JEE - Historique et Usage

Plan

- Présentation et historique : Une surcouche à JSE
- Conteneurs de Servlets et Serveurs d’application
- Servlets en détails
- L’approche de la programmation défensive

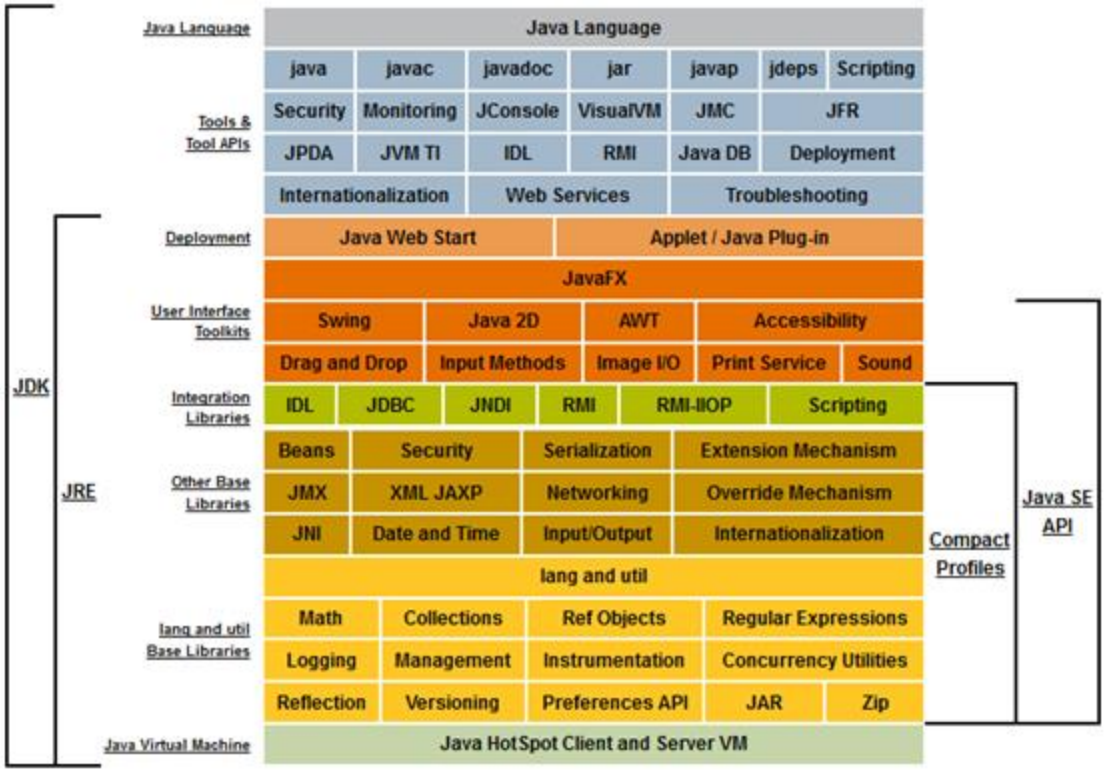
JEE

Présentation et historique

Présentation et historique

Utilisé jusqu'ici :

J2SE



Présentation et historique

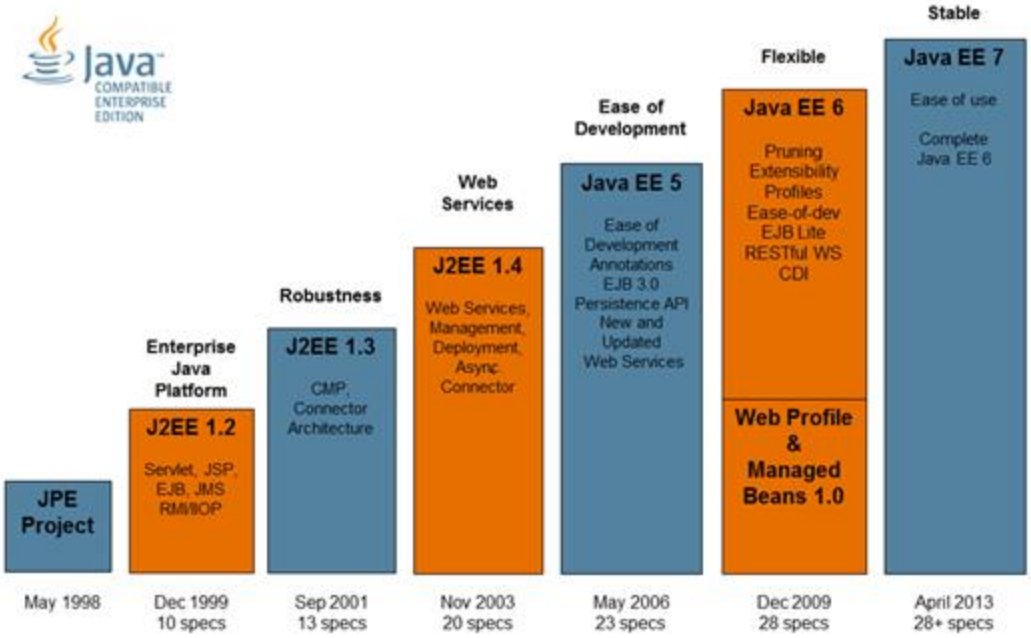
J2SE (SE pour Standard Edition) forme le kit de développement et d'exécution classique du monde Java pour les applications dites "de poste de travail" : s'exécutant sur une seule machine.

