ISIMA

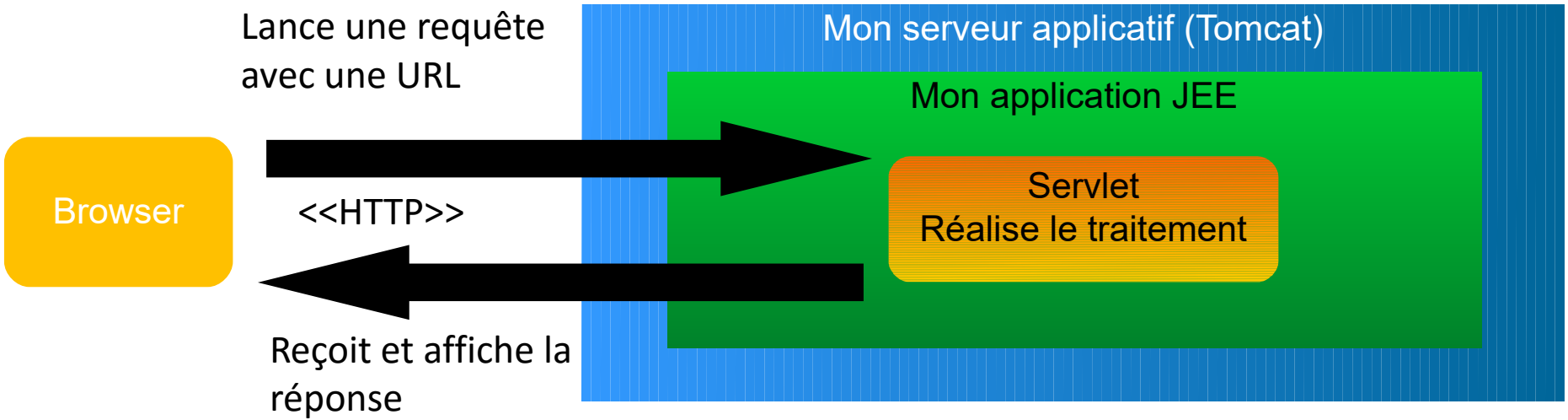
Java EE : Développer une application Web en Java

Les filters !

Rappel sur le premier cours

- La Servlet
- La servlet permet d'exécuter facilement un service Web en Java
  - Il permet de manipuler la réponse et d'envoyer un contenu défini par le serveur
  - Il est possible de récupérer des paramètres du client pour les utiliser afin d'effectuer un traitement

Un schéma récapitulatif



Limitations

- La servlet sert à réaliser un service
- Elle traite une seule requête à la fois
- Il existe une fonction de JEE qui permet d'appliquer un traitement commun à un ensemble de Servlet délimité

Le filter

- Le filter permet d'appliquer un traitement commun à plusieurs Servlet
- Pas besoin de modifier les Servlet pour rajouter les nouveaux comportements !

Pourquoi utiliser le Filter ?

- Un service commun à tous
  - Par exemple un système qui mesure le temps de traitement des Servlets
- Pour un service n'ayant pas de rapport avec ce que réalise la Servlet
  - En HTTP, il existe des informations d'entêtes (le header HTTP). Le user agent, par exemple, est l'identifiant du navigateur :
    - Pour l'afficher systématiquement, ou même l'enregistrer, il est préférable d'utiliser un Filter
    - Pour en savoir plus sur le header http : https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
- Vous n'aurez à modifier aucune de vos Servlets existantes

Configurer le filter

- Utilisation du mapping des Urls pour déclencher ou non le filtre
  - Ex, soit **/welcome/hello**, **/welcome/hi** et **/goodbye** trois Servlets, si je configure un filtre avec un mapping :
  - **/*** le filtre sera exécuté pour **/welcome/hello**, **/welcome/hi** et **/goodbye**
  - **/welcome/*** le filtre sera exécuté pour **/welcome/hello** et **/welcome/hi**
  - **/goodbye** le filtre sera exécuté uniquement pour **/goodbye**

