ISIMA

Java EE : Concevoir une application Web

Durée de vie des objets !

Previously in cours de JEE

- Un résumé
- Nous avons étudié
  - Les Servlets, les filters
  - La présentation avec un framework type bootstrap
  - Ajax et les interfaces dynamiques avec JQuery
- Toutefois nous n'avons étudié que des problèmes « isolés »
  - Comment pouvons nous créer une application Web complète ?
  - Comment pouvons nous agencer les objets et répondre aux problématiques du Web ?

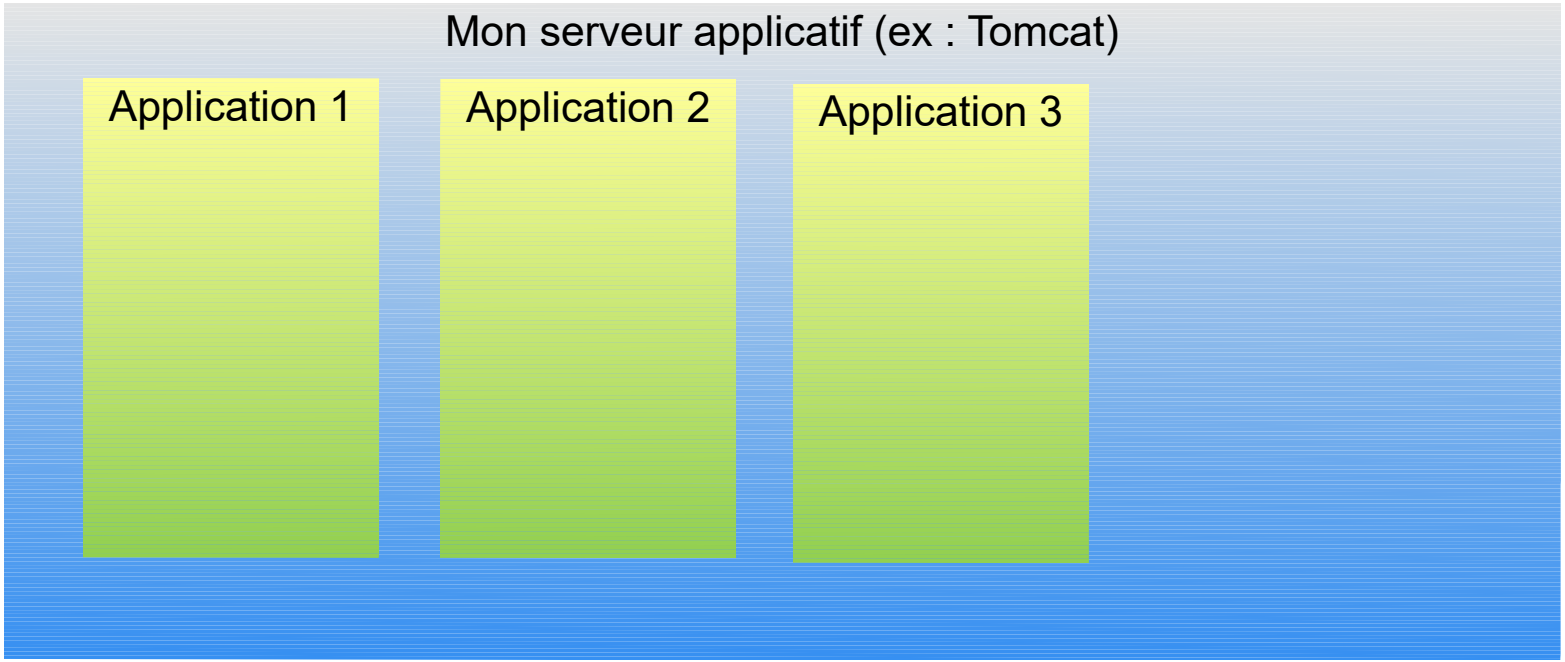
Revenons sur le serveur applicatif

- Maintenant que nous sommes habitués à le manipuler
- Le serveur applicatif est considéré comme un container lourd
  - Par opposition aux containers légers types Spring, Guice ou Pico
  - A noter que JEE contient un container léger CDI
- Le serveur applicatif nous apporte donc la possibilité de « contenir nos applications » et par prolongement de « contenir les objets de nos applications »
- Voyons ce que cela signifie

Reprenons la base

- Le SA
- Le SA contient des applications
  - Chacune indépendante les unes des autres
    - Elles peuvent avoir leurs propres bibliothèques, leurs propres dépendances
  - Elles ont leurs propres durées de vies
    - Elle peuvent être relancées indépendamment
    - Redémarrer le serveur d'application relance toutes les applications