JEE (EE pour Entreprise Edition), anciennement J2EE et devenu récemment Jakarta EE est son pendant pour les applications "d'entreprise".

C'est un ensemble d'APIs pour le création d'applications dites distribuées.

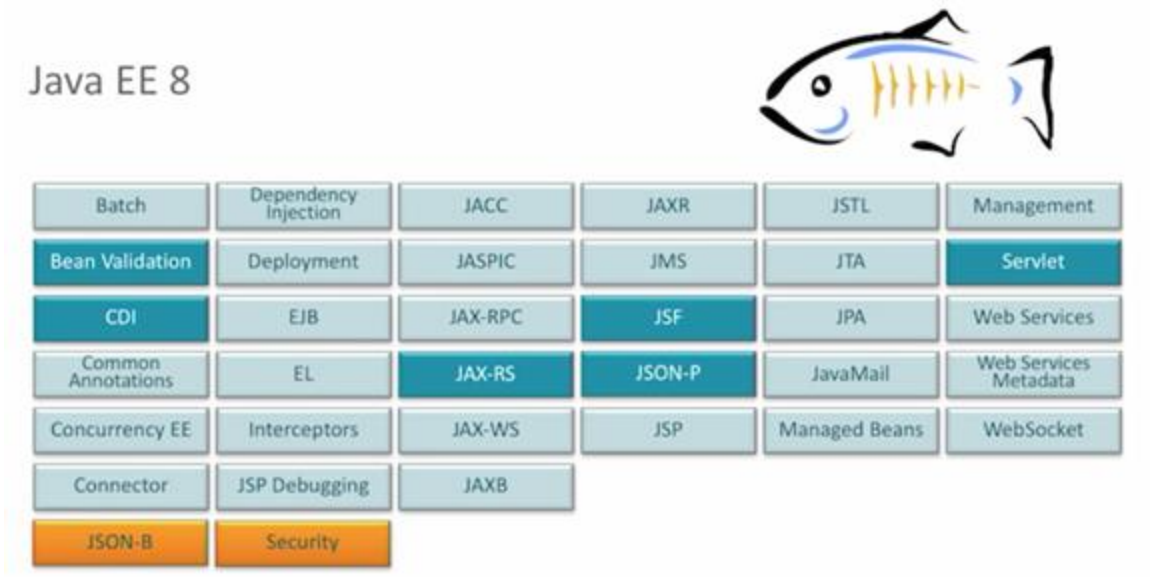
Présentation et historique

Une vie presque aussi longue que celle de Java!



Présentation et historique

Dernière release en septembre 2017 :



Présentation et historique

Principaux composants de JEE (1/4)

CDI : Context Dependency Injection. Permet l'injection de dépendances entre classes.

Bean Validation : Permet la validation des objets de type Pojo (Plain Old Java Object : propriétés, getter & setter, constructeurs)

Interceptors : Permet d'intercepter des appels et d'appliquer des comportements ou du code à ces derniers.

Concurrency EE : Sert à la manipulation des traitements parallèles, via la gestion des threads.

Principaux composants de JEE (2/4)

JPA : Java Persistance API. Permet la sauvegarde des données, notamment par l'utilisation d'ORM (Object Relational Mapping)

JTA : Java Transaction API. Gestion des Transactions au sein de l'application.

EJB : Entreprise Java Beans. Permet la création de services distribués, avec injection de dépendances, traitement des transactions, etc. Utilisé dans des contextes très particuliers.

JMS : Java Message Service. Permet d'envoyer des messages, de communiquer entre composants, de manière asynchrone

Principaux composants de JEE (3/4)

Servlet: API permettant de traiter les requêtes HTTP

WebSocket : Permet une communication TCP, utilisée pour l'HTML 5

JSF : Java Server Faces. Propose un ensemble de composants pour gérer l'IHM d'une application Web

JAX-RS: Permet d'écrire des Web Services de type REST, servant par exemple à exposer des données depuis une base de données

Principaux composants de JEE (4/4)

JSON-P: Permet de construire/parser les données représentées au format JSON

Batch Applications: Permet d'exécuter des batchs, applications de traitements de fond ne nécessitant pas d'interaction humaine.

JavaMail: Permet d'envoyer des mails

…

Il en existe d'autres, dont l'usage est à découvrir lors du besoin.

JEE

Application et serveurs

Hardware Vs Software !

Dans la suite de notre cours, nous parlons quasi exclusivement de serveurs dits “Logiciels” ou “Software”. Il s’agit de programmes dont le but est d’exécuter les applications que nous, développeurs WEB, produisons.

Les serveurs physiques, ou “Hardware” désignent les machines sur lesquelles seront exécutées nos applications et leurs serveurs applicatifs.

Les serveurs JEE

En JEE, il existe 2 principaux types de serveurs, répondent à des problématiques différentes et donc à des besoins différents :

- Conteneurs Web : Tomcat, Jetty, WebSphere, ...
- Serveur Applicatif : GlassFish, Weblogic, WildFly, ...

