08/05/2023
Version 2

JEE

JÉRÉMY PERROUAULT

RAPPELS

HTTP & RAPPELS

PROTOCOLE HTTP

- Quelques commandes HTTP
  - GET
  - POST
  - PUT
  - DELETE

PROTOCOLE HTTP

- Les requêtes HTTP composées de
  - Commande HTTP, URL et version du protocole utilisé
  - Champs d'en-tête [optionnels]
  - Corps de la requête [optionnel]

- L'URL est formée de la manière suivante
  - [protocol]://[host]:[port][request_path]?[query_string]
  - http://localhost:8080/formation-web?id=34
  - http://localhost:8080/formation-web?idProduit=42&fournisseurId=2
  - https://www.google.fr/

PROTOCOLE HTTP

- La réponse HTTP est composée de :
  - Statut et version du protocole utilisé
  - Champs d'en-tête [optionnels]
  - Corps de la réponse

PROTOCOLE HTTP

Code	Type	Message	Définition
200	Succès	OK	Requête traitée avec succès
400	Erreur (côté client)	Bad Request	La syntaxe de la requête est erronée
401		Unauthorized	Une authentification est requise
403		Forbidden	Le serveur refuse d'exécuter la requête. S'authentifier n'y changera rien.
404		Not Found	La ressource n'a pas été trouvée
405		Method Not Allowed	Méthode de requête non autorisée
408		Request Time-out	Temps d'attente d'une réponse serveur écoulé
500	Erreur (côté serveur)	Internal Server Error	Erreur interne du serveur
502		Bad Gateway	Mauvaise réponse envoyée à un serveur intermédiaire par un autre serveur
503		Service Unavailable	Service temporairement indisponible

PROTOCOLE HTTP

- Le protocole HTTP est un protocole dit « déconnecté »
  - Nativement, le serveur ne garde pas en mémoire l'historique des requêtes d'un client
  - Il n'y a pas de cohérence client / serveur
  - Exemple
    - Un client devra envoyer son nom d'utilisateur et son mot de passe à chaque requête pour que le serveur puisse le reconnaître
      - Avec toute autre information nécessaire au bon fonctionnement de l'application Web
  - Pour assurer cette cohérence, l'utilisation de 2 mécanismes est primordiale
    - D'abord côté client, avec les Cookies
    - Puis côté serveur, avec les Sessions

PROTOCOLE HTTP

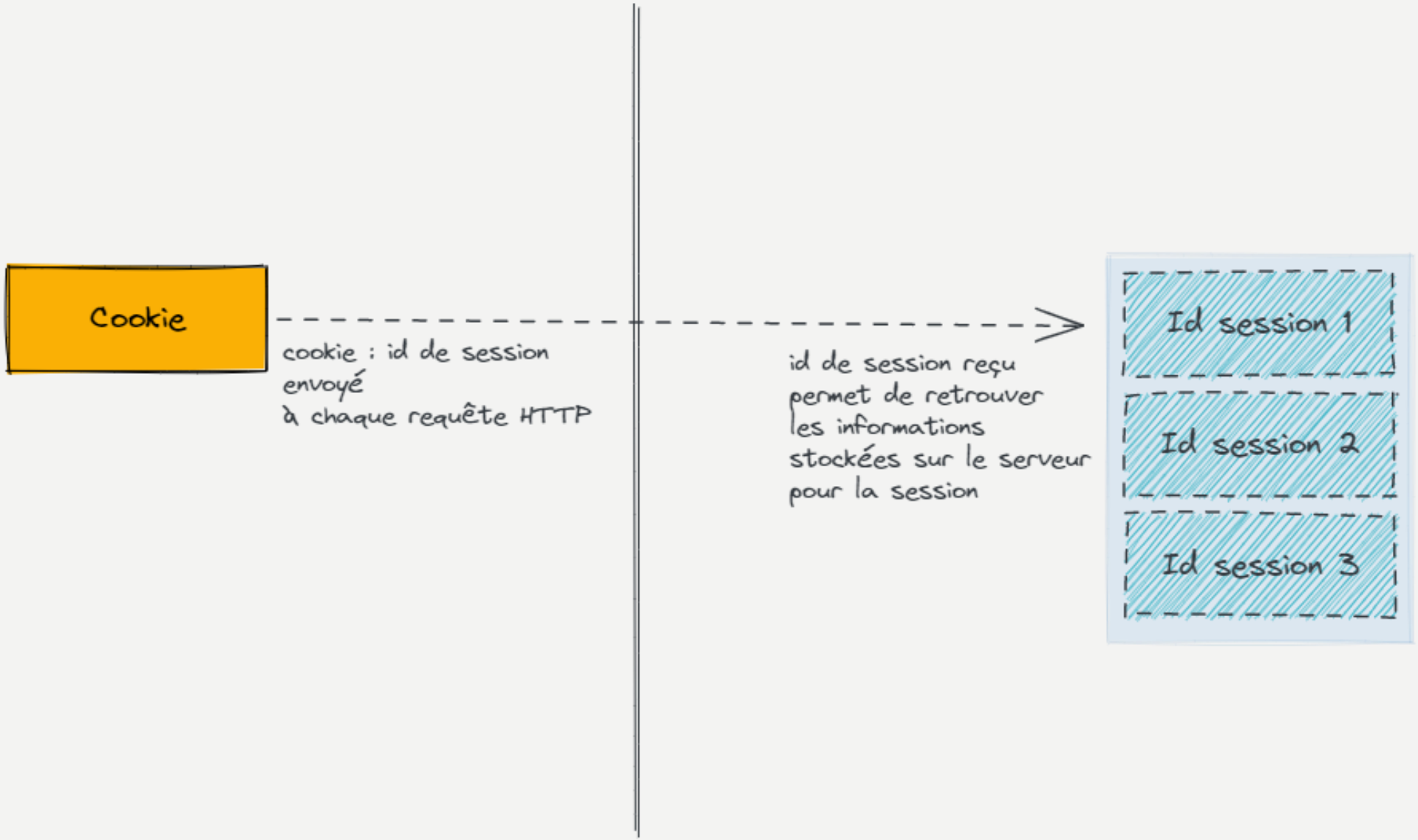
- Le cookie est une donnée stockée sur le poste client (navigateur)
- Envoyé à chaque requête vers l'hôte pour lequel le cookie a été stocké

PROTOCOLE HTTP

- Espace alloué sur le serveur (donnée stockée sur le serveur)
- Permet la persistance de données
  - En Java, les informations sont stockées dans l'objet persistant **HttpSession**
- Permet de maintenir la cohésion entre utilisateur et la requête

- L'identifiant de session est stocké dans un cookie du navigateur !