Le serveur applicatif



ISIMA

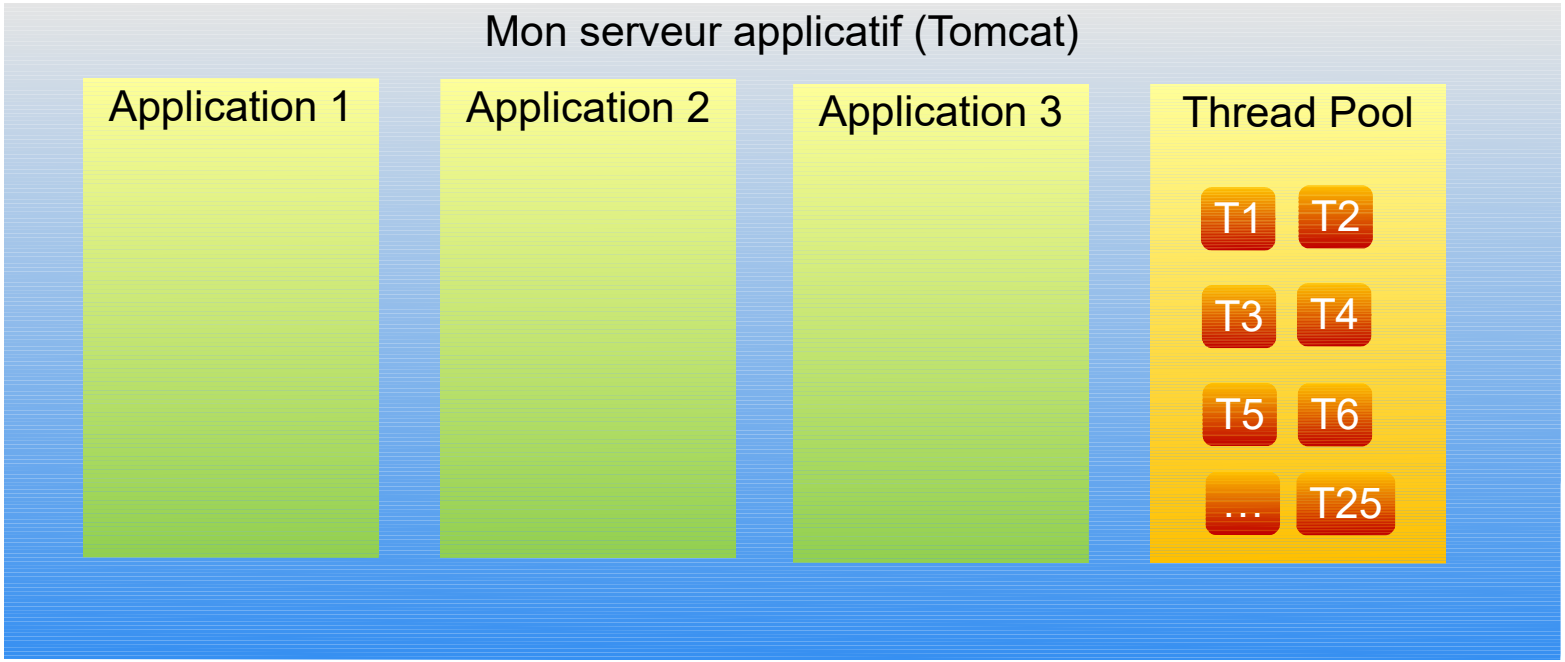
Java™

ORACLE

Les Threads

- Ze pool
- Pour pouvoir exécuter les applications le processus Tomcat créé des Threads
- Il est possible de paramétrer le nombre de Threads en attente
- Le pool est partagé entre les applications
  - http://tomcat.apache.org/tomcat-8.0-doc/config/executor.html

Le serveur applicatif



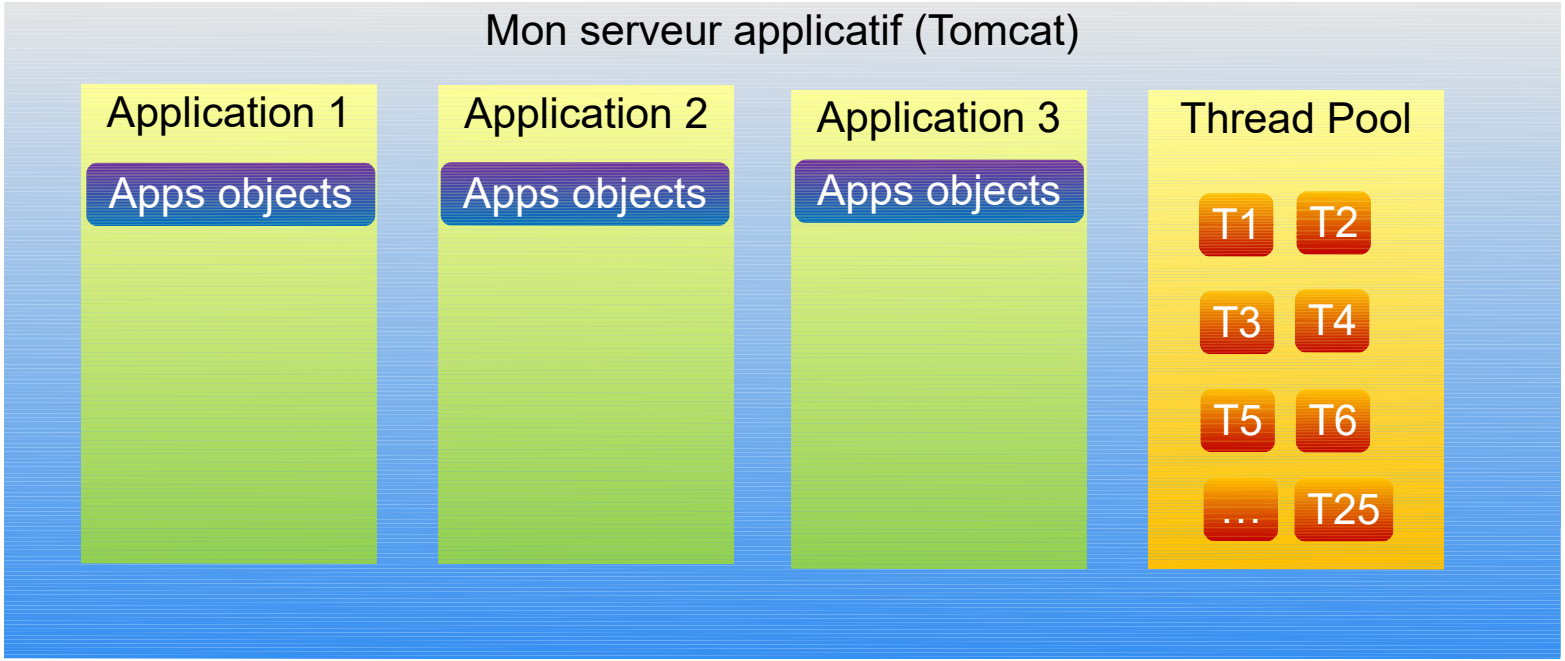
Un mot sur les Thread

- Il y'a des Thread réservés par Tomcat pour l'administration et la réception des requêtes
- Les Threads sont là pour traiter les requêtes et peuvent être mis en attente si la file d'exécution est pleine
  - Ou une erreur 503 est renvoyée
- L'exécution d'une requête à un instant T revient à prendre un Thread, exécuter la requête puis la rendre au pool pour réutilisation
  - Attention à l'usage de l'objet ThreadLocal ! Il faut bien libérer les objets à la fin

Durée de vie de notre application

- Le cycle de vie de l'application
- Le cycle de vie est assez basique
  - Le serveur démarre
  - L'application démarre
  - Le serveur dirige les requêtes vers l'application
  - L'application peut-être arrêtée et redémarrée à part
- Il existe une durée de vie (ou scope) **applicative**
  - Le programmeur peut positionner des objets afin qu'ils soient conservés pendant toute l'exécution de l'application

Le serveur applicatif

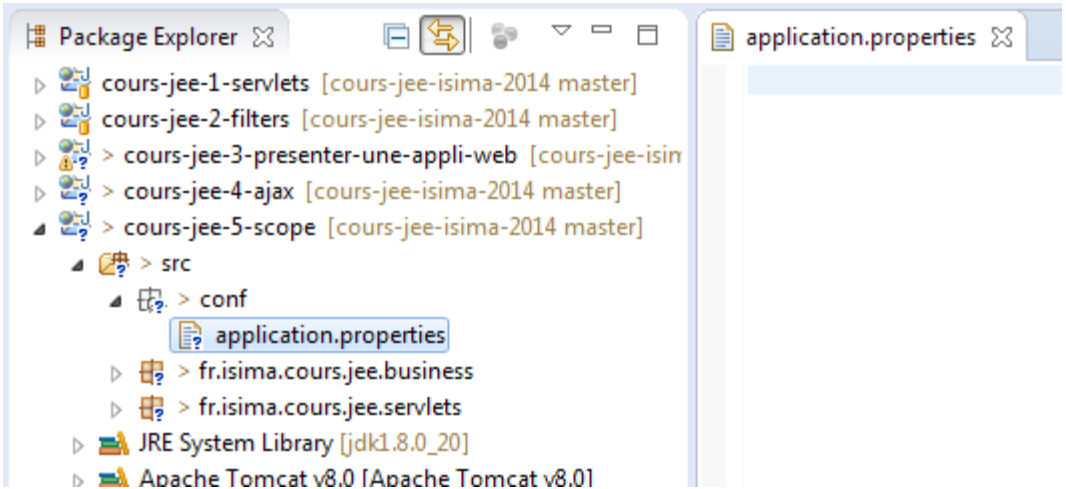


Comment sauvegarder des objets de durée de vie applicative ?

