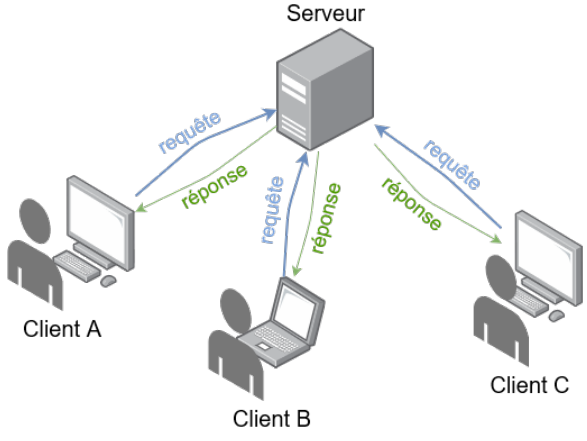
PROJET DE DÉVELOPPEMENT LOGICIEL

CLIENT - SERVEUR

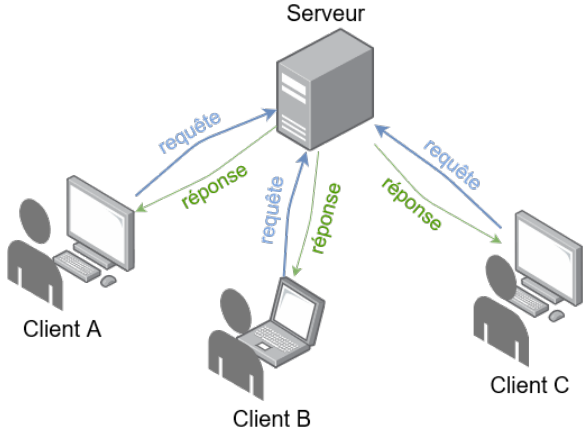
LE MODÈLE CLIENT - SERVEUR



Échange d'un **service**

- Le **client** demande l'exécution d'un service
- Le **serveur** réalise le service
- Client et serveur souvent localisés sur **deux machines distinctes**
  - mais ce n'est pas obligatoire
  - le client peut lui-même être un autre serveur

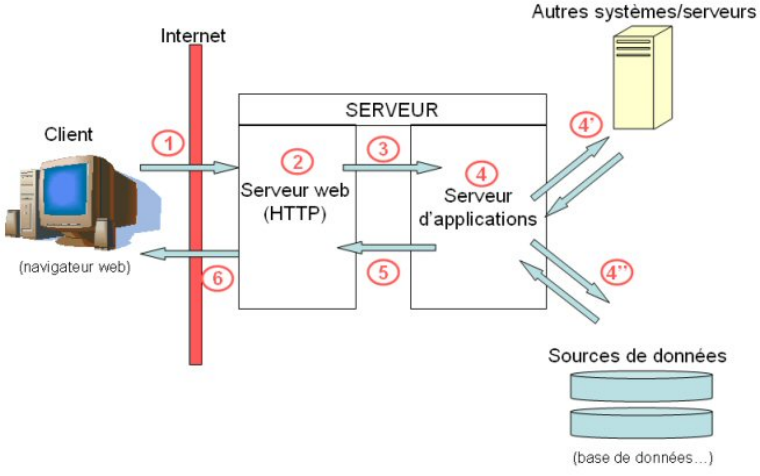
LE MODÈLE CLIENT - SERVEUR



Communication par **messages**

- **Requête** : paramètres d'appel + spécification du service requis
- **Réponse** : résultats + indicateur d'exécution ou d'erreur
- Communication **synchrone** ou **asynchrone**
  - sur machines distinctes, protocole de communication réseaux

APPLICATION WEB



Application dont l'interface est visible dans un **navigateur web**

- **nécessairement** des programmes côté serveur
- parfois une partie côté client
- communication via le **protocole HTTP**

DIFFÉRENTS TYPES D'APPLICATIONS WEB

Site Web dynamique

- Ensemble de **pages dynamiques simples**
  - éventuellement inter-reliées
  - mais pas d'inclusion

- Exemples
  - Traitement des données d'un formulaire
  - Composition pour l'affichage de données du serveur

DIFFÉRENTS TYPES D'APPLICATIONS WEB

Application Web « localisée »

- Toute la programmation est **sur le serveur**
- Modèle(s) de structuration de l'application (couches, modules, composants)

- **Principes de structuration** :
  - affectation de responsabilités à chaque ressource
  - inclusion/appel des ressources les unes dans les autres
  - utilisation de patterns de conception (MVC, DAO...)

DIFFÉRENTS TYPES D'APPLICATIONS WEB

Application Web répartie

- Application localisée + **déport d'une partie côté client**
- Appel à d'autres ressources / composants
  - sur des machines distantes
  - dont on n'est pas nécessairement propriétaire
- Requière des **mécanismes de communication** (*middleware*)
  - Exemples : RPC, CORBA, Services Web, REST...

EXEMPLES DE LANGAGES UTILISÉS CÔTÉ SERVEUR

- Php
- Java
- Python
- Javascript
- .NET (C#, VB)
- Ruby
- Perl
- Go
- Rust
- ...

LE PROTOCOLE HTTP

LE PROTOCOLE HTTP

- **HTTP** : **H**yper **T**ext **T**ransfer **P**rotocol
- Dédié au Web (origine : CERN, 1990)
- Fonctionne en mode client / serveur
- Protocole **sans état**
  - Gestion légère des transactions
    - aucune information conservée (sur le serveur) entre 2 connexions
    - permet au serveur HTTP de servir plus de clients
  - Nécessite un **mécanisme de gestion des sessions**
    - cookie, Id dans l'URL, champ caché de formulaire...

FORMAT DES REQUÊTES

- **Commande**
  - `GET, POST, HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT`
  - Ressource = URL à partir de la racine du serveur
  - version HTTP (0.9 → 2.0)
- **En-têtes** (*header*)
  - nom de l'en-tête `:` valeur de l'en-tête
- **Une ligne vide**
- **Contenu** / **corps** (*body*)
  - passage de paramètres à traiter par le serveur

LA MÉTHODE GET

- Méthode standard de **requête** d'un document
- Corps de la requête toujours vide
- Ajout de **paramètres** après le nom de la ressource

```
GET /cgi-bin/prog.cgi?email=toto@site.fr&pass=toto&s=login HTTP/1.1
```

- Remarques :
  - Données transmises en **clair et visibles dans l'URL**
  - Taille limitée de l'URL (4Ko)

LA MÉTHODE POST

- Transmission des données **dans le corps** de la requête

```
POST /cgi-bin/prog.cgi HTTP/1.1
User-Agent: Mozilla/5.0 (compatible;MSIE 6.0;Windows NT 5.1) Host: localhost
Accept: */*
Content-type: application/x-www-form-urlencoded Content-length: 36

email=toto@site.fr&pass=toto&s=login
```

- Données également transmises **en clair**

FORMAT DES RÉPONSES

- **Type de la réponse**
  - version HTTP (0.9 → 2.0)
  - code de la réponse
  - description du code
- **En-têtes** (*header*)
  - nom de l'en-tête : valeur de l'en-tête
- **Une ligne vide**
- **Contenu** / **corps** (*body*)
  - Ressource encodée en fonction du **type MIME** spécifié

CODES DE RÉPONSE

- Résultat de la requête : **succès ou échec**
- En cas d'échec, la réponse décrit la raison
- **Classes de codes**
  - 100-199 : information
  - 200-299 : succès
  - 300-399 : redirection
  - 400-499 : échec dû au client
  - 500-599 : échec dû au serveur

Plus d'infos : https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

UNE TRANSACTION TYPIQUE (1)

Requête : client → serveur

1. demande du document `test.html`

```
GET /test.html HTTP/1.1
```

2. envoi des informations d'en-tête :
   - configuration du client
   - documents acceptés

```
User-Agent: Mozilla/5.0 (compatible;MSIE 6.0;Windows NT 5.1)
Accept: image/gif, image/jpeg
```

3. envoi d'une ligne vide (fin de l'en-tête)
4. envoi du contenu (vide dans cet exemple)

UNE TRANSACTION TYPIQUE (2)

Réponse : serveur → client

1. code indiquant l'état de la requête

```
HTTP/1.1 200 OK
```

2. envoi des informations d'en-tête :
   - configuration du serveur
   - document demandé

```
Date: Tue, 30 Sep 2008 06:11:28 GMT
Server: Apache/1.3.34 (Debian) PHP/5.2.1
Content-Length: 97
Content-Type: text/html; charset=iso-8859-1
```

3. envoi d'une ligne vide (fin de l'en-tête)
4. envoi du contenu (si la requête a réussi)