PROTOCOLE HTTP



INTRODUCTION

INTRODUCTION À JAKARTA EE

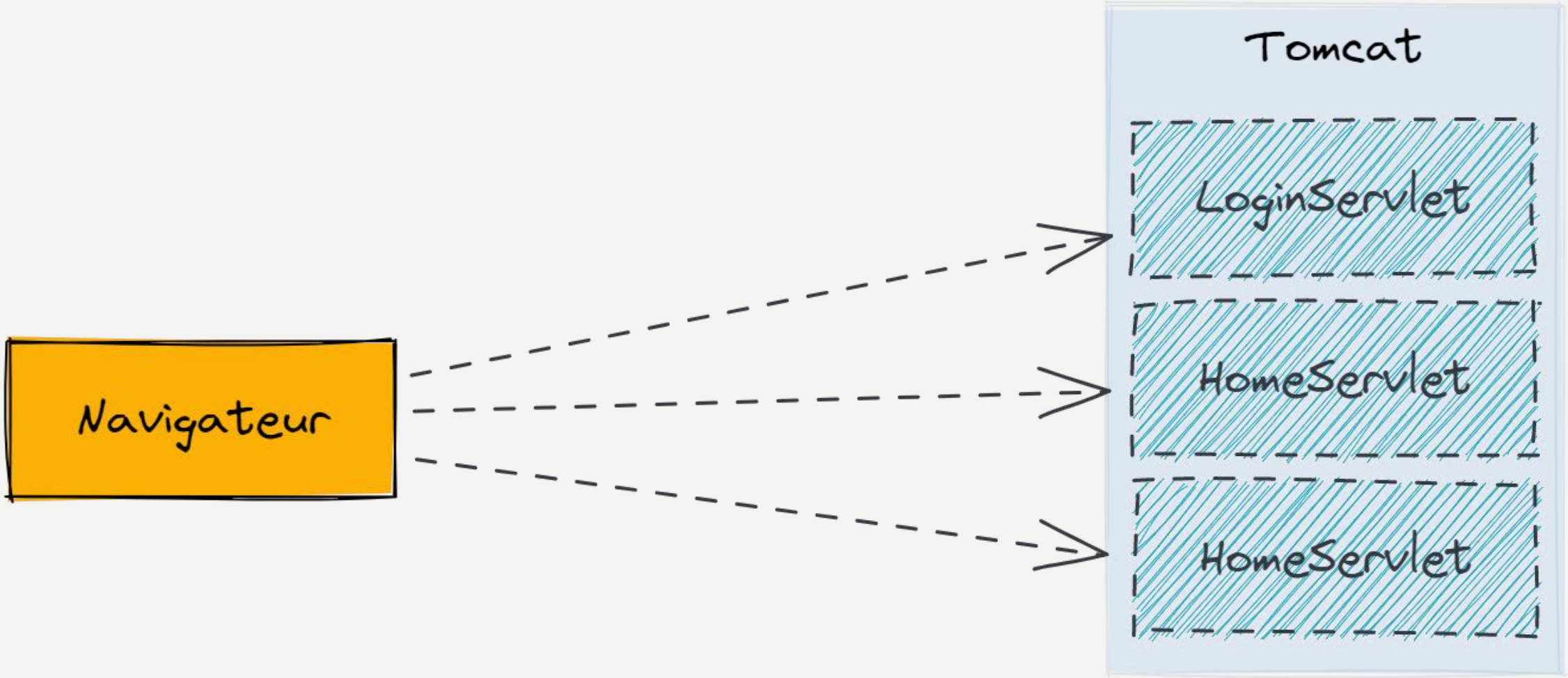
INTRODUCTION

- **JakartaEE** (anciennement **JavaEE**) est une norme, il faut choisir son implémentation
  - **Tomcat**
  - **JBoss**
  - **Glassfish**
  - …
- Application en couches **MVC**

INTRODUCTION

- Le serveur d'application joue le rôle de l'application principale **JAVA**
  - Il a accès à la console et peut imprimer des informations et des erreurs
  - C'est un programme qui existe et a été écrit
    - On n'écrit plus de classe avec la méthode statique `main`
  - C'est un programme un peu différent
    - On pourra y accéder grâce à des points d'accès (appelés **Servlet**) avec un client **HTTP** (navigateur web)
  - Les projets web seront exécutés au sein de ce serveur d'application

INTRODUCTION



INTRODUCTION

Servlet	Listener	Filter
Point d'accès Accessible par une requête **HTTP** (**GET**, **POST**, **PUT**, **DELETE**, ..,) sur une URL spécifiée http://localhost:8080/projet-web/home *On appelle ça le "mapping"* C'est elle qui génère la réponse **HTTP** que le navigateur interprétera	Ecouteur qui déclenche une action lorsqu'un évènement se déclenche	Permet de filtrer des requêtes

INTRODUCTION

- Dans une application **JAVA** classique
  - Un fichier **JAR** (**J**ava **AR**chive) est créé (classes compilées et empaquetée)
- Dans une application **JAVA** web
  - Un fichier **WAR** (**W**eb **AR**chive) est créé (classes compilées et empaquetée)

- Pour un projet **MAVEN**, il faudra donc choisir dans les options du projet
  - Un package **WAR**

EXERCICE

- Télécharger et dézipper **Apache Tomcat 10**
- Configurer **Apache Tomcat 10** sur **Eclipse**
  - Ajouter un nouveau **Server**
- Démarrer le serveur
  - Vérifier que ça fonctionne en allant sur cette adresse http://localhost:8080/
  - Une erreur 404 s'affiche si tout va bien !