ISIMA

Java™

ORACLE

Let's Filter

- Reprenons les 3 servlets que nous venons d'évoquer **/welcome/hello**, **/welcome/hi** et **/goodbye**
- Elles affichent simplement un message « Hello ! », « Hi ! » ou « Good bye ! »

```
@WebServlet("/goodbye")
public class GoodByeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        resp.getWriter().write("Goodbye !");
     }
}
```

MeasureExecutionTimeFilter

- Nous allons ajouter un filter qui mesure le temps de traitement de toutes les Servlets
- Il va afficher le temps sur la console pour que notre exemple soit parlant

MeasureExecutionFilter

```
@WebFilter("/*")
public class MeasureExecutionDurationFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws IOException,
ServletException {
        // utilisation de l'import static import static java.time.LocalTime.now;
        final LocalTime before = now();
        chain.doFilter(req, resp);
        final LocalTime after = now();
        final Duration duration = Duration.between(before, after);
        final HttpServletRequest httpReq = (HttpServletRequest) req;
        System.out.println("Execution of " + httpReq.getServletPath() + " took " + duration.toMillis() + "ms");
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
    }

    @Override
    public void destroy() {
    }

}
```


Déclaration en détail

- `@WebFilter("/*")`
  - Permet de déclarer le filtre
  - En paramètre de l' annotation, le filtre **/*** signifie que toutes les servlets sont concernées
- `public class MeasureExecutionTimeFilter implements Filter`
  - La classe doit implementer l'interface java.servlet.Filter
  - Celle-ci contient 3 méthodes que nous allons détailler

ISIMA

Java™

ORACLE

Déclaration en détail

```
public void doFilter(ServletRequest req, ServletResponse resp,
FilterChain chain) throws IOException, ServletException
```

- La méthode prends 3 paramètres
- **ServletRequest**, **ServletReponse**
  - Les paramètres sont du type *ServletRequest* et *ServletResponse* pour permettre l'utilisation d'autres protocoles qu'HTTP (au lieu de *HttpServletRequest* et *HttpServletResponse*)
- **FilterChain**
  - Cet objet permet, par sa méthode *doFilter()*, de continuer la chaîne d'execution de la requête
  - D'un point de vue conceptuel, il serait préférable d'avoir une méthode d'interruption plutot que de continuité. Il ne faut pas oublier l'appel sinon votre filtre bloquera toutes les requêtes !

Déclaration en détail

```
@WebFilter("/*")
```

- Permet de déclarer le filtre
- En paramètre de l' annotation, le filtre **/*** signifie que toutes les servlets sont concernées

```
public class MeasureExecutionDurationFilter implements Filter
```

- La classe doit implementer l'interface java.servlet.Filter
- Celle-ci contient 3 méthodes que nous allons détailler

ISIMA

Java™

ORACLE

Déclaration en détail

```
public void init(FilterConfig arg0) throws
ServletException
```

- Cette méthode est appelée une fois au démarrage de votre container
- Certains Framework exploite cet évènement (ex l'integration Web de Guice)
- Si vous avez un fichier web.xml vous pouvez ajouter des valeurs que vous allez retrouver dans le paramètre FilterConfig

Déclaration en détail

```
public void destroy()
```

- Evenement de destruction du filter
- Il est appelé au moment où le container arrête l'application

Comment la mesure est-elle réalisée ?

- Le détail du corps de doFilter()

```
final LocalTime before = now();
chain.doFilter(req, resp);
final LocalTime after = now();
final Duration duration = Duration.between(before, after);
```

- Pour effectuer les mesures, utilisons la nouvelle API de gestion du temps de Java 8 (inspirée du projet JodaTime)
- Voir LocalTime et Duration
- A noter l'appel du doFilter() entre le before et le after