- Reprenons notre exemple d'heure courante
- L'application utilise un format de date
  - Ce format de date peut-être amené à changer en fonction des besoins des clients
- Créons un fichier de configuration application.properties
  - Les fichiers properties sont des fichiers standards Java
  - Ils sont couramment utilisés et sont suffisants pour des données non structurées (sinon il faut opter pour du xml, du json, du yml…)

Le fichier sous Eclipse

- Le répertoire de création est « conf ». C'est une convention bien utile



Ajoutons une propriété

- Sous les fichiers properties, il est possible d'associer à une clé une valeur sous cette forme
  - Cle=valeur
- Le commentaire commence par #
- Le contenu du fichier :

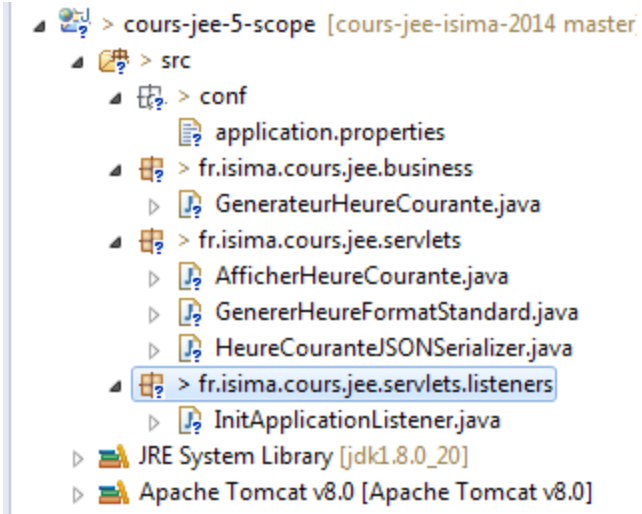
```
#Le format de date utilise par le generateur d'heure courante
format.date.heure.courante='le' dd/MM/yyyy ' à ' HH:mm:ss
```

Qui va lire le fichier de configuration ?

- Il faut écouter le démarrage de l'application
- JEE prévoit de pouvoir écouter des évènements liés à la vie de l'application !
- Pour cela nous allons utiliser « un listener »
- Le ServletContextListener est informé de l'arrêt et de la relance du serveur d'application

InitApplicationListener

- Créons la classe dans un nouveau package



A quoi ressemble-t-elle ?

```
@WebListener
public class InitApplicationListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        final Properties props = new Properties();
        try {
            props.load(InitApplicationListener.class.getResourceAsStream("/conf/application.properties"));
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException("Unable to start application. Can't read application.properties file " +
e.getMessage(), e);
         }
        sce.getServletContext().setAttribute("application-configuration", props);
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {

    }

}
```

La classe en détail

- `@WebListener`
  - Signale au container que la classe suivante est un listener

```
public class InitApplicationListener implements
ServletContextListener
```

  - L'interface ServletContextListener correspond à l'écoute d'évènements au démarrage et à l'arrêt du serveur

```
sce.getServletContext().setAttribute("application-
configuration", props);
```

  - Enregistrement des propriétés dans une map (cle=>valeur) dont la durée de vie est égale à celle de l'application (scope applicatif)

ISIMA Java ORACLE

Intégrons la modification dans notre application

- Modifions le générateur
- Il demande maintenant un format en paramètre du constructeur

```
public class GenerateurHeureCourante {

    private final String dateFormat;

    public GenerateurHeureCourante(String dateFormat) {
        this.dateFormat = dateFormat;
    }

    public String nowWithStandardFormat() {
        return DateTimeFormatter.ofPattern(dateFormat).format(LocalDateTime.now());
    }
}
```

Modifions les Servlet

- Une classe abstraite pour mutualiser

```
abstract class AbstractServletWithGenerateurHeure extends HttpServlet {

    final protected GenerateurHeureCourante creerHeureCourante() {
        final Properties properties = (Properties)
                getServletContext().getAttribute("application-configuration");
        if (properties == null) {
            throw new IllegalStateException("Impossible de charger la configuration avec la
cle application-configuration");
        }
        return new
                GenerateurHeureCourante(properties.getProperty("format.date.heure.courante"));
    }
}
```

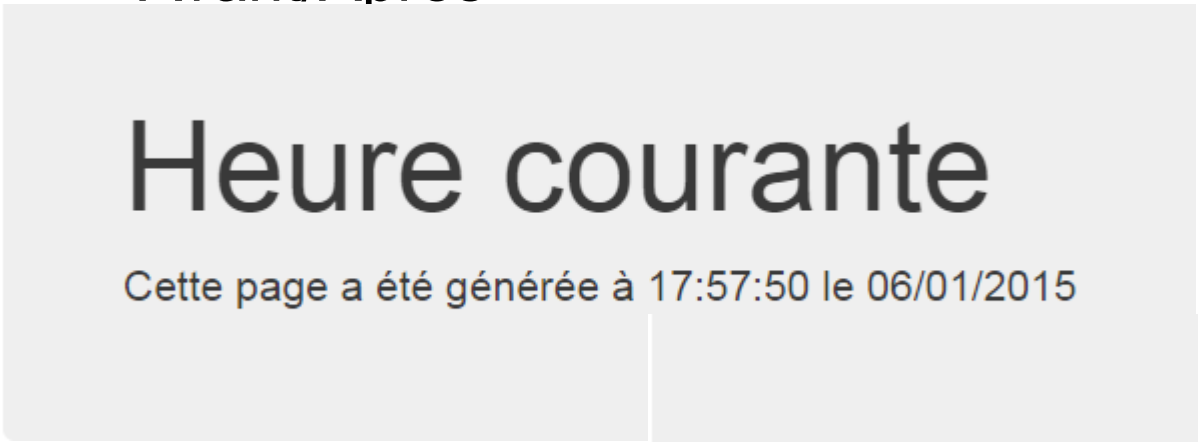

ISIMA

Java™

ORACLE

Testons

- Avant/Apres



AVANT

APRES

Résumons cette 1ère étape

- Scope applicatif
- Le scope applicatif permet d'avoir des objets communs à toute l'application
- C'est la durée de vie idéale pour
  - La configuration
  - Les initialisations de pools (pools de BDD par exemple)
  - Tout ce qui doit être accédé une seule fois
- Utiliser un scope applicatif est à bien des égards plus propre que d'abuser du pattern singleton
  - L'implémentation getInstance() est à banir avec un container leger type spring

Et les utilisateurs ?