SÉMANTIQUE DES MÉTHODES HTTP

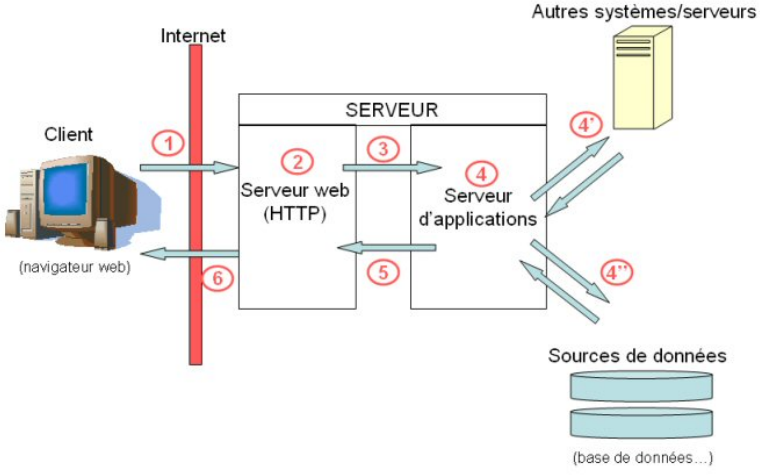
Selon les spécifications IETF :

	Safe	Idempotent	Cacheable
GET	✔	✔	✔
POST	✘	✘	O
PUT	✘	✔	✘
DELETE	✘	✔	✘

- **Safe** = sans effet de bord
- **Idempotent** = même résultat si appliquée plusieurs fois

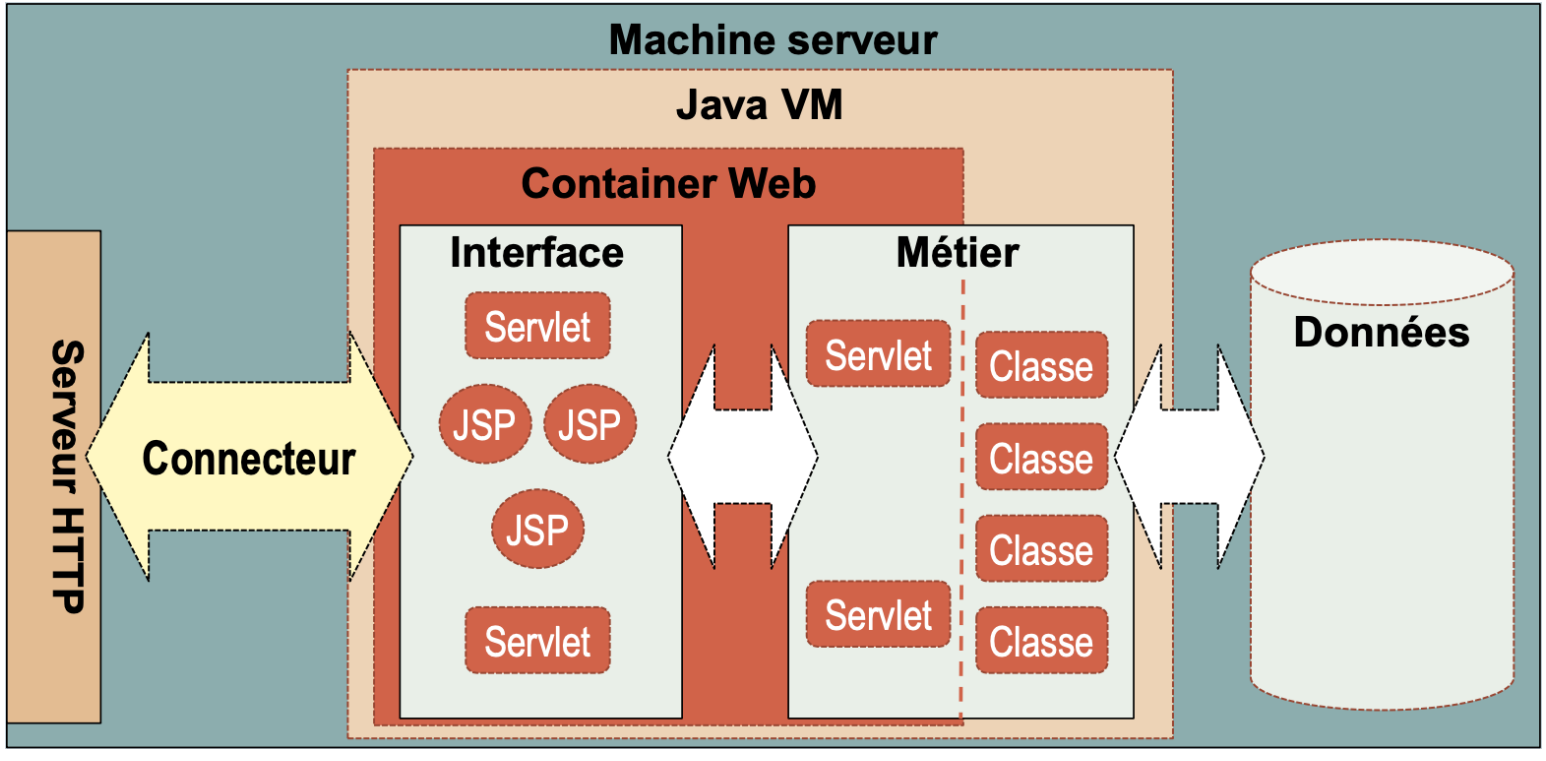
PROGRAMMATION CÔTÉ SERVEUR EN JAVA

PRINCIPE GÉNÉRAL



1. Le **Client** envoie la requête HTTP
2. Le **Serveur web** reçoit la requête HTTP
3. Le **Moteur de servlets** encapsule la requête dans un objet Java
4. Traitement de la requête et génération de la réponse dans un objet Java → **Composants Java**
5. Le **Moteur de servlets** désencapsule la réponse
6. Le **Serveur web** envoie la réponse HTTP au client

PRINCIPE GÉNÉRAL



Objets d'encapsulation :

`HTTPServletRequest` / `HTTPServletResponse`

SERVEURS DE RÉFÉRENCE

Implémentent la **Java Servlet API**

- **Tomcat** (Apache)
  - Référence en matière de **moteurs de servlets**
  - Contenu
    - Serveur web : Apache
    - Connecteur : `mod_jk` (*Jakarta*)
    - Moteur de servlets : *Catalina*

- **Jetty** (Eclipse Foundation)
  - Serveur web + conteneur de servlets « léger »

SERVLETS : DÉFINITION

Implémentation Java d'un mécanisme de requête/réponse

- Initialement : **indépendant d'un protocole**
- Avec **encapsulation des données dans des objets**
  - génériques
  - spécifiques à HTTP
    - méthode
    - type MIME de la réponse
    - headers
    - session
    - cookies

SERVLETS : CONCRÈTEMENT

- **Objet (classe) Java**
  - composant d'application
  - derrière un serveur (Web, mais pas seulement)
  - associé à une URL sur le serveur

- Dans un « **Container** »
  - pas d'accès direct au serveur
  - accès protégé aux autres objets métier de l'application
  - gestion avancée par le container

SERVLETS : EXEMPLE DE CODE

```
import javax.servlet.*;
import javax.servlet.http.*;

public class NewServlet extends HttpServlet {
  public void init(ServletConfig config) throws ServletException {
    super.init(config); ... }

  public void destroy() { ... }

  protected void doGet(HttpServletRequest request,
         HttpServletResponse response)
   throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><head><title>Hello page</title></head>");
    out.println("<body><h1>Hello "+ request.getParameter("name") +
   "</h1></body></html>");
  }
}
```

(HTTPServlet, Servlet API V2)

SERVLETS : BILAN

- **Avantages**
  - Composants simples : classes Java
  - Codage minimum (cycle de vie, traitement de la requête)
    ⇒ autres aspects assurés par le conteneur
  - **Sûrs**
    - isolation du serveur par le conteneur
    - « rigueur » de l'orienté-objet

- **Inconvénients**
  - Nombreux `out.println()`
  - Code HTML généré difficile à lire

JAKARTA SERVER PAGES (JSP)

- **Principe**
  - Page web **dynamique** codée comme si elle était statique
  - Ne mettre du code que lorsque c'est nécessaire
    - code « HTML-like » compilé en code Java
    - programmation impérative
  - Équivalent au PHP ou ASP pour Java

- **Mêmes fonctionnalités qu'un** `HttpServlet`
  - Accéder aux mêmes données / objets qu'une servlet
  - Inclure ou rediriger la requête vers une autre servlet/JSP
  - JSP compilée en servlet à la première utilisation