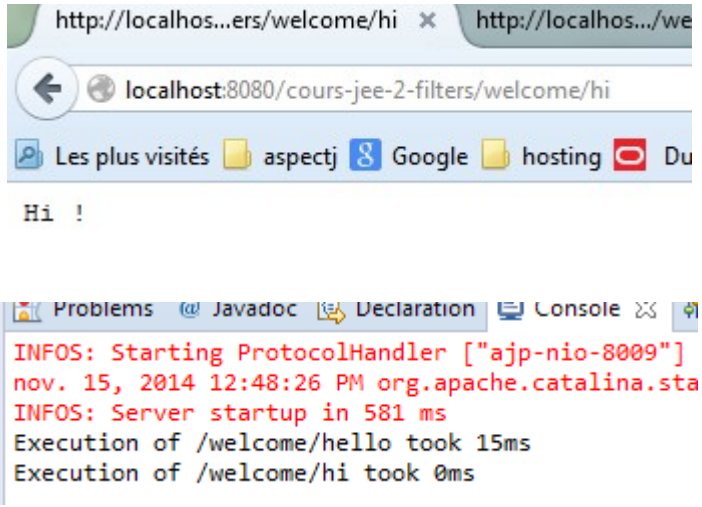
Comment la mesure est-elle réalisée ?

- Le détail du corps de doFilter()

```
final HttpServletRequest httpReq = (HttpServletRequest) req;
System.out.println("Execution of " + httpReq.getServletPath()
          + " took " + duration.toMillis() + "ms");
```

- On cast le ServletRequest en HttpServletRequest pour accéder à la méthode *getServletPath()*
- Ensuite un System.out affiche le temps de traitement

Testons



- Afficher des éléments sur la sortie standard est une mauvaise pratique. Elle est juste utile dans notre exemple
- Dans la réalité nous sauvegardrions nos mesures en base de données, dans des fichiers ou d'autres formes plus exploitables

Ecrivons un filtre limité

- Comme notre système est exigent sur la politesse, nous pouvons appeler **/welcome/hello** et **/welcome/hi** uniquement si un paramètre nick est fournit. Le système ne salut que les personnes qui se sont présentées !
  - Si le nick n'est pas transmis, nous emmètrerons une erreur 401 spécifiant un problème d'authentification.

NickMandatoryFilter

```
@WebFilter("/welcome/*")
public class NickMandatoryFilter implements Filter {

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws
IOException, ServletException {
        final HttpServletResponse httpResp = (HttpServletResponse) resp;
        final String nick = req.getParameter("nick");
        if (nick == null) {
            httpResp.sendError(401, "Je ne dis pas bonjour aux inconnus");
        } else {
            chain.doFilter(req, resp);
         }
     }
…
}
```

Déclaration en détail

```
@WebFilter("/welcome/*")
```

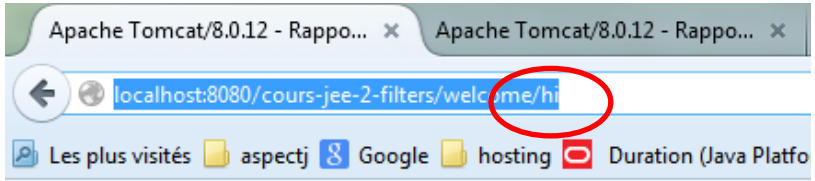
- En paramètre de l' annotation, le filtre **/welcome/*** signifie que seuls les servlets dont le chemin commence par **/welcome/** sont concernées

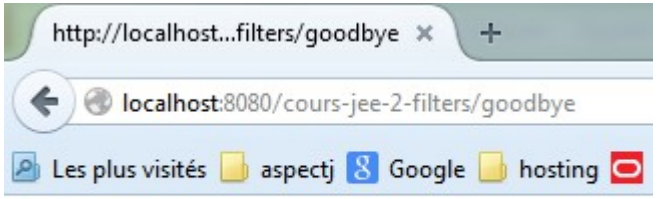
```
httpResp.sendError(401, "Je ne dis pas bonjour aux inconnus");
```

- Une erreur HTTP standard 401 pour signaler que la ressource n'est pas autorisée

ISIMA Java ORACLE

Testons





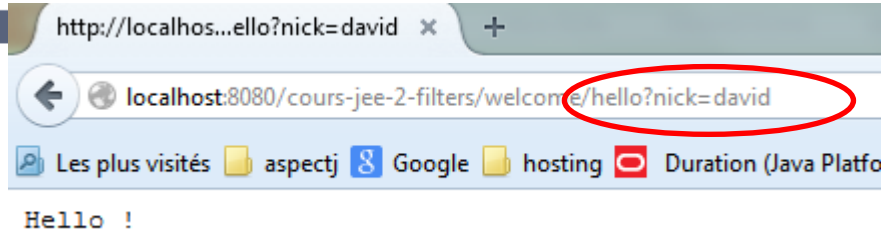
Etat HTTP 401 - Je ne dis pas bonjour a

type Rapport d''état

message Je ne dis pas bonjour aux inconnus

description La requête nécessite une authentification HTTP.