- Il existe deux manières de stocker les informations liées aux utilisateurs :
  - Les stocker coté client
    - Soit utiliser un cookie (qui sera connu du serveur)
    - Soit utiliser une base intégrée au navigateur (localstorage/sessionstorage)
  - Les stocker coté serveur

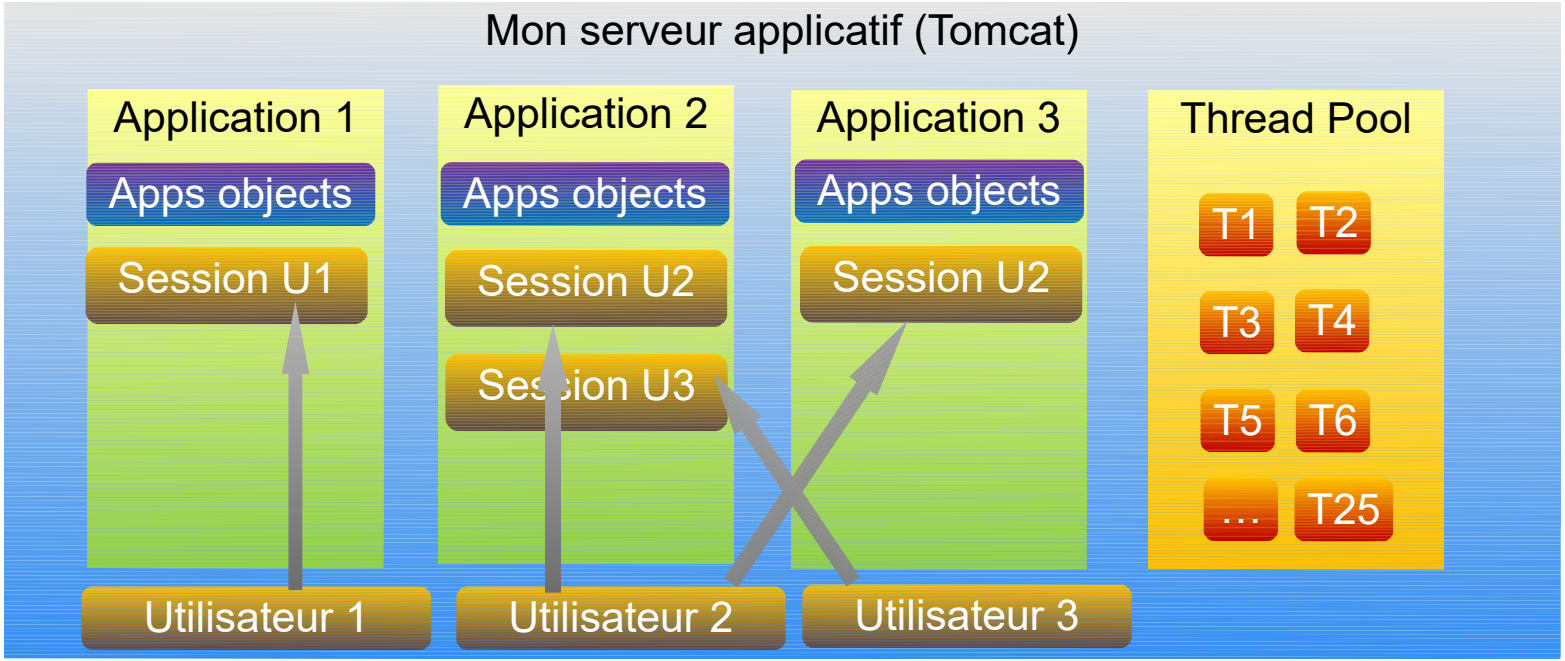
Avantages et inconvénients des deux solutions

- **Coté Client**
  - *Avantages*
    - L'espace mémoire est déporté coté client
    - La scalabilité (le client n'est pas lié à un serveur précis)
  - *Inconvénients*
    - La sécurité : montrer des données qui peuvent être sensibles
- **Coté serveur**
  - *Avantages*
    - La sécurité des données qui peuvent être masquées coté serveur
  - *Inconvénients*
    - La scalabilité (ou alors mise en place de partage de session complexe et couteux).
    - Les performances pour des utilisateurs nombreux

Traitement du stockage

- Dans ce cours nous traiterons uniquement du stockage coté serveur
- Nous allons voir comment gérer nos sessions applicatives sous JEE

Le serveur applicatif



ISIMA

Java™

ORACLE

Créer des objets dans les sessions utilisateurs

- En insérant directement en session l'objet dans un service (une Servlet dans notre cas)
  - Un cas classique est le chargement d'informations utilisateurs après sa connexion
- En utilisant un listener de session
  - L'évènement est envoyé dés la première utilisation de l'application par l'utilisateur. Il est lié au cookie de session (JSESSIONID)
  - Il est utile si l'on veut des informations dès que l'utilisateur commence à utiliser l'application

