Formation Tapestry 5

Une Approche Composant du développement WEB

mercredi 8 décembre 2010

Objectifs

- A l'issue de la formation, les participants auront appréhendé et pratiqué les concepts mis en œuvre par le Framework Tapestry (Composant, Template, Services, IOC)
- Les participants seront capables d'utiliser Tapestry pour développer une application frontale WEB en exploitant au mieux les outils et mécanismes inhérents proposés par Framework

Plan

- Présentation Générale
  - Qu'est-ce que Tapestry ?
  - Composant / Page
  - Modèle de traitement des requêtes
  - Cycle de vie
  - Installation et paramétrage
- Mécanismes de base
  - Généralités
  - Traitement de la requête
  - Méthodologie
  - Composants de base
  - Gestion des formulaires

Atos Origin

Plan

- **Composants avancés**
  - **BeanEditForm**
  - **Grid**
  - **BeanModel**
  - **Upload**
- **Mécanismes avancés**
  - **Extension du mécanisme de validation**
  - **Prise en charge de nouveaux types**
  - **Composants utilisateur**
  - **Internationalisation**
  - **Assets**
  - **Services et Injection de dépendances**

Atos Origin

Présentation Générale

Atos Origin

Présentation Générale

Qu'est-ce que Tapestry ?

- **Tapestry est un Framework de développement WEB qui propose une méthode de travail structurée**
- **La dernière version stable préconisée est la 5.1.0.5**
- **Cette version applique les derniers concepts de développement**
- **Elle propose aussi des ponts vers les technologies majeures du marché (Hibernate, Spring…)**

- **Objectif principal :**
  - **Faciliter et rendre efficace le développement: un minimum de configuration, composants réutilisables, rechargement à chaud etc.**

6 *Turning Client Vision into Results*

Le Framework Tapestry est un framework Web, développé par Howard Lewis Ship. Avant de devenir un Projet Apache, Tapestry était intégré dans le projet Jakarta

L'une des principales caractéristiques du framework Tapestry est sa simplicité. En effet, toutes les classes proposées par le framework sont de simples classes java (POJO : Plain Old Java Object). Le fait que les composants Tapestry soit de simple POJO, permet de définir Tapestry comme un framework léger. Tapestry propose également des rapports d'erreurs très détaillés, facilitant la phase de développement.
Le rechargement à chaud (avec Jetty), permet de visualiser les modifications, sans relancer le serveur.

La dernière version stable du Framework (5.2.4) permet d'utiliser les concepts de base du JDK 1.5 (Annotations, les types génériques…).
Le Framework nous propose également des ponts vers le FRONT et le MIDDLE permettant d'utiliser d'autres technologies (Hibernate, Spring).

Permet de créer des applications WEB…

- **…Dynamiques**

  Tapestry propose des mécanismes simples pour dynamiser des Templates de type HTML et interagir avec un modèle Java

- **…Robustes**

  Tapestry est un Framework de développement WEB abouti et propose des solutions pragmatiques à la plupart des problématiques liées au Framework « boîte à outils » ou orientés « action » (passage d'information, gestion de la session, gestion des ressources)

- **…Scalable**

  Tapestry exploite au minimum la session HTTP ce qui permet d'avoir des applications qui s'adaptent plus facilement aux architectures de production (Cluster)

L'utilisation de Tapestry permet de créer des applications web dynamique, grâce à des templates qui interagissent avec des classes Java.

La dernière version du Framework est la 5.2.4. Grâce à toutes ces versions, nous pouvons dire que Tapestry répond actuellement à toutes les problématiques liées au Framework : gestion des ressources, gestion des sessions.

- **S'appuie sur les standards de développement de l'API de Servlets**
- **Peut être déployé sur la majorité des serveurs d'application: Tomcat, Jetty, Glassfish**
- **Technologies exploitées**
  - **Interfaces + POJO (Java classique)**
  - **Annotations (JDK 1.5)**
  - **Types génériques (JDK 1.5)**
  - **IOC (via Javassist)**
  - **Aspect (via Javassist, les patterns « Interceptor », « Decorator »…)**
- **Objectif:**
  - **Rendre les applications moins sensibles aux évolutions du produit Tapestry**

Tapestry s'appuie sur des standards de développement de l'API de Servlets (Request, Response…). Cependant la manipulation de ces standards est transparente pour le développeur.

Tapestry se base sur les technologies :

- Interface + POJO : Il est inutile d'étendre des classes propres à Tapestry. Cette caractéristique permet d'être indépendant de la technologie Tapestry.
- IOC : géré par Tapestry

Atos Origin

Présentation Générale

Qu'est-ce que Tapestry ?

- **Une application Tapestry consiste en un ensemble de pages constituées de composants**
- **Le Framework gère :**
  - **Le traitement des requêtes (Analyse, Dispatch)**
  - **La création des URLs**
  - **La gestion de l'état des pages (Persistance des données)**
  - **La validation des données saisies par l'utilisateur**
  - **L'internationalisation**
  - **La gestion des erreurs et la génération des rapports d'exception**

9 *Turning Client Vision into Results*

Les URLs proposées par Tapestry permettent de répondre en partie aux problématiques de référencements des applications dynamiques

La gestion de la session utilisateur est transparente pour les développeurs qui ne doivent plus utilisés directement l'objet HttpSession.

Tapestry fournit un rapport d'erreur détaillée qui facilite le développement et le débogage des applications

- **Développer une application Tapestry, c'est…**
  - **Créer des Templates HTML**
  - **Associer chaque Template à une classe Java**
  - **Développer en pensant « Objet » et non plus « HTTP », « Request »…**
  - **Augmenter l'utilisation des composants (facilité du déploiement, réutilisabilité, Approche POJO…)**

- **Le projet Tapestry 5 est constitué de plusieurs modules :**
  - **Tapestry-ioc : Socle de développement du projet**
  - **Tapestry-core : Framework de développement WEB (utilise tapestry-ioc)**
  - **Tapestry-annotation : Contient les annotations à utiliser pour le développement d'application Tapestry 5**
  - **Tapestry-upload : Sous-projet pour la prise en charge de l'upload (est chargé par tapestry-core)**
  - **Tapestry-test : utilisé pour intégrer des technologies de test unitaire et fonctionnel dans les applications Tapestry**
- **Il existe d'autres sous-projets qui sortent du cadre de cette formation (tapestry-hibernate, tapestry-spring…)**

- **Techniquement**
  - **Le chargement du Framework se fait grâce à un Filtre J2EE**
  - **L'interception des requêtes de l'utilisateur se fait via un « mapping »**
- **Liens utiles :**
  - **http://tapestry.apache.org/tapestry5.1/**
  - **http://wiki.apache.org/tapestry/Tapestry5HowTos**
  - **http://canalweb.atosworldline.com/FR/RUB/5093219/Tapestry-5.htm**

- **Un composant Tapestry est constitué de deux éléments fondamentaux**
  - **Une classe Java**
    - POJO (Aucune classe à étendre)
    - Un constructeur sans paramètres (les autres ne sont pas pris en compte)
    - Des annotations Tapestry (Déclaration des paramètres, Agrégation de services et d'autres composants …)
    - Des annotations Tapestry pour intervenir durant les phases du rendu
  - **Un Template HTML**
    - Document XML valide contenant essentiellement du (X)HTML
    - Document associé au namespace Tapestry
    - L'extension des composants est « .tml » pour « Tapestry Markup Language »

- **Une page et un composant sont structurellement identiques :**
  - **Une classe Java**
  - **Un Template**
- **Les différences concrètes entre un composant et une page**
  - **Le nommage (package différent)**
  - **Un composant ne gère pas l'évènement « activate »**
- **Ils ne sont pas écrits dans le même but mais s'appuient sur les mêmes mécanismes**

- **Tapestry distingue deux types de requêtes**
  - **Action**
  - **Rendu**
- **L'utilisateur est toujours redirigé vers une requête de type Rendu après une action**
- **Objectifs**
  - **Eviter certains désagréments liés aux actions de l'utilisateur dans son navigateur : refresh, back, F5 qui exécute plusieurs fois la même action …**
  - **Proposer des formats d'URL « bookmarkable » via le mécanisme d'activation/passivation des pages**
- **Lien utile :**
  - **http://tapestry.apache.org/tapestry5/tapestry-core/guide/pagenav.html**

- **Détails**
  - **Action**
    - Génère un évènement sur un composant spécifique dans une page donnée
    - La valeur de retour de l'évènement définira ce qui sera retourné au client (le plus souvent un « redirect » vers une page de type Rendu)
      - Ex : Vérification du login/password et renvoi vers la page principale de l'application
  - **Rendu**
    - Exécute une page et renvoie le flux HTML généré au client
    - Une page peut nécessiter un contexte d'activation (Ex : l'identifiant de l'élément que la page se charge d'afficher)
      - Ex : Affichage d'une fiche produit

- **Action : gestion des évènements**

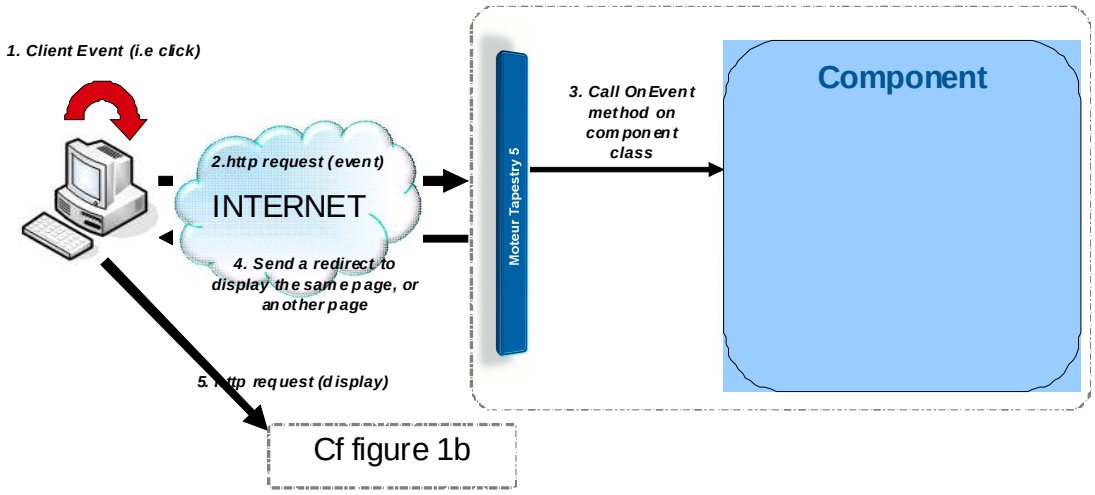


Figure 1a : Evènement sur un composant d'une page

- **Gestion du Rendu**

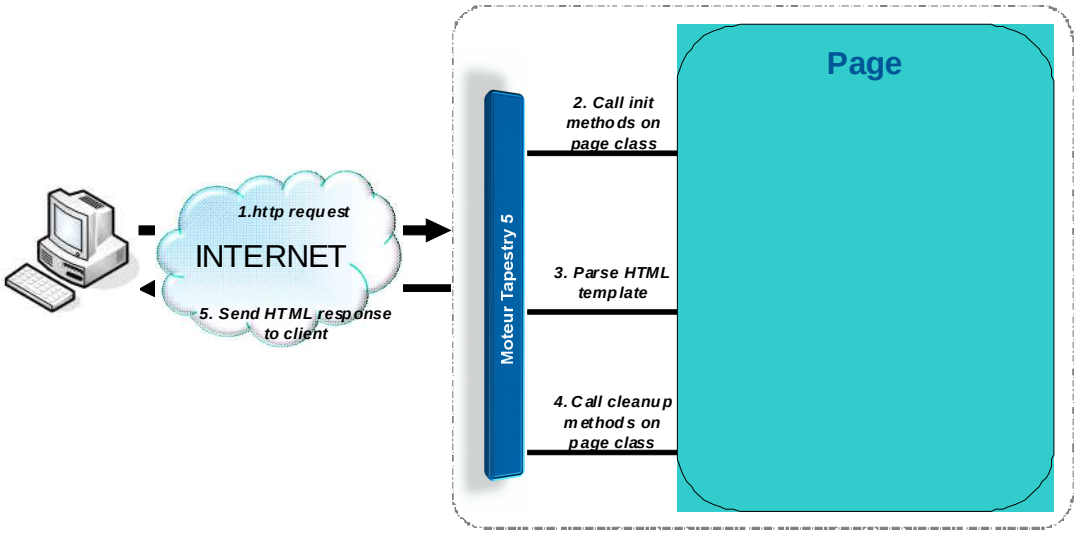
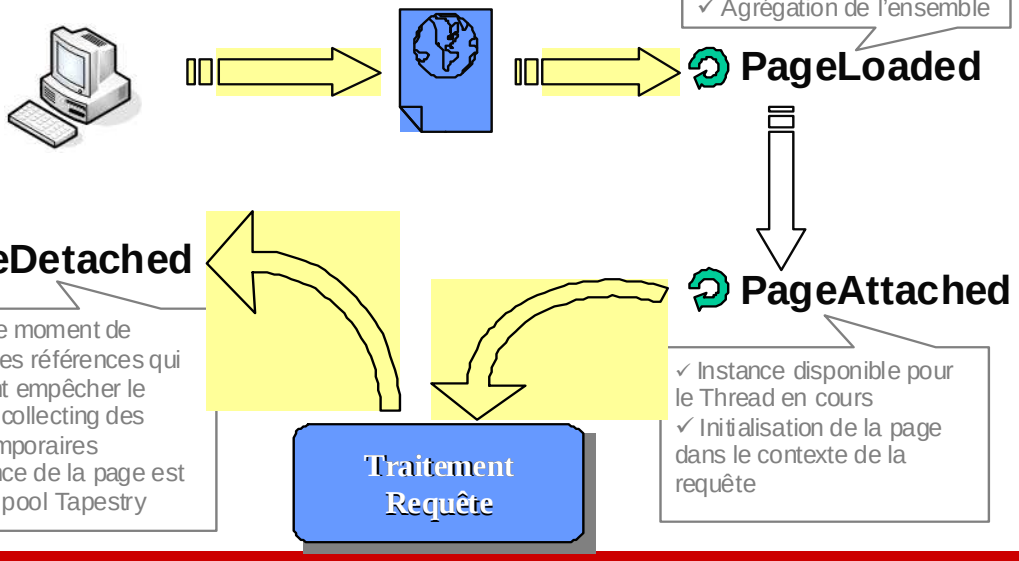
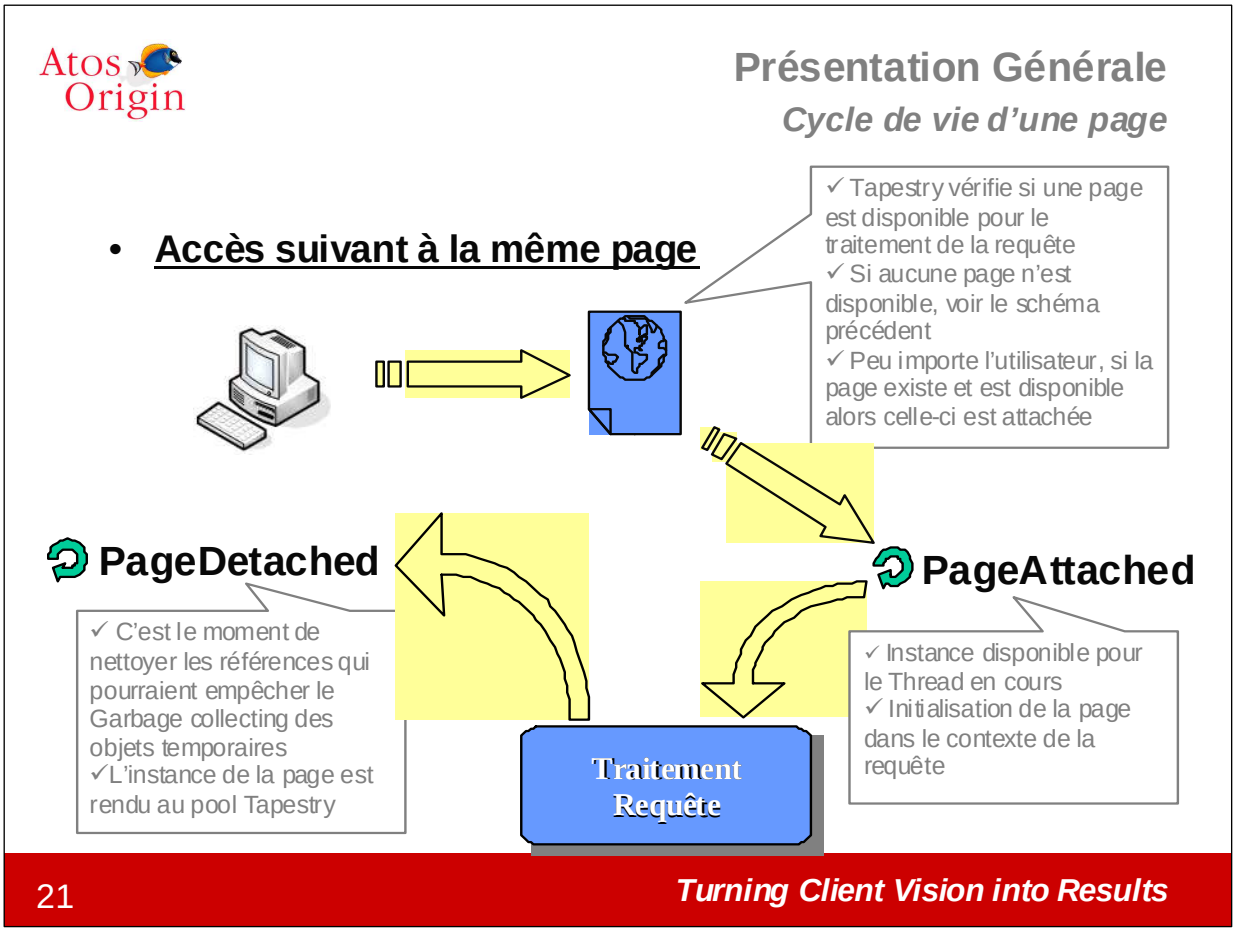