ARCHITECTURE

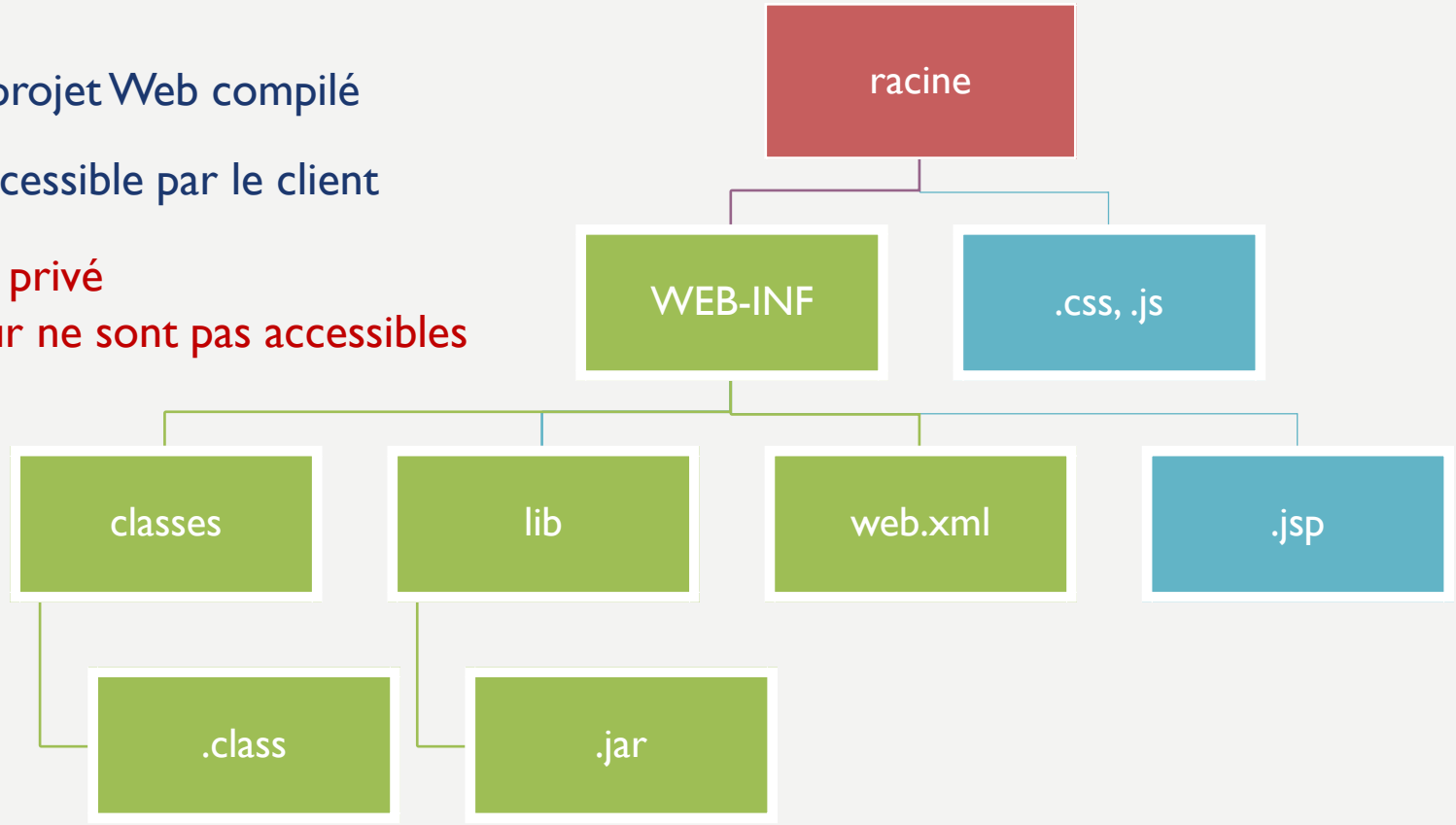
LES ARCHITECTURES

ARCHITECTURE

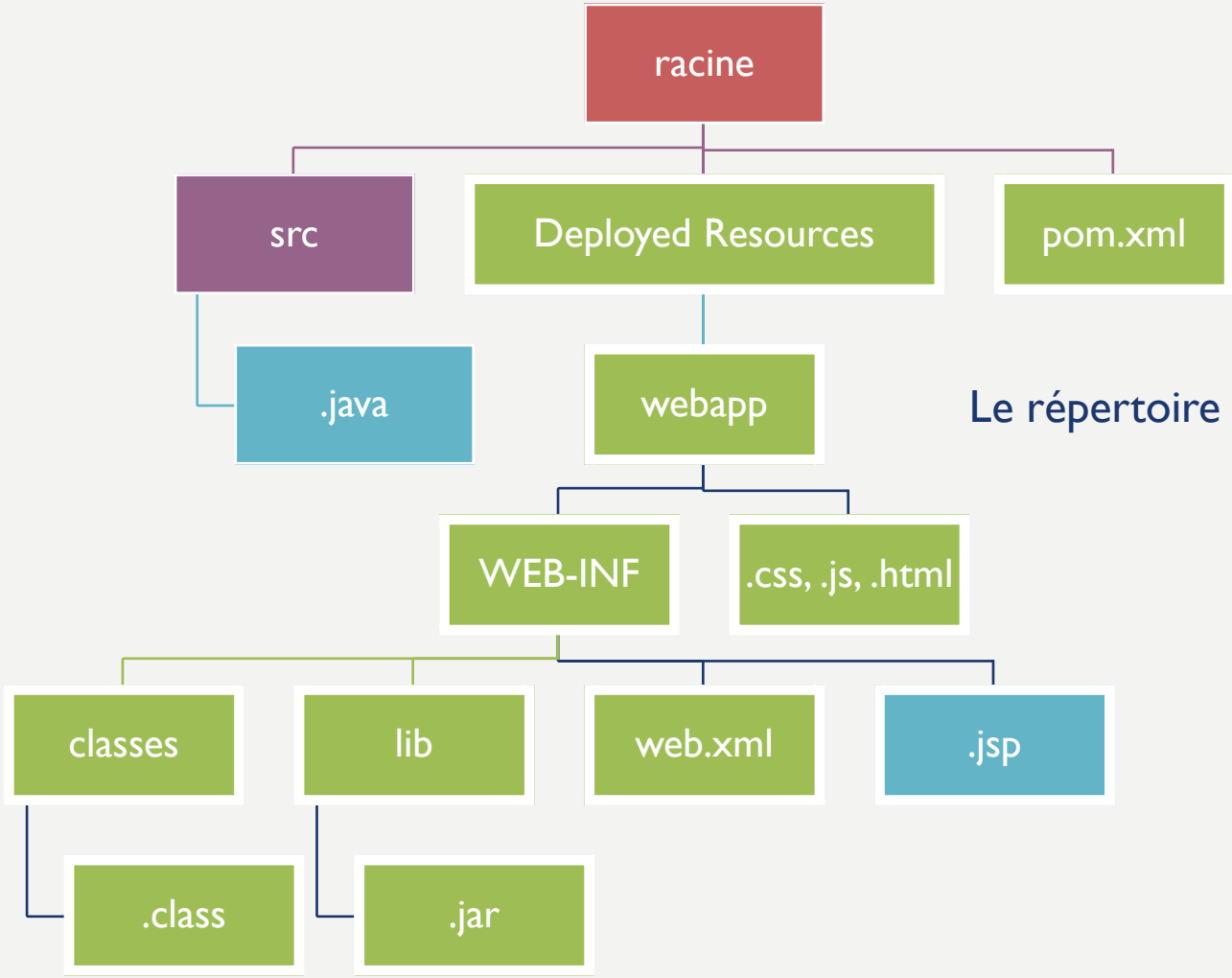
C'est l'arborescence d'un projet Web compilé