JAVABEANS

- **Définition** (1996)
  - Composants logiciels réutilisables
  - En pratique, une classe Java **standardisée**
- **Structure**
  - Constructeur sans paramètre
  - Attributs privées accessibles par des méthodes publiques
    `get, set, is, to` ⇒ permet l'**introspection**
  - Sérialisable ⇒ permet la **persistance**
- **Intérêt**
  - Permet l'assemblage de composants
  - Communication via des **évènements**
  - En Web, accessible depuis une JSP

PATTERN WEB MVC

- **Principe**
  - Pattern MVC côté serveur
  - Implémentation en Java : servlets / JSP / JavaBeans

- **Modèle**
  - Contient le domaine de l'application
  - Peut utiliser d'autres patterns
  - Implémenté sous forme de classes / interfaces simples (Beans, POJOs)

PATTERN WEB MVC

- **Contrôleur**
  - Gère les paramètres des requêtes
  - Lie modèle et vue
  - Implémenté par une (ou plusieurs) servlet(s)

- **Vue**
  - Encapsule la création des pages Web de réponses
  - Données préparées par le contrôleur ou récupérées du modèle
  - Implémentée par des JSP

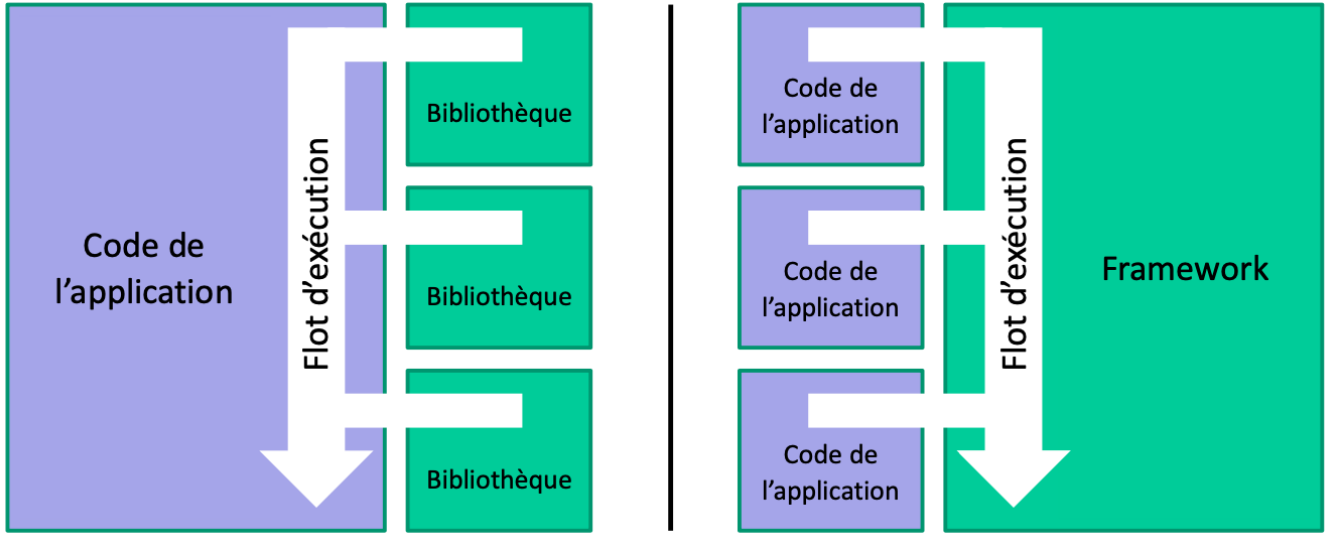
OUTILS D'AIDE AU DÉVELOPPEMENT

La **réutilisation** comme principe général de conception

- Même démarche qu'en conception « classique »
  - Ne plus développer *from scratch*
  - Gagner du temps
  - Se placer dans des conditions réelles de conception

- **Spécificité des outils Web**
  - Nombreux
  - Hétérogènes
  - Notion de *framework* (vs. bibliothèque) plus répandue

DIFFÉRENCE BIBLIOTHÈQUE / FRAMEWORK



- **Qui contrôle le flot d'exécution de l'application ?**
  - votre code
  - un des outils que vous utilisez ⇒ **inversion de contrôle (IoC)**
- En programmation classique : d'où provient le `main` ?
- En MVC : qui dirige le contrôleur ?

FRAMEWORKS CÔTÉ SERVEUR

- Serveur Web déjà un *framework* en soi
- *Framework* Web = **couche d'abstraction** supplémentaire
  - doit apporter une valeur ajoutée :
    - pattern MVC
    - services annexes (sécurité, interface avec les BDD...)
  - doit être nécessaire pour la réalisation du cahier des charges
  - Exemple de **fonctionnalités**
    - routage des requêtes
    - négociation de contenus (e.g., formats d'image)
    - renvoi automatique d'erreurs HTTP

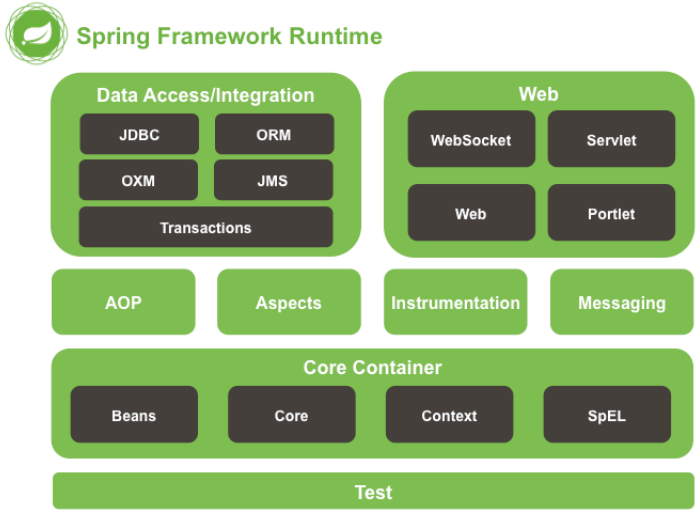
DIFFÉRENTS TYPES DE FRAMEWORKS

- **Push-based**
  - Contrôleur utilisant des *beans* pour calculer les contenus
  - Ces contenus sont « poussés » à la Vue
  - Exemples : Struts, Spring

- **Pull-based** (ou *component-based*)
  - Vue « tire » les contenus de plusieurs Contrôleurs dédiés à des tâches spécifiques
  - Plusieurs Contrôleurs participe à la création d'une seule Vue
  - Exemples : Struts2, Tapestry, JBoss Seam

https://en.wikipedia.org/wiki/Comparison_of_web_frameworks

LE FRAMEWORK SPRING



- Architecture autour d'un « **conteneur léger** »
  - les composants sont des POJOs
- Intégration de fonctionnalités d'autres projets Open Source
  - Struts, Hibernate, JUnit, AspectJ, JSF...

https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html

SPRING CORE CONTAINER

- Implémente le **pattern IoC**
  - création, cycle de vie et dépendances des composants
  - sorte de fabrique évoluée
- Applique la **configuration** de l'application
  - via des fichiers XML
  - par **annotations**
  - par programmation
- Fournit un **contexte applicatif**
- Fournit des **services annexes**

COMPOSANTS

- *Spring Beans* = POJOs **instanciés** par le conteneur
- Avec **injection de dépendances**
  - par constructeur :

```
public class UserService {
    private UserDao userDao;
    public UserService(UserDao userDao) {
        this.userDao = userDao;
    }
}
```

  - par propriétés (*setters*) :

```
public class UserService {
    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
      this.userDao = userDao;
    }
}
```

CONFIGURATION PAR ANNOTATIONS

Annotations de classes : stéréotypes

- `@Component` : composant générique
- `@Repository` : dérive de `@Component`, dédié à la **persistance**
- `@Service` : dérive de `@Component`, dédié aux **objets métiers du modèle**
- `@Controller` : dérive de `@Component` , dédié au **contrôleurs Spring MVC**

```
import org.springframework.stereotype.Repository;

@Repository
public class UserDao implements Dao {
}
```

https://www.jmdoudoux.fr/java/dej/chap-spring_core.htm

CONFIGURATION PAR ANNOTATIONS

`@Autowired` sur un **attribut**, un *setter* ou un constructeur :

```
@Service
public class UserService {
  @Autowired
  private UserDao userDao;
}
```

- Injection automatique de dépendances **basée sur le type**
- Ne s'applique qu'aux *Spring beans*

CONFIGURATION PAR ANNOTATIONS

`@Autowired` sur un attribut, un ***setter*** ou un constructeur :

```
@Service
public class UserService {
  private UserDao userDao;

  @Autowired
  public void setUserDao(UserDao userDao) {
    this.userDao = userDao;
  }
}
```

- Injection automatique de dépendances **basée sur le type**
- Ne s'applique qu'aux *Spring beans*

CONFIGURATION PAR ANNOTATIONS

`@Autowired` sur un attribut, un *setter* ou un **constructeur** :

```
@Service
public class UserService {
  private UserDao userDao;

  @Autowired
  public UserService(UserDao userDao) {
    this.userDao = userDao;
  }
}
```

- Injection automatique de dépendances **basée sur le type**
- Ne s'applique qu'aux *Spring beans*

CONFIGURATION PAR ANNOTATIONS

`@Autowired` sur un attribut, un *setter* ou un constructeur.