Figure 1b : Requête d'affichage de page

- **Tapestry utilise des « pools » d'objets pour stocker et réutiliser les instances des pages / composants créés**
- **La connaissance du cycle de vie des composants permet d'exécuter les actions d'initialisation au moment opportun**
- **Tapestry nous permet d'exécuter des actions aux différentes étapes du cycle de vie des composants via trois annotations de méthode :**
  - **@PageLoaded : Une fois la page créée et associée aux composants qui la constituent. Permet d'exécuter des traitements qui ne peuvent se faire au moment de l'instanciation et qui nécessite par exemple la présence des services injectés**
  - **@PageAttached : Lorsqu'une page est associée au thread traitant notre requête**
  - **@PageDetached : Lorsque le traitement de la requête est terminé et que la page est rendu au pool**

- **Premier accès à une page**





Le nombre d'instance de page disponible dans le pool est configurable dans le fichier AppModule.java .

- **Objectifs**
  - **Pas de XML**
  - **Peu verbeux**
  - **Exploiter au maximum les normes de nommage**
  - **Utiliser les variables « système »**

Modification du fichier « web.xml » :

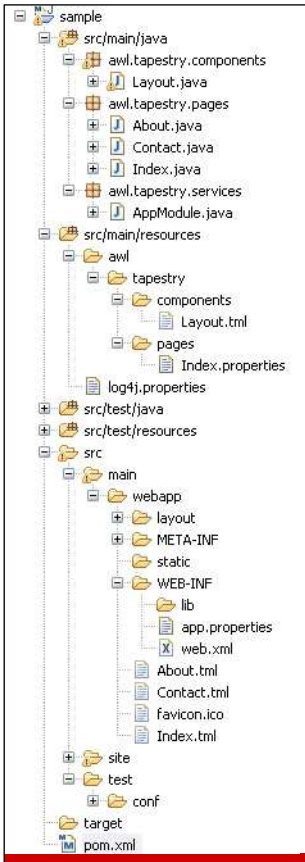
1. Déclarer le package racine de l'application

```
<context-param>
    <param-name>tapestry.app-package</param-name>
    <param-value>net.atos.mm.formation.tapestry</param-value>
</context-param>
```

2. Déclarer le Filtre Tapestry

⚠ L'attribut « filter-name » définit aussi le nom de l'application

```
<filter>
    <filter-name>app</filter-name>
    <filter-class>org.apache.tapestry5.TapestryFilter</filter-class>
</filter>

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



Installation et Paramétrage

Organisation des répertoires

- **Préconisations pour l'organisation du projet Maven**
  - **Deux répertoires de sources**
    - Un répertoire « src/main/java » destiné à contenir les classes Java de votre application
    - Un répertoire « src/main/resources » contenant les fichiers de ressources de l'application (Template HTML « *.tml », fichiers de ressources embarquées)
  - **Les classes compilées ainsi que les Templates doivent être disponibles dans le « classpath » de votre application**
  - **A la racine de vos documents WEB**
    - Un répertoire « static » pour contenir les informations purement statiques de l'application

Atos Origin

Installation et Paramétrage

- **Une fois le package de l'application défini, ajouter les sous-package**
  - **« root.package ».pages : contient les pages l'application (Classe Java + Template HTML « .tml »)**
  - **« root.package ».components : contient les composants utilisateur**
  - **« root.package ».mixins : contient un type de composant particulier permettant d'ajouter des comportements**
  - **« root.package ».base : permet de créer des composants de base qui ne sont pas destinés à être utilisé directement**
- **Préconisation O&D**
  - **« root.package ».data : *non obligatoire***
- **Pour le paramétrage de l'application**
  - **« root.package ».services : contient les services utilisateur**
  - **Ajouter la classe « root.package ».services.AppModule pour configurer votre application**

La classe « *App*Module »

- Permet de charger et de paramétrer l'application via un ensemble de méthodes prédéfinies
- Permet de charger des librairies de composants externes
- ⚠ « *App* » représente le nom de l'application donné lors de la déclaration du filtre Tapestry

  Ex : Si le package racine de l'application est
  « net.atos.mm.formation.tapestry » et que le filtre a été nommé
  « *TapestryApp* »

  Tapestry s'attend à trouver la classe
  « net.atos.mm.formation.tapestry.services.TapestryAppModule »

http://canalweb.atosworldline.com/FR/ART/5090411/HOWTO-Make-a-reusable-component-library-with-Tapestry-5.htm

La classe « *App*Module »

- C'est la méthode « contributeApplicationDefaults » qui permet de paramétrer l'application en contribuant au service de paramétrage « ApplicationDefaults »
  - Remarque : Il est possible d'utiliser le fichier web.xml (context-param) et l'option « -D » de la JVM pour indiquer les valeurs des paramètres de l'application
- L'utilisation de context param est la meilleure façon pour externaliser le paramétrage
- Voici quelques paramètres utiles
  - tapestry.start-page-name : Page de démarrage de l'application
  - tapestry.supported-locales : Affecter la liste des langues supportées par l'application
  - tapestry.file-check-interval : Paramétrer le temps de latence entre deux vérifications des fichiers (Rechargement à chaud)
  - tapestry.compress-whitespace : Supprime les espaces et retour à la ligne non visible dans la page HTML retournée
  - tapestry.production-mode : Le détail des exceptions ne sera pas complet si elles surviennent
- Certaines de ces constantes sont disponibles en Java dans l'interface « SymbolConstants »
  - http://tapestry.apache.org/tapestry5.1/guide/conf.html

TP1 : Installation et Paramétrage

Concepts abordés

- Configuration d'une application WTP
- Installation et paramétrage de Tapestry
- Premiers pas avec Tapestry

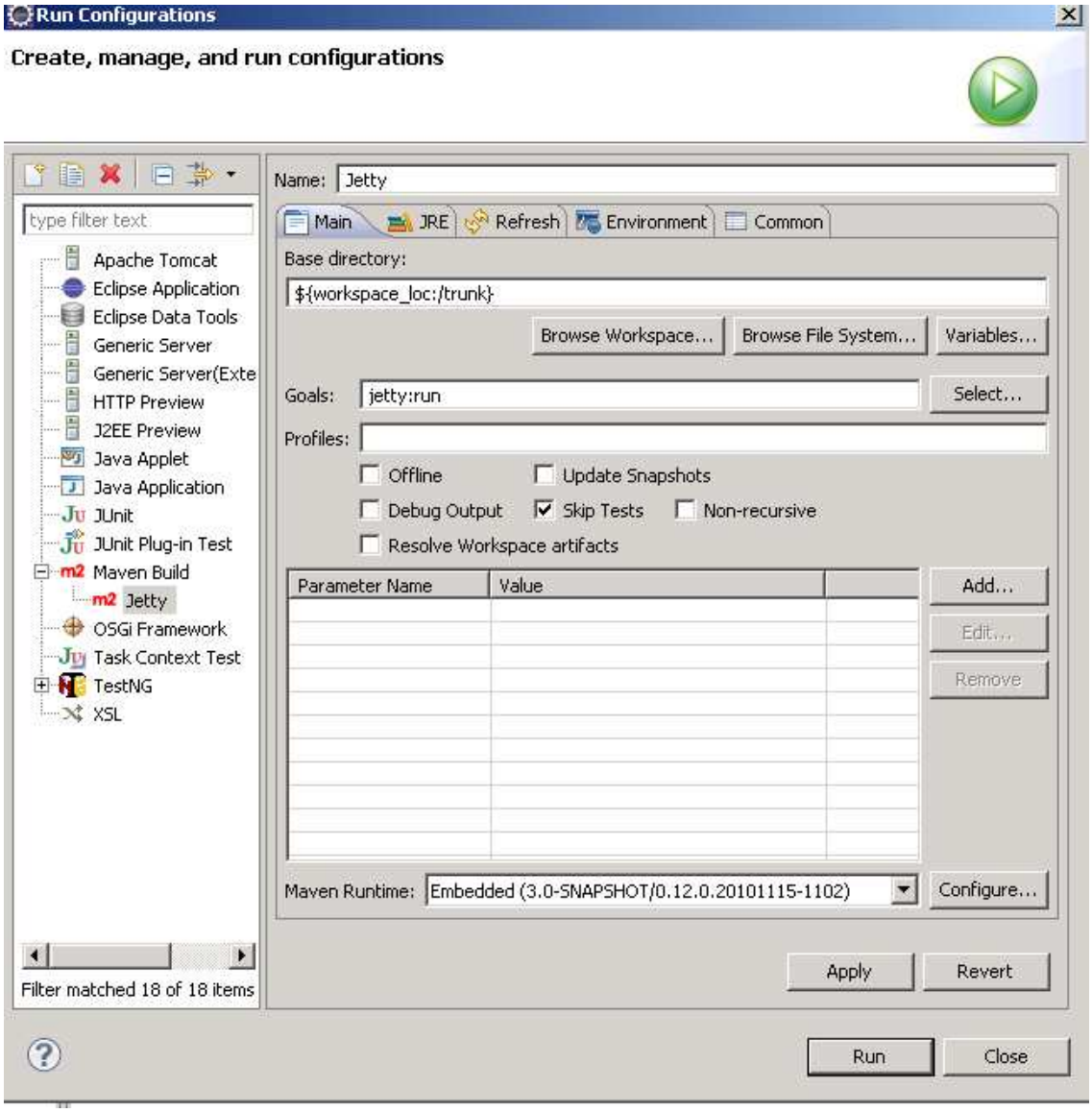
Enoncé

Présentation de l'organisation des répertoires et préparation de l'environnement avec le formateur :

1. Importer le projet
    a. File > Import > Existing Maven Project

Nous pouvons importer un projet Maven, grâce au plugin m2eclipse. Maven est outil qui permet d'automatiser la production d'un application. Pour notre formation, il téléchargera toutes les JAR externes (tapestry-core, tapestry-ioc…) utiles à notre application.

2. Vérifier que vous utilisez un JDK 1.5 (Installed JRE dans les préférences Java d'Eclipse)
3. Créer un Run pour permettre le lancement de jetty
    a. Run > Run Configuration
    b. Créer un nouveau build maven.
        i. Name : jetty run
        ii. Base directory : ${project_loc}
        iii. Goals : jetty:run
        iv. Cocher « Skip Test »



Jetty est identique à Tomcat. La seule différence est que Tomcat ne prend pas en compte le rechargement à chaud, car il utilise son propre ClassLoader, qui rentre en conflit avec celui de Tapestry.

4. Ouvrir le fichier web.xml et remarquer la déclaration du filtre Tapestry ainsi que les paramètres de contexte associé.
5. Implémenter la classe « *App*Module » où « *App* » est le nom que vous avez donné à votre application
    a. Modifier le corps de la méthode « contributeApplicationDefaults ». Cette méthode contient un paramètre de type « Map » qui permet de mapper les propriétés par défaut de l'application (clé = valeur)

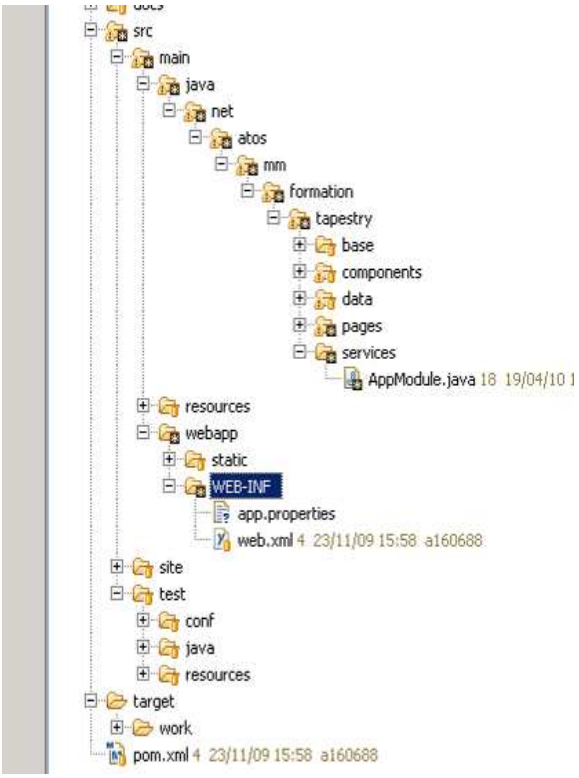
b. Spécifier une page de départ à l'aide de la clé « tapestry.start-page-name » : « Welcome »

```
public static void contributeApplicationDefaults(
        MappedConfiguration<String, String> configuration) {

    // Set here the code modify the start page name to "Welcome"
    configuration.add("tapestry.start-page-name", "Welcome");

    configuration.add(SymbolConstants.COMPRESS_WHITESPACE, "false");
    configuration.add(SymbolConstants.PRODUCTION_MODE, "false");
}
```

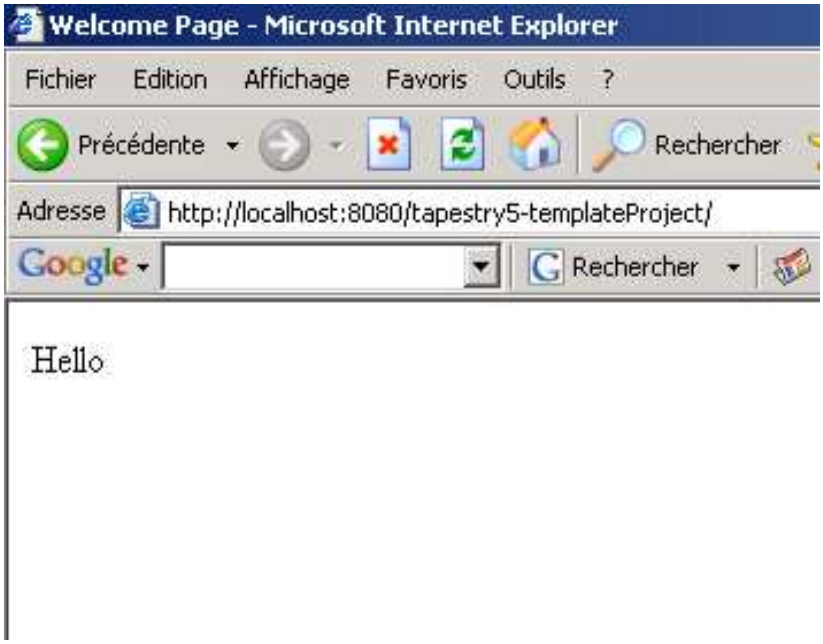
La Constante **COMPRESS_WHITESPACE** permet d'agir sur le rendu du code HTML. Si elle est à true, le code HTML sera compressé.

La Constante **PRODUCTION_MODE** permet d'avoir des rapports d'erreurs détaillés.

6. Ajouter un fichier « app.properties » dans le répertoire WEB-INF où « app » est le nom que vous avez donné à votre application



7. Démarrer l'application et se connecter à la racine « /tapestry5-templateProject » de l'application, observer le résultat



Le résultat obtenu correspond au contenu HTML du template Welcome.tml. On peut y voir qu'il n'est composé que de code HTML, aucun composant Tapestry n'y est associé.

Atos Origin

Etude des Mécanismes de Base

Atos Origin

Mécanismes de Base

- **Généralités**
- Traitement des requêtes
- Méthodologie
- Composants de base
- Gestion des formulaires
- Autres composants

- **Les Templates Tapestry sont des fichiers respectant la syntaxe XML**

⚠ **Pour pouvoir utiliser les caractères entités (Ex : «   »), utiliser un des DOCTYPE (X)HTML**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```


Si une page contient plusieurs composants contenant tous leur déclaration de DOCTYPE, seul le premier sera renvoyé à l'utilisateur.

- **Pour utiliser les composants Tapestry, il faut déclarer le « namespace » Tapestry**

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd"
  <head>
    <title>Hello World Page</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

- **Préciser l'encodage du fichier Template, ceci vous permettra notamment d'utiliser des caractères accentués ISO**

⚠ **Cette balise doit être la première du fichier Template (avant même le DOCTYPE) sinon le parser génère une erreur**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
…
</html>
```


Par défaut, l'encodage du fichier sera l'UTF-8.