Apache Tomcat/8.0.12



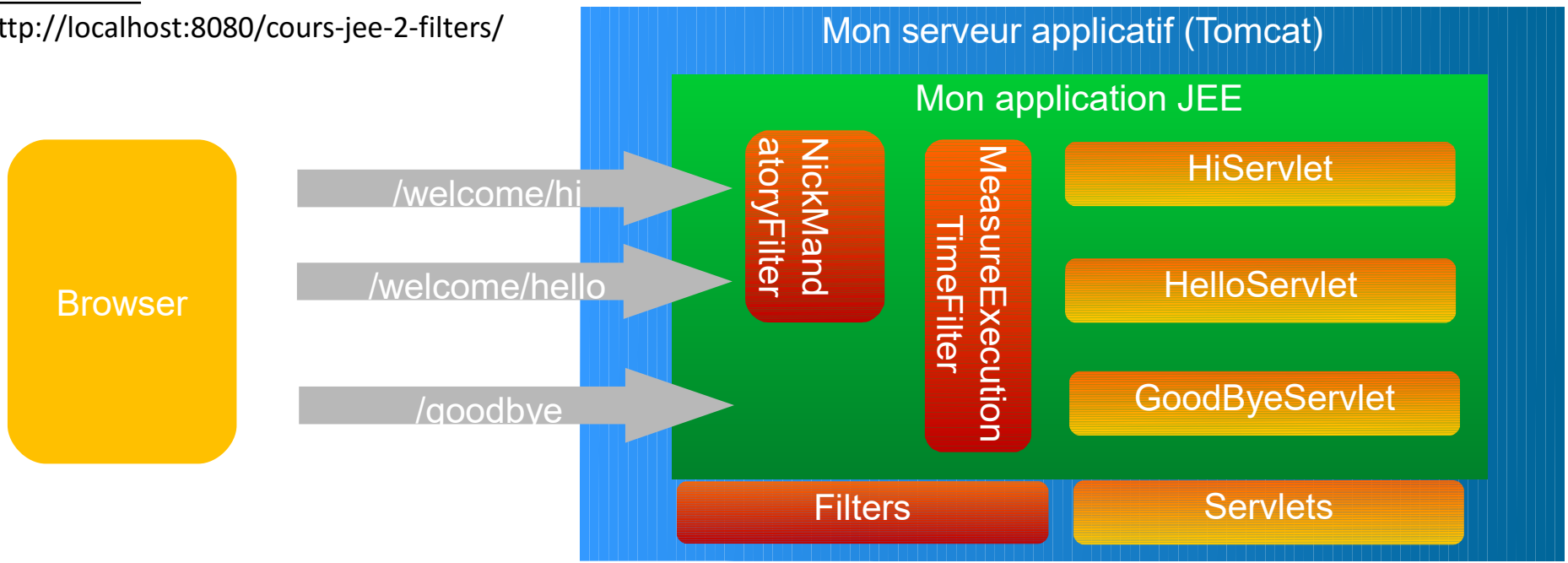
ISIMA

Java™

ORACLE

Notre exemple, en schéma

Url de base :
http://localhost:8080/cours-jee-2-filters/

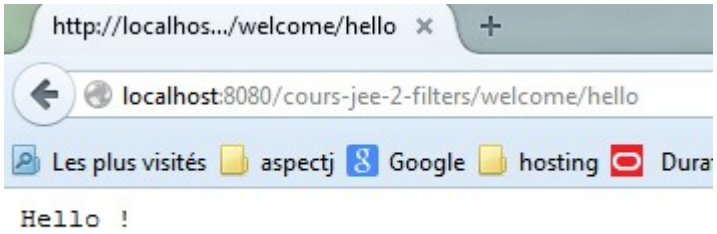


Une dernière modification sur le pattern

- Essayons un autre pattern
- Essayons d'utiliser un pattern **/w***
- Si ce pattern fonctionne, nous devrions obtenir le même résultat que précédemment

```
@WebFilter("/w*")
public class NickMandatoryFilter implements
Filter
```