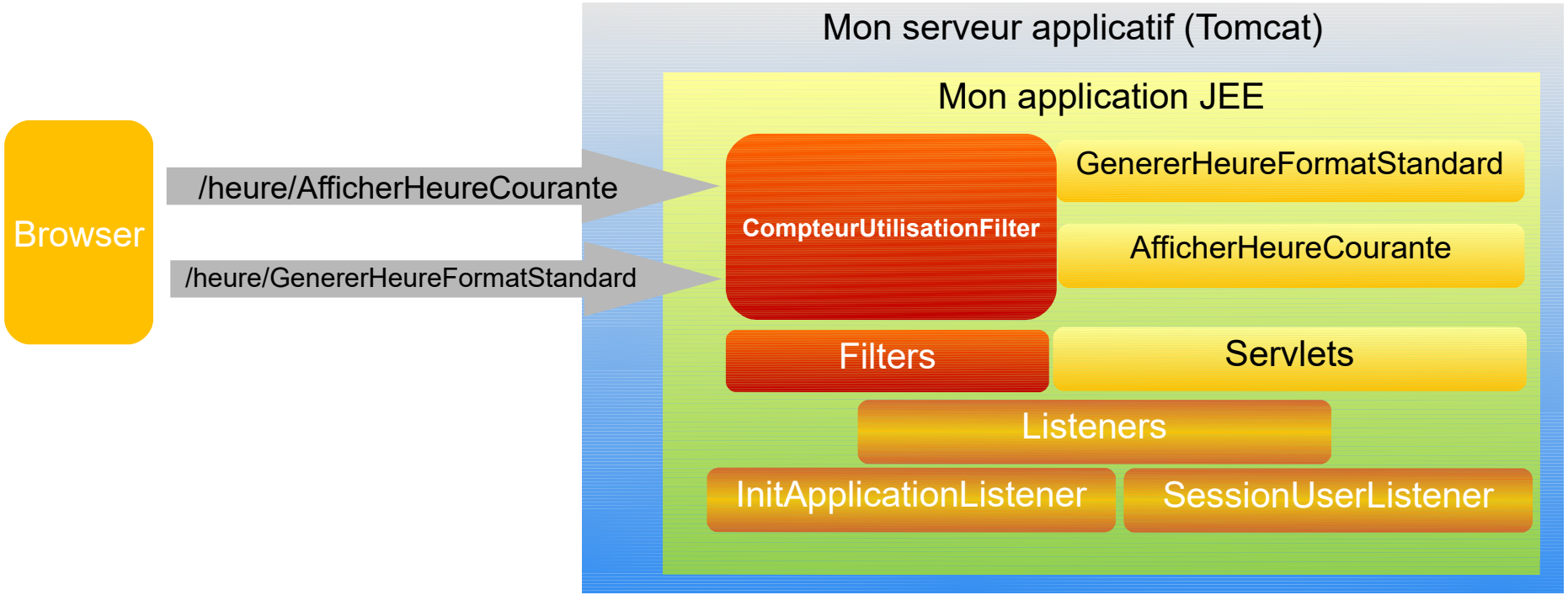
Détruire des objets en session

- Pour écouter la destruction des sessions de façon sure, seul le listener est efficace
- Une servlet de destruction des sessions ne prévient pas de tous les cas
  - Exemple session timeout
  - Fermeture du navigateur sans arrêter la session
- Pour l'arrêt du serveur, Tomcat sérialise par défaut les sessions qui sont restaurées ensuite
  - http://tomcat.apache.org/tomcat-8.0-doc/config/manager.html

Ecrivons un exemple

- Comptons le nombre d'appel par utilisateur
- Pour cela nous allons créer un objet en session qui va stocker le nombre d'appel au service d'affichage de l'heure
- Au moment de détruire la session nous allons l'afficher sur la sortie standard
- Nous allons utiliser un filtre pour compter les appels comme lors du TP2

A quoi cela va-t-il ressembler ?



Nous avons plusieurs composants à créer

- Le compteur qui va stocker le nombre de requêtes pour une session utilisateur
- Le SessionUserListener qui va créer notre compteur et lors de la destruction de la session va afficher le nombre de requêtes
- Le filtre qui va incrémenter le nombre de requête utilisateur
- Deux petits arrangements supplémentaires :
  - Le changement des path à surveiller pour rajouter « heure »
  - Une servlet d'invalidation des sessions pour nos tests

CompteurDemandeAffichageHeure

```
public class CompteurDemandeAffichageHeure implements Serializable {

    private final AtomicLong nombreDeDemandeParIhm;

    public CompteurDemandeAffichageHeure() {
        nombreDeDemandeParIhm = new AtomicLong();
    }

    public void addDemande() {
        nombreDeDemandeParIhm.incrementAndGet();
    }

    public long getNombreDemande() {
        return nombreDeDemandeParIhm.get();
    }

    @Override
    public String toString() {
        return "Nombre de demande d'affichage de l'heure : " + getNombreDemande();
    }
}
```

Quelques notes

```
public class CompteurDemandeAffichageHeure
implements Serializable
```

- Il est important que l'objet en session soit serializable pour qu'en cas d'arrêt relance, Tomcat puisse restaurer les instances en l'état
  - Vous pouvez aussi choisir de ne pas restaurer l'état, étudiez bien votre besoin
- Pour le reste il s'agit d'une classe très simple incrémentant un compteur

SessionUserListener

```
@WebListener
public class SessionUserListener implements HttpSessionListener {

    public static final String COMPTEUR_HEURE_COURANTE = "compteur-heure-courante";

    @Override
    public void sessionCreated(HttpSessionEvent create) {
        final HttpSession session = create.getSession();
        System.out.println("Session create " + session.getId());
        session.setAttribute(COMPTEUR_HEURE_COURANTE, new CompteurDemandeAffichageHeure());
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent hse) {
        final HttpSession session = hse.getSession();
        System.out.println("Fin de la session de l'utilisateur " + session.getId());
        final CompteurDemandeAffichageHeure compteur = (CompteurDemandeAffichageHeure)
                        session.getAttribute(COMPTEUR_HEURE_COURANTE);
        System.out.println(compteur);
    }
}
```

Les points remarquables

```
public class SessionUserListener implements
HttpSessionListener
```

- L'interface HttpSessionListener pour écouter les évènements liés aux sessions

```
 public void sessionCreated(HttpSessionEvent
create)
```

- L'évènement de création de la session

```
public void sessionDestroyed(HttpSessionEvent hse)
```

- Idem pour la destruction

CompteurUtilisationFilter

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

    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain chain) throws IOException,
ServletException {
        try {
            System.out.println("Debut du filtre de comptage des heures");
            chain.doFilter(req, resp);
        } finally {
            // Les evenements de creation de session n'arrivent qu'apres
            // l'execution des filtres
            // il faut donc incrementer le compteur qu'a posteriori
            // l'execution du final permet de s'assurer que cette instruction
            // n'est jamais oubliee
            final CompteurDemandeAffichageHeure compteur = findCompteurAffichageHeure(req);

            compteur.addDemande();
            System.out.println("Fin du filtre de comptage des heures");
        }
    }
//A suivre sur la diapo suivante
}
```


ISIMA

Java™

ORACLE

CompteurUtilisationFilter

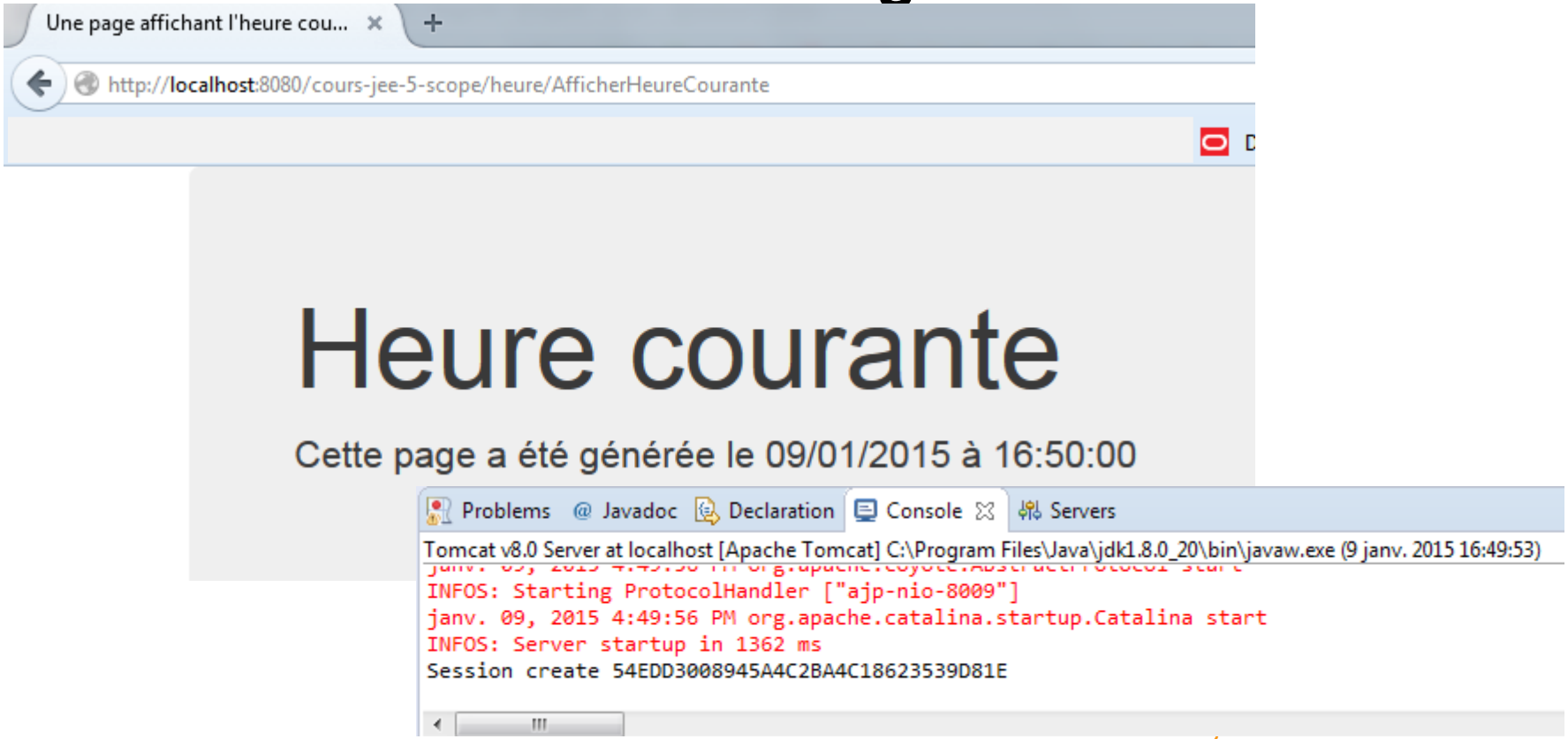
```
@WebFilter("/heure/*")
public class CompteurUtilisationFilter implements Filter {