- Possibilité de préciser si l'injection est **obligatoire** ou non :
  `@Autowired(required=false)`
- `BeanCreationException` levée si :
  - aucun *bean* de ce type est trouvé (si `required=true`)
  - plusieurs *beans* de ce type sont trouvés et l'élément annoté n'est pas un tableau ou une collection

CONFIGURATION PAR ANNOTATIONS

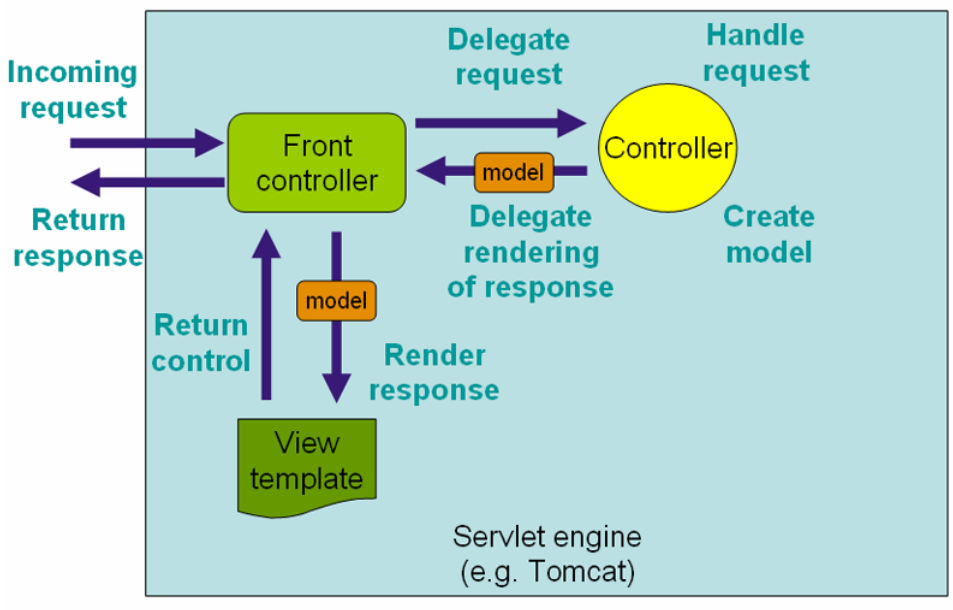
Portée (*scope*) : annotation `@Scope`

- *singleton* (par défaut) : une seule instance créée par le conteneur
- *prototype* : une nouvelle instance à chaque demande du conteneur
- *request, session, application, websocket* : spécifiques au conteneur web
- *user-defined*

```
@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

https://www.baeldung.com/spring-bean-scopes

SPRING WEB MVC



- **Front controller** : `DispatcherServlet` (fournie par Spring)
- **Contrôleurs délégués** : composants (`@Controller`)

https://docs.spring.io/spring-framework/reference/web/webmvc.html

COMPOSANTS MVC

Contrôleur annoté

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {
  private final AppointmentBook appointmentBook;

  @Autowired
  public AppointmentsController(AppointmentBook appointmentBook) {
    this.appointmentBook = appointmentBook;
  }

  @GetMapping
  public String get() {
    return "appointments/today";
  }
}
```

COMPOSANTS MVC

Méthodes de service (*Handler methods*)

- Annotées avec `@RequestMapping` (ou `@GetMapping`, `@PostMapping`...)
- Permettent :
  - de récupérer les paramètres de la requête
  - de faire du *data binding* entre les paramètres et le modèle
  - d'appeler les *beans* concernés
  - de passer les infos (du modèle) nécessaires à la vue pour générer la réponse

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-methods

COMPOSANTS MVC

Méthodes de service (*Handler methods*) : signature « flexible »

- **Arguments**
  - Objet du model : `@ModelAttribute`

```
@PostMapping("/owners/{ownerId}/pets/{petId}/edit")
public String processSubmit(@ModelAttribute Pet pet) {
    // method logic...
}
```

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-methods

COMPOSANTS MVC

Méthodes de service (*Handler methods*) : signature « flexible »

- **Arguments**
  - Objet du model : `@ModelAttribute`
  - Paramètres ou corps de la requête : `@RequestParam`, `@RequestBody`

```
@GetMapping
public String setupForm(@RequestParam("petId") int petId, Model model) {
  Pet pet = this.clinic.loadPet(petId);
  // method logic...
}
```

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-methods

COMPOSANTS MVC

Méthodes de service (*Handler methods*) : signature « flexible »

- **Valeur de retour**
  - `String` : nom d'une Vue
  - Objet `View`
  - Objet `ModelAndView`
  - `@ResponseBody`
  - Jackson JSON

https://www.baeldung.com/jackson-json-view-annotation

LE FRAMEWORK SPRING : BILAN

- **Avantages**
  - Légèreté
  - S'appuie sur des solutions open source éprouvées
  - Très utilisé
  - Documentation abondante
- **Faiblesses**
  - Complexité croissante
    - Nombreux sous-projets
    - Trop de « magie » ?
  - Concurrence de **Jakarta EE** (J2EE) devenu plus simple

⇒ **Spring Boot** : simplifier la configuration des projets

SPRING BOOT

Objectif : créer facilement une application Java

- Avec très peu de configuration
- Faire tourner une application Spring (Web ou non)
- Pas de génération de code ni de fichier de configuration XML
- Interface en ligne de commande pour démarrer plus rapidement
- Création d'un projet vide avec *Spring Initializr* :

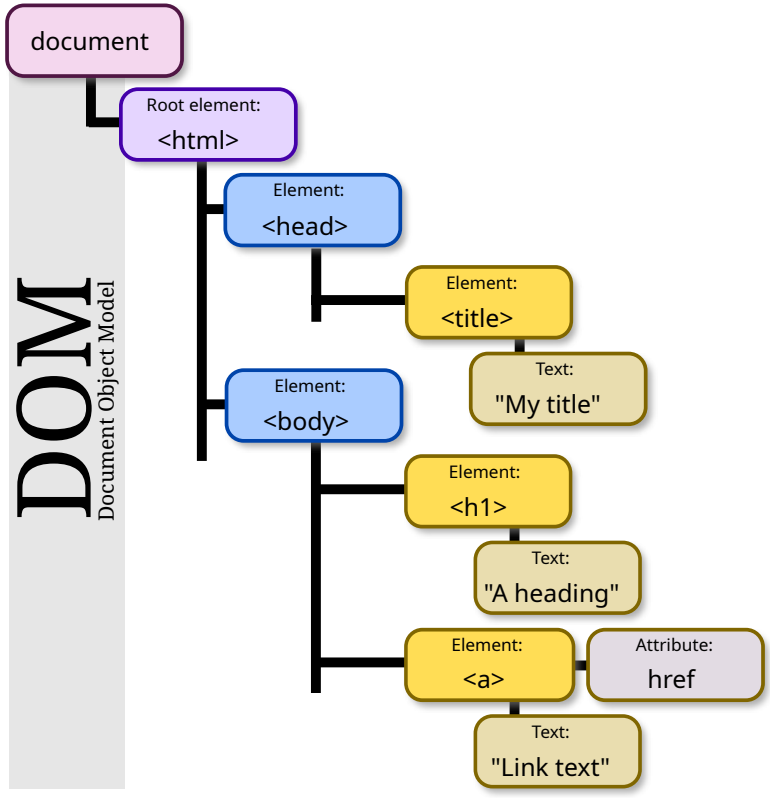
https://spring.io/projects/spring-boot

PROGRAMMATION CÔTÉ CLIENT EN JAVASCRIPT / TYPESCRIPT

OFFRE CÔTÉ CLIENT

Navigateur Web

- **Moteur de rendu :** HTML ⇒ DOM tree ⇒ Affichage



OFFRE CÔTÉ CLIENT

Navigateur Web

- **Moteur de rendu** : HTML ⇒ DOM tree ⇒ Affichage
- **Moteur de scripting**
  - Objectifs :
    - manipulation du DOM tree
    - échanges de données asynchrones
    - logique applicative côté client
  - Premières versions :
    - **JavaScript** : Netscape Navigator 2.0 (mars 1996)
    - JScript : MS Internet Explorer 3.0 (août 1996)
    - Standard commun : ECMAScript-262 (juin 1997)

https://en.wikipedia.org/wiki/ECMAScript

MOTEURS JAVASCRIPT

- **Interprétés**
  - Compilation JIT en bytecode ou code natif
  - Implémentent un sous-ensemble de ES6

