le frontend**: l'affichage des pages exécuté dans le navigateur
- **le backend**: le traitement des données exécuté sur le serveur
- **la base de données**: la gestion de la persistance

LE FRONTEND

Java n'est pas le meilleur langage pour faire des applications web.

Il est plus adapté d'utiliser JavaScript pour faire des applications web.

LA BASE DE DONNÉES

Il existe plusieurs types de bases de données:

- SQL : MySQL, PostgreSQL, Oracle, SQL Server, ...
- NoSQL : MongoDB, Cassandra, Redis, ...

LE BACKEND

C'est ici que nous allons nous concentrer.

`Spring boot` est un framework adapté pour la création d'application backend.

LE BACKEND

Le backend est le cœur de l'application.



LE RÔLE DU BACKEND

Le backend est responsable de:

- la gestion des données
- la gestion du métier

LE BACKEND ET LE FRONTEND

Le frontend envoie des requêtes au backend grâce à des **API**.

Une **API** est une interface de programmation qui permet de communiquer avec un autre programme.

COMMUNICATION ENTRE BACKEND ET FRONTEND

Pour communiquer entre le Front et le Back nous allons avoir besoin
d'utiliser des protocole

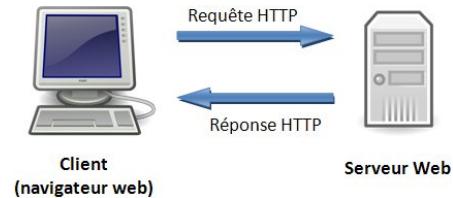
Un protocole est un ensemble de règles et de codes de langage qui
définissent comment se déroule la communication

HTTP

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.

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"nom": "Dupont", "prenom": "Jean"}
```

HOST

Le **Host** est l'adresse du serveur.

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"nom": "Dupont", "prenom": "Jean"}  
7
```

URI + HOST = URL

Une URL est une Uniform Resource Locator.

Une URL est une URI qui contient le Host.

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"nom": "Dupont", "prenom": "Jean"}
```

MÉTHODE

La méthode est la requête que le client veut effectuer.

Ici **POST**:

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"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:

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"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"}:

```
1 POST /api/clients HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4 Content-Length: 41
5
6 {"nom": "Dupont", "prenom": "Jean"}
```

LES RÉPONSES

- 1.
- 2.
- 3.

LES RÉPONSES

1. Le client envoie une **requête** au serveur.
- 2.
- 3.

27.1

LES RÉPONSES

1. Le client envoie une **requête** au serveur.
2. Le serveur traite la **requête**.
- 3.

27.2

LES RÉPONSES

1. Le client envoie une **requête** au serveur.
2. Le serveur traite la **requête**.
3. Le serveur envoie une **réponse** au client.

27.3

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**:

```
1 HTTP/1.1 200 OK
2 Content-Type: application/json
3 Content-Length: 41
4
5 {"nom": "Dupont", "prenom": "Jean"}
```

HTTPCATS



200
OK

HTTPDOG

[httpdog](http://httpdog.com)

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.

EXAMPLE:

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

EXAMPLE:

```
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**.

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`.

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
```

45.1

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`.

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.

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

47.1

CRÉATION D'UN CLIENT

Pour créer un client, on ajoute le nom de la ressource au pluriel à l'URL 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:

MODIFICATION D'UN CLIENT

Il existe deux méthodes pour modifier un client:

- **PUT**: on modifie l'ensemble des champs de la ressource.

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**.

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"}
```

50.1

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.

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**.

SUPPRESSION D'UN CLIENT

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

DELETE /clients/1

QUIZ!

QUESTION 1

Je veux récupérer l'ensemble des voiture:

- GET /voiture
- POST /voitures
- GET /voitures
- POST /voiture

QUESTION 2

Je veux récupérer la voiture ayant l'identifiant 1:

- GET /voitures/1
- GET /voiture/1
- POST /voitures/1
- POST /voiture/1

QUESTION 3

Je veux récupérer les voitures ayant le nom "Clio":

- POST /voiture?nom=Clio
- GET /voitures?nom=Clio
- GET /voiture?nom=Clio
- POST /voitures?nom=Clio

QUESTION 4

Je veux récupérer la marque de la voiture ayant l'identifiant 1:

- GET /voitures/1/marque
- GET /voiture/1/marque
- GET /voiture/marque/1
- GET /voitures/1/marque
- GET /voitures/marques

QUESTION 5

Je veux récupérer l'ensemble des marques de voitures:

- GET /voiture/marques
- GET /voiture/marque
- GET /voitures/marque
- GET /voitures/marques

QUESTION 6

Je veux récupérer le siège 4 de la voiture 2 du client 1:

- GET /clients/1/voiture/2/sieges/4
- GET /clients/1/voiture/2/siege/4
- GET /clients/1/voitures/2/sieges/4
- GET /clients/1/voitures/2/siege/4

QUESTION 7

Quelle est la méthode HTTP pour créer un client?