- **Inclure une ressource statique (Ex : image) dans le Template**
  - **Utiliser l'expansion « ${asset:<filepath>} »**
  - **Tapestry vérifie l'existence du fichier et crée une URL adaptée pour que le client accède à la ressource**

- **L'utilisation de l'expansion « asset: » rend le Template inutilisable pour le travail avec le Studio**
- **Tapestry vérifie que la ressource existe et génère une URL**

Les Assets sont des fichiers qui peuvent être téléchargés en complément des fichiers HTML : des images, des fichiers javascript, des feuilles de styles …

L'utilisation du mot clé asset lors de la déclaration d'une ressource statique, dans un template, permet au framework Tapestry

- de vérifier si le fichier existe
- de générer des URLs propres à chaque ressource, permettant ainsi de les mettre dans le cache.

- **L'expansion « ${asset:<filepath>}, détails « filepath » :**
  - **Utiliser le préfixe « context: » si l'image se situe dans le répertoire de documents de l'application**

    **Ex : **

  - **Utiliser le préfixe « classpath » si l'image est dans le classpath de l'application**

    **Ex : **

⚠ **Préférer l'utilisation du préfixe « context: »**

Pour la production, ne pas utiliser le préfixe « classpath: » pour éviter à Tomcat de servir les ressources statiques et faciliter le travail de déploiement

- **Apparence des composants et personnalisation**
  - **Utilisation des CSS**
    - Les composants utilisent les feuilles de style CSS pour paramétrer leur apparence
    - Tapestry inclut par défaut un lien vers une CSS propriétaire « default.css »
    - Il est possible de redéfinir les styles par défaut en injectant dans notre page une feuille de style
  - **Pour inclure une feuille de style dans votre Template, utiliser le même mécanisme qu'avec les images pour l'attribut « href »**

```
<style type="text/css">@import url(${asset:context:static/css/style.css});</style>
```

⚠ **Toutes les « css » définies par l'utilisateur passent devant celles fournies par défaut Tapestry**

- **Accès aux attributs de la classe dans un Template via les expansions**
  - **Ex : pour accéder à l'attribut « Name » de la classe**
    - Créer les accesseurs associés à cet attribut

```
public String getName() {
  return name;
}
public void setName(String aName){
  name = aName;
}
```

    - Utiliser l'expansion `${name}`

- **Accès aux attributs de la classe dans un Template via les expansions**
  - **Fonctionne aussi pour les objets complexes ayant eux-mêmes des propriétés**

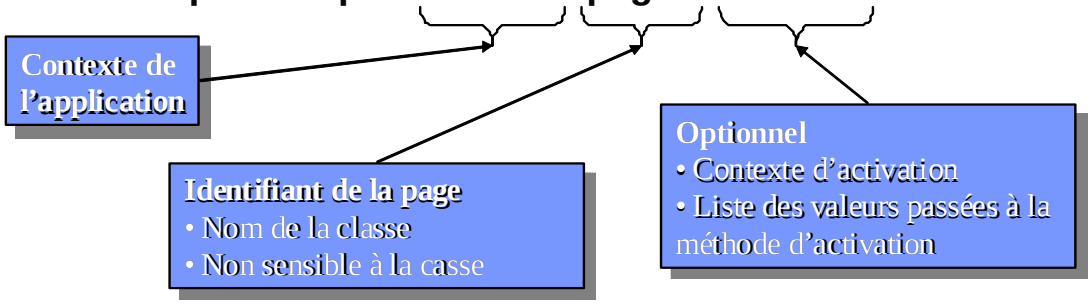
Ex : « `${monObjet.name}` » sera traduit en « `getMonObjet().getName()` »

  - **Si l'objet est susceptible d'être « null », utiliser « ?. » pour effectuer une vérification et éviter une exception. Ex: « `${monObjet?.name}` »**
  - **N.B. : Si l'objet est « null » c'est un chaîne vide qui s'affiche**

- **Utilisation des « ResourceBundle »**
  - **Les composants utilisent des fichiers de ressources par défaut pour obtenir des messages génériques**
  - **Ils utilisent aussi les messages de ressources associés aux pages de l'application pour obtenir certaines informations (labels, messages d'erreur, titre de colonne…)**
  - **Pour associer un fichier de ressources à « MaPage.java », créer le fichier « MaPage.properties »**
  - **Catalogue global : WEB-INF/*AppName*.properties**
- **Accès aux messages depuis le Template**
  - **Utiliser l'expansion ${message:key} où « key » est la clé du message qui permet de stocker un message dans le fichier de propriétés**

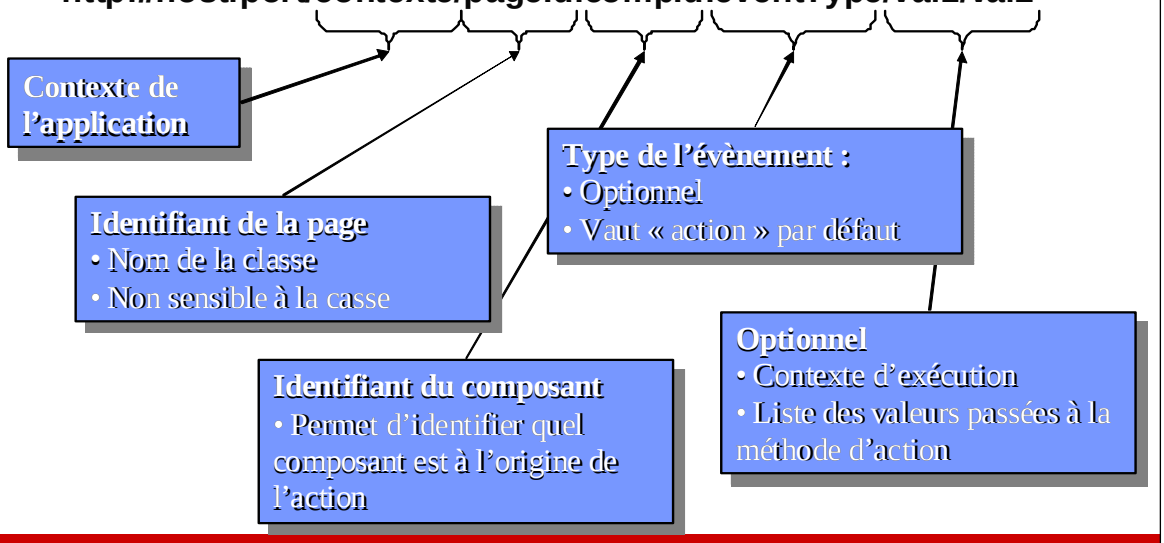
- **Tapestry gère son propre format d'URL**
  - **Améliorer la lisibilité**
  - **Faciliter le référencement**
  - **Proposer des URLs « bookmarkable »**
- **Requête de type « Rendu »**

http://host:port/contexte/pageId/val1/val2



- **Requête de type « Action »**

http://host:port/contexte/pageId.compId:eventType/val1/val2



TP 2 : Hello World

Concepts abordés

- Template Tapestry et pages Tapestry
- Inclusion de ressources statiques
- Catalogue global de messages
- Expansion de type « message » et « asset »

Enoncé

Compléter le Template Tapestry de la page « Welcome » :

1. Ajouter un lien vers le fichier css « style.css » situé dans le répertoire « static/css ». Utiliser la balise « style » dans la partie Header du document

```
<head>
    <meta name="copyright" content="Atos Origin" />
    <title>Welcome Page</title>
    <style type="text/css">@import url(${asset:context:static/css/style.css});</style>
</head>
```

L'utilisation du mot-clef context indique que notre ressource statique se trouve dans la webapp de notre application

L'utilisation du mot-clef asset permet de bénéficier des traitements realises par Tapestry : Vérification de l'existence de la ressource, génération d'URL. Tapestry utilise le numéro de version (paramètrable dans l'AppModule) pour générer les URLs de nos ressources statiques. Ainsi la première fois que l'utilisateur lancera notre application, toutes les ressources seront récupérées, et grâce à la génération d'URLs, elles seront mises dans le cache de l'utilisateur. Le numéro de version étant dans l'URL, dès qu'une nouvelle version sera disponible, toutes les ressources seront récupérées une nouvelle fois car leurs URLs auront changés.

2. Afficher l'image « logo.gif » située dans le répertoire « static/images/logo.gif ». Utiliser la balise HTML « img »

```
<p><img src="${asset:context:static/images/logo.gif}" /></p>
```

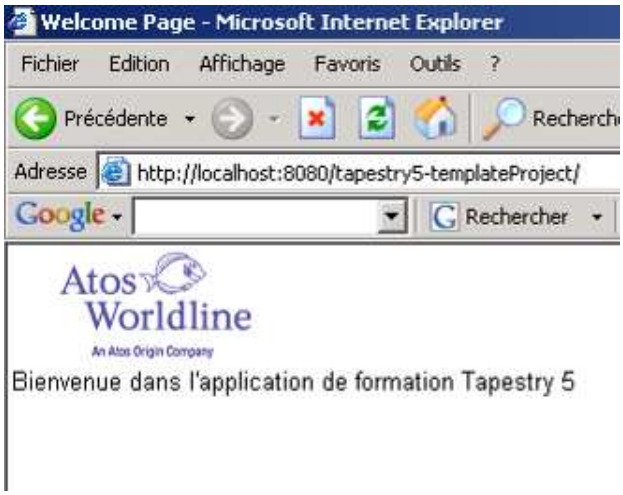
3. Ajouter un message de bienvenue dans le fichier de ressources global « app.properties »

```
welcome=Bienvenue dans l'application de formation Tapestry 5
```

4. Afficher ce message dans une balise « p » HTML

```
<p>${message:welcome}</p>
```

5. Tester l'affichage de la page « Welcome »



Note : Visualiser le code de la page résultante et observer les URLs. Par défaut celles-ci sont optimisée par rapport à la ressource demandée (donc relatives si possible). Il est possible de forcer la génération d'URL absolue, modifier la classe « AppModule » pour fixer le paramètre « tapestry.force-absolute-uris » à « true ». Pour information, certaines des constantes de configuration Tapestry sont situées dans la classe « SymbolConstants ».

Redémarrer l'application, et observer les nouvelles URLs générées.

- **Tapestry utilise les annotations pour identifier le rôle des variables d'instance ou des méthodes des pages de l'application**
  - Persistance des variables
  - Injection de dépendances (Agrégation de composant, Page pour la navigation, Service)
  - Identification des méthodes d'activation
  - Identification des phases de rendu
  - Identification des paramètres pour les composants

- **Les variables d'instances de pages ne sont pas associées à un utilisateur par défaut**
- **Par défaut, elles sont réinitialisées à la fin de chaque requête**
- **Trois annotations pour gérer la persistance**
  - @Persist : Stockage d'une information liée à une page pour la session utilisateur
  - @SessionState : Partage d'informations entre toutes les pages de l'application pour une session utilisateur donnée
  - ⚠ **Ne pas confondre avec le scope « application » du J2EE**
  - @Retain : « lazy loading »
- ⚠ **Seules ces annotations doivent être utilisées pour gérer la persistance des informations dans les pages et composants de l'application**
- ⚠ **Eviter d'utiliser le mot clé « static »**

@Persist : A chaque fois qu'un utilisateur va retourner sur la page, il retrouvera sa valeur pour cette variable.
@SessionState : Cette annotation est identique à **@Persist**, mais la variable est disponible sur toutes les pages.
@Retain : Cette annotation permet que la valeur soit initialisée lors de la création de l'instance d'une page. Si on récupère une instance de page qui a déjà été créée, la variable annotée en @Retain ne sera pas réinitialisée.

@Persist

- **Permet de faire persister une information dans une page donnée pour un utilisateur donné**
- **Par défaut, c'est la session HTTP qui est utilisée pour stocker l'information, il existe d'autres stratégies**
- **Pour préciser explicitement la stratégie de stockage**
  - **@Persist("session") : Utiliser la session HTTP**
  - **@Persist("flash") : Stocker l'information en session et la supprimer au premier accès**
  - **@Persist("client") : Stocker l'information dans un champ de l'URL ou dans un champ caché pour les pages contenant des formulaires**
- **Mettre la variable à 'null' pour la supprimer de la session**

L'une des applications de l'annotation **@Persist** est le passage de paramètre entre 2 pages. Par exemple : Nous sommes dans une Page A. Nous créons une instance de la page B, et nous manipulons l'un de ses attributs qui est annoté **@Persist**. Après une redirection vers la page B, nous pourrons récupérer la valeur de cet attribut, initialisé dans la page précédente.

@SessionState

- **Cette annotation indique à Tapestry qu'il doit conserver l'information pour la session utilisateur**
- **La visibilité de la variable est « l'ensemble des pages de l'application »**

  Ex :

  @SessionState
  private User loggedUser;

  Cette variable permet d'obtenir une référence sur l'utilisateur authentifié depuis toutes les pages de l'application
- ⚠ **C'est le type qui définit une variable « session state » et non l'identifiant de celle-ci comme pour le @Persist. Autrement dit, on ne peut pas avoir deux variables « session state » de même type avec des valeurs différentes.**
- **Pour vérifier si cette variable existe, créer un booléen <varName>Exists**

  Ex :

  private boolean loggedUserExists

  Cette variable est instrumentée automatiquement par Tapestry si la variable « loggedUser » est déclarée dans la classe et est annotée @SessionState

  Note : @SessionState remplace l'annotation dépréciée @ApplicationState
- **Mettre la variable à 'null' pour la supprimer**

@Retain

- **La portée est limitée au cycle de vie d'une instance de la page et n'est pas liée à l'utilisateur**
- **Cette annotation indique à Tapestry que la variable ne doit pas être réinitialisée à la fin du Traitement de la requête**
- **L'information n'est pas liée à l'utilisateur, mais cela permet d'éviter la création d'objet lourd à chaque requête**

  Ex :

  @Retain
  private UserManager manager;

- **Autres annotations**
  - **@InjectPage : permet d'injecter une instance de page dans une autre. L'instance ainsi créée par Tapestry pourra être utilisée dans le cadre de la navigation**
  - **@Inject : injecte un élément de base de Tapestry , ou un élément configuré depuis un module applicatif**
    - Placée devant une variable d'instance, cette annotation indique à Tapestry qu'il doit injecter une instance de l'objet dans la page
    - Permet à Tapestry de gérer le cycle de vie de ce type d'objet
    - Répond seulement pour un nombre défini de Types que nous verrons au fil de l'exposé (Ex : java.util.Locale, org.tapestry.ioc.Messages, Log …)

Type	Annoté par…	Description fonction et durée de vie
Page	-	Une instance de page peut être utilisée pour plusieurs utilisateurs, l'état de la page pour un utilisateur donné est retrouvé à chaque requête pendant la phase « pageAttached »
Variable d'instance de page	Sans annotation	Une variable d'instance non annotée est toujours initialisée à sa valeur par défaut (déclaration) en fin de traitement (action ou rendu)
Variable d'instance de page	@Persist	La variable est disponible pour une session utilisateur donnée dans la page en question. A chaque requête, ce type de donnée est initialisée en fonction du client pendant la phase « pageAttached »
Variable d'instance de page	@Retain	Dans ce cas si la valeur de la variable d'instance change au cours de la requête, elle n'est pas réinitialisée à la fin de la requête, mais elle n'est pas liée à l'utilisateur

Type	Annoté par…	Description fonction et durée de vie
Variable d'instance de page	@SessionState	L'information est disponible dans toutes les pages de l'application. L'utilisateur peut retrouver l'information dans toutes les pages en fonction du type de la variable.
Variable d'instance de page	@Inject	Il s'agit d'un service automatiquement injecté par Tapestry dans la page. La variable existe à chaque requête, la durée de vie de l'objet en question est défini au moment de la déclaration ou de l'écriture du service : • Une instance par Thread • Une instance pour tous les Threads de l'application
Variable d'instance de page	@InjectPage	Il s'agit d'une instance de page qui permettra à l'utilisateur de réaliser de la navigation tout en modifiant l'état de la dite page. ⚠ Les variables modifiées sur l'instance de page doivent être persistantes pour conserver leur état (@Persist)

TP 3 : Persistance

Concepts abordés

- Scopes de persistance (@Persist, @Retain)

Ce TP permet de préparer le générateur de nombre aléatoire qui sera utilisé pour le jeu Hilo, et a pour but de mettre en évidence la différence entre les scopes de stockage « Persist » et « Retain ». Il a pour but de mettre la différence entre la persistance d'information en session utilisateur et la persistance des variables d'instance au sens Java. Pour rappel, les données de type @Persist permettent de persister des informations propres à l'utilisateur alors que le @Retain ne fait qu'empêcher Tapestry de réinitialiser les variables d'instances des pages lors du retour de la page dans le pool.