- **Nouvelles versions** (ES 7-11, TypeScript)
  - Programmation d'applications structurées côté client
  - Nouveaux éléments syntaxiques
  - **Transpilables en ES5**

BASES DU LANGAGE JAVASCRIPT

- **Typage dynamique**
  - variables typées (entiers, réels, chaines de caractères...)
  - mais conversions de types implicites

```
var a;
var b;
a = 5;           // a est un entier
b = '5';         // b est une chaines de caractères
var c = (a===b); // c est un booléen
b = 10;          // b devient un entier
c = a + b;       // c devient un entier égal à 15
```

- **Flexible & permissif**
  - Pas/peu d'erreurs à l'exécution
  - Utilisation de variables hors scope

BASES DU LANGAGE JAVASCRIPT

Recommandations :

- Déclarer les variables avec `const` et `let` plutôt que `var`
- Gérer les erreurs avec `try` et `catch`
- Utiliser les fonctionnalités des versions récentes
- Utiliser le mode *strict*

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

BASES DU LANGAGE JAVASCRIPT

- Langage **fonctionnel**
  - Fonctions = « citoyens de première classe »
  - mais **pas fonctionnel pur** :
    les fonctions peuvent modifier des variables externes
- Langage **object orienté prototype** (jusqu'à ES5)
  - types associés aux instances et non aux classes
  - approche « dictionnaire »
- Langage **événementiel**
  - Mécanismes de *callback*
  - Pattern **Observer** : *eventListener*

BASES DU LANGAGE JAVASCRIPT

- **Objet = dictionnaire**
  - ensemble de propriétés et de méthodes
  - couples `nom : valeur`
  - `this` pour référencer l'objet courant

```
let jonSnow = {
  first : 'Jon',
  last : 'Snow',
  isAlive : undefined,
  resurrect : function() {
    this.isAlive = true;
  }
}

jonSnow.isAlive = false;
jonSnow.resurrect();
```

BASES DU LANGAGE JAVASCRIPT

- Programmation **orientée prototype**
  - constructeur
  - instanciation avec new

```
function Person(first, last) {
  this.first = first,
  this.last = last,
  this.isAlive = undefined,
  this.resurrect = function() {
    this.isAlive = true;
  }
}

let jonSnow = new Person('Jon','Snow');
jonSnow.resurrect();
```

BASES DU LANGAGE JAVASCRIPT

- Programmer de l'**orienté object** (ES6)
  - `class`, `constructor`, `static`, `extends`, `super`
  - Propriétés publiques
  - Pas de classes abstraites ou d'interfaces

```
class Person {
  constructor(first, last) {
    this.first = first;
    this.last = last;
    this.isAlive = undefined;
  }

  resurrect() {
    this.isAlive = true;
  }
}

let jonSnow = new Person('Jon','Snow');
jonSnow.resurrect();
```

JAVASCRIPT OBJECT NOTATION (JSON)

- **Sérialisation / désérialisation d'objets JS**
  - Dictionnaires
  - Tableaux
- **Primitives simples :**

```
let jonSnowJSON = '{ "first" : "Jon", "last" : "Snow", "isAlive" : true }';
let jonSnow = JSON.parse(jonSnowJSON);
jonSnowJSON = JSON.stringify(jonSnow);
```

TYPESCRIPT

- JavaScript + **typage strict**
- Langage créé par Microsoft en 2012
- Fondé sur ES2015
- Spec : https://www.typescriptlang.org/

TYPESCRIPT

Fonctionnalités principales

- **Typage explicite** des variables et des fonctions (encouragé)
- Inférence de types (pour les types non déclarés)
- Programmation **orientée-objet** (classes + interfaces)
- **Généricité**
- **Modularité**
- Composants réutilisables (*Mixins*)
- Annotations (décorateurs `@` comme en Java)

TYPESCRIPT : EXEMPLE

```
interface Person<T> {
  first?: T;
  last: T;
  isAlive: boolean | undefined;
  resurrect() : void;
}

let jonSnow: Person<string> = {
  last : 'Snow',
  isAlive : undefined,
  resurrect : function() { this.isAlive = true; }
}
```

TYPESCRIPT : TYPES PRIMITIFS

- `boolean`

```
let vrai: boolean = true;
```

- `number`

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

- `string` + *template*

```
let fullName: string = `Titi Toto`;
let sentence: string = `Hello, my name is ${ fullName }`;
```

TYPESCRIPT : TYPE PRIMITIF `symbol`

- Type défini dans la spec ES6
- Clés de propriétés **uniques** et **cachées**

```
let s = Symbol("id");
let user = { name: "Toto", [s]: 42};  // Les crochets indiquent que s est un symbole et non un
alert(user[s]); // 42
```

- `Symbol()` = *factory* qui génère un token unique (paramètre `string` optionnel)

```
console.log(Symbol("toto") == Symbol("toto")); // false
```

- Uniquement accessibles par : `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()` et `Object.assign()`

TYPESCRIPT : TYPES PRIMITIFS `null` ET `undefined`

- Sous-types de tous les autres types
- Intérêt : **vérifier les erreurs d'initialisation**
  en compilant avec `--strictNullChecks`

```
let test: string;
console.log(test); // Compilation : error TS2454: Variable 'test' is used before being assigne
let s: string = null; // Compilation : error TS2322: Type 'null' is not assignable to type 'st
```

- Si nécessaire, utiliser les ***union types*** :

```
let s: string | null = null; // Compilation : OK
```

TYPESCRIPT : AUTRES TYPES

- `tuple` : tableaux de taille fixe

```
let x: [string, number];
```

- `enum`

```
enum UserResponse { No = 0, Yes = 1 }
```

- `any` : pas de validation de type

```
let list: any[] = [1, true, "free"];
```

- `void` : retour de fonction ou variable que `null` ou `undefined`

```
function warnUser(): void { console.log("This is my warning message"); }
let unusable: void = undefined;
```

TYPESCRIPT : AUTRES TYPES

- `never` : fonction ne terminant jamais

```
function error(message: string): never {
    throw new Error(message);
}

function infiniteLoop(): never {
    while (true) { }
}
```

- `object` : n'importe quel type **non primitif**

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

TYPESCRIPT : ASSERTIONS DE TYPES

- Permettent de faire des « *cast* »
- 2 syntaxes :

```
let someValue: any = "this is a string";

let strLength1: number = (<string>someValue).length;

let strLength2: number = (someValue as string).length;
```

TYPESCRIPT : CLASSES

- Constructeurs, accesseurs
- Visibilité des membres `public` (défaut), `private`, `protected`
- Modifieurs : `readonly`, `static`, `abstract`
- Héritage (simple)

```
class Animal {
    protected name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    constructor() { super("Médor"); }
    bark() { console.log(this.name + ' says: Woof! Woof!'); }
}

const dog = new Dog();
dog.bark();
dog.move(10);
```

TYPESCRIPT : INTERFACES

- Définition de **nouveaux types** (propriété et méthodes)

```
interface ClockInterface {
    currentTime: Date; // propriété
    setTime(d: Date);  // méthode
}
```

- **Héritage** entre interfaces et classes (signature des méthodes)

```
class Control {
    private state: any;
}
interface SelectableControl extends Control {
    select(): void;
}
class Image implements SelectableControl {
    select() { }
} // Error: Property 'state' is missing in type 'Image'.
```

TYPESCRIPT : INTERFACES GÉNÉRIQUES

```
// Déclaration de l'interface
interface GenericIdentityFn {
    <T>(arg: T): T;
}

// Fonction qui l'implémente (sans le savoir)
function identity<T>(arg: T): T {
    return arg;
}

// Type checking OK
let myIdentity: GenericIdentityFn = identity;
```

TYPESCRIPT : MODULES

- Fondés sur les modules d'ES2015
- Module = fichier contenant une directive `import` ou `export`

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

```
import { ZipCodeValidator } from "./ZipCodeValidator";
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
import * as validator from "./ZipCodeValidator";
```

- Chaque module a son propre *scope*

https://www.typescriptlang.org/docs/handbook/modules.html

FRAMEWORKS JAVASCRIPT

DES BIBLIOTHÈQUES AUX FRAMEWORKS

Début des années 2000 :

Web centré **documents et formulaires** ⇒ centré **application**

- Essor des **Applications Web Riches** (RIA)
- Toujours beaucoup de logique côté serveur
- Développement d'**AJAX** et de **REST**
- Nombreuses **bibliothèques JS**
  - très permissives
  - offrant une fonctionnalité particulière

RICH INTERNET APPLICATIONS (RIA)