Le répertoire racine est accessible par le client

Le répertoire *WEB-INF* est privé
Tous les fichiers à l'intérieur ne sont pas accessibles



ARCHITECTURE



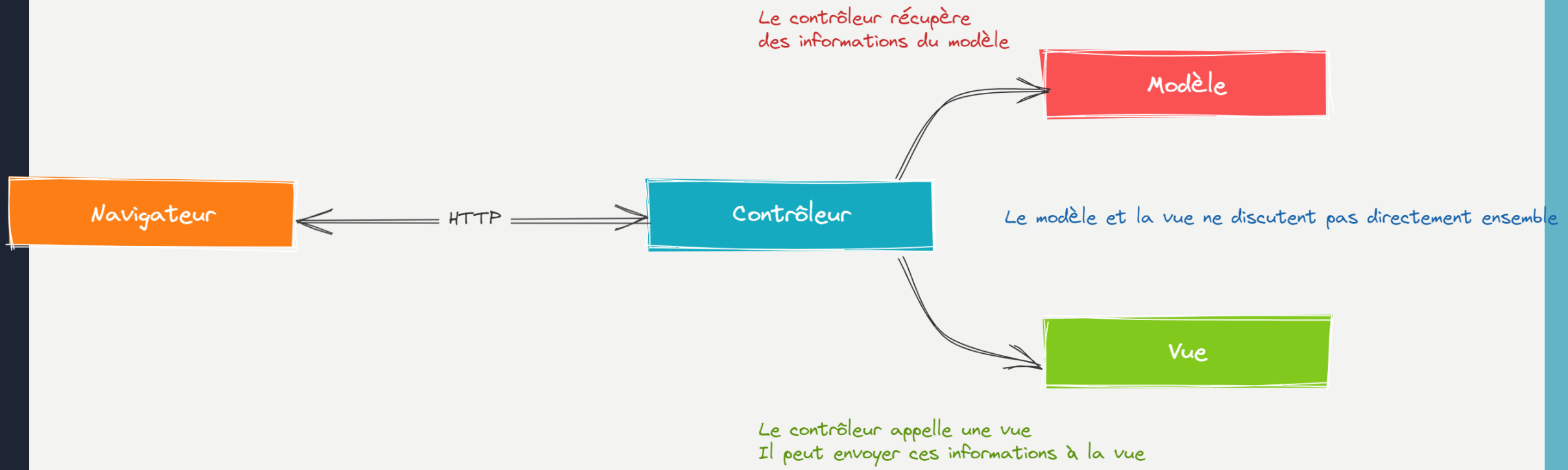
Le répertoire **webapp** est en fait la "racine compilée"

ARCHITECTURE

- Dans le répertoire *WEB-INF*
  - On retrouve les fichiers privés
  - Mais surtout le fichier de configuration du projet Web, le fichier *web.xml*

ARCHITECTURE

Application MVC - Modèle - Vue - Contrôleur



SERVLETS

LES SERVLETS

LES SERVLETS

- Classe **JAVA** qui traite les requêtes **HTTP**
  - Hérite de la classe abstraite `jakarta.servlet.http.HttpServlet`
- Implémentation (au besoin) des méthodes choisie
  - `doGet` Requête **HTTP** `GET`
  - `doPost` Requête **HTTP** `POST`
  - `doPut` Requête **HTTP** `PUT`
  - `doDelete` Requête **HTTP** `DELETE`

- Chaque **Servlet** doit être mappée sur une *URL*
  - *Sans prendre en compte le nom du serveur ni le nom du projet*

LES SERVLETS

- Les **Servlets** sont gérés par le conteneur de **Servlets**

- Un conteneur gère le cycle de vie et la vie des instances
  - Le conteneur de **Servlets** gère le cycle de vie des **Servlets** et tout ce qui s'articule autour
  - C'est lui qui instancie, utilise et détruit une **Servlet**

LES SERVLETS

- Initialisation du serveur
  - Création du pool de threads auxquels les requêtes seront affectées
  - Création des **Servlets** indiquées comme devant être initialisées au démarrage
- Première requête
  - Chargement de la **Servlet** (si pas déjà fait au démarrage)
  - Méthode `init()` invoquée par le conteneur
  - Création des objets `Request` et `Response` spécifiques à la requête
- Autres requêtes
  - La méthode `service()` est invoquée dans un nouveau thread
  - Le conteneur donne les paramètres `Request` et `Response` à la **Servlet**
- Déchargement de la **Servlet**
  - La méthode `destroy()` est appelée

LES SERVLETS

- Déclaration des mapping (URL, chemin d'accès web vers la **Servlet**)
  - Configuration *web.xml*