Enoncé

1. Observer la sortie standard lors des accès à la page « Welcome » et remarquer les traces concernant l'initialisation des variables « seed » et « sessionStart »

Ces deux variables dont déclarées à chaque rafraichissement de la page Welcome.

2. Ajouter les annotations adéquates sur les variables « seed » et « sessionStart » d'instance en fonction des commentaires

La variable **sessionStart** doit stocker en session la première date d'accès à cette page. Nous préférons donc @Persist à @SessionState, car cet attribut est propre à cette page.

Pour la variable **seed**, nous utiliserons l'annotation @Retain, car cette variable d'instance ne doit pas être réinitialisée.

3. Ajouter dans le Template HTML de la page « Welcome » une expansion permettant d'afficher la valeur de la variable « sessionStart » juste après le message de bienvenue

```
<p>${message:welcome} (Session Start : ${sessionStart})</p>
```

4. Effacer les traces de la sortie standard
5. Exécuter la page une première fois et visualiser la sortie standard, exécuter la page une deuxième fois et observer la sortie standard

Nous remarquons que les deux variables **seed** et **sessionStart** ne sont initialisées qu'une seule fois. Si vous ouvrez la même page avec un autre navigateur, vous vous apercevrez que seule la variable sessionStart est initialisée.

Par contre la variable **seed** n'est pas réinitialisée, car quelque soit le navigateur nous utilisons la même instance de la page.

6. Effectuer une modification quelconque de la classe, ré-exécuter la page et observer la sortie standard.

Seule la variable « random » a été réinitialisée car l'instance de la page a été recréée suite à la modification alors que la variable « sessionStart » est récupérée depuis la session et contient toujours la même valeur

- **Deux méthodes pour insérer un composant dans une page**
  - **Utiliser le namespace Tapestry pour localiser le composant « t:component »,** Ex : <t:beaneditform …></t:beaneditform>
  - **Utiliser l'instrumentation invisible via l'attribut « t:type » qui permet d'identifier un tag HTML comme étant un composant** Ex : <tr **t:type**="loop"…>…</tr>
- ⚠ **Préférer l'instrumentation invisible**
  - **Permet de conserver un aperçu statique de la page une fois celle-ci dynamisée**
  - **Facilite le travail avec le Studio**
- ⚠ **Identifier les composants à l'aide « t:id »**

Pour obtenir une référence sur le composant dans la classe, utiliser l'annotation « @Component »

Ex :

Dans le Template

```
<form t:type="form" t:id="monForm"…>…</form>
```

Pour obtenir une référence dans la classe de la page

```
@Component(id="monForm")
private Form monForm;
```

- **Passer des paramètres aux composants dans le Template**
  - La plupart des composants Tapestry possèdent des paramètres accessibles via les attributs XML
  - La conversion dans le type de destination se fait de manière implicite par Tapestry
  - On utilise un préfixe pour indiquer où trouver la valeur du paramètre (« prop: », « literal: »…)
- ⚠ **Ne pas utiliser les expansions pour préciser la valeur d'un paramètre. Dans la plupart des cas, celle-ci ne sera pas réévaluée par Tapestry**

Atos Origin

Généralités
Interaction Page / Composant

- **Passage de paramètres et namespace**
  - Il n'est pas obligatoire de préciser le namespace « t: » devant les paramètres affectés au composant
  - Cependant, l'utilisation du namespace pourra vous permettre d'éviter des erreurs de validation si vous choisissez d'utiliser par exemple des validateurs XHTML

- **Passage de paramètres et préfixe**
  - **« prop:xxx » (préfixe par défaut), dans ce cas la méthode « getXxx() » de la classe Java associée à la page est appelée**
  - **Ex :**

```
<input t:type="textfield" t:value="prop:login"… />
```

```
public String getLogin(){
  return this.login;
}
public void setLogin(String aLogin){
  this.login = aLogin;
}
```


- **Passage de paramètres et préfixe**
  - **Utiliser le préfixe « literal:xxx » , dans ce cas l'attribut utilisera la chaîne « xxx » pour valeur et une conversion pourra éventuellement s'opérer**
  - **Ex :**

```
<div t:type="errors" t:banner="literal:Message d'erreur"  … />
```

- **Passage de paramètres et préfixe**
  - **« message:xxx » permet d'extraire une valeur d'un fichier de propriétés**
  - **Ex :**

MaPage.tml :

```
<div t:type="errors" t:banner="message:errorMsg"  … />
```

MaPage.properties :

```
errorMsg=Message d'erreur
```


- **Deux exemples simples**
  - **Faire un lien vers une page**

```
<a t:type="pagelink" t:id="monLien"
   t:page="maPage">Lien</a>
```

  - **Exécuter une action**

```
<a t:type="actionlink" t:id="monAction">Action</a>
```

Documentation de référence des composants :
http://tapestry.apache.org/tapestry5/tapestry-core/component-parameters.html

TP 4 : Utilisation de composants Simples

Concepts abordés

- Utilisation d'un composant utilisateur simple
- Page et fichier de ressources
- Utilisation de composants Tapestry simple (actionlink, pagelink)
- Passage de paramètres

Liens

http://tapestry.apache.org/tapestry5/tapestry-core/guide/templates.html

Enoncé

1. Préparation du Template « Welcome.tml »
    a. Ajouter le DOCTYPE « xhtml1-strict » (cf. Template de la page « Login »)
    b. Ajouter le NAMESPACE Tapestry à la racine (cf. Template de la page « Login »)
2. Remplacer l'image par le composant Banner fournit avec le TP

```
<div t:type="banner"/>
```

1. Créer un lien de type action ayant pour identifiant « hilo »

```
<p><a t:type="actionlink" t:id="hilo">Start Hilo Game</a></p>
```

2. Créer un fichier de ressources spécifique à la page Welcome au même niveau que le fichier Template
3. Ajouter un lien de navigation vers la page de Login

```
<p><a t:type="pagelink" t:page="login">${message:enter)</a></p>
```

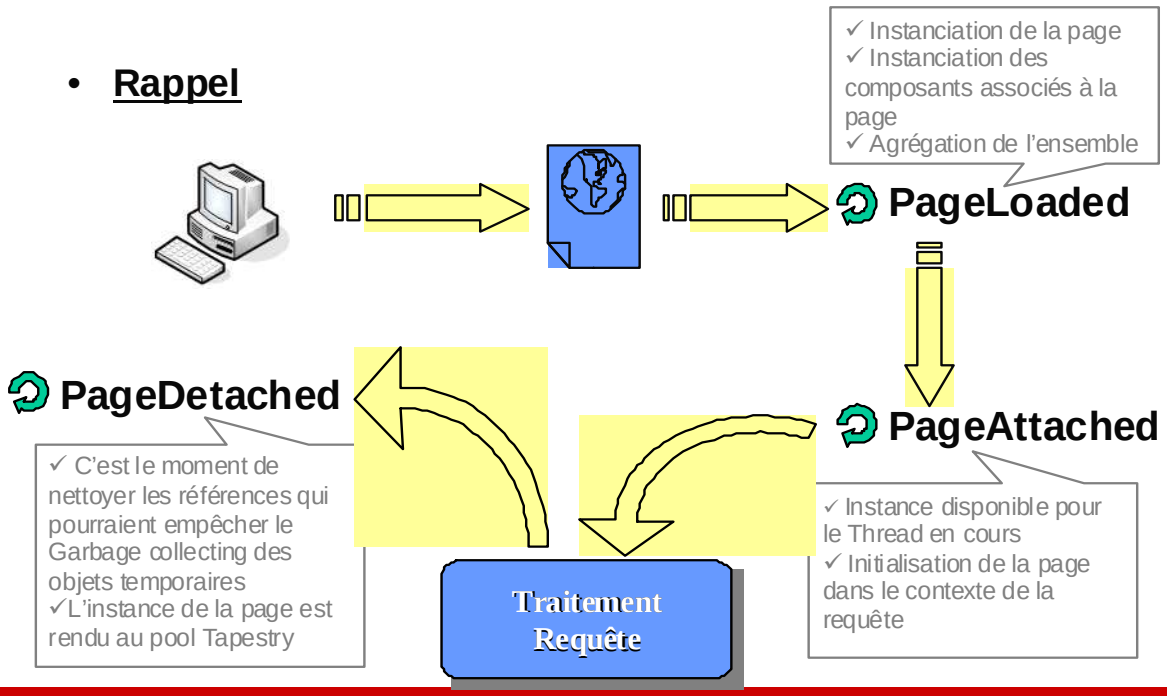
4. Redémarrer le serveur (pour prendre en compte les nouveaux fichiers de propriétés), tester la page et les liens que vous avez ajouté

Quand nous ajoutons de nouveaux fichiers properties, il est nécessaire de redémarrer le serveur Jetty. Ensuite, si vous le modifiez, le rechargement à chaud fonctionne.

Tapestry 5 Training Application

Bienvenue dans l'application de formation Tapestry 5 (Session Start : Thu Dec 09 17:36:23 CET 2010)
Start Hilo Game
Enter portfolio application

- **Rappel**



Mécanismes de Base

- Généralités
- **Traitement des requêtes**
- Méthodologie
- Composants de base
- Gestion des formulaires
- Autres composants

Atos Origin

Traitement des requêtes
Activation

- **Toutes les requêtes entrantes (action ou rendu) passent par la phase d'activation**
- **L'activation permet à la page de restaurer son état interne à partir des données encodées dans l'URL (contexte d'activation)**
- **Permet d'éviter le stockage d'informations en session**
- **Une méthode d'activation sera annotée par @OnEvent("activate")**

Ex :

```
@OnEvent("activate")
private void setupManager(){
   this.manager = UserManager.getInstance();
}
```


Atos Origin

Traitement des requêtes
Activation

- **Une méthode d'activation peut contenir des paramètres**
- **Ces paramètres sont passés dans l'URL de l'application via le contexte (en gras ci-dessous)**

Ex : http://localhost:8000/user/**3**

- **Dans ce cas la méthode d'activation peut contenir un paramètre de type long**

```
@OnEvent("activate")
private void getUserFromDb(long id){
  …
}
```

Traitement des requêtes
Passivation

- **De la même manière qu'il existe une méthode d'activation, il est possible de définir la passivation d'une page**

```
@OnEvent("passivate")
public long retrieveUserFromDb(){
  return this.id;
}
```

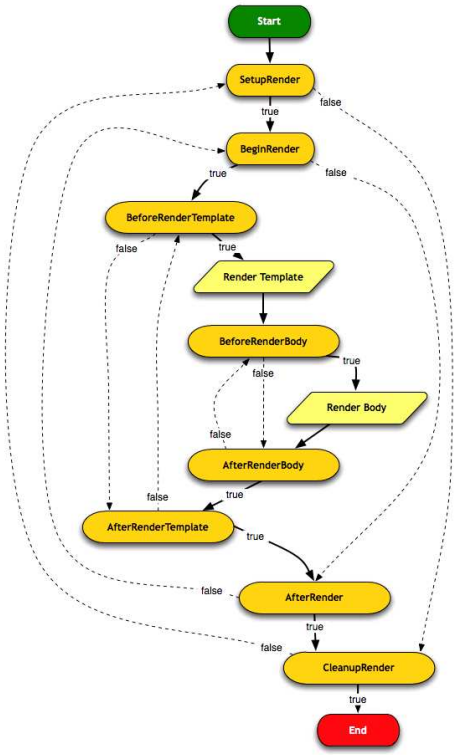
- **Une méthode de passivation doit renvoyer les éléments nécessaires à la réactivation de la page**
- **Cette méthode est utilisée par Tapestry pour modifier les URLs faisant référence à la page en question en y ajoutant le contexte d'action**
- **Ceci permet d'obtenir des URLs « Bookmarkables »**

Traitement des requêtes
Rendu - automate

- **Le rendu des composants est basé sur un automate à états**
- **Le processus de rendu est composé de parties simples**
  - **Facilité d'implémentation ou de surcharge**

Atos Origin

- **Les phases orange correspondent chacune à une annotation**
  - **SetupRender**
  - **BeginRender**
  - **...**
- **Les méthodes marquées par ces annotations sont appelés les « méthodes de rendu »**
- **Ces méthodes sont appelées par Tapestry pour afficher une page**
- **Peuvent renvoyer « void » ou un booléen**
  - **false: flèche en pointillé**
  - **« void » ou true: flèche pleine**



Tapestry propose différents états lord d'un rendu d'un composant. Ces états font référence à des annotations, qui peuvent être associées à des méthodes. Ces méthodes seront exécutées dans l'ordre des états.

On peut par exemple décider qu'à l'état BeforeRenderBody, des données soient affectées à des champs d'un formulaire. Ensuite à l'état AfterRenderBody, on peut par exemple réinitialiser certaines variables.

Cet automate à état permet de naviguer entre chaque état. Si nous retournons true, nous passons à l'état suivant, sinon nous revenons à l'état précédent.

Nous ne sommes pas obligés d'implémenter toutes ces méthodes.

Atos Origin

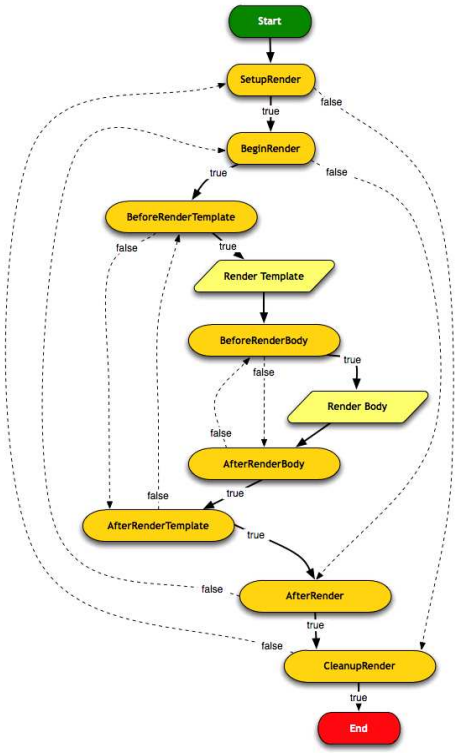
Traitement des requêtes

Rendu – Méthodes de rendu

- **Les méthodes des phases de rendu peuvent prendre un paramètre du type « MarkupWriter »**
- **Cet objet va permettre de modifier le fichier DOM généré à partir du Template par Tapestry**
- **Méthodes optionnelles**
- **La manipulation du DOM peut engendrer des problèmes de performances**
- **Plutôt utiliser le Template dans la mesure du possible**

Atos Origin

```
public class Count {
 @Parameter
 private int _start = 1;

 @Parameter(required = true)
 private int _end;

 @Parameter
 private int _value;
 private boolean _increment;

 @SetupRender
 void initializeValue() {
   _value = _start;
   _increment = _start < _end;
 }

 @AfterRender
 boolean next() {
   if (_increment) {
     int newValue = _value + 1;
     if (newValue <= _end) {
       _value = newValue;
       return false;
     }
   } else {
     int newValue = _value - 1;
     if (newValue >= _end) {
       _value = newValue;
       return false;
     }
   }
   return true;
  }
}
```



69 *Turning Client Vision into Results*

Cet exemple implémente des méthodes que pour deux états.