Testons



- Le filtre n'est plus appliqué !
- Le joker * ne peut s'appliquer qu'après un /
- C'est une limitation à connaître qui vous évitera certains maux de tête
- Une autre limitation : avec les annotations, il est impossible de spécifier l'ordre d'éxécution des filtres

Servlets et Filters doivent être ThreadSafe

- Attention les ressources utilisées par les Servlet et les Filters doivent être ThreadSafe
- En effet, une seule instance de Servlet ou de Filter est utilisé par toutes les applications
- Dans une application Web, chaque requête est exécutée par un Thread
  - En conséquence, si deux requêtes arrivent en même temps sur un filtre, il peut y'avoir des conséquences imprévus !

Exemple de ThreadSafe

- Un petit cas de test

```
@WebFilter("/*")
public class CounterFilter implements Filter {

    private long counter;

    @Override
    public void doFilter(ServletRequest req, ServletResponse rep, FilterChain chain) throws IOException,
ServletException {
        counter++;
        System.out.println("Nombre de requete " + counter);
        chain.doFilter(req, rep);
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
        counter = 0;
    }
…
}
```

Faisons un test de charge

- Les tests de charge vont plus loin que les tests de bon fonctionnement et permettent de mettre en avant certaines faiblesse d'un système
  - Fuite mémoire
  - Problèmes de performances
  - Bugs
- Généralement, le meilleur moyen est de s'appuyer d'outils comme Jmeter
- Dans notre cas, nous allons créer une classe de test

Que voulons nous faire ?

- Nous allons vérifier que notre application gère les requêtes multi-clientes
- Nous allons appeler une url couverte par le filtre
  - http://localhost:8080/cours-jee-2-filters/hi?nick=loadTest
- Le filtre doit compter exactement le nombre d'appel que nous allons effectuer dans notre test de charge

Comment écrire un code en //

- Quelques règles à respecter
- Depuis le début Java est vendu comme étant un langage ou la programmation // est facile
  - Les Thread en Java 1.0
  - Les Executors en Java 1.5
  - Les Parallel Stream en Java 1.8
- Nous allons utiliser ces derniers. Notre code doit etre découpé en 2 phases :
  - Dans la première il faut créer les tâches à executer
  - Dans le deuxieme temps on execute les tâches

Tests de charge

```
public class TestCharge {

    public static void main(String[] args) {
        final int nombreAppel = 10000;
        final List<Runnable> tasks = new ArrayList<>();

        // Cette instruction est l'équivlant d'un for int i = 0; i< max en java8
        // Pour chaque element on ajoute un element
        IntStream.range(0, nombreAppel).forEach((range) -> tasks.add(createCallHi(range)));

        // Cette instruction est un raccourci pour executer dans un thread
        // chaque fonction. Le gain en de code est très important
        tasks.parallelStream().forEach(e -> e.run());
    }
}
```


ISIMA

Java™

ORACLE

Détaillons notre fonction

- ```
  final int nombreAppel = 10000;
  ```
  - Le nombre d'appel que nous allons effectuer
- ```
  final List<Runnable> tasks = new ArrayList<>();
  ```
  - La liste des fonctions à appeler en // sera stockée dans cette liste
  - Runnable est une interface avec une methode et pas de paramètre utilisée dans l'API Thread de Java (depuis la version 1.0 d'ailleurs)

Détaillons notre fonction

- Première phase, créons nos tâches

```
IntStream.range(0, nombreAppel).forEach((range) ->
tasks.add(createCallHi(range)));
```

- Cette instruction remplace en Java 8 l'equivalant de ce bout de code

```
for (int i = 0; i < nombreAppel; i++) {

    tasks.add(createCallHi(i));

}
```

- Nous verrons la méthode createCallHi par la suite

ISIMA

Java™

ORACLE

Détaillons notre fonction

- Deuxième phase, on exécute

```
tasks.parallelStream().forEach(e -> e.run());
```