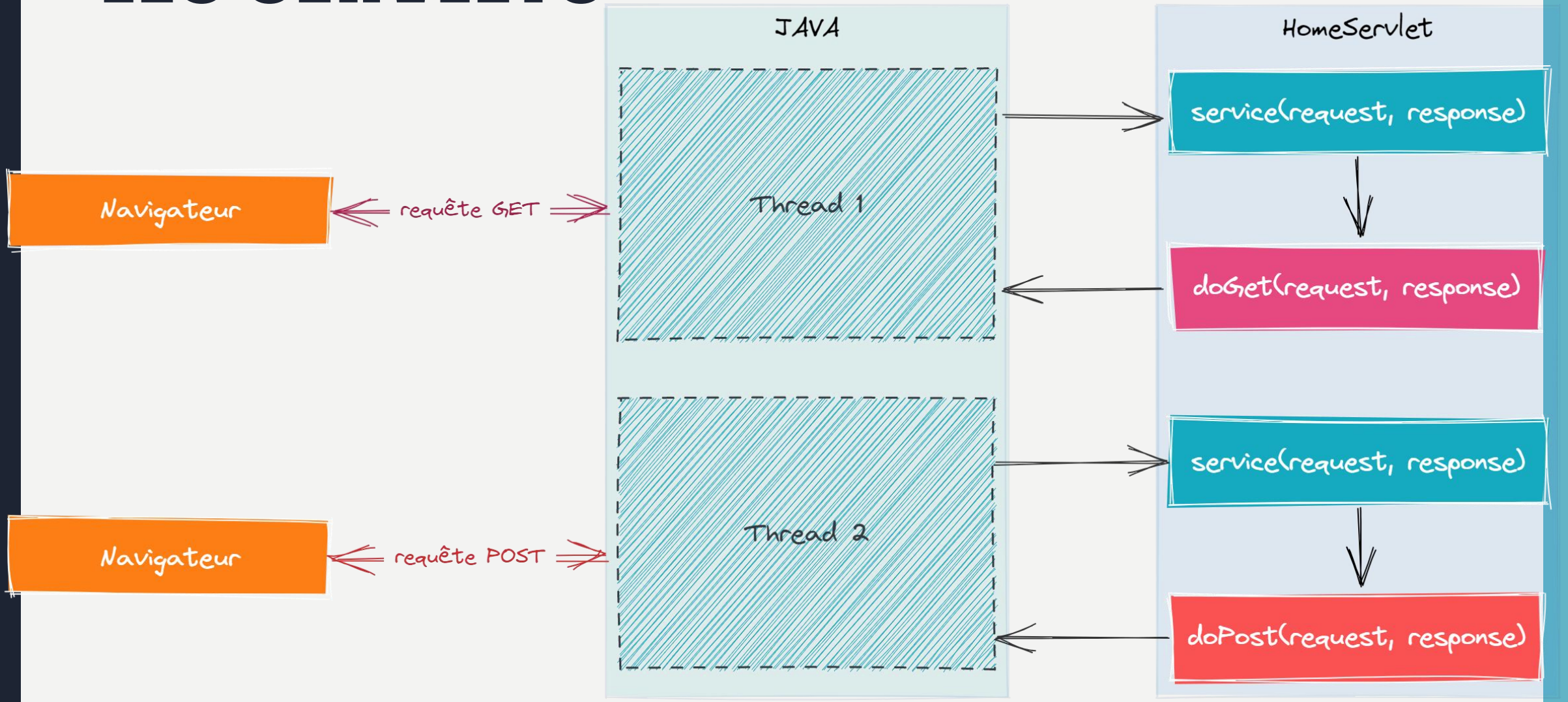
```
<servlet>
  <servlet-name>Home</servlet-name>
  <servlet-class>fr.formation.servlet.HomeServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Home</servlet-name>
  <url-pattern>/home</url-pattern>
</servlet-mapping>
```

  - Annotation

```
@WebServlet("/home")
public class HomeServlet extends HttpServlet { }
```

LES SERVLETS



La *Servlet* **HomeServlet** est mappée sur "/home"

LES SERVLETS

- `HttpServletRequest` et `HttpServletResponse` sont injectés dans les méthodes
  - `doGet`, `doPost`, ...
- `HttpServletRequest` contient des informations sur la requête **HTTP**
  - Les paramètres de la requête
  - Les attributs
  - La session utilisateur
  - *Toute autre information envoyée du client vers le serveur*

- `HttpServletResponse` contient les éléments de la réponse **HTTP** qui sera envoyée
  - C'est avec cet objet qu'on pourra modifier la réponse **HTTP**
    - Ecrire du contenu **HTML** par exemple, ou demander une redirection
  - *Tout autre élément envoyé du serveur vers le client*

LES SERVLETS

- `HttpServletRequest`
  - Récupérer un paramètre de requête
    - http://localhost:8080/mon-projet/home?username=babar

```
String myUsername = req.getParameter("username");
```

  - Ajouter un attribut de requête

```
req.setAttribute("nomVariable", "valeur");
```

  - Récupérer la session de l'utilisateur

```
Object myVariable = req.getSession();
```

  - Récupérer un attribut de la session de l'utilisateur

```
Object myVariable = req.getSession().getAttribute("variableDeSession");
```

LES SERVLETS

- `HttpServletResponse`
  - Spécifier le type de contenu de la réponse **HTTP** (contenu **HTML** dans l'exemple)

```
resp.setContentType("text/html");
```

  - Ajouter du contenu à la réponse **HTTP**

```
resp.getWriter().println("Bonjour le monde !");
```

  - Rediriger vers une autre **Servlet** ou une autre *URL*

```
resp.sendRedirect("autreServlet");
```

EXERCICE

- Créer un nouveau projet
  - Ajouter le fichier *web.xml* (Clique droit > Java EE Tools > Generate Deployment Descriptor Stub)
- Ajouter les dépendances
  - **jakarta.servlet-api** (scope "`provided`")

- Créer une **Servlet** `HomeServlet`
  - La mapper sur */home*
  - La méthode **HTTP** `GET` doit retourner un flux **HTML** « Bonjour le monde ?! »
- Exécuter ce projet sur le serveur **Apache Tomcat**

LES SERVLETS

- On distingue les paramètres des attributs
  - Les paramètres sont des chaînes de caractères envoyées par l'utilisateur (toujours dans une requête)
    - `GET` ou `POST`
  - On ne peut que lire l'information
    - `getParameter()` sur l'objet `Request`
  - Les attributs sont des objets stockés sur le serveur d'application
    - Stockés dans un scope spécifié
  - On peut lire et écrire de l'information
    - `setAttribute()` pour sauvegarder un attribut
    - `getAttribute()` pour récupérer un attribut

EXERCICE

- Modifier la **Servlet** `HomeServlet`
  - Dans la méthode GET, attendre un paramètre « username »
  - Retourner un flux **HTML** « Bonjour "le nom d'utilisateur du paramètre" »

JSP

LES VUES JSP

LES VUES JSP

- **Servlet**

```
resp.setContentType("text/html");

resp.getWriter().println("<!DOCTYPE html>");
resp.getWriter().println("<html>");
resp.getWriter().println("<head>");
resp.getWriter().println("<title>Ma première page</title>");
resp.getWriter().println("</head>");
resp.getWriter().println("<body>");
resp.getWriter().println("<p>Allô le monde ?!<p>");
resp.getWriter().println("</body>");
resp.getWriter().println("</html>");
```

JSP

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ma première page</title>
  </head>