La méthode initializeValue permet d'initialiser deux variables. La méthode next() teste une variable est inférieure, supérieure ou égale à la variable end.

Si elle n'est pas égale, Tapestry retourne false, et le traitement retourne à l'état BeginRender. Comme aucune méthode n'est déclarée entre les états @SetupRender et @AfterRender, Tapestry reboucle sur la méthode next(), tant que la valeur est différente.

Traitement des requêtes
Rendu – SetupRender

@SetupRender

- **Dans la méthode associée à cette annotation, on peut préparer le rendu pour le composant**
  - **Lire des paramètres**
  - **Créer des variables temporaires**
  - **Éventuellement appeler le MIDDLE pour pré-charger des données**
  - **…**
- **Renvoyer « true » ou « void » fait avancer l'automate vers la phase « BeginRender »**
- **Renvoyer « false » saute cette phase, et ce rend directement dans la phase « CleanupRender »**

Traitement des requêtes
Rendu – CleanupRender

@CleanupRender

- **Le pendant de « SetupRender », permet de « nettoyer » les variables inutilisées ou les connexions inutiles**
- **Renvoyer « false » nous fait retourner sur « SetupRender »**

Traitement des requêtes
Rendu – Annotations vs Nommage

- Il est possible de remplacer l'utilisation des annotations par une norme de nommage spécifique pour les phases de rendu
- Dans ce cas, le nom de la méthode est le nom de l'annotation, avec le premier caractère en minuscule:
  - setupRender()
  - beginRender(), etc
- Le nom des méthodes est très générique
- Si le Framework évolue, le « re-factoring » peut devenir complexe
- De manière générale, préférer l'utilisation des annotations et l'écriture de noms de méthodes explicites, ex : « setupRender » devient « setupListeDesPays »

Traitement des requêtes
Action

- Les requêtes de type « Action » peuvent être générées par des composants tels que « actionlink »
- Une action est identifiée par
  - Le type d'évènement (Ex : action, submit, failure…)
  - L'identifiant du composant émetteur
- Utiliser l'annotation OnEvent pour identifier les méthodes d'action

```
@OnEvent(value="action", component="select")
 void valueChosen(int aValue) {
   value = aValue;
}
```

Atos Origin

Traitement des requêtes
Action - OnEvent

@OnEvent(value="…", component="…")

- **L'annotation OnEvent permet d'identifier les méthodes qui traiteront les requêtes de type « Action »**
- **Cette annotation possède deux paramètres**
  - **value : nom de l'évènement qu'il faut matcher**
  - **component : identifiant du composant émetteur**
- **Par défaut « value » vaut « action » qui est émis par les composants de type « actionlink » ou « form »**
- ⚠ **Si le paramètre « component » est omis, la méthode sera exécutée pour tous les événements du type défini par le paramètre value**

Atos Origin

Traitement des requêtes
Action - Navigation

- **La valeur de retour des méthodes d'action définit la requête de rendu qui sera exécutée juste après l'action par Tapestry**
  - **« void » ou « null » : la page courante servira de rendu**
  - **String : Tapestry redirige l'utilisateur vers la page identifiée par la chaîne de caractère**

  Ex : return "Welcome";
  - **Class : Tapestry redirige l'utilisateur vers la page associée à cette classe**

  Ex : return Welcome.class;

Atos Origin

Traitement des requêtes
Action - Navigation

- **Valeur de retour des méthodes d'action (suite)**
  - **Link : Cet objet Tapestry permet de rediriger l'utilisateur. Ce type d'objet pourra être créé via l'objet « ComponentResources » à injecter préalablement dans la page**
  - **StreamResponse : Il s'agit d'une interface permettant de renvoyer du contenu binaire à l'utilisateur.**
  - **URL (API J2SE) : pour rediriger l'utilisateur vers cette URL**
  - **Autre : Tout autre type est considéré comme une erreur**

Atos Origin

Traitement des requêtes
Action - Navigation

- **Valeur de retour des méthodes d'action (suite)**
  - **Page : Il s'agit dans ce cas de renvoyer une instance de page injectée préalablement via l'annotation @InjectPage. Ceci permet d'effectuer du traitement sur l'instance avant de rediriger l'utilisateur vers celle-ci**

Ex :

```
@InjectPage
private Welcome welcome;

@OnEvent("action")
private Object back(){
   welcome.setBackMessage("…");
   return welcome;
}
```

Atos Origin

Traitement des requêtes
Action – Annotations vs Nommage

- **Il est possible de remplacer l'utilisation des annotations par une norme de nommage spécifique pour les méthodes d'action**
- **Le nom de ces méthodes doit commencer par le préfixe « on », suivi du nom de l'action (capitalisé)**
- **Pour préciser l'élément à l'origine de l'évènement, suffixer par « From » suivi de l'identifiant du composant capitalisé.**
- **Ex :**

```
void onActionFromSelect(int aValue) {
 value = aValue;
}
```

- **Si le Framework évolue, le « refactoring » peut devenir complexe**
- **De manière générale, préférer l'utilisation des annotations et l'écriture de noms de méthodes explicites**

TP 5 : Initialisation du jeu Hilo (1/2)

Concepts abordés

- Gestion des scopes de persistance (@Persist)
- Gestion des évènements Tapestry (@OnEvent)
- Gestion de la navigation entre les pages (@InjectPage)

Voici la cinématique du jeu dans l'application :

1. La page « Welcome » permet d'initialiser le jeu en définissant un nombre à deviner
2. Une fois le jeu initialisé, c'est la page « Guess » qui propose à l'utilisateur la liste des numéros disponibles
3. A chaque tentative de l'utilisateur, la page « Guess » vérifie si l'utilisateur a trouvé le bon numéro
    a. Si l'utilisateur a réussi il est renvoyé vers la page « Welcome » qui lui affiche le nombre de tentatives
    b. Si l'utilisateur échoue, la page « Guess » s'affiche de nouveau avec un message adapté en fonction de sa dernière sélection

Enoncé

Initialisation du jeu : Implémenter la classe « Welcome »

1. Ajouter l'annotation adéquate sur la variable d'instance « guess » en fonction des commentaires

```
/**
 * This object can be returned by event method to redirect to the
 * corresponding page. This page must be injected by Tapestry
 */
@InjectPage
private Guess guess;
```

La variable guess fait référence à une page de notre projet. Pour pouvoir utiliser un attribut ou une méthode de cette page, il faut l'instancier et ajouter l'annotation @InjectPage

2. Renommer la méthode « onactivate » en « initializeRandomizer » et ajouter l'annotation correspondante qui permet d'exécuter cette méthode sur l'évènement « activate »

```
@OnEvent(EventConstants.ACTIVATE)
public void initializeRandomizer()
{
    if (seed == null)
    {
        seed = System.currentTimeMillis();
        System.out.println("Initialize randomizer with seed :" + seed);
    }
    if (sessionStart == null)
    {
        sessionStart = new Date();
        System.out.println("Initialize first page access time :" + sessionStart);
    }
}
```

Pour pouvoir associer une méthode à un événement de la page (ou d'un composant), il faut utiliser l'annotation @OnEvent. Cette annotation prend en paramètre le nom de l'événement.

La Constante **EventConstants.Activate** (disponible que pour les pages) correspond au moment est activé entièrement.

3. Annoter la méthode « startHiloGame() » pour qu'elle réponde à l'évènement « action » du lien « hilo » créé dans le TP précédent
4. Implémenter le corps de la méthode « startHiloGame() » pour initialiser le jeu en utilisant l'instance de la page Guess et la méthode « setupGame », rediriger ensuite l'utilisateur vers cette même page en utilisant la variable « guess » pour valeur de retour de la méthode d'action

```
/**
 * This method must be call on actionlink to setup the hilo Game and
 * redirect the user to the Guess Page.
 *
 * @return Guess Tapestry handle this object to redirect the user to thus pa
 */
@OnEvent(value = EventConstants.ACTION, component = "hilo")
private Guess startHiloGame()
{

System.out.println("Initializing Hilo Game");

// Setup hilo game by using the Guess injected page
// There is a corresponding setup method in Guess class
guess.setupGame(seed);

return guess;
}
```

Le paramètre component de l'annotation @OnEvent permet de spécifier le composant sur lequel la méthode est associée.

Pour initialiser le jeu, il suffit de faire appel de la méthode setupGame de la classe Guess, grâce à son instanciation guess.

Atos Origin

Mécanismes de Base

- Généralités
- Traitement des requêtes
- **Méthodologie**
- Composants de base
- Gestion des formulaires
- Autres composants

Atos Origin

Méthodologie
Tapestry et les appels Middle

Quand Appeler le modèle ?

- **Si l'appel doit être exécuté pour toutes les requêtes (action ou rendu) avant l'exécution effective**
  - **Utiliser la phase d'activation @OnEvent("activate")**
- **Pour une requête de type « Rendu », utiliser la phase « SetupRender » pour initialiser des données d'affichage**
- **Exemple : Pour obtenir des données d'initialisation de champs de formulaires ou de certaines zones de la page (Ex : Liste de pays pré-remplie)**

⚠ **Ne pas utiliser les annotations de rendu pour appeler le MIDDLE**

⚠ **Ne pas utiliser les accesseurs pour appeler le MIDDLE**

Quand Appeler le modèle ?

- Dans les méthodes de type « Action » qui permettent de traiter les évènement provenant de l'utilisateur
- Exemple : action « success » pour le traitement des formulaires
- Ce type de méthode permet d'identifier clairement le fonctionnel de l'application et les accès à la couche métier

Comment écrire une vue ?

- Exploiter au maximum le Template HTML
- Utiliser des accesseurs et mutateurs simples pour l'accès aux données de la classe
- ⚠ Éviter l'insertion de code de présentation dans la classe de Traitement via les phases de rendu
- ⚠ Éviter l'utilisation de l'objet « MarkupWriter » fourni en entrée des méthodes de gestion du cycle de rendu

Atos Origin

Mécanismes de Base

- Généralités
- Traitement des requêtes
- Méthodologie
- **Composants de base**
- Gestion des formulaires
- Autres composants

Atos Origin

Composants de base
ActionLink

- **Le composant ActionLink crée une URL associée à une action.**
- **L'URL identifie**
  - **la page qui contient le composant**
  - **L'identifiant du composant à l'intérieur de la page**
  - **Les données additionnelles de contexte**
- **URL d'exemple : http://localhost:8080/chooser.select/3**
  - **Page : « chooser »**
  - **Identifiant : « select »**
  - **Contexte : « 3 »**
- **Quand des données de contexte supplémentaires sont présentes, elles sont ajoutées à l'URL**

Lorsque l'utilisateur actionnera ce composant, Tapestry générera une URL :
`http://myhost/tapestry/mapage.monActionAFaire/Context`

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/ActionLink.html

Rappel : Comment associer une méthode d'action à un lien de type ActionLink ?

```
<p> Choose a number from 1 to 10: </p>
<p t:type="count" t:end="10" t:value="index">
  <a t:id="select" t:type="actionlink" t:context="index">${index}</a>
</p>
```

⚠ **Sur l'exemple ci-dessus, Tapestry détecte la boucle et génère des identifiants uniques**

```
@OnEvent(value = "action", component = "select")
void valueChosen(int aValue) {
    value = aValue;
}
```


- **Tapestry a :**
  - **Identifié la méthode valueChosen() comme méthode à invoquer**
  - **Converti la valeur contextuelle d'une chaîne vers un entier**
  - **L'a passée à la méthode**
- **Le composant ActionLink n'émet que des événements de type « action » ce qui explique l'annotation :**

  @OnEvent(value = "action", component = "select")

Atos Origin

Composants de base
If

- org.apache.tapestry5.corelib.components.If
- **Permet d'afficher une information de manière conditionnelle**

Nom	Type	Drapeaux	Par défaut	Préfixe par défaut	Description
else	org.apache.tapestry5.Block			prop	Permettre d'indiquer un block dans le cas où le test échoue
negate	boolean			Prop	Permet d'indiquer une propriété qui inverserait le test
test	boolean	Required		prop	Si la propriété vaut « true » le corps est affiché, sinon utilisation de l'attribut « else » ou rien n'est affiché

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/If.html

Ex : Si la variable d'instance « displayMessage » vaut « true » alors le « Message… » s'affiche

```
<p t:type="if" t:test="displayMessage">
Message…
</p>
```

TP 6 : Implémentation du jeu Hilo

Concepts abordés

- Gestion des scopes de persistance (@Persist)
- Evènements et contexte d'exécution (@OnEvent)
- Gestion de la navigation entre les pages (@InjectPage)
- Utilisation de composants de base Tapestry (if)

Enoncé

Implémentation du jeu : Implémenter la classe « Guess »

1. Ajouter les annotations sur les variables d'instance de la classe en fonction des commentaires. Attention, toutes les variables d'instance ne nécessitent pas une annotation

```
/**
 * This variable is used to store in session the last message for the user
 */
@Persist
private String message;

/**
 * This variable is used to store the current user choice
 */

private int guess;

/**
 * This variable is used to store in session the target number
 */
@Persist
private int target;

/**
 * Used to count the user attempts
 */
@Persist
private int count;

/**
 * Used to redirect the user after a successful hit and then display the
 * game result
 */
@InjectPage
private Welcome welcomePage;

/**
```

Les variables message, target et count doivent être stockées en session. Comme ils sont réservées à cette page, nous préférons l'annotation **@Persist**.

L'annotation **@InjectPage** est obligatoire pour instancier la page Welcome.

2. Ajouter l'annotation adéquate sur la méthode « verifyChoice » pour lui permettre de répondre aux liens de type « action » identifiés par « link »

```
@OnEvent(value = EventConstants.ACTION, component = "link")
private Object verifyChoice(int userChoice)
{
```

Ainsi, la méthode verifyChoice sera exécutée lorsque l'utilisateur fera une action sur le composant identifié par link.

3. Implémenter le corps de cette méthode pour, en cas de succès, affecter le message de succès sur la page de bienvenue avec le nombre de tentatives et rediriger l'utilisateur vers cette même page

```
@OnEvent(value = EventConstants.ACTION, component = "link")
 private Object verifyChoice(int userChoice)
{

count++;

if (userChoice == target)
{
    welcomePage.setMessageFromHilo(String.format("You have been successful in %d hits", count));
    return welcomePage;
}

if (userChoice < target)
{
    message = String.format("%d is too low.", userChoice);
} else
{
    message = String.format("%d is too high.", userChoice);
}

return null;
}
```

Si le résultat est correct, nous appelons la méthode setMessageFromHilop de la classe Welcome. Puis, nous redirigeons le visiteur vers cette page.

