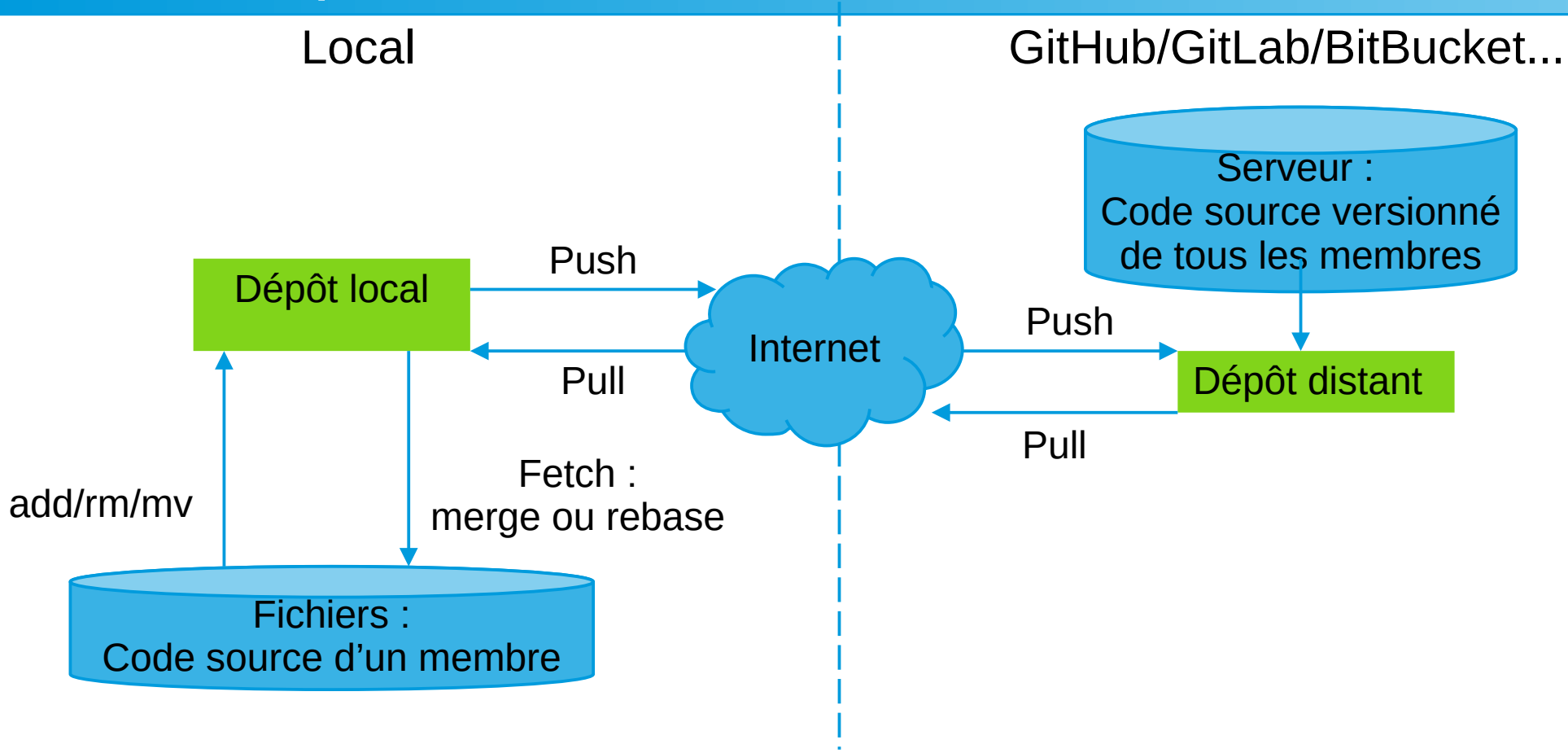
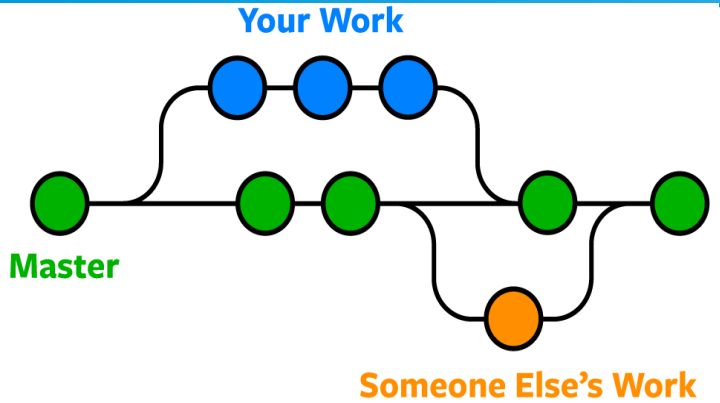
Principes de Git : Architecture décentralisée