  <body>
    <p>Allô le monde ?!<p>
  </body>
</html>
```

LES VUES JSP

- Pénible d'écrire la vue (**HTML**) dans un fichier **JAVA**
- **JSP** est une **Servlet** compilée et auto-indéxée
  - Déployée par le conteneur de **Servlet**
  - Les variables `request` et `response` sont accessibles

LES VUES JSP

- Dans une **JSP**, on peut inclure du **JAVA**, ou des instructions pour fabriquer / compiler
  - Directive

```
<%@ directive options %>
```

  - Code **JAVA** (Scriptlet)

```
<% ... %>
```

  - Expression (Impression de la valeur d'une variable)

```
<%= ... %>
```

  - Commentaire **JAVA**

```
<%-- ... --%>
```

LES VUES JSP

- 3 directives possibles
  - `page` informations relatives à la page
  - `include` identifie des fichiers à inclure
  - `taglib` indique que la page utilise une bibliothèque de balises (similaire au **XML NS**)

- Quelques exemples
  - Préciser l'encodage

```
<%@ page pageEncoding="UTF-8" %>
```

  - Ajouter une `taglib`

```
<%@ taglib uri="http://lurl-de-la-lib" prefix="leprefix" %>
```

LES VUES JSP

- On peut utiliser les Expressions Langages (EL) en **JSP** en utilisant la syntaxe suivante
  - `${ … }`

```
${ variable }
```

```
${ 5 + 5 }
```

  - Permet de lire une variable qui se trouve, **en tant qu'attribut**, dans un des scopes existants
  - Pour lire un paramètre de requête
    - Utiliser l'attribut `param`

```
${ param.nomParametre }
```

EXERCICE

- Créer une **JSP** *home.jsp*
  - Afficher « Bonjour "le nom d'utilisateur du paramètre" »

SCOPES

LES SCOPES DE VARIABLES

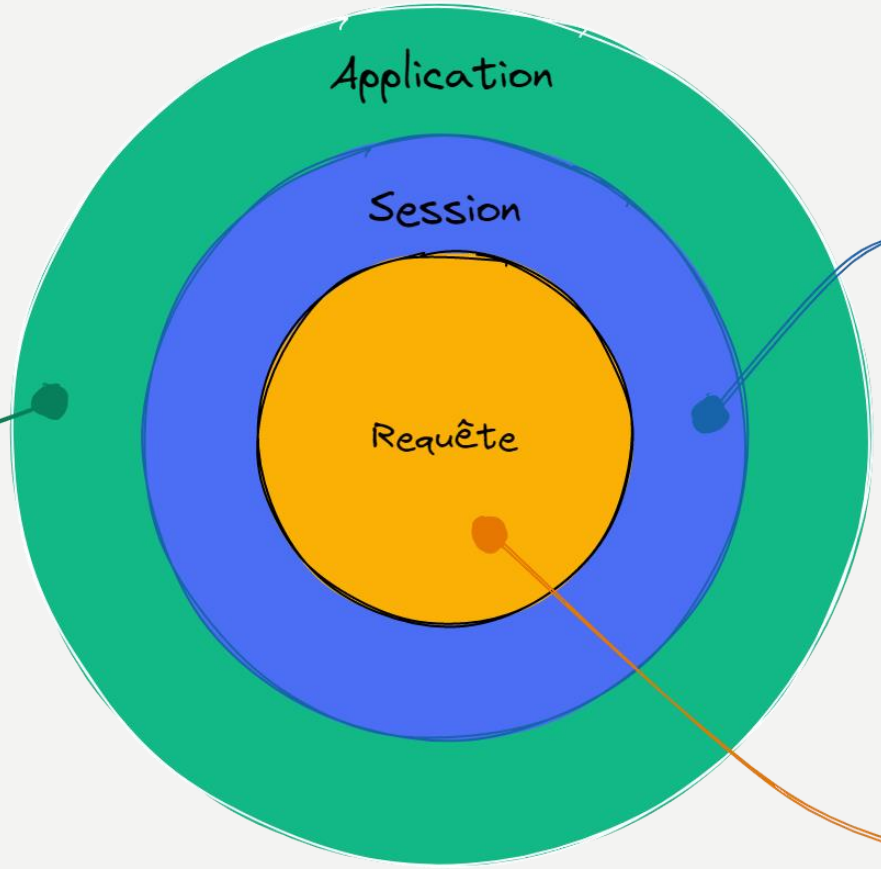
LES SCOPES DE VARIABLES

- Il existe 3 *scopes* (portées), selon le cycle de vie

Scope / Portée	Servlet	JSP
Application	`getServletContext().getAttribute("attr")`	`applicationScope["attr"]`
Session	`req.getSession().getAttribute("attr")`	`sessionScope["attr"]`
Request	`req.getAttribute("attr")`	`requestScope["attr"]`

- Implémentés grâces aux attributs
  - De contexte applicatif
  - De session
  - De requête

LES SCOPES DE VARIABLES



Démarrage de la session utilisateur
jusqu'à l'inactivité
Valeurs des variables
différentes pour chaque utilisateur

A chaque nouvelle requête HTTP

Démarrage de l'application
jusqu'à l'arrêt
Valeurs des variables
identiques à tous les utilisateurs

DÉLÉGATION

DÉLÉGUER LA VUE

DÉLÉGUER LA VUE

- Pour répondre au modèle **MVC**
  - **Servlet** joue le rôle de `Contrôleur`
  - **JSP** joue le rôle de la `Vue`

- Il faut rendre les pages **JSP** inaccessibles (les placer dans */WEB-INF/views/*)
- Il faut déléguer la requête de la **Servlet** vers la vue **JSP**

DÉLÉGUER LA VUE

- Délégation de transfert
  - Contexte de **Servlet** ➔ `Dispatcher` ➔ `Forward`
  - Transférer la suite du traitement à une autre **Servlet** ou à une vue **JSP**