- **Web-based**
  - Paradigme client-serveur
  - Protocole HTTP
- **Expérience utilisateur** proche des **applications natives**
  - Traitement de l'interface côté client (JavaScript, CSS, DOM)
  - Échanges client-serveur asynchrones (AJAX)
- **Logique métier complexe**
  - Outils « évolués » de modélisation, conception, développement

DES BIBLIOTHÈQUES AUX FRAMEWORKS

Fin des années 2000 :
essor des **boîtes à outils**

- La plus emblématique : jQuery
- Autres : lodash, Underscore

⇒ **standardiser** les comportements des navigateurs

DES BIBLIOTHÈQUES AUX FRAMEWORKS

2010 : HTML5

- Implémentations (incomplètes) des standards existants
- Standards eux-mêmes incomplets

⇒ ***polyfill*** pour homogénéiser les comportements des navigateurs

https://en.wikipedia.org/wiki/Polyfill_(programming)

DES BIBLIOTHÈQUES AUX FRAMEWORKS

Années 2010's : frameworks JS

- Émergence des **Single Page Applications** (SPA)
- Déplacement de la logique vers le client
- Structuration du code
- Développement du *tooling* :
  - modules
  - préprocessing CSS
  - exécution de tâches

https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_web_frameworks

OBJECTIF GLOBAL

Faciliter le développement de **Single Page Applications (SPA)**

- Gérer la logique métier
- S'adapter aux terminaux mobiles
- Interagir avec le(s) serveur(s)
- Optimiser les performances

SINGLE-PAGE APPLICATION (SPA)

- Application côté client dans **une unique page Web**
  - une seule page à charger ⇒ **fluidité**
  - contient toutes les **vues** de l'application
  - *hash* (#) de l'URL pour sélectionner la vue à afficher

- **Pattern MV* complet côté client**
  - *Model* : JavaScript local + échanges asynchrones de données avec le serveur
  - *View* : ensemble de `<section>` dans la page HTML
    + mécanisme de *templating*
  - *Controller / Presenter / View-Model* : routage + gestion de événements + ...

ROUTAGE PAR HASH

- **Principe**
  - Intercepter le changement de *hash* dans l'URL

```
<a href="#/page1">Page 1</a>
<a href="#/page2">Page 2</a>
```

  - Récupérer les (éventuels) paramètres
  - Appeler une fonction JavaScript
- **Outils**
  - Événement `hashchange` et `window.location.hash`
  - Supporté par tous les navigateurs et simple à mettre en place
  - Mais peu lisible et mal adapté au référencement

⇒ **alternative en HTML5** : objet `window.history`, mais plus de configuration

LIENS ENTRE MODÈLE ET VUE

Deux approches :

- *One-way data binding*
  - modification d'une propriété du modèle
    ⇒ mise à jour de la vue

- *Two-way data binding*
  - modification d'une propriété du modèle
    ⇒ mise à jour de la vue
  - action sur la vue ⇒ mise à jour du modèle

TEMPLATING CÔTÉ CLIENT

- **Principe** :
  - Produire une vue à partir de données
  - Exemple côté client : Mustache + Handlebars
- **Interpolations** (texte, variables, expressions JS)

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}
```

- **Directives** (`if`, `for`, `on`)

```
<p v-if="seen">Now you see me</p>

<a v-on:click="doSomething"> ... </a>
```

COMPOSANTS

Créer ses éléments HTML

en fonction des besoins métiers de l'application

- Leur attribuer :
  - un nom
  - une vue (*template* HTML, CSS)
  - un comportement (animations, évènements...)
- Exemples : Web Components

INTRODUCTION À VUE.JS

- **Framework** initialement « complètement côté client »
- **Objectif** : déporter une partie de la logique métier d'une application localisée côté client
- **Écosystème de plugins** :
  - routeur
  - gestion d'états
  - *Server-Side Rendering*
  - ...

FONCTIONNEMENT

- Chaque élément d'interface est un **composant Vue** (HTML, JS/TS, CSS)
- **Lien modèle / vue**
  - *One-way data binding* : la vue réagit automatiquement aux changements du modèle
  - *Two-way data binding* : à faire explicitement depuis la vue
- Structure de l'application définie dans la configuration du framework

CRÉATION D'UN COMPOSANT

- Fichier `.vue` : *Single-File Component* (SFC)
- 3 blocs : `<script>`, `<template>` et `<style>`

```vue
<script setup>
import { ref } from 'vue';

const count = ref(0);
</script>

<template>
  <button @click="count++">You clicked me {{ count }} times.</button>
</template>

<style>
button {
  background-color: red;
}
</style>
```

https://vuejs.org/guide/essentials/component-basics.html

UTILISATION D'UN COMPOSANT

`import` dans le bloc `<script>`

```vue
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>

<template>
  <h1>Here are many child components!</h1>
  <ButtonCounter />
  <ButtonCounter />
  <ButtonCounter />
</template>
```

https://vuejs.org/guide/essentials/component-basics.html

TEMPLATING

- **Interpolation** : `{{ }}`
- **Directives** : préfixées par `v-`
  - `v-if` : affichage conditionnel
  - `v-for` : affichage de listes
  - *Two-way data binding*
    - `v-model` : gestion des formulaires
    - `v-on` (ou `@` + événement) : écouteur d'événements de l'interface

https://vuejs.org/guide/essentials/template-syntax.html

PROGRAMMATION RÉACTIVE

Exemple : **tableur**

```
let A0 = 1;  // dependence
let A1 = 2;  // dependence
let A2;

function update() { // (side) effect
  A2 = A0 + A1;
}

whenDepsChange(update) {
  // 1. track when a variable is read (A0 or A1),
  // 2. if variable read during update execution, make update a subscriber to that variable,
  // 3. detect variable mutated, notify all its subscriber effects to re-run.
}
```

VARIABLES RÉACTIVES

Intercepter l'**accès aux propriétés d'un objet**

- via un *Proxy* redéfinissant `get` / `set`
- via un nouvel objet servant de référence

VARIABLES RÉACTIVES : `reactive()`

- **Type objet** (objets, tableaux, collections)
- Réactivité **en profondeur** par défaut

```
const obj = reactive({ nested: { count: 0 }, arr: ['foo', 'bar'] })

function mutateDeeply() {
  obj.nested.count++;
  obj.arr.push('baz');
}
```

- Crée un *Proxy* de l'objet

```
const raw = {}
const proxy = reactive(raw)
// proxy is NOT equal to the original.
console.log(proxy === raw) // false
```

- Référence à ce *Proxy* doit rester constante,

≡sinon perte de la réactivité

VARIABLES RÉACTIVES

Mise à jour **asynchrone** du DOM

⇒ `nextTick()` pour attendre

```
import { nextTick } from 'vue'

const state = reactive({ count: 0 })

function increment() {
  state.count++;
  nextTick(() => {
    // DOM à jour
  })
}
```

VARIABLES RÉACTIVES : `ref()`

- **Tous les types** (primitifs et objets)
- Retourne un **nouvel objet**
  - valeur accessible via `.value` dans le `script`
  - déréférencement automatique dans le `template`

```
<script setup>
import { ref } from 'vue';

const count = ref(0);

function increment() {
  count.value++
}
</script>

<template>
  <button @click="increment">You clicked me {{ count }} times.</button>
</template>
```

PROGRAMMATION RÉACTIVE : `watchEffect()`

```
import { ref, watchEffect } from 'vue'

const A0 = ref(0);
const A1 = ref(1);
const A2 = ref();

watchEffect(() => {
  // tracks A0 and A1
  A2.value = A0.value + A1.value;
})

// triggers the effect
A0.value = 2;
```

https://vuejs.org/guide/essentials/watchers#watcheffect

PROGRAMMATION RÉACTIVE : `computed()`

```
import { ref, computed } from 'vue';

const A0 = ref(0);
const A1 = ref(1);
const A2 = computed(() => A0.value + A1.value);

A0.value = 2;
```

Remarque : résultat mis en cache,
seulement réévalué si les dépendances changent

https://vuejs.org/guide/essentials/computed.html

PROGRAMMATION RÉACTIVE : `watchEffect()`

```
import { ref, watchEffect } from 'vue';

const count = ref(0);

watchEffect(() => {
  document.body.innerHTML = `count is: ${count.value}`;
})