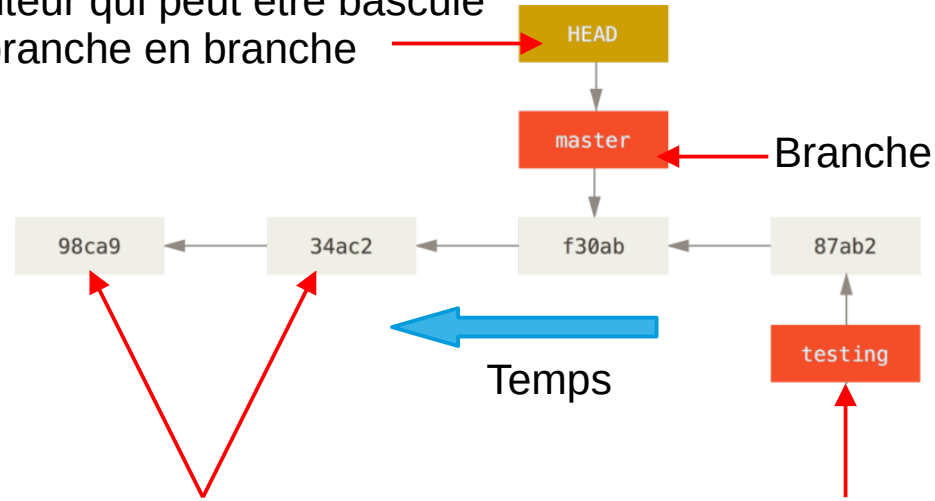
Principes de Git : Les branches



Cela sert à :

- **Travailler à plusieurs** chacun de son côté
- Développer de nouvelles **features**
- Isoler et corriger les **bugs**
- Faire des **tests**
- Maintenir **plusieurs versions** différentes du même projet

Pointeur qui peut être basculé de branche en branche

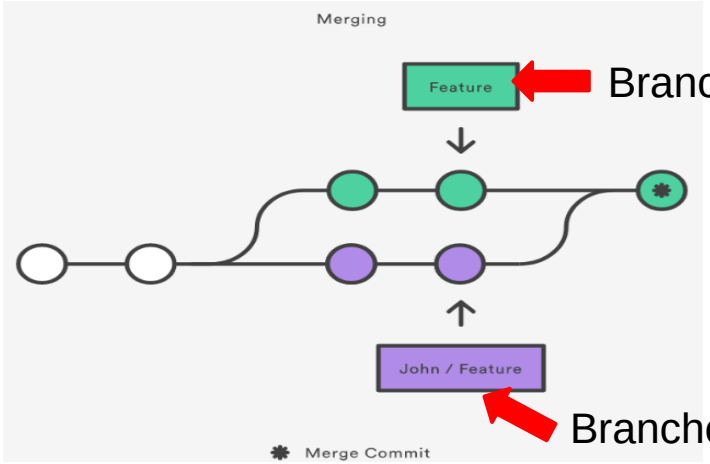


Les versions des fichiers sont indexées avec des Hash

Branche qui diverge vers une version du projet

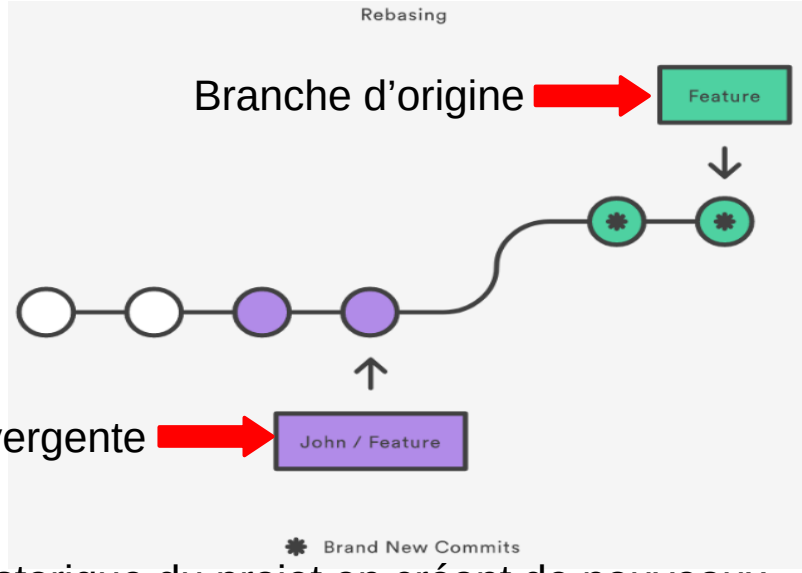
Principes de Git : Merge VS Rebase

Permet de transférer des changements d'une branche à une autre et de résoudre les conflits de version.



Merge :

- 1 nouveau commit dans la branche fusionnée
- Simple et non risqué : pas d'altération de branche
- Pollue l'historique : projet difficile à suivre avec des divergences superflues dans tous les sens



Rebase :

- Réécrire l'historique du projet en créant de nouveaux commits pour chaque commit de la branche d'origine
- Historique linéaire et plus simple à suivre
- Risqué : permet de modifier des commit manuellement + perte d'informations possible

GitHub Actions : Mettre en place un système de CI pour Java

Exécuter automatiquement les tests unitaires à chaque push sur GitHub.