Les serveurs JEE

- Les conteneurs Web sont plus légers en terme d'espace disque, plus rapide au démarrage, et sollicitent moins la machine sur laquelle ils sont exécutés.

- En contrepartie, les conteneurs Web offrent moins de services qu'un serveur d'application : toutes les briques disponibles dans le JEE ne peuvent pas être utilisées.

Les serveurs JEE

- Les serveurs d'Application sont conçus pour mieux résister à la charge et pour de plus grandes applications. Ils offrent, en principe, plus de service que ne le font les conteneurs Web (JNDI, JMS, Monitoring, …)
- Les serveurs d'application contiennent un conteneur Web
- Les serveurs d'application sont plus gros, nécessitent une expertise à cause d'un paramétrage poussé, et souvent payants.

> *On n’a qu’à toujours prendre un serveur d’applications! C’est mieux et ca peut tout faire!*
>
> *-Un développeur enthousiaste*

Il n'y a pas de bons et de mauvais serveurs !

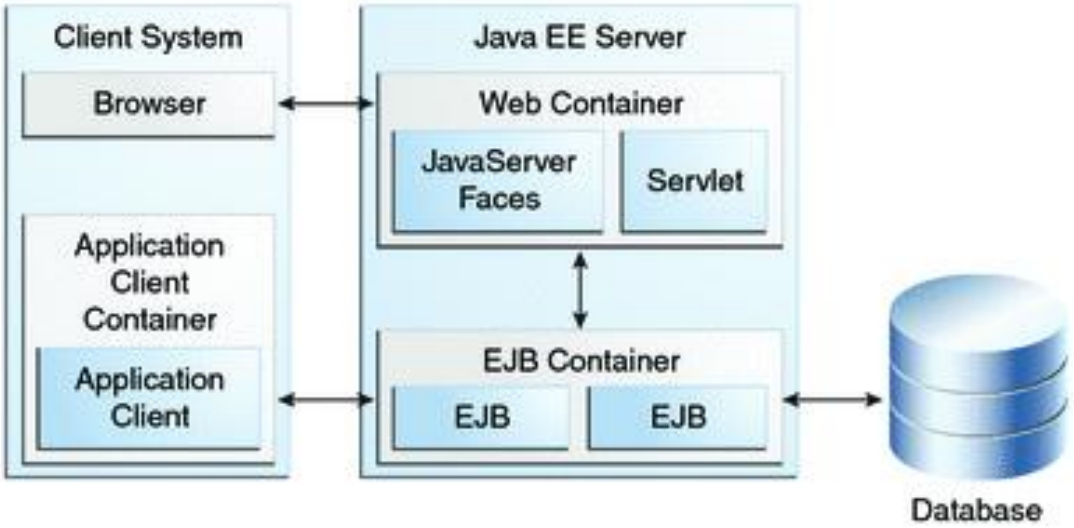
L'essentiel est de trouver celui qui correspond à votre besoin : il n'est pas toujours nécessaire de sortir l'artillerie lourde pour de petites applications.

Il est donc important d'envisager dès le début les usages et nécessités de votre application.

Il est toutefois possible de passer d'un conteneur Web à un serveur d'applications sans trop de difficultés. **Attention ! L'inverse est souvent faux!**

Conteneurs de Servlets et Serveurs d'application

- Les conteneurs Web sont des conteneurs de Servlet
- Les serveurs d'application sont des conteneurs d'EJB



Conteneurs de Servlets et Serveurs d'application

Capacité de déploiement :