// updates the DOM
count.value++;
```

https://vuejs.org/guide/essentials/watchers#watcheffect

PROGRAMMATION ÉVÉNEMENTIELLE

Objet `DOM Event`

- Émis lorsqu'un élément DOM subit des **interactions**
  - Lorsqu'un utilisateur clique sur la souris : `click`
  - Quand une page Web / une image est chargée : `load`
  - Quand la souris passe sur un élément : `mouseover`
  - Lorsqu'un champ de saisie est modifié : `change`
  - Lorsqu'un formulaire HTML est soumis : `submit`
- Peut être **intercepté** à l'aide de code JavaScript

PROGRAMMATION ÉVÉNEMENTIELLE

- **Deux processus en parallèle**
  - *Principal* : déroulement des traitements et association des événements à des fonctions de callback
  - *Callbacks* : récupèrent et traitent les événements
- **Deux syntaxes**
  - attributs HTML spécifiques (`onclick`, `onload`)
  - ajout d'`eventListeners` en JavaScript

```
monElementDOM.addEventListener("click", maFonctionCallback);
```

FONCTIONS DE RAPPEL (CALLBACK)

Fonction passée en paramètre à une autre fonction

- **Intérêt** : exécuter du code au sein d'un fonction
  - sans savoir ce qu'il va faire
  - en suivant une interface de programmation
- **2 syntaxes :**
  - sans paramètre : directement le nom de la fonction

```
setTimeout(callbackFunction);
```

  - avec paramètre : encapsulation dans une fonction anonyme

```
setTimeout(function() { callbackFunction("toto"); });
```

PROMESSES

- **Objectifs** :
  - lancer du code **asynchrone ou différé**
  - réagir spécifiquement
    - en cas de succès (*fullfilled*)
    - en cas d'erreur (*rejected*)
    - à la fin de l'exécution (*settled*)
  - conserver des performances acceptables

PROMESSES

- **Création** (avec passage de paramètre)

```
function maFonctionAsynchrone(nbDecimales) {
    return new Promise((resolve, reject) => {
        try {
            const pi = computePiIn30seconds(nbDecimales);
            resolve(pi);
        } catch(e) {
            reject("machine too slow!");
        }
    });
}
```

- **Appel**

```
maFonctionAsynchrone(150).then(
      function(result) { console.log(result); }
  ).catch(
      function(error) { console.error(error); }
  ).finally(
      function() { alert("Résultat disponible dans la console"); }
);
```

PROMESSES

- **Chaînage :** `.then` retourne une nouvelle promesse

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 2

});
```

PROMESSES : async / await

- **Simplifier l'écriture de promesses**
  - async : faire retourner une promesse

```
async function f() { return 1; }

f().then(alert); // 1
```

  - await : faire attendre le résultat d'une promesse (dans une fonction async)

```
async function maFonctionSynchrone() {
    try {
        const result = await maFonctionAsynchrone(150);
        console.log(result);
    } catch(error) {
        console.error(error);
    }
    alert("Résultat disponible dans la console");
}
```

REQUÊTE ASYNCHRONE AU SERVEUR

Deux solutions :

- **Asynchronous Javascript And XML (AJAX)**
  - Transfert de données en XML via un objet `XMLHttpRequest` ou un contrôle ActiveX `XMLHTTP` (Internet Explorer)
  - Mécanisme d'*events* et *callbacks*

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

- **Fetch API**
  - Primitives de plus haut niveau que `XmlHttpRequest`
  - Récupérer du texte ou du JSON (pas de XML)
  - Encapsulation dans des promesses

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

FETCH API

- Un seul paramètre obligatoire : **URL**
- Options dans un **objet JSON**

```
fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
      'Content-Type': 'application/json; charset=utf-8'
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer
    body: JSON.stringify(data) // body data type must match "Content-Type" header
  });
```

FETCH API

- Renvoie **une réponse**
  - format du corps en fonction du *Content-Type*
- **Erreur** levée en cas de :
  - problème réseau
  - réponse non conforme au résultat attendu (parsing JSON)
- **Pas d'erreur HTTP** ⇒ vérifier `Response.status` à la main

```
async function fetchData(url) {
  let response = await fetch(url);

  if (response.ok) { // if HTTP-status is 200-299
    let json = await response.json();
  } else {
    console.error("HTTP-Error: " + response.status);
  }
}
```

ROUTING AVEC VUE.JS

- `vue-router` : plugin recommandé
- `Hash` et `History` possibles

```
// 1. Define route components (can be imported from other files)
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. Define some routes
const routes = [ { path: '/foo', component: Foo },
   { path: '/bar', component: Bar } ]

// 3. Create the router instance and pass the `routes` option
const router = new VueRouter({
  routes // short for `routes: routes`
})

// 4. Create and mount the root instance.
const app = new Vue({
  router
}).$mount('#app')
```

https://router.vuejs.org/

OUTILS COMPATIBLES AVEC VUE.JS

- **NPM** : gestionnaire de paquets JavaScript
- **Webpack** : packaging / optimisation
- **Vite** : serveur de dev.
- **ESlint** : analyse statique du code
- ...

Remarque : Vue fournit sa propre CLI : `vue-cli`

REPRESENTATIONAL STATE TRANSFER (REST)

AVANT : LES SERVICES WEB

- **Architectures Orientées Services**
- Ensemble de technologies (SOAP, WSDL, UDDI, BPEL...)
- **Échange de messages (XML, JSON)**, pas de pages Web
- Expose une **API** à un client AJAX ou un autre service
  - ensemble d'opérations disponibles (*verbs*)
  - ex. : `login`, `addUserToGroup`, `createReservation`
- Utilise (ou pas) HTTP comme protocole de transport

SOAP : SIMPLE OBJECT ACCESS PROTOCOL

- Mécanisme **très puissant** pour les applications Web complexes
- Pas si « simple » (spécification volumineuse)
- Pas si interopérable (encodage des messages)
- Pas si standard / réutilisable :
  - l'API d'un service dépend des choix de conception
  - couplage fort entre fournisseur et consommateur
- **Passage à l'échelle** problématique
  - pas de cache
  - bande passante importante

⇒ Perte des « bonnes propriétés » qui ont fait le succès du Web

REPRESENTATIONAL STATE TRANSFER (REST)

Proposé en 2000 par Roy Fielding dans sa thèse

1. Analyse des bonnes propriétés du Web
2. Recherche des raisons de leur conception
3. Généralisation sous forme d'un style architectural applicable au Web des machines
4. Définition de contraintes pour le respect de ce style

CONTRAINTES (1/4)

1. **Uniform interface**

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application state

⇒ **conception orientée-ressources**

⇒ changements d'états par des **liens hypermédias**

CONTRAINTES (2/4)

2. **Client/Server** :
   separation of concerns
3. **Stateless** :
   each request must contain all of the information necessary to understand the request

   ⇒ serveur ne gère par **l'état de l'interaction avec le client**

CONTRAINTES (3/4)

4. **Cache** :
   response implicitly or explicitly labeled as cacheable or non-cacheable
5. **Layered System** :
   each component cannot "see" beyond the immediate layer with which they are interacting

⇒ **passage à l'échelle**

CONTRAINTES (4/4)

6. **Code on demand** (optional) :
   allows client functionality to be extended by downloading and executing code in the form of applets or scripts