    private CompteurDemandeAffichageHeure findCompteurAffichageHeure(ServletRequest
req) {
        final HttpServletRequest httpReq = (HttpServletRequest) req;
        final CompteurDemandeAffichageHeure compteur = (CompteurDemandeAffichageHeure)
              httpReq.getSession().
                   getAttribute(SessionUserListener.COMPTEUR_HEURE_COURANTE);
        // en cas d'arret relance du serveur le compteur peut-etre null
        if (compteur == null) {
            throw new
                NullPointerException(SessionUserListener.COMPTEUR_HEURE_COURANTE);
        }
        return compteur;
    }
}
```


ISIMA

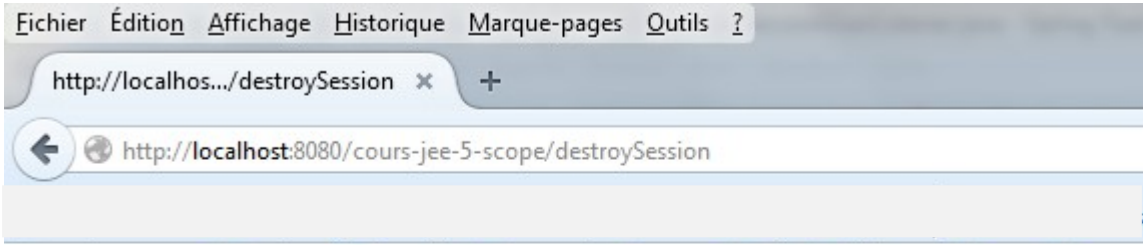
Java™

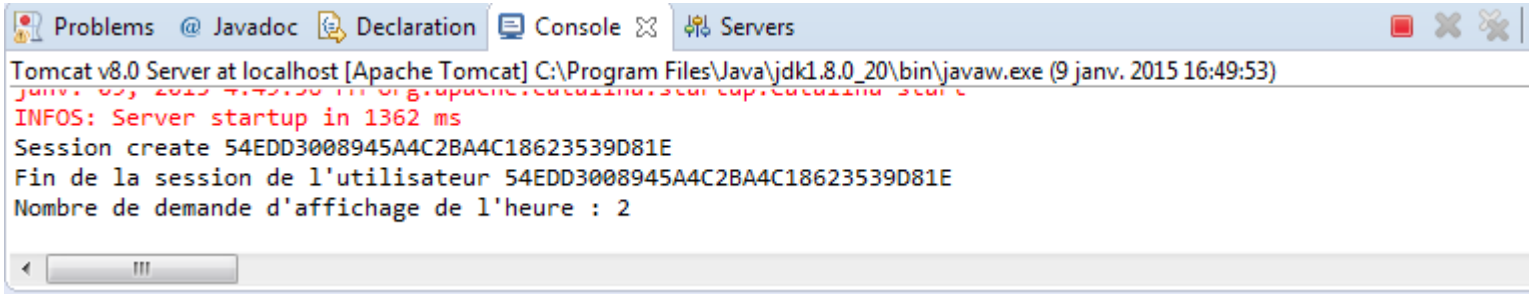
ORACLE

Testons sous notre navigateur



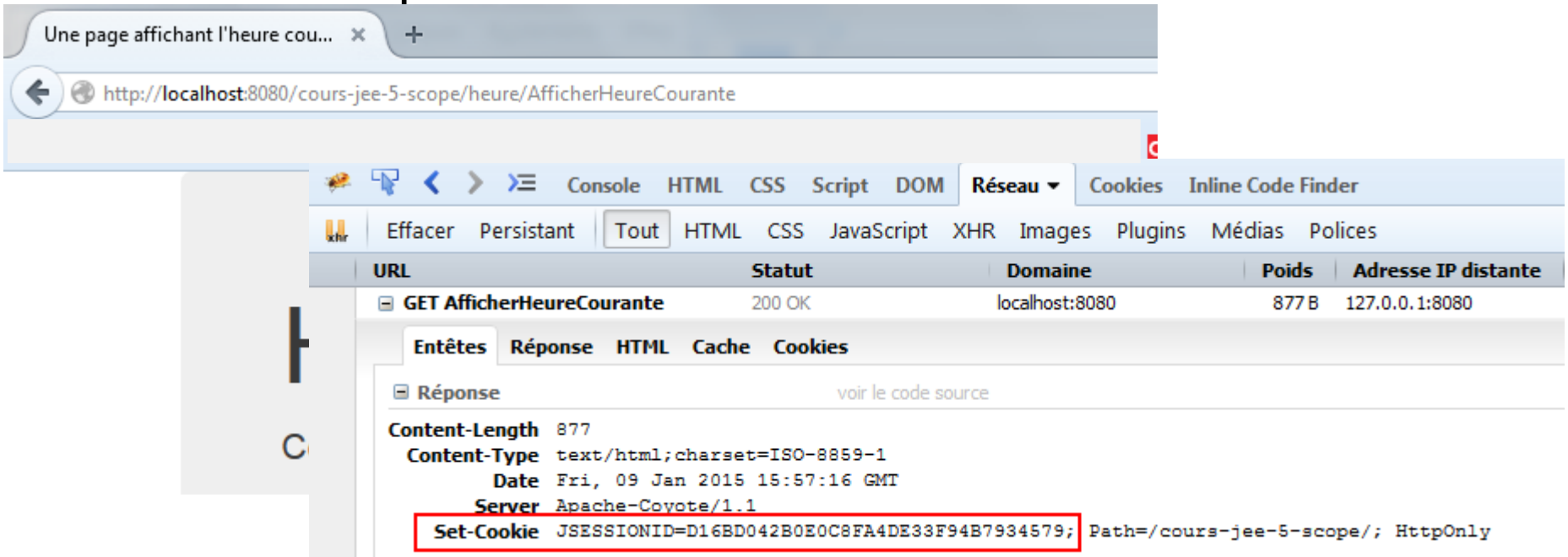
Arrêt du serveur



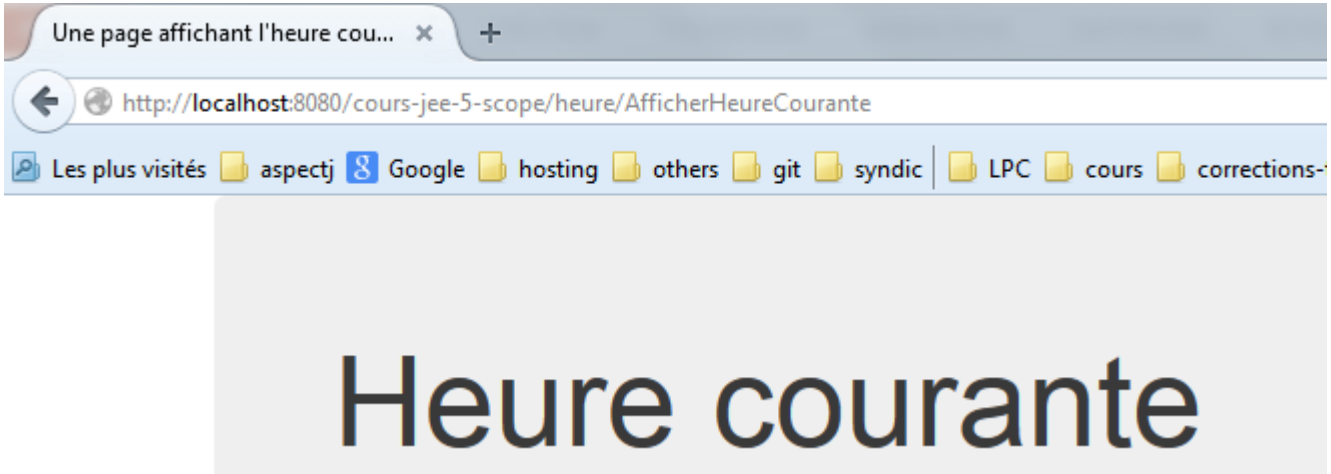


Quelques subtilités de la sérialisation

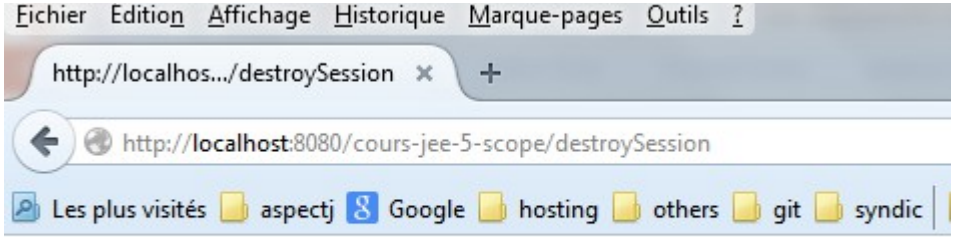
- Vérifions que la restauration de session fonctionne bien

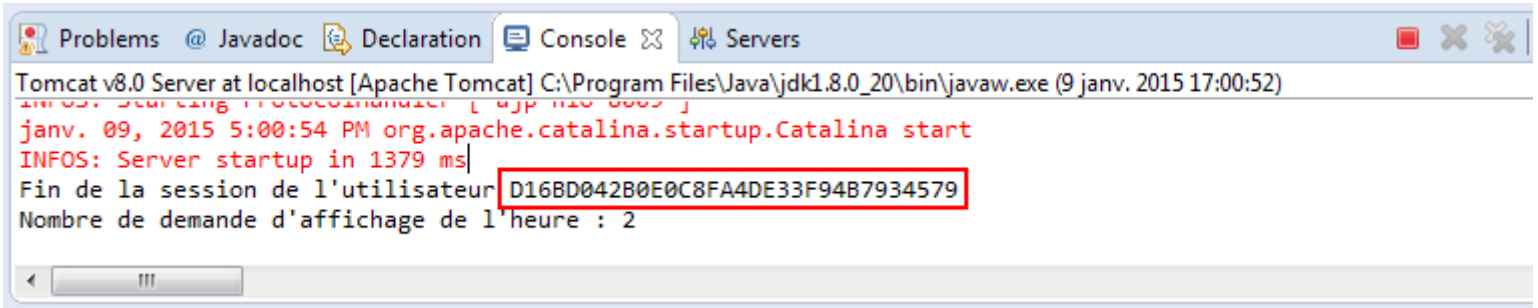


Redémarrons le serveur



Constatons le changement





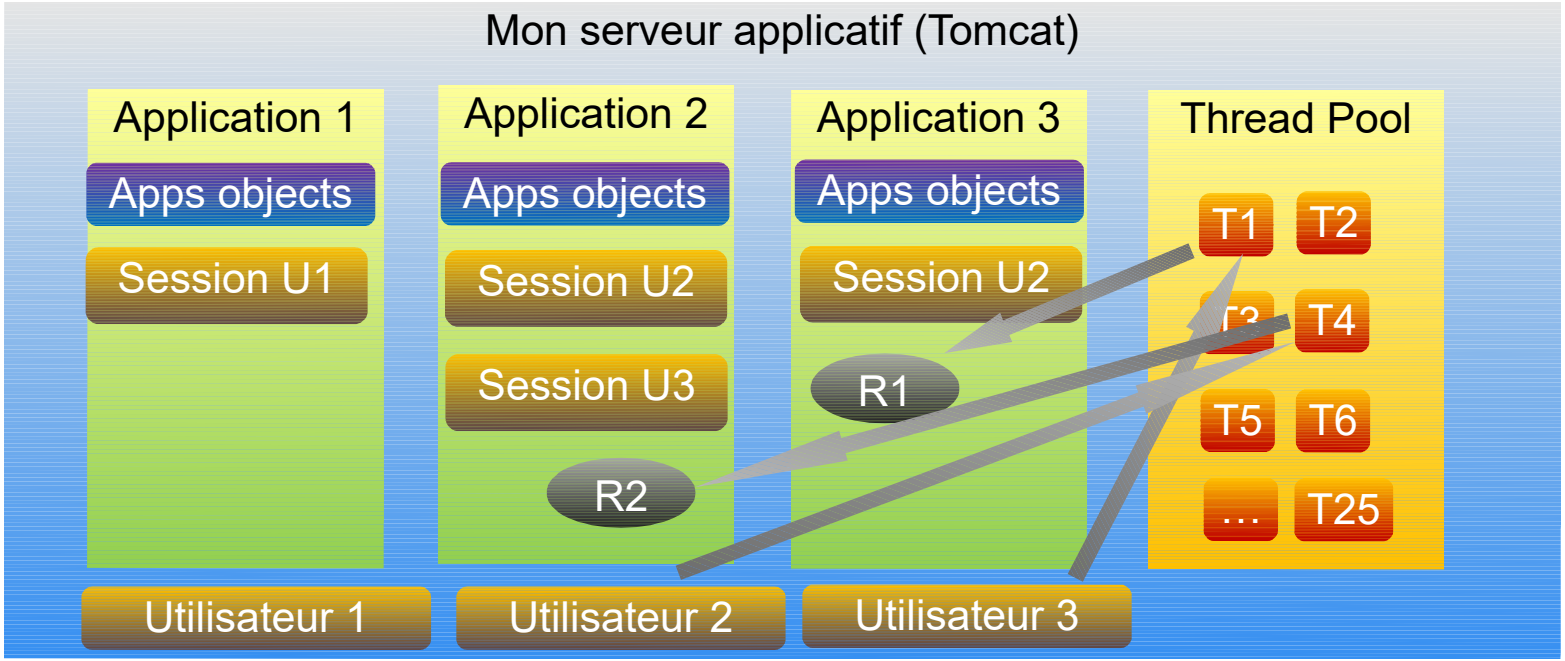
Résumons ce 2ème point

- La gestion des sessions
- Si vous choisissez de gérer vos sessions coté serveur utilisez le listener adéquat
  - Il vous permettra de conserver correctement vos sessions que ce soit sur du timeout ou sur des fermetures imprévues coté client
- Coté client, la conservation des informations peut poser des problèmes de sécurité

Dernier scope de notre cours

- Le scope « request »
- Il nous reste une durée de vie disponible dans les applications JEE
  - La durée de vie request
  - Elle correspond au temps d'exécution d'une requête
  - Ce scope permet de stocker des objets qui sont détruits à la fin de la requête
- Cela correspond à un appel d'une servlet

Le serveur applicatif



Le durée de vie Request

- Un objet de type Request ne peut-être accédé que depuis un seul thread du serveur applicatif
  - Il a donc la grande qualité d'être ThreadSafe
  - Privilégiez cette durée de vie, sauf si le coût de création des objets est grand
    - Dans ce cas, il faut réfléchir à mettre en place un cache
    - Eviter de surcharger la session d'objets inutiles
  - N'optimisez pas à priori, c'est souvent inutile et source de beaucoup de problèmes

Pourquoi utiliser le scope request ?

- Il permet de transférer des informations entre les différents intervenants de la requête
  - Exemple
    - Le filter passe des informations à la Servlet
      - Relativement rare car le filter est souvent un composant technique indépendant
    - La servlet passe des informations à la JSP
- Typiquement nous l'avons utilisé dans AfficherHeureCourante