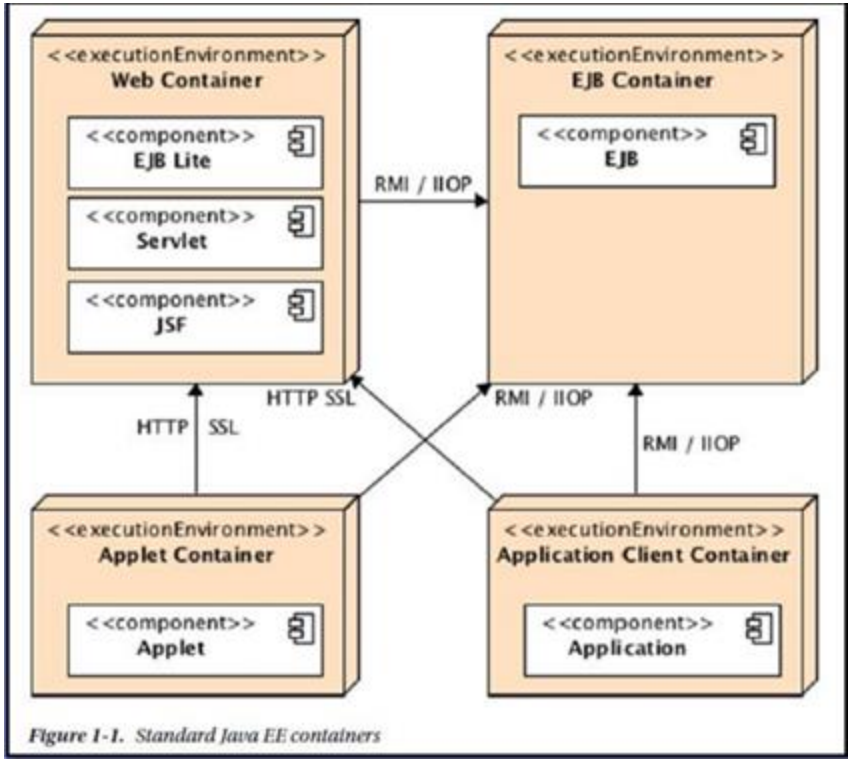


Figure 1-1. Standard Java EE containers

Conteneurs de Servlets et Serveurs d'application

Pour approfondir le sujet, rien de mieux que la doc !

https://javaee.github.io/tutorial/overview.html#BNAAW

Conteneurs de Servlets et Serveurs d'application

Plusieurs serveurs -> Plusieurs applications

Les Conteneurs de Servlets et d'EJB ne prennent pas en charge les mêmes applications !

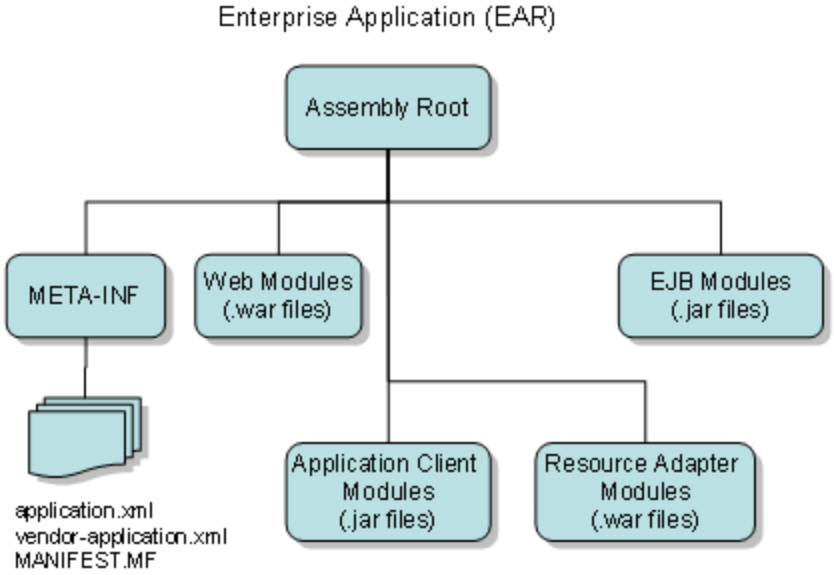
- Les **JAR** : Java ARchive, parfois exécutables. Ils servent surtout en tant qu' API, une bibliothèque de fonctions. Ils ne sont pas directement déployés sur le serveur mais contenus par les autres applications
- Les **WAR** : Web ARchive. Ils sont prévus pour être déployés dans un conteneur Web. Ils ont une structure commune, contenant notamment des JAR, et reconnue par le conteneur.
- Les **EAR** : Entreprise ARchive. Ils sont prévus pour être déployés dans un serveur d'application. Ils contiennent le plus souvent des WAR et des JAR

Structure des applications

Chacun de ces formats contient un fichier **.xml** appelé le "deployment descriptor".

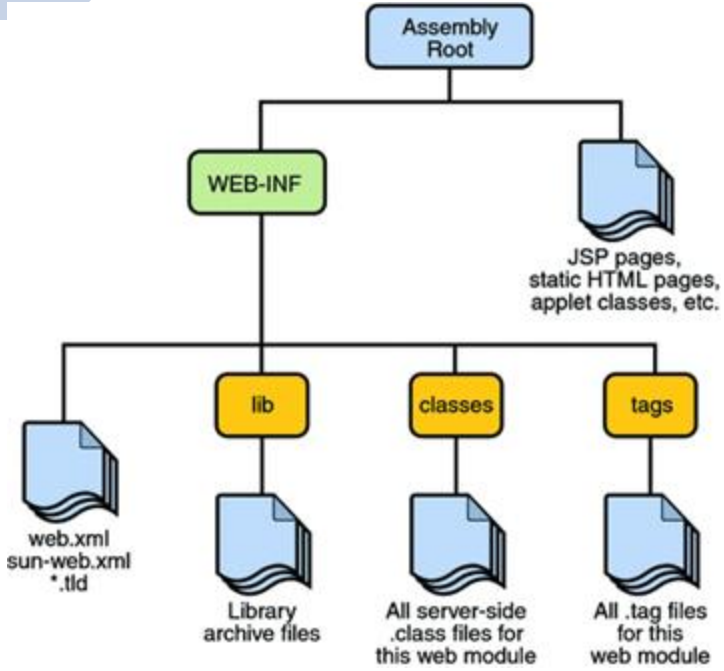
C'est grâce à ce fichier que le serveur sait comment déployer et utiliser l'application que l'on souhaite exploiter.