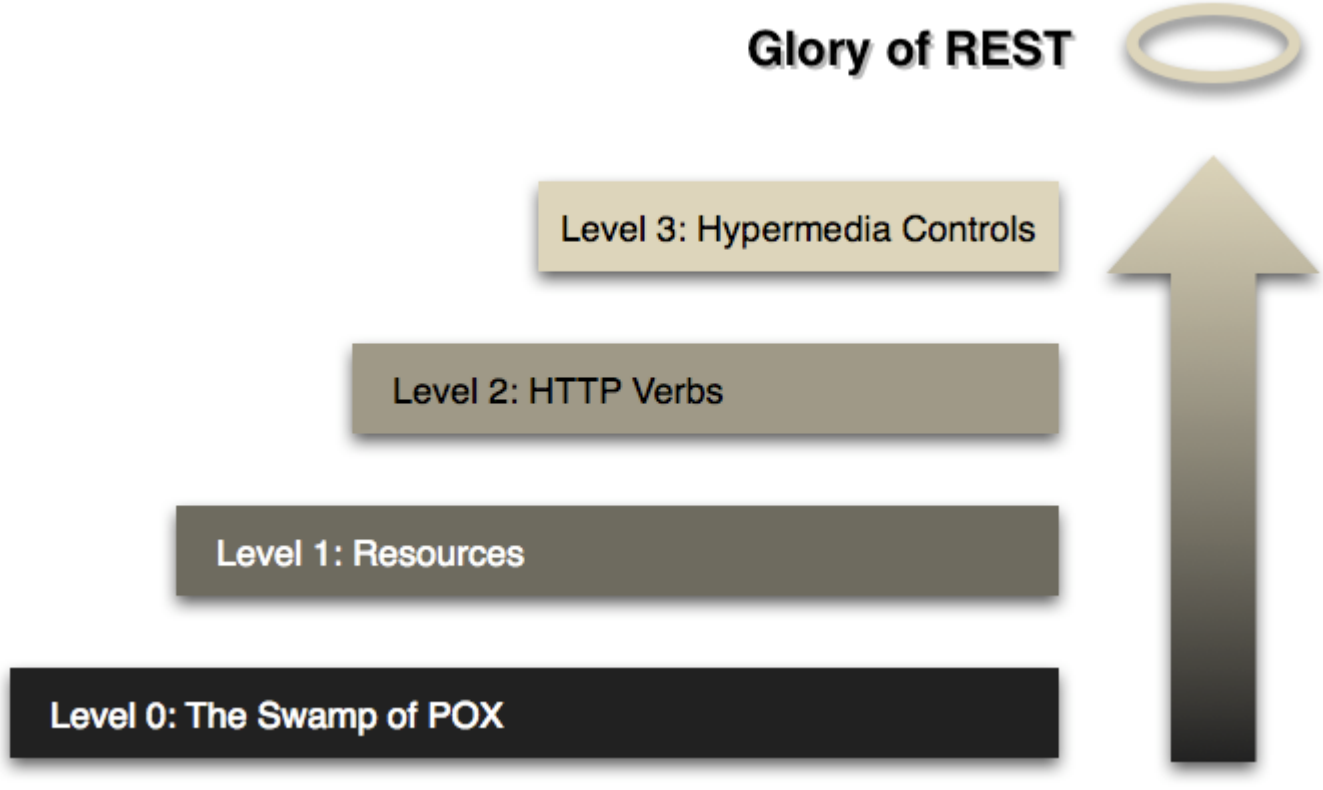
⇒ **évolution de l'interface**

REST, CONCRÈTEMENT

Contraintes théoriques ⇒ « recettes » opérationnelles

- Approche orientée-ressources
- Transactions sans état
- Forme des URL
- Utilisation avancée de HTTP
- Flux applicatif : HATEOAS
  (*Hypertext As The Engine Of Application State*)

MODÈLE DE MATURITÉ DE RICHARDSON



(crédit : billet de blog de Martin Fowler)

APPROCHE ORIENTÉE-RESSOURCE

Le serveur gère et expose des ressources

(pas l'interaction avec le client)

- **Ressource**
  - objet du domaine
  - identifiée par un nom
  - exposée par son **URL** (unique)
  - échangée sous forme de **représentations** (structure de la réponse)
  - **auto-décrite** (`Content-Type` de l'en-tête)

APPROCHE ORIENTÉE-RESSOURCE

Modèle de ressources arborescent

- Deux types :
  - **instance** (objet métier)
  - **collection** d'instances
- Les ressources « portent » les cas d'utilisation
- Cas d'utilisation (CU) liés à **l'état interne** des ressources
  - état interne pas accessible par le client

TRANSACTIONS SANS ÉTAT

Le client gère le contexte de l'interaction avec le serveur

Le serveur ne conserve aucune trace des interactions passées

- La connexion est dite **sans état** (*stateless*)
  - le client maintient ses « variables de session »
  - on élimine un goulot d'étranglement
  - on économise des ressources

⇒ **Gain en performances et en scalabilité**

TRAVAILLER EN « STATELESS »

1. **Passer le contexte dans la requête**
   - En paramètres : en fonction de la méthode HTTP
     - dans l'URL

```
https://api.qwant.com/api/search/web?count=10&q=test&t=web&locale=fr_FR&uiv=4
```

     - dans le corps de la requête

```
{"query":"test","language":"francais","pl":"ext-ff","lui":"francais","cat":"web"}
```

   - Dans les headers HTTP
     - authorization
     - cookies

TRAVAILLER EN « STATELESS »

2. **Récupérer le contexte côté serveur**
   - Authentification, autorisations
     ⇒ **filtres**
   - Informations nécessaires au traitement de la requête
     ⇒ **Contrôleur**
3. **Traiter la requête en interrogeant le modèle**

Remarque : cas particulier de **l'authentification sans état**

FORME DES URL

URL libres, mais **bonnes pratiques** :

- **Version** : à la racine de l'arbre
- Noms pluriels pour les collections
- ID d'instances comme sous-ressources des collections
- Séparer les éléments par des **slashs** (« / »)
- Pas de slash final
- **Paramètres** : filtres, tri, pagination

```
https://monserveur.com/api/v3/users/toto/friends/1
https://monserveur.com/api/v3/messages?author=toto&sort=+title,-index
https://monserveur.com/api/v3/messages?offset=100&limit=25
```

FORME DES URL

URL URL pour les requêtes de **CU « opérationnels »**

- Utiliser un verbe
- Raccrocher à l'URL de la ressource à laquelle se rapporte le CU

```
https://monserveur.com/api/v3/users/login
https://monserveur.com/api/v3/users/toto/playlist/play
```

- Pas toujours évident : que choisir ?

```
https://monserveur.com/api/v3/users/logout
https://monserveur.com/api/v3/users/toto/logout
```

UTILISATION AVANCÉE DE HTTP

Utiliser les **méthodes HTTP** pour les opérations

- **C**reate : `POST /users`
- **R**ead : `GET /users` ou `GET /users/{id}`
- **U**pdate : `PUT /users/{id}`
- **D**elete : `DELETE /users/{id}`

UTILISATION AVANCÉE DE HTTP

Bonnes pratiques pour les codes de statut

- Renvoi d'un contenu :
  - `200 OK` ou `206 Partial Content`
- Création : `201 Created` + lien vers la ressource dans le header `Location`
- Modification :
  - `200 OK` + représentation, si nécessaire
  - `204 No content`, sinon
- Suppression : `204 No content`
- URL incorrecte : `405 Method not allowed`

UTILISATION AVANCÉE DE HTTP

Négociation de contenus

⇒ renvoyer une représentation de ressource
en fonction des **préférences du client** spécifiées par :

- Entête HTTP

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Language: en,en
Accept-Encoding: gzip, deflate
```

- Extension dans l'URL (uniquement le format de sérialisation)

```
https://monserveur.com/api/v3/users/toto.html
https://monserveur.com/api/v3/users/toto.xml
https://monserveur.com/api/v3/users/toto.json
```

UTILISATION AVANCÉE DE HTTP

Négociation de contenus

- Si type demandé non disponible, réponse vide avec le code :
  - `300 Multiple Choices` + liste de choix
  - `406 Not Acceptable` + liste de choix
- Sinon, envoi du media type (ex. MIME types) dans la **réponse**
  - **Syntaxe** :

    `type "/" subtype ["+" suffix] *[";" parameter]`

```
Content-Type: application/json
Content-Type: application/vnd.ms-excel
Content-Type: text/html; charset=utf-8
Content-Type: application/x.foo+json; schema="https://example.com/my-hyper-schema#"
```

HYPERMEDIA AS THE ENGINE OF APPLICATION STATE (HATEOAS)

API **auto-découvrable** en renvoyant des **contrôles hypermédias**
= **liens** (URI) vers les ressources + **description**

- Dans l'entête HTTP :

```
Location: https://monserveur.com/api/v3/users/toto
Link: <https://monserveur.com/api/v3/users/titi>; rel="previous"; title="previous user";
```

- Dans le corps de la réponse :

```
{
  "userId":"toto",
  "links": [
    { "href": "https://monserveur.com/api/v3/users/toto/messages",
      "rel": "messages",
      "type" : "GET" }]
}
```

HYPERMEDIA AS THE ENGINE OF APPLICATION STATE (HATEOAS)

Le client **parcourt les liens** grâce aux **contrôles hypermédias**

- En présentant l'information à l'utilisateur et en le laissant choisir
- En « hard-codant » la logique applicative côté client
  (quel intérêt de la description ?)
- En utilisant la description pour déduire automatiquement les actions à effectuer
  - description sérialisée sous forme d'un media type
  - schéma standardisant le media type
    (HAL, Collection+JSON, JSON-API, Siren, etc.)

WEB API

- **REST** fournit des principes pour :
  - alléger les serveurs
  - permettre l'évolution des clients

- Créer des applications
  - en exposant des **ressources** sur un serveur
  - en documentant l'API du serveur : **Web API**
  - en réalisant le **client séparément**
    - en Javascript dans un navigateur Web
    - sous forme d'applications bureau ou mobile
    - sur un autre serveur
    - sur des composants intermédiaires qui exposent / filtrent / transforment ces données

REST A AUSSI DES INCONVÉNIENTS

- **Augmentation de la bande passante**
  - Toute l'information nécessaire au traitement de la requête doit « voyager » dans la requête
- **Nécessite des clients « intelligents »**
  - Authentification sans état
- **Manque d'un standard de description d'interfaces**
  - de nombreuses propositions (WADL, Hydra, OpenAPI)
  - mais aucune ne s'est encore complètement imposée