```
this.getServletContext()
  .getRequestDispatcher("/WEB-INF/views/home.jsp")
  .forward(req, resp);
```

DÉLÉGUER LA VUE

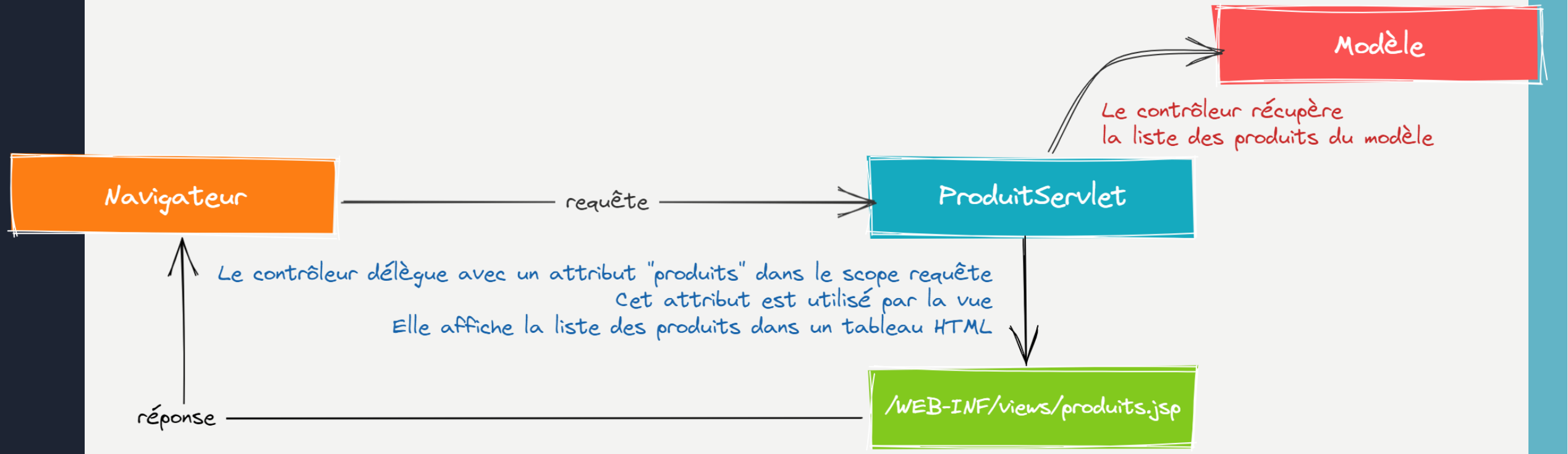
- Délégation de transmission
  - Requête ➔ Ajout d'un attribut (attention à l'ordre, l'attribut doit être inséré avant la délégation !)
    - Dans le scope de requête

```
req.setAttribute("nomUtilisateur", "Jeremy");
```

  - Lecture de l'attribut dans la **JSP**

```
<p>${ nomUtilisateur }</p>
```

DÉLÉGUER LA VUE



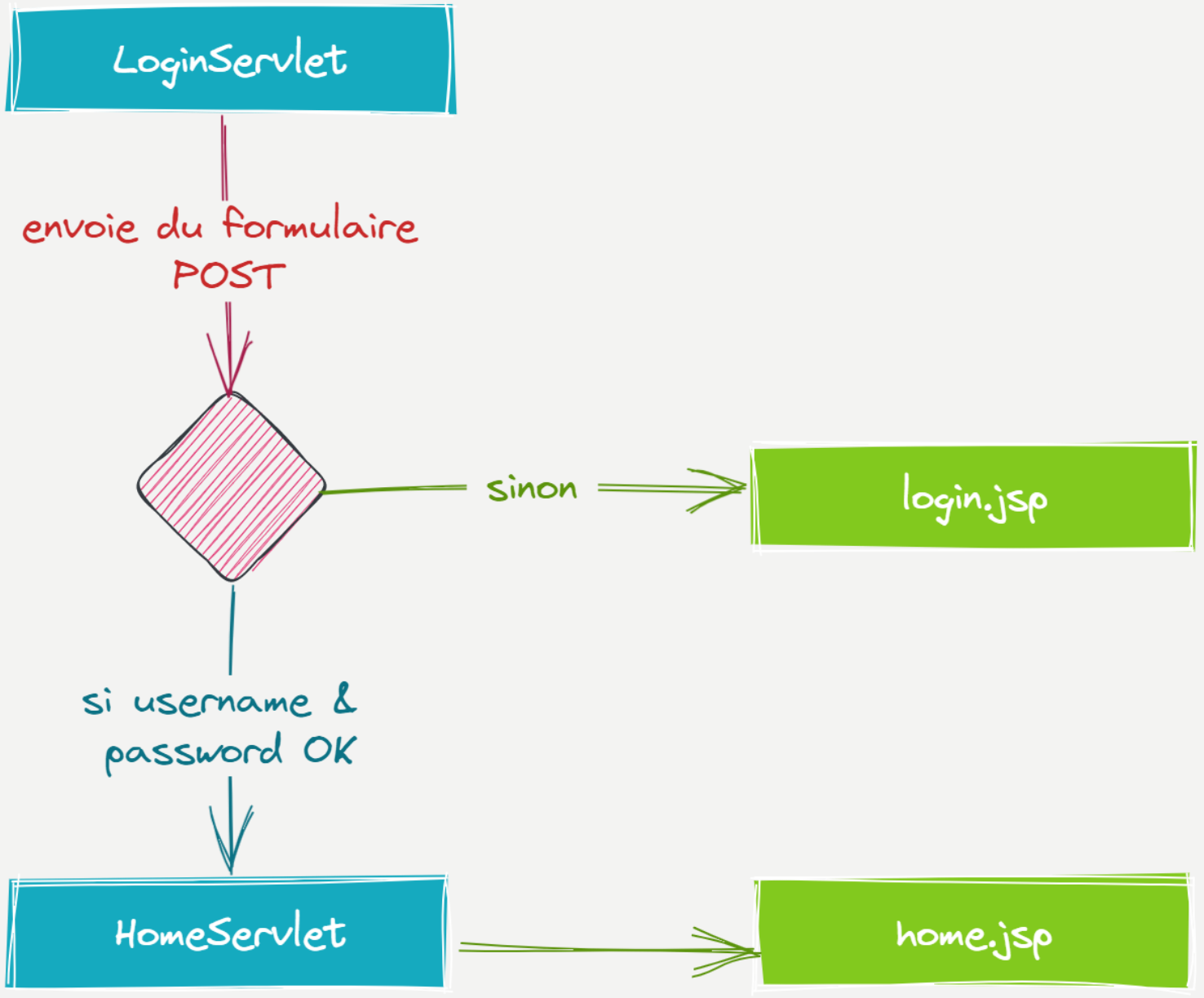
EXERCICE

- Masquer la **JSP** *home.jsp* au public
- Reprendre la Servlet `HomeServlet`
  - Déléguer vers la vue **JSP** *home.jsp*
- Créer une **Servlet** `ProduitServlet`
  - Créer un nouveau `Produit` à ajouter en tant qu'attribut de la requête
  - Afficher le nom du produit dans la **JSP**

EXERCICE

- Créer une Servlet `LoginServlet`
  - Afficher un formulaire de connexion (username / password)
  - Si `username = "Admin"` et `password = "123456$"`
    - Récupérer le `username` et le stocker dans une variable de session
    - Rediriger vers la Servlet `HomeServlet` après la saisie du formulaire (requête POST)
    - La page *home.jsp* doit afficher le nom d'utilisateur enregistré en session
  - Sinon
    - Réafficher le formulaire de connexion

EXERCICE


LoginServlet
envoie du formulaire POST
sinon
login.jsp
si username & password OK
HomeServlet
home.jsp

JSP / JSTL

DE LA LOGIQUE DANS LES VUES JSP

JSP / JSTL

- Possible d'étendre le vocabulaire **JSP** avec **JSTL**
  - **JSP S**tandard **T**ag **L**ibrary
  - Utilisation d'une `taglib`
    - Dépendances **JSTL**
      - `jakarta.servlet.jsp.jstl-api` (de `jakarta.servlet.jsp.jstl`)
      - jakarta.servlet.jsp.jstl (de `org.glassfish.web`)

JSP / JSTL

- Inclure la `taglib` dans la **JSP**