Dans le cas des EAR, il se nomme généralement application.xml



Structure des applications

Dans le cas des WAR, il se nomme généralement web.xml



Conteneurs de Servlets et Serveurs d'application

Synthèse

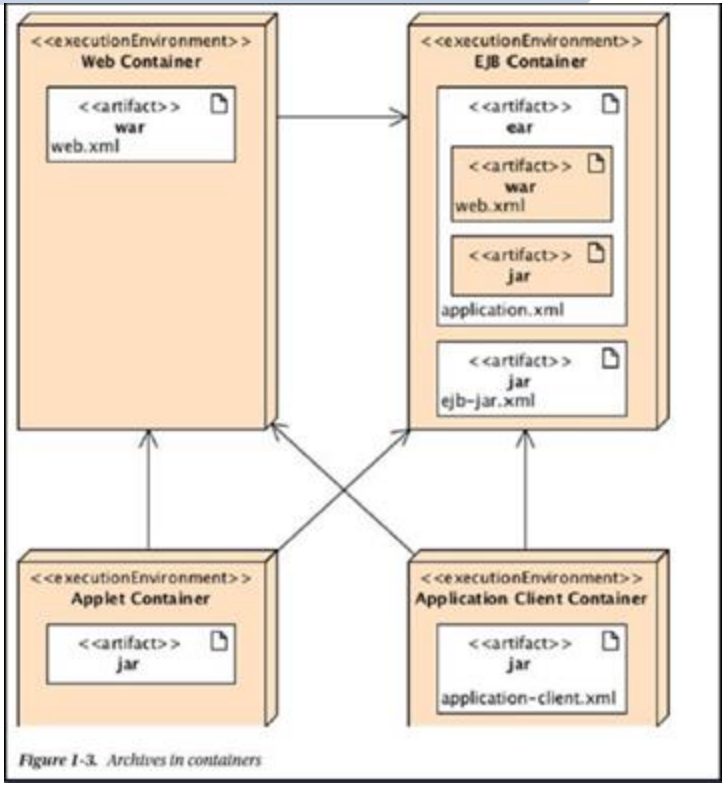


Figure 1-3. Archives in containers

Déploiement et accessibilité

L’ essentiel de notre projet utilisera par la suite un conteneur de servlet, ainsi que des applications WAR.

Une fois déployée, une application est visible sur le file system. On y retrouve la structure précédemment décrite : lib, classes, etc. Le serveur se “contente” d’extraire l’archive et d’exploiter son contenu.

Déploiement et accessibilité

Il existe plusieurs solutions pour déployer une application:

- Console du serveur d'application
- Via le file system et la capacité du serveur à détecter les éléments à déployer
- Par une API distante
- Par un plugin au sein de l'IDE
- ...

La création de l'archive à déployer s'appelle le **packaging**. Il sera vu ultérieurement.

Déploiement et accessibilité

Pour solliciter l'application, le plus simple est d'utiliser son **URL** (Uniform Resource Locator). C'est l'adresse à laquelle elle peut être appelée.

Exemple d'URL : http://127.0.0.1:8080/MonApplication

- **Http** est le protocole utilisé pour l'appel
- **127.0.0.1** correspond à l'adresse de la machine qui héberge le conteneur Web ou le serveur applicatif
- **:8080** sert à préciser le port d'écoute du serveur
- **MonApplication** est le nom choisi pour l'application déployée. Il est précisé dans le deployment descriptor.

Conteneurs de Servlets et Serveurs d'application

Derrière cette url on peut ajouter d'autres éléments, on se trouve dans le contexte applicatif: css, html, jsp, etc.

Exemple d'URL : http://127.0.0.1:8080/MonApplication/MaPage.html

Ceci est par exemple une page html qui serait déployée au sein de l'application.

Cela permet de vérifier son bon déploiement ou plus simplement son fonctionnement.

Conteneurs de Servlets et Serveurs d'application

Il est bien sur possible de déployer plusieurs applications sur un même serveur :

http://127.0.0.1:8080/MonApplication et

http://127.0.0.1:8080/AutreApplication

PEUVENT désigner deux applications différentes.

JEE

Servlets en détails

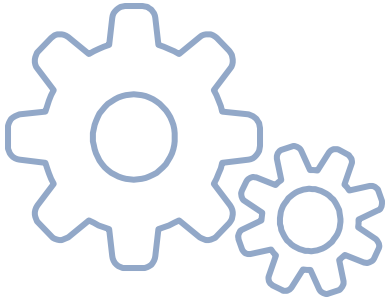
Servlets en détails

Définition

Une servlet est une classe de l'API JEE/Servlet. Elle est conçue pour être déployée et exécutée dans un conteneur de servlet (type Apache Tomcat).

Son rôle est d'écouter les requêtes HTTP entrantes et d'y répondre. Elle peut répondre par une page HTML, du JSON, du XML,...

PAS DE NEW !