Procédure automatique :
- Téléchargement de JDK 19
- Build du projet avec Maven
- Exécution des tests avec Maven

Dans le répertoire du projet :
- Créer le répertoire : .github/workflows/
- Y ajouter un fichier .yml avec le contenu suivant

```yaml
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        java: [ 19 ]
    name: Java ${{ matrix.java }}
    steps:
      - uses: actions/checkout@v3
      - name: Setup Java JDK
        uses: actions/setup-java@v3.5.1
        with:
          distribution: 'temurin'
          java-version: ${{ matrix.java }}
      - name: Build and Execute tests with Maven.
        run: |
          mvn -B package
          mvn -B test --file pom.xml
```

Écrire des objets en JSON

Parser des types Objets qui ne sont pas des String

Utiliser la **convention Java Bean** ⇒ un moyen de récupérer les propriétés d'une classe qui possède **un getter et un setter**.

Pour les classes : Il existe une Interface BeanInfo avec une méthode Introspector.getBeanInfo(beanType) qui créé un BeanInfo à partir d'une classe : pour que ça marche il faut que le nom du getter soit préfixé par « get » ou « is » .

```
public class Person {
  String getName() { ... }
  void setName(String name) { ... }
}
```

On a besoin de récupérer le nom et l'accesseur/getter d'une classe/record. Grace à 2 méthodes, Il est possible de récupérer un array contenant cela.

⇒ **Classe : BeanInfo.getPropertyDescriptors()**

⇒ **Record : Class.getRecordComponents()**

Pour chaque élément de l'array retourné, on récupère :

- Le nom de la classe/record via **element.getName()**
- Le getter via **element.getReadMethod()** **OU** l'accesseur via **element.getAccessor()**
  - ⇒ Ces 2 méthodes renvoient une instance de **Method** que l'on peut appeler avec **method.invoke(instance, arguments)**.

Écrire des objets en JSON

Mettre les informations des classes/records en cache pour éviter de les recalculer à chaque fois

Utiliser un cache de type **ClassValue** pour y stocker les informations de chaque classe/record.

Principe : La méthode ClassValue.get(type) récupère la valeur associée à type si elle est déjà dans le cache sinon elle appelle la méthode computeValue(type) pour la calculer et la mettre en cache.

On doit donc override **computeValue** pour récupérer et parser les propriétés d'une classe/record en prenant en compte 2 cas de figure :

- **Soit la propriété est de type « primitif** » ⇒ il faut la parser en JSON.
  Exemple :

```
public class Person {
  String getName() { ... }
```

```
System.out.println(jsonWriter.toJSON(new Person("toto")));
```

→ {"name": "toto"}

- **Soit la propriété est un Bean ou un record** ⇒ il faut appeler récursivement **toJSON** avec en paramètres : la propriété renvoyée par le getter du Bean ou par l'accesseur du record.
  Exemple :

```
record Address(String street) { }
record Person(String name, Address address) { }
```

```
new Person("Bob", new Address("21 Jump Street"));
```

→ {"name": "Bob", "address": {"street": "21 Jump Street"}}

- On utilise une Interface fonctionnelle qui prend un **JSONWriter** et un bean ou un record et qui renvoie un String correspondant à la notation en JSON de la propriété du bean ou du record.

Écrire des objets en JSON
Gestion de la configuration et d'une annotation

Permettre à l'utilisateur de définir une représentation en String d'un type donné.
Exemple : définir un format de DateTime.

Une configuration = Function<Object, String> que l'on va appeler dans la méthode qui parse les objets si l'objet à parser est du même type que celui configuré (dans notre exemple : LocalDateTime.class).

- On associe chaque configuration au type de l'objet qu'elle représente : HashMap<Class<?>, Function<Object, String>>.
- Pour garantir que le type de la clé soit le même que celui du paramètre de la Function, on utilise un type paramétré. On définit donc une méthode **configure** permettant d'ajouter une configuration dans la Map. Prototype de **configure** :

public <T> void configure(Class<T> cls, Function<T, String> functionToApply)

Pour ajouter la configuration donnée dans la HashMap :
map.putIfAbsent(cls, functionToApply.compose(o -> cls.cast(o))); ⇒ D'abord on cast l'objet donné (o) dans le même type que la clé (T) pour pouvoir appeler la Function avec cet objet.

Avec notre exemple, on obtient :

```
writer.configure(LocalDateTime.class, time -> time.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));
writer.toJSON(LocalDateTime.of(2021, 6, 16, 20, 53, 17)));
```

→ **"2021-06-16T20:53:17"**

L'utilisateur peut mettre une annotation sur un getter/accesseur pour changer le nom d'une propriété JSON.
Exemple : @JSONProperty("first-name")

Si le getter du bean/record à parser a une annotation, on récupère sa valeur et on remplace le nom de la propriété JSON par cette valeur.

⇒ **Récupérer l'annotation du getter : getter.getAnnotation(JSONProperty.class)**
⇒ **Récupérer la valeur de l'annotation : annotation.value()**