4. Implémenter le Template pour
    a. Ajouter les liens d'actions qui permettront d'exécuter la méthode « verifyChoice » avec le chiffre sélectionné par l'utilisateur
    b. Afficher le message « message » qui indique au joueur quelle direction prendre

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">

    <head>
        <meta name="copyright" content="Atos Origin" />
        <title>Guess Page</title>
        <style type="text/css">@import url(${asset:context:static/css/style.css});</style>
    </head>

    <body>

        <!-- Replace the message here with the dynamic value from Guess Page -->
        <div class="pForm">
            <span class="commts">${message}</span>
        </div>

        <div class="pForm">
            <h2>Make a guess between one and ten:</h2>
            <span class="pForm" t:type="loop" t:source="1..10" t:value="guess">
                <!--  Add and action link here to execute the verifyChoice method with the specified number -->
                <a t:type="actionlink" t:id="link" t:context="guess">${guess}</a>
            </span>
        </div>

    </body>

</html>
```

Nous utilisons le composant Loop, qui permettra d'afficher des chiffres de 1 à 10. Chaque chiffre correspondra à un actionLink, qui exécutera la méthode verifyChoice quand l'utilisera aura fait son choix. Le paramètre guess sera passé à cette fonction (grâce à l'attribut t:context qui fait référence à la valeur en cours).

Tous les ActionLink doivent nomalement avoir un ID différent (ici link). C'est Tapestry qui générera les ID automatiquement (link, link_0, link_1…)

5. Dans la page « Welcome », implémenter le Template HTML pour afficher la variable « messageFromHilo » si elle existe et qu'elle n'est pas vide. Utiliser le composant Tapestry « if » et la méthode « getLastHiloMessage » pour effectuer ce test

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsc

    <head>
        <meta name="copyright" content="Atos Origin" />
        <title>Welcome Page</title>
        <style type="text/css">@import url(${asset:context:static/css/style.css});</style>
    </head>

    <body>

        <div t:type="banner"/>

        <p>${message:welcome} (Session Start : ${sessionStart})</p>

        <p>
            <a t:type="actionlink" t:id="hilo">Start Hilo Game</a>
            <span t:type="if" t:test="lastHiloMessage">
                (${messageFromHilo})
            </span>
        </p>

        <p><a t:type="pagelink" t:page="login">${message:enter}</a></p>

    </body>

</html>
```

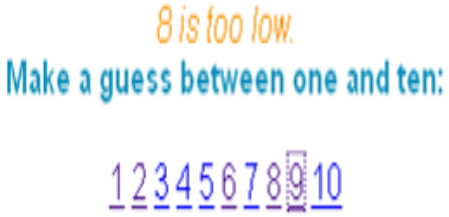
Nous utilisons le composant IF, qui permet de tester une condition avant d'afficher le corps du composant. Pour cela nous utilisons l'attribut t:test qui fait référence à une méthode (getlastHiloMessage), déclarée dans la classe java correspondant à ce template. Cette méthode indique si le message est nul.

```
/**
 * This method is used by the Tapestry "if" component to display or not a
 * message from Hilo Game in the welcome page.
 *
 * @return true if hilo has been played and successful, false otherwise
 */
public boolean getLastHiloMessage()
{
return (messageFromHilo != null && !("".equals(messageFromHilo)));
}
```

Si le test est vrai, nous affichons le résultat de la variable messageFromHilo, grâce à son getter getMessageFromHilo

6. Tester l'application Hilo

Si votre choix est différent de la solution, un message est affiché dans la page Guess.



Si vous trouvez la solution, vous êtes redirigé vers la page Welcome et message indiquant le nombre de tentative est affiché.

7. Modifier la page « Guess » pour rendre cette URL bookmarkable et autonome (initialisation du jeu, gestion de l'état de la page…) Utiliser le contexte d'activation/passivation pour stocker la graine et le nombre de tentatives. Dans ce cas c'est la page Guess qui initialisera un Randomizer avec une graine fournie par la page appelante.

```
@OnEvent(EventConstants.ACTIVATE)
private void init(int guess, int count, long seed)
{
    Random random = new Random(seed);

    this.seed = seed;
    this.guess = guess;
    this.count = count;
    this.target = random.nextInt(10) + 1;
}

@OnEvent(EventConstants.PASSIVATE)
public Object[] getContext()
{
    return (new Object[] { Integer.valueOf(guess), Integer.valueOf(count), Long.valueOf(seed) });
}
```

La première modification à faire est de supprimer les annotations correspondant **@Persist** aux variables target et count. On va faire transiter ces variables dans l'URL.

La méthode d'Activation permet de restituer l'état interne de la page à partir de ces 3 paramètres. Ces paramètres devront être renseignés dans l'URL.

La méthode de Passivation va permettre de poster ces variables dans l'URL. On renvoie simplement les paramètres que nous désirons intégrer aux URLs..

Aller plus loin

Utiliser la stratégie de persistance « flash » pour le message de succès dans la page « Welcome » et vérifier que le message ne s'affiche qu'une seule fois

Note : Il faut redémarrer le serveur après la modification de la stratégie de persistance

```
/**
 * This variable is used to store in session the last message for the user
 */
@Persist("flash")
private String message;
```

Pour implémenter une stratégie de persistance flash, il suffit de rajouter le paramètre **flash** à l'annotation **@Persist**.

Atos Origin

Mécanismes de Base

- Généralités
- Traitement des requêtes
- Méthodologie
- Composants de base
- **Gestion des formulaires**
- Autres composants

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Form.html

Atos Origin

Gestion des formulaires
Form

- **Tapestry fournit le composant « Form » et une suite d'autres composants qui peuvent être inclus dans le corps de ce composant pour…**
  - **Faciliter la création de formulaire HTML**
  - **Faciliter la conversion des données saisies par l'utilisateur**
  - **Faciliter la validation des champs du formulaire**
  - **Proposer une séquence de traitement à la soumission du formulaire**
  - **Gérer l'association entre le champ de formulaire et une propriété de JavaBean**

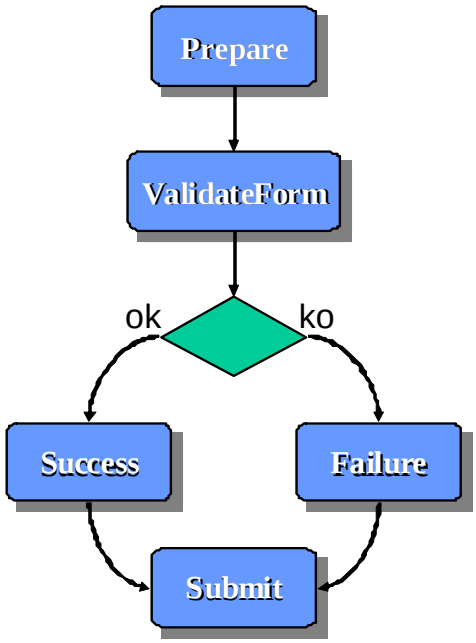
Par défaut, la méthode utilisée par les formulaires est la méthode POST.

Gestion des formulaires
Form

- **L'objet « Form » permet de dynamiser rapidement un formulaire HTML standard pour l'associer à des propriétés d'objet Java**
- **Un formulaire émet plusieurs événements**
  - « prepareForSubmit » ou « prepareForRender » (invoqué respectivement pour la soumission et le rendu avant la phase « prepare » qui lui est invoqué dans tous les cas)
  - « prepare » (Avant le rendu ou avant le processus de soumission)
  - « validateForm »
  - « success » ou « failure » (fonction du résultat de la validation)
  - « submit »
- **Pour éviter les problèmes liés à l'évolution du produit utiliser les constantes de la classe « EventConstants » pour les phases de traitement du formulaire (et pour tous les évènements Tapestry de manière générale)**
- **Utiliser l'annotation @OnEvent pour associer un évènement à une des phases de traitement du formulaire**

Gestion des formulaires
Form – Chronologie des évènements

- **Voici les phases de traitement du formulaire lorsque l'utilisateur soumet le formulaire**
- ⚠ **L'évènement « Submit » est toujours exécuté, peu importe le résultat de la validation**
- ⚠ **Toujours implémenter une méthode pour le traitement de l'évènement « failure »**
- ⚠ **Ce mécanisme peut paraître masqué lorsque la validation cliente est activée**



- ⚠ **Utiliser l'instrumentation invisible pour pouvoir travailler plus facilement avec le studio**
- ⚠ **Toujours identifié clairement un formulaire dans une page via l'attribut « t:id » pour plusieurs raisons**
  - **Inclure plusieurs formulaires dans une page**
  - **Identifier clairement l'origine d'un évènement lors de l'écriture des méthodes d'action**
  - **Récupérer explicitement l'instance du composant dans la page**

```
<form t:type="form" t:id="loginForm"> … </form>

@Component(id="loginForm")
private Form form;
```


Atos Origin

Gestion des formulaires
Form – Règles d'écriture

- **Identification des éléments dans un formulaire**
  - **« t:id » : Permet d'obtenir une référence sur l'instance du composant dans la page et de générer les attributs HTML « id » et « name » si le « clientId » n'est pas précisé**
  - **« t:clientId » : Permet de générer les attributs HTML « id » et « name ». Cet identifiant ne peut pas être utilisé pour obtenir une référence sur l'instance du composant dans la page**
  - **Par défaut : si aucun identifiant n'est précisé, Tapestry utilise le type du composant pour générer les identifiants « client » et « serveur »**
- **Tapestry ajoute un suffixe automatiquement si le composant est présent plusieurs fois dans une page (Ex : boucle, grille…)**

- **L'écriture des méthodes d'action du formulaire doit se faire de manière explicite en précisant**
  - **L'identifiant du formulaire à l'origine de l'évènement**
  - **Le type de l'évènement**

⚠ **Le risque si l'on applique pas cette règle est de voir les méthodes s'exécuter indifféremment pour tous les formulaires de la page**

```
@OnEvent(value=EventConstants.PREPARE,component="log
  inForm")
public void processLogin(){
…
}
```

TextField et PasswordField

- **Le premier permet d'associer un champ de type « text » à une variable d'instance de la page**
- **Le deuxième permet d'associer un champ de type « password » à une variable d'instance de la page**
- **Ils possèdent la même liste de paramètres**
  - value (obligatoire) : Propriété, identifiant de la variable d'instance à laquelle sera associé le champ
  - clientId : String, identifiant client du composant. Si ce champ n'est pas précisé alors Tapestry s'appuie sur le paramètre « t:id »
  - label : Permet de modifier le label associé au champ
  - disabled : Propriété de type booléen, si la propriété vaut « true » alors le champ de saisie est désactivé

Le paramètre value fait référence à une variable de la classe. Cela suppose qu'il existe un getter pour cette variable.

Liens vers la documentation :
http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/PasswordField.html

http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/TextField.html

⚠ **Dans la mesure du possible, préférer l'instrumentation invisible**

Ex :

- **Dans le Template**

```
<input t:type="textfield" t:value="login" t:id="login"
type="text" name="Login" />
```

- **Dans la classe**

```
private String login;

public String getLogin(){
  return login;
}

public void setLogin(String pLogin){
  this.login = pLogin;
}
```

CheckBox

- **Permet d'associer un champ HTML de type « checkbox » à une variable d'instance de type booléen (obligatoire)**
- **Liste de paramètres**
  - value (obligatoire) : **Propriété de type booléen**, identifiant de la variable d'instance à laquelle sera associé le champ
  - clientId : String, identifiant du composant
  - label : Permet de modifier le label associé au champ
  - disabled : Propriété de type booléen, si la propriété vaut « true » alors le champ de saisie est désactivé

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Checkbox.html

DateField

- **Ce composant complexe permet de créer un champ de saisie de type « Text » accompagné du calendrier Javascript « WebFX Datepicker »**
- **La variable d'instance associée à ce composant doit être du type « java.util.Date »**
- **Listes paramètres**
  - value (obligatoire) : **Propriété de type « java.util.Date »**, identifiant de la variable d'instance à laquelle sera associé le champ
  - clientId : String, identifiant du composant
  - label : Permet de modifier le label associé au champ
  - disabled : Propriété de type booléen, si la propriété vaut « true » alors le champ de saisie est désactivé
  - format : Format de la date (utiliser les notations J2SE)

http://webfx.eae.net/dhtml/datepicker/usage.html

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/DateField.html

Label

- **Ce composant permet de générer un label pour une propriété du formulaire**
- **Il possède un attribut obligatoire**
  - **for : identifiant du composant pour lequel l'attribut est créé**
  - ⚠ **Ce paramètre permet aussi de générer l'attribut XHTML « for » qui permet de donner le focus au champ de saisie associé**
- **Par défaut, le label sera équivalent au nom du composant remise en forme**
  - **Ex : « bookingDate » devient « Booking date »**
- **Pour paramétrer le label d'un champ via les fichiers de ressources, ajouter une clé du type « <id>-label »**
  - **Ex : bookingDate-label=Date de réservation**

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Label.html

Errors

- **Ce composant permet d'afficher la liste des erreurs courantes sur le formulaire**
- **Ce composant n'a pas de paramètre obligatoire, mais deux paramètres optionnels**
  - **banner : permet de modifier le message d'annonce des erreurs. Il peut s'agir d'une propriété (prop:…) ou d'un message (message:…)**
  - **class : classe CSS utilisée pour le composant « div » englobant les erreurs (par défaut : t-error)**
- **Pour modifier l'aspect actuel du composant d'erreur, redéfinir la classe « t-error » dans votre fichier CSS**
  - **div.t-error, div.t-error div, div.t-error ul, div.t-error li …**

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Errors.html

Submit

- **Ce composant permet de créer un bouton submit pour le formulaire**
- **Ce composant n'a pas de paramètre obligatoire**
  - disabled : Propriété de type booléen, si la propriété vaut « true » alors le champ de saisie est désactivé
- **Par défaut, la propriété label est obtenu dans le fichier de ressources de la page grâce à la clé « <id>-label » où « id » est l'identifiant du champ submit**
- **Ce composant émet un évènement « selected » qui permet de l'identifier quel bouton est à l'origine de la soumission**

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Submit.html

Gestion des formulaires
Form – Validation de premier niveau

- La validation des champs du formulaire peut se faire manuellement via les méthodes d'actions, cependant il existe un ensemble de validations de premier niveau qui peuvent être réalisées automatiquement par Tapestry
- Pour ajouter une contrainte sur un champ, il existe deux possibilités
  - Attribut « validate » (Textfield, PasswordField, DateField…) : Cet attribut permet de déclarer une liste de contraintes
  - Utiliser l'annotation @Validate sur les accesseurs des variables d'instances associées au Formulaire
- Dans les deux cas, on précisera une liste de validateur séparés par une virgule

Gestion des formulaires
Form – Validation de premier niveau

- Il existe plusieurs validateurs fournis avec Tapestry
  - Required : Le champ doit être obligatoire
  - Max : Valeur maximum pour les champs de type « Number »
  - Min : Valeur minimum pour les champs de type « Number »
  - MaxLength : Longueur maximum pour les champs de type « String »
  - MinLength : Longueur minimum pour les champs de type « String »
  - Regexp : Permet d'affecter une expression pour la validation des champs de type « String »

- **Utilisation via les annotations**

```
 @Validate("required,minLength=6")
public void setPassword(String pPassword){
  this.password = pPassword;
}
```

- **Utilisation du paramètre validate**

```
 <input t:type="passwordfield" t:id="password"
t:value="password" t:validate="required,minLength=6" />
```

⚠ **Tapestry ne permet pas l'utilisation conjointe des deux méthodes et celles-ci sont exclusives**

L'utilisation de ce type d'écriture peut être problématique lorsque la valeur du validateur est une chaîne complexe (Expression régulière) dans ce cas, utiliser un fichier de propriété pour donner une valeur à un validateur (cf. suite)

- **Spécifier une valeur pour les contraintes du type « validator=… » (ex : regexp=.*@.*)**
  - **Ajouter une clé du « <id>-<validator> » dans le fichier de propriétés**
  - **Ex : Dans le template**

```
<input t:type="textfield" t:id="email" t:value="email"
   t:validate="regexp" />
```

Dans le fichier de propriétés :

email-regexp=.*@.*

- **Modifier le message d'erreur associé à un champ**
  - **Ajouter une clé du type « <id>-<validator>-message » dans le fichier de propriétés associé à la page**
    - id : identifiant du champ dans le formulaire
    - validator : identifiant du validateur
  - **Ex : email-regexp-message=L'email doit contenir le caractère '@'**

La validation Client est désactivable.

Atos Origin

Gestion des formulaires
Form – Validation de second niveau

- **Le formulaire est associé à un objet du type « ValidationTracker » qui permet de manipuler la liste des erreurs dans la classe de la page**
- **Utiliser une instance du formulaire pendant la phase « validate » pour ajouter des contrôles de second niveau**

```
@Component(id="loginForm")
private Form loginForm;


@OnEvent(value=EventConstants.VALIDATE_FORM,component="
loginForm")
public Object sndLevel(){
  loginForm.recordError("Erreur de second niveau");
  return Index.class;
}
```


La validation de Second Niveau correspond à la validation fonctionnelle.

- **La classe « Form » possède d'autres méthodes**
  - **getDefaultTracker() : pour obtenir l'instance de l'objet « ValidationTracker » associé**
  - **clearErrors() : pour supprimer la liste de erreurs**
  - **recordError(Field, String) : pour assigner une erreur à un champ particulier du formulaire. La plupart des composants de type Formulaire implémentent l'interface « Field »**
  - **isValid() : renvoie « true » si le « ValidationTracker » ne contient pas d'erreurs**
  - **getHasErrors() : renvoie « true » si le « ValidationTracker » contient des erreurs**

Les méthodes isValid et getHasErrors peuvent être utile pour la phase de Submit. Si on n'a pas implémenté les phases de SUCCESS et de FAILURE, ces méthodes permettent d'avoir l'information.