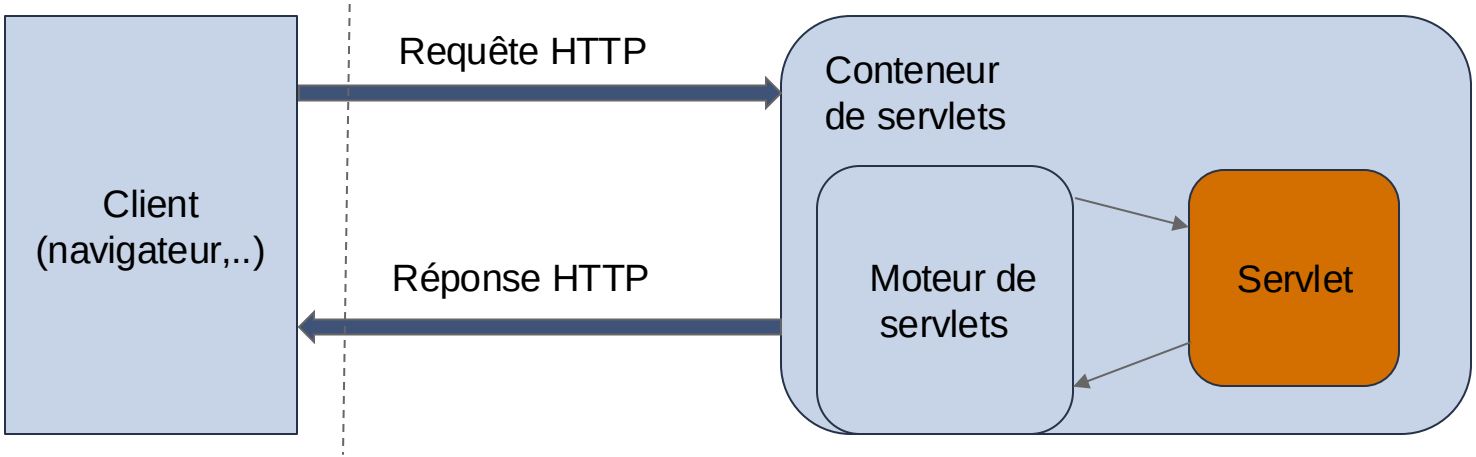


Une servlet ne doit jamais être instanciée à la main! 😐

Le serveur est seul le maître du cycle de vie d'une servlet. Il la crée lui même et se charge de la manipuler. 🙂

Servlets en détails

Echange HTTP Classique



Servlets en détails

Echange HTTP Classique

1. Le client émet une requête à destination du serveur localisé par l'url
2. Le serveur reçoit la requête et la transmet à l'application. Il l'identifie grâce à son contexte dans l'url
3. L'application transfère la requête à la servlet écoutant la partie de l'url concernée
4. La servlet traite les éléments (production de JSON, etc)
5. Le serveur renvoie les éléments sortis de la servlet au client

Servlets en détails

Lier serveur et servlet

- Les Servlets doivent hériter de **HttpServlet**
- Elles offrent les méthodes suivantes :
  - doGet pour les requêtes GET
  - doPost pour les requêtes POST
  - …
  - Il en existe une pour chaque verbe HTTP usuel.

Servlets en détails

Servlet basique

```
public class FirstServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp)
            throws ServletException, IOException {
        System.out.println("Hello World");

    }
}
```

Servlets en détails

Précisions

La classe **HttpServletRequest** permet de récupérer les éléments de la requête entrante vers le serveur.

HttpServletResponse sert à fournir les éléments de réponse au client.

Le type de retour des méthodes do…(HttpServletRequest, HttpServletResponse) est **toujours void**. Le contenu de l'objet HttpServletResponse formera les données renvoyées.

L'annotation @Override est facultative, elle apporte toutefois une garantie que l'on modifie bien ce que l'on souhaite.

Servlets en détails

Réponse

HttpServletResponse possède plusieurs méthodes permettant d'écrire la réponse.

Le cas le plus simple est d'utiliser la méthode **getWriter()** qui retourne un **PrintWriter**.

Sa méthode **write(String s)** permet de spécifier le contenu envoyé : HTML, ...

Servlets en détails

Mapping

Pour que le serveur comprenne que la Servlet écoute une certaine URL, il faut passer effectuer le mapping entre :

- Un fragment de la route ou de l'url, généralement la dernière partie.

  ET

- La classe de type HttpServlet écrite

Ce lien s'effectue de deux manières différentes.

Servlets en détails

Mapping XML

Il faut ajouter les balises comme ci dessous au fichier web.xml

```
<servlet>
  <servlet-name>FirstServlet</servlet-name>
  <servlet-class>FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>FirstServlet</servlet-name>
  <url-pattern>/hello.do</url-pattern>
</servlet-mapping>
```

Servlets en détails

Mapping XML

La seconde balise `<servlet-mapping>` sert à définir le lien entre l'URL et la servlet en elle même

`<servlet-name>` est à remplir avec le nom de la servlet précédemment défini dans la première balise

`<url-pattern>` est à remplir avec le fragment de l'url qui déterminera quel servlet utiliser.

Servlets en détails

Mapping avec annotations

Cette possibilité n'est offerte que pour les applications utilisation l'API de Servlet en version 3.0 ou supérieure.

```
@WebServlet(urlPatterns = "/hello.do")
public class FirstServlet extends HttpServlet {
```

Le code ci dessus est équivalent au mapping XML précédemment décrit. La propriété urlPatterns est à valoriser avec le fragment de l'url qui déterminera quel servlet utiliser.