- Ce code permet d'executer chaque appel de e.run() dans un thread
- Java s'occupe de créer le pool de thread et de le gérer sans que nous nous en préoccupions

Dernière fonction

- createCallHi

```
    private static Runnable createCallHi(int range) {
        System.out.println("Creating caller of range " + range);
        return () -> {
            try (InputStream is = new URL("http://localhost:8080/cours-jee-2-filters/welcome/hi?
nick=loadTest").openStream()) {
                System.out.println("Executing call " + range);
                is.read(new byte[50]);
            } catch (final Exception e) {
                System.err.println("Call of range " + range + " failed");
                e.printStackTrace();
             }
         };
    }
```

Détail de la fonction

- Syntaxe Java 8

- `return () -> {}`
  - Définis le corps de la méthode
  - Elle ne prends pas de paramètres
  - {} le corps à écrire

```
try (InputStream is = new
URL("http://localhost:8080/cours-jee-2-
filters/welcome/hi?nick=loadTest").openStream()) {
```

  - Le nouveau try() de Java 7
  - Il s'occupe de fermer les connexion tout seul
  - Pas besoin d'appeler is.close() et de gérer des cas complexes

Détail de la fonction

- Le corps pour la route

```
is.read(new byte[50]);
```

- Permet simplement de lire la réponse du serveur
- Dans notre cas ce sera toujours « hi »

```
catch (final Exception e) {

    System.err.println("Call of range " + range + " failed");
```

- Recupère l'erreur et affiche un stacktrace

Executons !

- Avec 5 appels
  - Nos traces coté client
- Creating caller of range 0
- Creating caller of range 1
- Creating caller of range 2
- Creating caller of range 3
- Creating caller of range 4
- Executing call 3
- Executing call 2
- Executing call 1
- Executing call 4
- Executing call 0

- Coté serveur
  - Nombre de requête 2
  - Nombre de requête 3
  - Nombre de requête 3
  - Nombre de requête 2
  - Nombre de requête 4
- Oops : au lieu d'afficher 5 appels, nous avons 2 fois 3
- Le décompte est mauvais
- Notre filter ne gère par correctement la parallélisme !!!

ISIMA

Java™ ORACLE

Modifions notre filter

- Les objets Atomic, des objets très puissant !
- En Java 5, ont été introduit les objets Atomic
- Ils permettent d'utiliser des types sûrs sans utiliser de bloc synchronized, ils sont donc très performant
  - Les blocs synchronized ne peuvent être utilisé que par un Thread à la fois ! Si il y'a de nombreux thread en même temps cela peut-être très lent
  - Les opérations synchronisées doivent donc être très courte !
- Remplaçons notre long par un AtomicLong

CounterFilter 2 !

```
@WebFilter("/*")
public class CounterFilter implements Filter {

    private AtomicLong counter;

    @Override
    public void doFilter(ServletRequest req, ServletResponse rep, FilterChain chain) throws IOException,
ServletException {
        System.out.println("Nombre de requete " + counter.incrementAndGet());
        chain.doFilter(req, rep);
    }

    @Override
    public void init(FilterConfig arg0) throws ServletException {
        counter = new AtomicLong(0);
    }
    …
}
```

Relançons notre test !

- Nombre de requête
  - Nombre de requete 1
  - Nombre de requete 3
  - Nombre de requete 2
  - Nombre de requete 4
  - Nombre de requete 5
- Comme vous pouvez le voir le nombre est bon
- Le 3 et le 2 sont affichés dans le désordre
  - C'est un comportement tout à fait normal
- Voyons voir si notre programme tiens une charge de 10000 :
  - Creating caller of range 9999
- Coté serveur voyant le décompte :
  - Nombre de requete 9998
  - Nombre de requete 9999
  - Nombre de requete 10000
- Parfait, ça fonctionne. Notre filtre tiens la charge !
- Le programme s'exécute très vite !

En résumé

- Le filter
  - Permet d'effectuer des traitements avant et après l'exécution de la Servlet.
    - Contrôles
    - Modifications d'informations
    - Etc…
  - Ces traitements peuvent s'appliquer à une, plusieurs ou toutes les Servlets
  - Faites attentions aux ressources partagées : elle peuvent créer des anomalies !