Il existe d'autres composants Tapestry susceptible de vous intéresser dans la gestion de vos formulaires :

- Hidden
- RadioGroup
- Select
- TextArea

TP 7 : Formulaire de login et d'enregistrement

Concepts abordés

- Dynamisation de Template Tapestry
- Les composants de type « formulaire » (form, errors, textfield, passwordfield, checkbox)
- Les évènements du processus de traitement d'un formulaire
- Les mécanismes de validation de base (@Validate)
- Interaction Template/Classe Java (@Component)
- Utilisation de scope session Application (@ApplicationState)
- Utilisation des scopes de stockage (@Retain)

Dans ce TP, nous allons aborder l'utilisation du composant « Form » via un scénario d'identification de l'utilisateur. Ce TP et le suivant ont pour but de vous faire manipuler les composants et annotations associées au principe de formulaire.

Enoncé

1. Compléter le Template de la page « Login »

    a. Déclarer un formulaire identifié par « verifyForm »

```
<form t:type="form" t:id="verifyForm" id="verifyForm" method="post">

</form>
```

    b. Dynamiser les champs du formulaire et les associer aux variables d'instance de la classe « Login.java »

    c. Mapper les noms des champs de formulaire sur les identifiants des attributs de la classe « Login.java »

```
<fieldset id="fsGeneral">
    <p>
        <label t:type="label" t:for="login" for="login">Login</label>
        <input t:type="textfield" t:value="login" t:id="login" type="text" id="login" value="" />
    </p>
    <p>
        <label t:type="label" t:for="password" for="password">Password</label>
        <input t:type="passwordfield" t:value="password" t:id="password" type="password" id="password" value="" />
    </p>
</fieldset>
```

Pour associer les champs du formulaire aux variables d'instances de la classe java, il suffit de déclarer le nom de cette variable comme étant la valeur de la propriété t:value. Tapestry récupérera la valeur grâce au getter de la variable.

Pour associer le label aux champs du formulaire, il suffit de paramétrer les attributs t:for des labels avec le nom du champ.

d. Utiliser le composant de type « Errors » pour afficher la liste des erreurs potentielles

```
<p><span t:type="errors" /></p>
```

2. Ajouter les annotations sur les variables pour paramétrer les scopes de stockage des différentes informations

```
/**
 * Used to store a UserManager instance on the page instance
 */
@Retain
private UserManager manager;

/**
 * This variable is used to set a reference on the authenticated after
 * verify process
 */
@SessionState
private User loggedUser;

/**
 * Used to have a reference on the form component
 */
@Component(id = "verifyForm")
private Form verifyForm;
```

@Component permet d'associer à un composant du template à sa déclaration dans la classe Java correspondante.

La variable loggedUser doit être visible sur toutes les pages. Nous allons donc préférer l'annotation @SessionState.

Grâce à l'annotation @Retain, le UserManager sera instancié lors de la création de l'instance de la page et ne sera pas réinitialisé lorsque Tapestry va remettre la page dans le pool.

3. Annoter et implémenter la méthode « activateManager » pour créer une instance de « UserManager » associée à la page grâce à la méthode statique « getInstance() »

```
@OnEvent(EventConstants.ACTIVATE)
public void activateManager() {
    // Instantiate UserManager
    if (manager == null) {
        manager = UserManager.getInstance();
    }
}
```

Cette méthode sera appelée automatiquement lorsque la page sera activée entièrement.

4. Ajouter les contraintes sur les « setter » associés aux champs du formulaire pour effectuer la validation du formulaire avant le « submit »
    a. Login : obligatoire
    b. Password : obligatoire

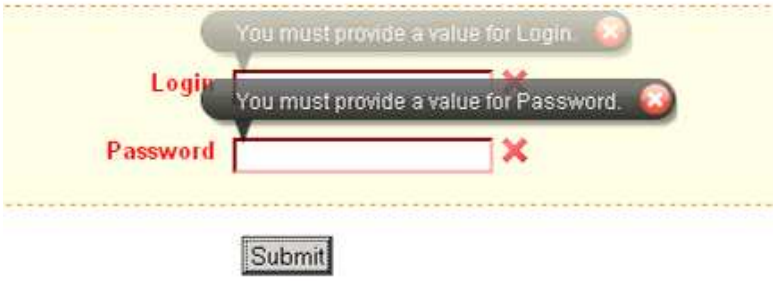
```
public String getLogin() {
    return login;
}

@Validate("required")
public void setLogin(String login) {
    this.login = login;
}

public String getPassword() {
    return password;
}

@Validate("required")
public void setPassword(String password) {
    this.password = password;
}
```

5. Annoter et implémenter la méthode de soumission pour
    a. Vérifier que l'utilisateur existe et qu'il a saisi un mot de passe correct
    b. En cas de succès
        i. Renvoyer l'utilisateur vers la page Main
        ii. Stocker l'objet « User » en session de manière à ce que cette instance soit visible des autres pages de l'application
    c. En cas d'échec ajouter un message d'erreur général et renvoyer l'utilisateur vers la page de Login

```
@OnEvent(value = EventConstants.SUCCESS, component = "verifyForm")
public Object verifyUser() {

    String errorMsg = "Wrong password or user doesn't exist...";

    // Verify if user exists
    User ttlUser = manager.getUserByLogin(login);
    if (ttlUser != null) {
        // Verify User Password
        if (password.compareTo(ttlUser.getPassword()) == 0) {
            loggedUser = ttlUser;
            return Main.class;
        } else {
            verifyForm.recordError(errorMsg);
        }
    } else {
        verifyForm.recordError(errorMsg);
    }

    return this;

}
```

Cette méthode sera exécutée quand le formulaire verifyForm sera envoyé correctement.

Si nous laissons les champs vides, des messages d'erreur apparaissent indiquant la structure des données attendues. Par contre, si nous renseignons les champs avec des données erronées, un message d'erreur s'affiche après avoir validé le formulaire.



Utilisation du composant Form (2/2)

Enoncé

L'objectif de ce TP est d'implémenter le formulaire d'enregistrement de l'utilisateur en vous inspirant de tout le travail réalisé dans les précédents TPs. Voici quelques indications pour la mise en œuvre de ce TP :

1. Ajouter les éléments nécessaires pour créer et conserver une instance de l'objet « UserManager »

```
@Retain
private UserManager manager;
```

Comme dans le TP précédent, nous allons utiliser l'annotation @Retain pour qu'une variable soit créer lors de la création d'une instance de la page, et qu'elle ne soit pas réinitialisée lorsque Tapestry la remet dans le pool.

2. Ajouter les variables d'instances qui permettront de collecter les informations du formulaire

```
private String login;

private String password;

private String email;

private int age;

private boolean admin;
```

Notre formulaire sera composé de 5 champs : login, password, adresse email, age et un flag qui indique si l'utilisateur sera un administrateur.

3. Ajouter une variable pour récupérer le composant identifié par « registerForm »

```
/**
 * Used to have a reference on the form component
 */
@Component(id = "registerForm")
private Form registerForm;
```

4. Créer les « getter » et « setter » des champs et ajouter les contraintes sur les champs du formulaire

a. Login : obligatoire, au minimum 5 caractères

```
@Validate("required,minLength=5")
public void setLogin(String login) {
    this.login = login;
}
```

b. Password : obligatoire, au minimum 6 caractères

```
@Validate("required,minLength=6")
public void setPassword(String password) {
    this.password = password;
}
```

c. Email : obligatoire, expression régulière « .*@.* »

```
@Validate("required,regexp=.*@.*")
public void setEmail(String email) {
    this.email = email;
}
```

5. Ajouter les éléments nécessaires pour que la méthode « verifyIfUserAlreadyExists » s'exécute une fois la validation de premier niveau terminée et implémenter le corps de la méthode pour

   a. Ajouter un message d'erreur global si l'utilisateur existe

   b. Laisser la méthode « submit » s'exécuter si l'utilisateur n'existe pas

```
/**
 * Used to control if the user already exists just before form submission.
 *
 * @return Register Page if user already exists
 */
@OnEvent(value=EventConstants.VALIDATE_FORM, component="registerForm")
public void verifyIfUserAlreadyExists() {
    if (manager != null) {
        User ttcUser = manager.getUserByLogin(login);
        if (ttcUser != null) {
            registerForm.recordError("User already exists.");
        }
    } else {
        registerForm.recordError("User Manager unavailable");
    }
}
```

Il faut absolument faire appel à la méthode recordError pour permettre à Tapestry de savoir s'il passe à l'événement SUCCESS ou FAILURE.

6. Ajouter les éléments nécessaires pour que la méthode « createUser » s'exécute en cas de succès des validations de premier et second niveau, implémenter le corps de cette méthode pour :

    a. Créer l'utilisateur via l'objet « UserManager » et rediriger l'utilisateur vers la page « Main » et stocker celui-ci en session

    b. Renvoyer l'utilisateur vers la page d'enregistrement si une erreur survient

```
/**
 * Used to process form submission.
 *
 * @return Register Page on creation failure, Main page on success
 */
@OnEvent(value=EventConstants.SUCCESS, component="registerForm")
public Object createUser() {

    User user = new User();
    user.setLogin(login);
    user.setPassword(password);
    user.setEmail(email);
    user.setAge(age);
    user.setAdminRigths(admin);

    // Try to add a new user
    if (manager != null) {
        try {
            // Add user
            manager.addUser(user);

            // Set application state logged user
            loggedUser = user;
        } catch (Exception ex) {
            registerForm.recordError("Error while adding user.");
            return Register.class;
        }
    } else {
        registerForm.recordError("User Manager unavailable");
        return this;
    }

    return Main.class;
}
```

7. Dynamiser le Template HTML pour

    a. Afficher les erreurs de saisie

```
<form t:id="registerForm" id="registerForm" method="post">

    <p><span t:type="errors" /></p>
```

    b. Effectuer le lien entre la classe et le formulaire via un formulaire identifié par « registerForm »

```
<form t:id="registerForm" id="registerForm" method="post">

    <p><span t:type="errors" /></p>

    <fieldset id="fsGeneral">
        <p>
            <label for="login">Login</label>
            <input t:type="textfield" t:value="login" type="text" id="login" value="" size="20" />
        </p>
        <p>
            <label for="password">Password</label>
            <input t:type="passwordfield" t:value="password" type="password" id="password" value="" size="20" />
        </p>
        <p>
            <label for="email">Email</label>
            <input t:type="textfield" t:value="email" type="text" id="email" value="" size="20" />
        </p>
        <p>
            <label for="age">Age</label>
            <input t:type="textfield" t:value="age" type="text" id="age" value="" size="20" />
        </p>
        <p>
            <label for="admin">Admin</label>
            <input t:type="checkbox" t:value="admin" type="checkbox" id="admin" />
        </p>
    </fieldset>

    <fieldset id="fsSubmit">
        <p><input t:type="submit" t:clientId="btSubmit" type="button" value="Submit" /></p>
    </fieldset>

    <p><span class="commts">&lt;&lt; <a t:type="pagelink" t:page="login">Back to index page</a></span></p>

</form>
```

Pour lier les champs du formulaire avec les variables d'instance de la classe java, il suffit de renseigner l'attribut t:value de chaque champ avec le nom de la variable.

8. Tester votre formulaire d'enregistrement

Atos Origin

Mécanismes de Base

- Généralités
- Traitement des requêtes
- Méthodologie
- Composants de base
- Gestion des formulaires
- **Autres composants**

Atos Origin

Autres Composants
Output

- org.apache.tapestry5.corelib.components.Output
- **Ce composant est utilisé pour formater la sortie**

Nom	Type	Drapeaux	Par défaut	Préfixe par défaut	Description
elementName	String		componentResources.elementName	prop	Le nom de l'élément, dérivé du Template du composant.
format	java.text.Format	Required		prop	Le formatage à appliquer à l'objet.
value	Object	Required		prop	La valeur à rendre (avant formatage). Si la valeur rendue est vide, aucune sortie ne sera produite.

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Output.html

Ex :

```
<p t:type="output"
t:value="currentPortfolio.bookingDate" t:format="dateFormat" />
```

- **Tapestry tente d'obtenir une instance de java.text.Format via la méthode « getDateFormat() »**
- **Le composant « output » utilise cette instance pour formater et affiche le contenu de la variable « currentPortfolio.bookingDate »**
- **Génèrera des informations du type**

```
<p>28/11/2007</p>
```

Atos Origin

Autres Composants
OutputRaw

- org.apache.tapestry5.corelib.components.OutputRaw
- **Utilisé pour produire du HTML brut au client**
  - **La sortie d'OutputRaw n'est pas filtrée**
  - **Les caractères spéciaux ainsi que les entités sont laissés tels quels**
  - **OutputRaw est surtout utilisé quand les balises sont fournies par une source extérieures, plutôt que construites avec Tapestry**

Nom	Type	Drapeaux	Par défaut	Préfixe par défaut	Description
value	String	Required		prop	La valeur à « rendre ».

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/OutputRaw.html

Ex :

```
<p t:type="outputRaw" t:value="article.asHtml"/>
```

- ⚠ **Ne pas utiliser d'expansion pour l'attribut « value ». Si l'on utilisait ${article.asHtml}le composant « outputRaw » ne réévalue pas l'expression**
- ⚠ **Par défaut, tous les attributs des composants sont « cachés »**
- ⚠ **Le fait de retirer l'expansion dans la valeur de l'attribut oblige Tapestry à créer un objet intermédiaire qui permet d'effectuer l'association entre le champ et la propriété, et non pas la valeur de la propriété**

Atos Origin

Autres Composants
Loop

- **Classe de boucle « de base »**
  - **Itérer sur une liste d'items**
  - **Le corps de la balise est exécuté**
  - **Permet d'associer une variable d'instance de la page à l'item courant**
  - **L'item courant peut alors être utilisé dans le corps de la boucle pour afficher des messages**

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Loop.html

Atos Origin

Autres Composants
Loop

Nom	Type	Drapeaux	Par défaut	Préfixe par défaut	Description
source	Iterable	Required			Définit la collection de valeur sur laquelle la boucle va itérer.
value	Object			prop	La valeur de l'item en cours, fixée avant le rendu.
index	int			prop	L'index dans la liste des items source.

Atos Origin

Autres Composants
Loop

Ex :

```
<p t:type="loop" t:source="1..10" t:value="guess">
    <a t:type="actionlink" t:id="link" t:context="guess">${guess}</a>
</p>
```

Tapestry parcourt les chiffres de 1 à 10 pour afficher des liens

- **source : Il aurait pu s'agir d'un lien vers une variable d'instance contenant une collection**
- **value : permet de stocker la valeur de l'itération courante**

TP 9 : Utilisation du composant Loop

- Gestion des variables de scope « ApplicationState »
- Le composant loop
- Interaction entre la page et le Template pendant la phase de rendu

L'objectif de ce TP est d'implémenter la page « Main » de l'application de gestion des portfolios. L'utilisation du composant « loop » nous permettra d'afficher la liste des portfolios dans un tableau HTML.

Enoncé

Avant d'utiliser le composant « loop », nous allons sécuriser la page « Main » en vérifiant s'il existe une variable « loggedUser » en session

1. Ajouter l'annotation pour permettre la récupération de la variable de session « loggedUser »

```
/**
 * Used to have a reference on the authenticated user
 */
@SessionState
private User loggedUser;
```

2. Créer la variable d'instance qui permettra de vérifier si la variable « loggedUser » existe

```
private boolean loggedUserExists;
```

Il est possible de déclarer une variable de type boolean, qui permettra de savoir si une variable de Session existe. Cette variable sera gérée automatiquement par Tapestry. Il suffit d'ajouter le suffixe Exists au nom de la variable que nous voulons tester.

3. Annoter et Implémenter la méthode « assertUserExists » pour renvoyer l'utilisateur vers la page « Login » s'il ne s'est pas « loggé » correctement

```
/**
 * Used to verify if the user is logged on
 *
 * @return the Index page if user doesn't exist in session, null otherwise
 */
@OnEvent(EventConstants.ACTIVATE)
public Object assertUserExists() {

    // Verify if user has logged in
    if (!loggedUserExists) {
        return Login.class;
    }

    return null;
}
```

Avant d'afficher le contenu des pages protégées, nous allons tester si l'utilisateur s'est identifié. Pour cela, il suffit d'annoter la méthode asserUserExists pour qu'elle soit exécutée lorsque la page a été entièrement activée.

Pour tester si l'utilisateur s'est identifié, il suffit de tester la variable loggedUserExists.