- POST
- PUT
- PATCH
- DELETE

QUESTION 8

Je veux modifier le solde d'un client:

- POST /clients/1/solde
- PUT /clients/1/solde
- PATCH /clients/1/solde
- DELETE /clients/1/solde

QUESTION 9

Je veux supprimer la voiture 2 du client 1:

- POST /clients/1/voiture/2
- PUT /clients/1/voiture/2
- PATCH /clients/1/voiture/2
- DELETE /clients/1/voiture/2

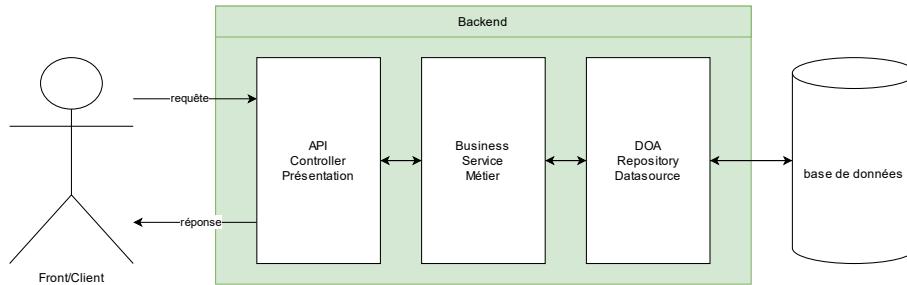
QUESTION 10

Je veux modifier le moteur de l'ensemble des voitures du client 1:

- POST /clients/1/voitures/moteur
- PUT /clients/1/voitures/moteur
- PATCH /clients/1/voitures/moteur
- DELETE /clients/1/voitures/moteur

L'ARCHITECTURE 3 COUCHES

L'architecture 3 couches est une architecture logicielle qui sépare les données, la logique métier et l'interface utilisateur.



LES COUCHES

- **Contrôleur:** c'est la couche qui reçoit les requêtes HTTP et qui les transmet à la couche métier.
- **Service:** c'est la couche qui contient la logique métier.
- **Repository:** c'est la couche qui contient la logique d'accès aux données.

LES CONTRÔLEURS

Les contrôleurs sont des classes qui reçoivent les requêtes HTTP et qui les transmettent à la couche métier.

Ils se chargent de retourner la réponse HTTP.

Ils ne font pas de connexions à la base de données ou de logique métier.

LES SERVICES

Les services sont des classes qui contiennent la logique métier.

Ils ne font pas de connexions à la base.

LES REPOSITORIES

Les repositories sont des classes qui contiennent la logique d'accès aux données.

Ils ne font pas de logique métier.

ANALOGIE DU RESTAURANT

L'on peut voir l'architecture 3 couches comme un restaurant.



LE SERVEUR

Le serveur est là pour prendre vos commandes et les transmettre à la cuisine.

LE SERVEUR

Le serveur est là pour prendre vos commandes et les transmettre à la cuisine.

Il est aussi là pour vous servir votre plat.

70.1

LE SERVEUR

Le serveur est là pour prendre vos commandes et les transmettre à la cuisine.

Il est aussi là pour vous servir votre plat.

Mais il n'est pas là pour cuisiner ou gérer les stocks.

70.2

LA CUISINE

La cuisine est là pour cuisiner les plats.

LA CUISINE

La cuisine est là pour cuisiner les plats.

Elle ne s'occupe pas de prendre les commandes ou de servir les plats.

LA CUISINE

La cuisine est là pour cuisiner les plats.

Elle ne s'occupe pas de prendre les commandes ou de servir les plats.

Elle ne s'occupe pas non plus de gérer les stocks.

71.2

LE STOCK

Le stock est là pour gérer les ressources et les entreposer.

LE STOCK

Le stock est là pour gérer les ressources et les entreposer.

Le stock ne s'occupe pas de prendre les commandes ou de servir les plats.

LE STOCK

Le stock est là pour gérer les ressources et les entreposer.

Le stock ne s'occupe pas de prendre les commandes ou de servir les plats.

Le stock ne s'occupe pas non plus de cuisiner.

CHAQU'UN À SA PLACE



SUITE

Mon premier backend