```
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
```

- Manipuler **JSTL**

```
<table>
  <c:forEach var="i" begin="0" end="7" step="1">
    <tr>
      <td>${ i }</td>
      <td>${ i * i * i }</td>
    </tr>
  </c:forEach>
</table>
```

JSP / JSTL

- **Servlet**

```
List<String> myUtilisateurs = new ArrayList<>();

myUtilisateurs.add("jeremy");
myUtilisateurs.add("anaïs");
myUtilisateurs.add("jessica");
myUtilisateurs.add("julie");

req.setAttribute("utilisateurs", myUtilisateurs);
```

JSP / JSTL

- JSP / JSTL

```
<table>
  <c:forEach items="${ utilisateurs }" var="utilisateur">
    <tr>
      <td>${ utilisateur }</td>
    </tr>
  </c:forEach>
</table>
```

JSP / JSTL

- Modifier la **Servlet** `ProduitServlet`
  - Constituer une liste de produits
  - Afficher la liste des produits dans un tableau **HTML**

EXERCICE

- Modifier *home.jsp* (utiliser `c:if`)
  - Afficher un message « vous n'êtes pas connecté »
    - Si l'utilisateur n'existe pas en session
  - Afficher le reste de la page dans l'autre cas

LISTENERS

ECOUTER DES ÉVÈNEMENTS

LES LISTENERS

- Un **Listener** écoute des évènements de l'API **Servlet** et déclenche des actions
  - Evènements Attribut (ajout & retrait)
    - Attribut d'application
    - Attribut de session
    - Attribut de requête
  - Evènements Cycle de vie (création & destruction)
    - Contexte d'application
    - Contexte de session
    - Contexte de requête

LES LISTENERS

- Chaque type de **Listener** possède son interface

Type de *Listener*	Interfaces
Attribut Application	*ServletContextAttributeListener*
Attribut Session	*HttpSessionAttributeListener*
Attribut Requête	*ServletRequestAttributeListener*
Contexte Application	*ServletContextListener*
Contexte Session	*HttpSessionListener* *HttpSessionActivationListener*
Contexte Requête	*ServletRequestListener*

- Il faut créer une classe qui implémente une ou plusieurs de ces interfaces

LES LISTENERS

- Une fois l'interface implémentée, il faut indiquer au serveur qu'il s'agit d'un **Listener**
  - Déclarer la classe dans le *web.xml* (ou via Annotation `@WebListener`)

```
<listener>
  <listener-class>fr.formation.listener.ApplicationDataInitializationListener</listener-class>
</listener>
```

- Lorsqu'un évènement a lieu, selon l'interface, la méthode implémentée sera appelée

EXERCICE

- Créer un écouteur qui, au chargement de l'application, crée une liste de produits
  - Utiliser l'interface `ServletContextListener`
  - Utiliser ce qui a été fait dans `ProduitServlet`
    - `ProduitServlet` ne doit plus créer la liste des produits
  - Afficher la liste des produits dans la page **JSP**

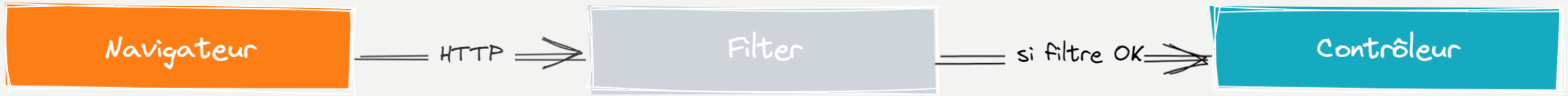
FILTERS

LES FILTRES

LES FILTRES

- Un **Filter** permet de filtrer une requête **HTTP**
- Il s'exécute avant la **Servlet**

- Il peut y avoir plusieurs **Filter** (c'est d'ailleurs souvent le cas, notamment en sécurité)
  - Ce chainage s'appelle **FilterChain**



LES FILTRES

- Il faut créer une classe qui implémente l'interface `jakarta.servlet.Filter`
  - La déclarer comme **Filter** dans *web.xml* (ou via Annotation `@WebFilter`)
- Contrairement au **Listener**
  - Le **Filter** ne s'applique que sur les requêtes dont le pattern URL correspond à l'URL demandée

```
<filter>
  <display-name>SecuriteFilter</display-name>
  <filter-name>SecuriteFilter</filter-name>
  <filter-class>fr.formation.filter.SecuriteFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>SecuriteFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

LES FILTRES

- La méthode `init` est appelée au démarrage du serveur
- La méthode `destroy` est appelée dès que le serveur s'arrête
- La méthode `doFilter` est appélée sur chaque pattern d'URL concerné
  - `chain.doFilter()` permet d'exécuter le **Filter** suivant ou la **Servlet** concernée si plus de **Filter**

```
public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
IOException, ServletException {
  HttpServletRequest request = (HttpServletRequest)req;
  HttpServletResponse response = (HttpServletResponse)resp;

  /* ... */

  chain.doFilter(request, response);
}
```

EXERCICE

- Créer un nouveau **Filter** `SecuriteFilter`
  - Si l'utilisateur n'est pas connecté
    - Refuser l'accès à toutes les pages, sauf la **Servlet** `LoginServlet`
    - Rediriger vers la page de connexion
  - ➔ Utiliser `getRequestURI()` de l'objet `HttpServletRequests`