Servlets en détails

Mapping XML

La première balise `<servlet>` sert à définir la servlet. C'est ce qui permet au serveur de détecter sa présence.

`<servlet-name>` est à remplir avec le nom de la servlet, il servira pour le mapping

`<servlet-class>`est à remplir avec la classe complète de la servlet, package compris. (NB: On nomme ce nom de type my.package.MyClass "fully qualified")

Servlets en détails

Scope et visibilité des éléments

Le scope est une portée, une visibilité dans laquelle des éléments sont visibles.

On s'en sert pour stocker des données ou les faire transiter entre différentes requêtes, ou même servlets.

Ils prennent la forme d'une map de type clé <-> valeur, où la clé est une chaîne de caractères. Cette paire clé <-> valeur est appelé **"Attribut"** d'un scope.

Servlets en détails

Scope et visibilité des éléments

Il existe 4 scopes, avec des usages et des portées différentes.

Nom du Scope	Durée
Application ou WebContext	Disponible pour toute l'exécution de la JVM
Session	Disponible pour la durée d'une session utilisateur
Requête	Disponible pour la durée de la requête
Page	Disponible pour la durée de l'affichage d'une page

Servlets en détails

Scope WebContext

Ce scope est accessible, et commun à toute l'application. Il est donc partagé entre tous les utilisateurs de l'application.

Il disparaît à la destruction de la JVM.

On y stocke généralement des objets complets pour les rendre accessibles entre différentes servlets (classe de calcul, etc). Voir chapitre sur Injection de dépendances.

Il est très rare, voire anormal, de devoir le manipuler soi même.

Servlets en détails

Scope Session

Ce scope est propre à une session d'un utilisateur et disparaît en même temps qu'elle.

On peut y faire figurer des éléments propres à un utilisateur qui ne doivent pas être partagés : panier de site eCommerce, heure de connexion, etc

On peut y accéder via la HttpServletRequest :

```
req.getSession().getAttribute( s: "Attribute");
```

Servlets en détails

Scope Request

Ce scope vit le temps d'un échange HTTP classique.

On peut y faire figurer des éléments à usage unique que l'on peut obtenir directement grâce à l'URL ou aux paramètres qu'elle contient.

Y figurera par exemple la description d'un article dont on connait le titre.

On la manipule directement via les classes HttpServletRequest et HttpServletResponse.

Servlets en détails

Scope Page

Ce scope vit le temps du calcul nécessaire pour le rendu d'un page HTML. C'est le plus restreint des 4. Il est notamment utilisé dans le cas de la génération de l'IHM depuis le backend, via les modules JSP et JSF.

Il sert à passer des éléments qui seraient affichés et renvoyés dans la réponse HTML.

Nous ne nous en servirons pas dans le cadre de notre projet.

Servlets en détails

Paramètres d'URL

Un paramètre de requête sert à affiner les informations que l'on récupère dans la réponse d'un appel HTTP.

Exemple : https://www.google.fr/search?q=foreach&ie=utf-8&oe=utf-8

Dans cette URL, 3 paramètres sont présents :

- "q" valorisé à "foreach"
- "ie" valorisé à "utf-8"
- "oe" valorisé à "utf-8"

Servlets en détails

Syntaxe des paramètres

Les paramètres se trouvent en fin d'URL.

Le début de la chaîne des paramètres est marqué par le caractère "?"

Il s'agit d'une Map de la forme clé <-> valeur, où la syntaxe est "clé = valeur"

Chaque paire est séparée par le caractère "&".

Ils peuvent être multi valués, auquel cas la syntaxe standard est "clé=valeur1&clé=valeur2"

Syntaxe des paramètres

Attention, quoique naturelle, la syntaxe “clé=valeur1,valeur2” n’est pas une norme

Elle n’est que tolérée!

Elle est ainsi tantôt acceptée, tantôt ignorée suivant l’API ou le serveur utilisé.

Servlets en détails

Récupération des paramètres

Dans une servlet, la récupération des paramètres se fait via la classe HttpServletRequest.

```
String parameter= req.getParameter( s: "produit");
```

Ici on récupère la valeur du paramètre dont la clé est “produit”.

Avec cette url: http://127.0.0.1:8080/Appli/search?produit=cuillere

Parameter vaudrait la chaine “cuillere”.

Servlets en détails

Stateful...

Dans le cas d'un traitement Stateful, l'application se souvient du client qui effectue la requête.

Il va chercher dans le scope Session ou dans le scope WebContext des éléments précédemment enregistrés.

Exemple: un historique de navigation ou d'appels

Servlets en détails

… Ou Stateless

Dans le cas d'un traitement Stateless, la requête contient tous les éléments nécessaires à son traitement.

Le seule scope utilisé est celui de la requête.

Exemple: une recherche

Servlets en détails

Stateless!

C’est le modèle prédominant dans les modèles actuels d’architecture généralement orientés **frontend/backend**.