```
final GenerateurHeureCourante generateurHeureCourante = creerHeureCourante();
request.setAttribute("currentDateAsString", generateurHeureCourante.nowWithStandardFormat());
```

Et le listener ?

- Le listener de request existe comme chaque scope
- Contrairement au filter il est exécuté systématiquement
  - requestInitialized est appelé avant tous les filtres
  - requestDestroyed après la servlet et les filtres
- Ce listener doit être utilisé dans le cas où
  - Tout le monde doit exécuter le traitement
  - Il faut l'exécuter avant ou après que tout le monde ait réalisé son travail
- Faisons un filtre qui affiche le contenu de la map « request »
  - J'utiliserais souvent le terme « la request »

Le DisplayRequestContentListener

```
@WebListener
public class DisplayRequestContentListener implements ServletRequestListener {

    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        System.err.println("Ajout d'un attribut exemple, attribut-positionne pour l'url " + ((HttpServletRequest)
sre.getServletRequest()).getRequestURI());
        sre.getServletRequest().setAttribute("attribut-postionne", "un attribut");
    }

    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        final ServletRequest sr = sre.getServletRequest();
        System.err.println("Affichage des attributs de la request");
        final Enumeration<String> en = sr.getAttributeNames();
        while (en.hasMoreElements()) {
            final String key = en.nextElement();
            final Object val = sr.getAttribute(key);
            System.err.println("L'objet suivant est associe a la cle " + key + " : " + val);
        }
    }
}
```

Voyons l'exécution

- Après avoir ajouté des traces dans AfficherHeureCourante et CompteurUtilisationFilter et appelé la Servlet AfficherHeureCourante, voici la sortie sur la console

```
Tomcat v8.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.8.0_20\bin\javaw.exe (12 janv. 2015 15:41:48)
Ajout d'un attribut exemple, attribut-positionne pour l'url /cours-jee-5-scope/heure/AfficherHeureCourante
Affichage des attributs de la request
L'objet suivant est associe a la cle currentDateAsString : le 12/01/2015  à  15:43:12
L'objet suivant est associe a la cle attribut-postionne : un attribut
Debut du filtre de comptage des heures
Afficher heure courante
Fin du filtre de comptage des heures
Ajout d'un attribut exemple, attribut-positionne pour l'url /cours-jee-5-scope/js/jquery-2.1.1.js
Affichage des attributs de la request
L'objet suivant est associe a la cle attribut-postionne : un attribut
Ajout d'un attribut exemple, attribut-positionne pour l'url /cours-jee-5-scope/js/bootstrap.min.js
Affichage des attributs de la request
L'objet suivant est associe a la cle attribut-postionne : un attribut
Ajout d'un attribut exemple, attribut-positionne pour l'url /cours-jee-5-scope/css/bootstrap-theme.min.css
Affichage des attributs de la request
L'objet suivant est associe a la cle attribut-postionne : un attribut
Ajout d'un attribut exemple, attribut-positionne pour l'url /cours-jee-5-scope/css/bootstrap.min.css
Affichage des attributs de la request
L'objet suivant est associe a la cle attribut-postionne : un attribut
```

Conclusion

- Un arbre d'objet
- Un bon programme orienté objet peut-être vu comme un « arbre »
  - Il faut réfléchir au bon propriétaire de l'objet
  - L'objet
    - donne les méthodes pour accéder aux services
    - fabrique les objets nécessaires à la réalisation de ceux-ci
      - Chaque sous objet fait de même
  - Chaque objet doit connaître et maitriser son propre environnement pas plus

Conclusion

- Comment choisir la durée de vie d'un objet ?
- Bien choisir la durée de vie aura un impact sur les performances applicatives
  - En fonction de la méthode choisie, des problèmes de sécurité ou de maintenabilité peuvent se poser
- Essayez de privilégier les objets avec une durée de vie courte
- Des objets avec des durées de vie différentes interagissent entre eux. Attention que des objets de durée de vie longue ne détiennent pas des références à des objets de durée de vie courte

Les durée de vie disponibles

- Une application JEE propose 3 durées de vie différentes