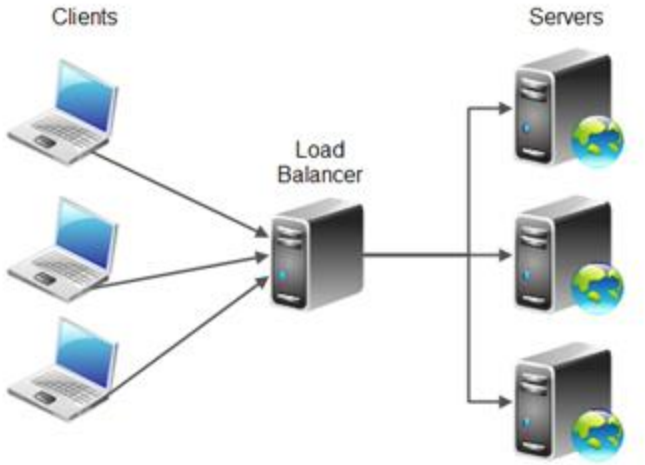
Le backend ne retient rien de la session.

C’est le frontend qui se charge de conserver les données nécessaires à la navigation

Servlets en détails

Pourquoi?

Il est fréquent de trouver plusieurs serveurs hardware et software en parallèle pour absorber la charge : on parle de scalabilité horizontale.



Servlets en détails

Si l'un des serveurs "tombe" ou n'est plus joignable, tous les clients qui l'utilisaient sont redirigés automatiquement vers un autre.

MAIS

Toutes les données sauvegardées en session ou dans le webcontext propres du serveur sont perdues.

NB : il existe des solutions pour le partage de données entre serveurs

Souvent complexes, rarement le meilleur choix

JEE

L’approche de la programmation défensive

Résiliance

Philosophie

La **Programmation défensive** est une forme de design visant à garantir la stabilité et la sécurité d'un ensemble applicatif.

Elle est axée sur l'amélioration du code source afin de

- Réduire le nombre de bugs
- Faciliter la récupération d'un comportement inattendu
- Ecrire du code réutilisable et "sûr"

Résiliance

Cadre

Pour aider le développeur dans son approche, quelques principes simples peuvent être mis en place au cours du processus de développement.

Il ne s'agit pas de vraies règles, simplement de quelques conseils aidant à la réalisation.

Certains de ces comportements sont devenus monnaie courante dans le monde du développement.

#1

Programmation par contrat

La programmation par contrat consiste, dans les grandes lignes, à établir un **contrat de dialogue** entre un appelant et un appelé.

En Java cela peut prendre la forme de la signature du méthode, d'une interface, …

On peut renforcer ces contrats par des conditions explicites d'entrée ou de sortie.

#1

Ne faites pas confiance à n'importe qui

Toutes les données venant de l'extérieur de l'application doivent être **vérifiées**. Elles peuvent potentiellement être **malveillantes** ou plus simplement entraîner des erreurs.

Le contrat ne suffit pas toujours, d'où l'ajout de préconditions.

Exemple : Une division par zéro, une durée négative, des requêtes SQL dans des champs textuels,...

#1

Utiliser les assertions

Java propose un mot clé **assert** qui permet de s'assurer qu'une condition est vraie.

Si la condition est fausse, une RuntimeException de type **AssertionError** est levée.

Si la condition est vraie, l'instruction suivante est exécutée.

NB: Dans la vraie vie, ce procédé est effectué autrement (if, framework, …)

Les **assert** sont souvent désactivés en production.

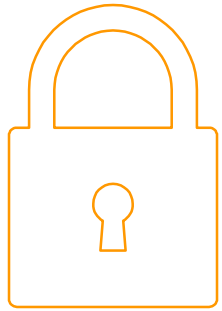
Exemple d'assertion

```
public void test(){
    assert false: "Erreur !!";
}
```

```
Picked up JAVA_TOOL_OPTIONS: -Djava.vendor='Sun Microsystems Inc.'

java.lang.AssertionError: Erreur !!
```

#2



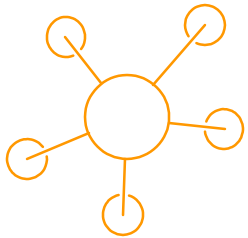
Sécuriser l'accès aux données

Beaucoup de failles applicatives proviennent de la **persistance des données**.

Déléguer cette tâche à une source de confiance, un **framework** le plus souvent, est judicieux.

Ce dernier étant en principe mis à jour régulièrement, il permet une correction rapide des failles, et sécurise le tout.

#3



Ne pas réinventer la roue

Réutiliser du code intelligent est souvent un bon moyen de gagner du temps.

Réécrire du code augmente les chances de “coder une erreur”. Il faut donc sécuriser au maximum le code existant et y faire appel autant que possible, à condition qu’il soit **fiable et testé**.

#4

Ecrire du code maintenable

Votre code doit être **lisible** facilement, comme celui d'un livre.

Cela évite les interprétations faites par d'autres développeurs, et donc réduit les risques d'erreur.

Limites

Pas gratuite

Bien évidemment, ces “règles”, notamment la 1e ont un coût sur le développement et les tests.

Il est par ailleurs fréquent de se prémunir de choses qui ne surviendront jamais. On rentre dans un système “**sur-sécurisé**”.

Par ailleurs cette surproduction de code peut faire tomber dans le vice qui amène à la “3e règle”.