Une fois la variable « loggedUser » vérifiée, afficher le login de l'utilisateur après le message de bienvenue.

```
<div id="main">
<p>Bonjour Mr <b>${loggedUser.login}</b>.</p>
```

Utiliser le composant « loop » pour parcourir la liste des portfolios de l'utilisateur et les afficher dans le tableau HTML. L'objectif est ici de placer un effet Zebra pour alterner la classe de la ligne en cours, utiliser les classes CSS « **tbl1** » et « **tbl2** »

1. Modifier le Template pour instrumenter la ligne « tr » effectuant l'affichage des valeurs du portfolio avec le composant « loop »
2. Utiliser la variable d'instance « currentPortfolio » pour stocker la ligne courante
3. Utiliser la variable d'instance « index » pour stocker l'indice de la ligne courante

```
<table border="1" cellpadding="0" cellspacing="0" class="forms">

    <tr>
        <th align="center" class="tableh">Label</th>
        <th align="center" class="tableh">Amount</th>
        <th align="center" class="tableh">Booking date</th>
    </tr>

    <tr t:type="loop" t:index="index" t:value="currentPortfolio" t:source="loggedUser.portfolios">
        <td align="center">${currentPortfolio.label}</td>
        <td align="center">${currentPortfolio.amount}</td>
        <td align="center">${currentPortfolio.bookingDate}</td>
    </tr>
</table>
```

Nous affichons dans un tableau le contenu d'une collection (loggedUser.portfolios) grâce au composant Loop. Les valeurs courantes du portfolio et de l'index sont définies par les variable index et currentPortfolio.

Ainsi, pour remplir chaque ligne, il suffit de récupérer les attributs de currentPortfolio.

4. Implémenter le corps de la méthode « getRowClass » pour définir la classe CSS de la ligne en cours
5. Utiliser cette même variable méthode pour créer l'effet Zebra

```
/**
 * This method is used for the zebra effect on the grid
 *
 * @return "tbl1" or "tlb2" in function of "index modulo 2"
 */
public String getRowClass() {
    // Implement here the choice of CSS class used to display a row
    // Implement here the choice of CSS class used to display a row
    if (index % 2 == 0) {
        return "tbl1";
    } else {
        return "tbl2";
    }
}
```

Pour notre effet Zebra, nous allons tester la valeur de l'index qui parcourt notre Collection, pour pouvoir alterner l'identifiant de la classe CSS. La valeur retournée par cette fonction, sera récupérée par l'attribut class de la balise <tr>.

```
<tr t:type="loop" t:index="index" t:value="currentPortfolio" t:source="loggedUser.portfolios" class="${rowClass}">
    <td align="center">${currentPortfolio.label}</td>
    <td align="center">${currentPortfolio.amount}</td>
    <td align="center">${currentPortfolio.bookingDate}</td>
</tr>
```

Atos Origin

Mécanismes de base

Template / Eléments de type Block

- **Tapestry fournit un élément de type Block**
  - ⚠ **Un « block » n'est pas un composant**
  - **Permet de créer des « micro-Template » dans un Template**
  - **Un « block » ne s'affiche pas par défaut**
  - **Un block possède un « Id »**
- **Pour Afficher les « block »**
  - **Utiliser l'annotation @Inject pour obtenir une référence au block dans la classe de la page**
  - **Utiliser le composant « Delegate »**
  - ***Ou utiliser les méthodes de rendu***

L'élement **block** permet, par exemple, de choisir l'affichage d'un block ou d'un autre

Atos Origin

Mécanismes de base
Template / Eléments de type Block

- **Pour les champs de type « Block », la valeur de l'annotation « Inject » est l'identifiant de l'élément « <block> » à l'intérieur du Template du composant**
- **L'identifiant est déterminé par le nom du champ. Quand ce n'est pas possible, on peut fournir une annotation « Id »**

```
@Inject
private Block foo;

@Inject
@Id("bar")
private Block barBlock;
```


Atos Origin

Mécanismes de base
Template / Eléments de type Block

- **Le composant « Delegate » dispose d'un seul paramètre obligatoire**
  - **to : la propriété qui définira le block à afficher, le type de cette propriété peut être un « Block » ou une instance de composant**

Atos Origin

Mécanismes de base

Template / Eléments de type Block

```
<html
xmlns:t="http://tapestry.apache.org/schema/
tapestry_5_1_0.xsd">
  <head>
    <title>Portail Page</title>
  </head>
  <body>
    <p t:type="delegate" t:to="chooseBlock"
/>
    <t:block id="block1">Edit
Mode</t:block>
    <t:block id="block2">View
Mode</t:block>
  </body>
</html>
```

```
@Inject
private Block block1;
@Inject
private Block block2;

public Block getChooseBlock(){
  if(…){
    return block1;
  }else{
    return block2;
  }
}
```


Par défaut, nos 2 blocks sont invisibles. C'est la méthode getChooseBlock qui va afficher l'un ou l'autre des blocs, selon une condition.

Atos Origin

Composants Avancés

Atos Origin

Composants Avancés

- **BeanEditForm**
- Grid
- BeanModel
- Upload

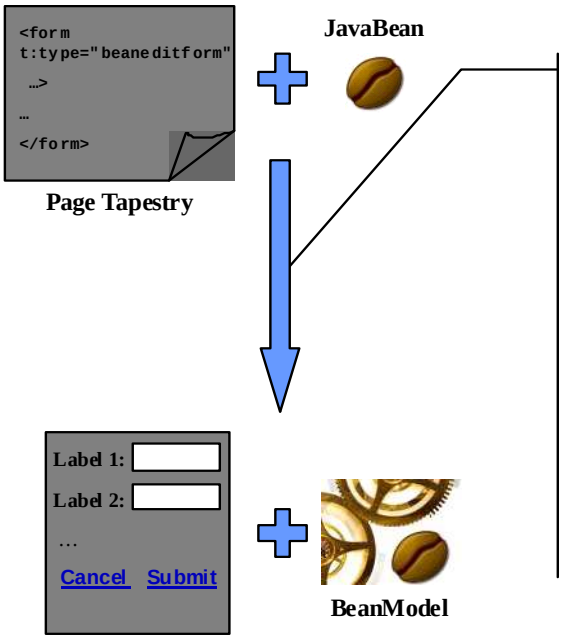
Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/BeanEditForm.html

- **Le BeanEditForm génère un formulaire de « création / mise à jour » à partir d'un JavaBean**
- **Fonctionnalités**
  - **Création automatique d'une instance du JavaBean associé**
  - **Association automatique « champ de formulaire » / attribut du JavaBean**
  - **Gestion des attributs non éditables**
  - **Validation par défaut en fonction du type des attributs**
  - **Ordonnancement implicite des attributs**
  - **Génération des labels**

Le BeanEditForm s'appuie sur deux composants pour l'analyse du JavaBean : le BeanEditor et le PropertyEditor.

Atos Origin

Composants Avancés
BeanEditForm



1. Analyse du Java Bean
2. Pour chaque attribut du JavaBean
   a. Identifier le type
   b. Analyse des validateurs
   c. Analyse de l'ordonnancement
   d. Analyse des propriétés non éditables
   e. Génération d'un modèle par défaut
3. Analyse du tag « BeanEditForm »
   a. Analyser les paramètres
   b. Identifier les blocs définis par l'utilisateur
   c. Modification du modèle si nécessaire
   d. Génération du formulaire HTML

Le « BeanModel » permet d'obtenir une représentation du JavaBean à créer et contient les outils pour créer les propriétés du Bean.
Le Template de l'utilisateur, conjointement au « BeanModel », généré permet de créer l'interface utilisateur pour la saisie du formulaire de création mise à jour du JavaBean

Liste des Types Supportés

Type	Rendu
String, Number	TextField
Enum	Select
Boolean	Checkbox
Date	TextField + Calendrier Javascript

- Liste extensible
- Résolution ascendante du type (Ex : un Integer est un Number…)
- Les types non gérés ne donnent lieu à aucun rendu
- ⚠ La personnalisation permet de remplacer ou d'utiliser d'autres composants pour un type donné

- **Un paramètre obligatoire**
  - **object :** préciser le JavaBean qu'il faut créer, doit être mappé sur un getter de la classe Java associée
- **Comportement**
  - **clientValidation :** booléen, indique si le contrôle du Formulaire doit être réalisé en JavaScript. Si « true » (par défaut), le composant génère le code nécessaire pour valider le formulaire avant le Post
  - **submitLabel :** Permet de modifier le nom du bouton « Submit » généré avec le formulaire HTML. Littéral (par défaut) ou clé de message (dans ce cas préfixer la valeur de l'attribut par « message: »)

- **Adapter le modèle et paramétrer le Bean**
  - **Un ensemble d'annotations sont fournies avec le composant BeanEditor**
    - NonVisual : permet de désactiver la saisie d'un attribut (Id technique par exemple)
    - ReorderProperties : permet de placer une contrainte sur l'ordre d'affichage d'une propriété
    - Validate : permet d'ajouter des validations sur les attributs du Bean
  - **Ces annotations doivent être placées sur les méthodes « get » ou « set » des attributs que l'on souhaite paramétrer**
  - **Attention ! La modification des annotations nécessite un redémarrage du serveur *si les classes modifiées sont en dehors des packages « pages » ou « components »***

```
public class MonBean {

  private long id;

  private String firstName;

  @Validate("required")
  public void setFirstName(
     String aFirstName){
    firstName = aFirstName;
  }

  public String getFirstName(){
    return firstName;
  }

  @NonVisual
  public void setId(long aId){
    id = aId;
  }

  public long getId(){
    return id;
  }

}
```

TP 10 : Utilisation simple du BeanEditForm

Concepts abordés

- Création d'une sous-application avec Tapestry
- Génération automatique de formulaire depuis un JavaBean
- Interaction Composant et JavaBean

L'objectif de ce TP est de manipuler le composant complexe « BeanEditForm ». Ce composant est l'un des principaux de la distribution, il permet de développer très rapidement à partir de Bean Java des interfaces de création/modification HTML

⚠ Les Templates fournis avec ce TP se situent dans un sous-package que nous considérons comme étant une sous-application permettant de traiter l'ajout de portfolios. Pour appeler la page directement depuis un navigateur, il suffira d'indiquer « <contexte>/porfolio/add ». Tapestry se charge de localiser la page « addPortfolio » en fonction du nom du sous-package

Enoncé

1. Ajouter le composant Menu dans votre application, le composant de type « menu » requiert un seul paramètre obligatoire pour l'instant
    a. **listOfActions** : la liste des actions de l'utilisateur. Une collection d'objet du type « Action »

```
<div t:type="menu" t:listOfActions="loggedUser.actions"></div>
```

On récupère les actions grâce aux actions de l'utilisateur loggé, grâce au getter getActions.

2. Paramétrer l'élément « form » du Template « AddPortfolio » pour utiliser le composant « BeanEditForm » de manière simple

```
<form t:type="beaneditform" t:object="portfolio">


</form>
```

3. Annoter et implémenter la méthode « activateManager »

```
/**
 * Used to create a User Manager instance on page activation.
 */
@OnEvent(EventConstants.ACTIVATE)
public void activateManager() {
    if(manager == null) {
        manager = UserManager.getInstance();
    }
}
```

L'implémentation de la méthode activeManager est identique que pour les autres pages de l'application.

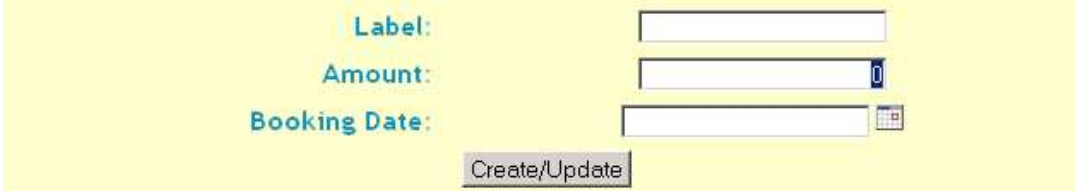
4. Annoter les variables d'instance de la classe

```
/** Used to add the created portfolio to the user */
@Retain
private UserManager manager;

/** Used to get the logged user from session */
@SessionState
private User loggedUser;
```

5. Annoter et implémenter la méthode « addPortfolioToUser » pour traiter l'évènement « success » du processus de soumission
    a. Ajouter le portfolio à l'utilisateur si celui-ci est bien loggé
    b. Rediriger l'utilisateur vers la page « Login » s'il ne s'est pas identifié avant d'accéder à cette page

```
@OnEvent(EventConstants.SUCCESS)
public Object addPortfolioToUser() {
    // Add portfolio to user or redirect the user to index page if not
    // logged in
    if(loggedUserExists) {
        manager.addPortfolioToUser(loggedUser, portfolio);
        return Main.class;
    }else{
        return Login.class;
    }
}
```

Comme précédemment, nous testons tout d'abord que l'utilisateur est identifié. Ensuite, nous ajoutons le nouveau portfolio à l'aide d'une méthode déclarée dans la classe UserManager.



Si vous visualiser le code HTML généré, Tapestry a ajouté de la validation cliente. En effet, il ajoute toutes les validations implicites. Ici par exemple, la variable Amount étant déclaré comme un Double, il ne peut pas avoir une valeur nulle.

- **Supprimer une propriété ou modifier l'ordre d'affichage**
  - **Par défaut c'est l'ordre de déclaration des attributs dans le JavaBean qui définit l'ordre d'affichage**
  - **Il est préférable d'utiliser les attributs d'affichage du composant**
    - **reorder : modifier l'ordre d'affichage des attributs**
      - Liste des propriétés du Bean séparées par des virgules
      - Non sensibles à la casse
      - Les attributs non présents sont placés en fin de liste
    - **Include/exclude : ajouter ou supprimer des attributs du modèle.**
      - Liste des propriétés du Bean séparées par des virgules
      - Non sensibles à la casse
  - **Attention : L'utilisation de ces attributs modifie le modèle**

```
<form t:type="beaneditform" reorder="firstname,lastname" exclude="id"
object="address">
…
</form>
```


Par défaut, Tapestry génère automatique un formulaire, avec touts les attributs déclarés dans la classe Portfolio.
Nous ne sommes pas obligés de déclarer toutes les variables dans la balise reorder. Tapestry placera les variables non déclarées dans la balise reorder à la fin de la liste.

- **Adapter la vue et modifier les composants d'affichage choisis**
  - **Si le type est résolu, le composant fournit un bloc d'édition par défaut**
  - **Pour redéfinir l'outil de saisie d'un attribut du JavaBean, utiliser les blocs « parameter »**
  - **Ex : Utiliser un composant d'affichage de type input type « password » pour l'attribut « password » du JavaBean**

```
<form t:type="beaneditform" t:object="loginCredentials">
  <t:parameter name="password">
    <label t:type="label" t:for="password"/>
    <input t:type="passwordfield" t:id="password"
t:value="loginCredentials.password" t:validate="required,minLength=6"/>
  </t:parameter>
</form>
```


- **Modifier les labels par défaut**
  - **Par défaut, ce sera le nom de la propriété remis en forme (Première lettre en majuscule, ajout d'un espace devant chaque Majuscule)**
  - **Pour redéfinir un label, il faut ajouter une clé dans le fichier de propriétés associé à la page : « property-label », Ex : pour un attribut email on ajoutera la clé « email-label »**
- **Liste des styles CSS à surcharger**
  - ***DIV.t-beaneditor* : Définit les attributs du « Div » contenant les lignes du BeanEditForm**
  - ***DIV.t-beaneditor-row* : Définit le style de chaque ligne (label + champ de saisie)**
  - ***DIV.t-beaneditor* LABEL:after : Définit le caractère séparateur entre le label et le champ de saisie**
  - ***DIV.t-beaneditor* LABEL : Définit le style des labels associés à chaque attributs**
  - ***t-beaneditor-submit* : Style du bouton de soumission du formulaire**

TP 11 : Paramètrage du BeanEditForm

Concepts abordés

- Utilisation des blocs « parameter »
- Utilisation des composants de type « formulaire » : label, textfield…
- Paramétrage de la validation via le Template HTML
- Modification des « css » par défaut
- Paramétrage du bouton « submit »

Nous avons pu voir grâce au TP précédent la facilité d'utilisation du composant « BeanEditForm », dans ce TP nous nous attacherons à identifier comment personnaliser le rendu. Pour éviter de modifier les Beans du modèle, nous allons ajouter les contraintes directement dans le Template HTML.

Enoncé

1. Ajouter un bloc « parameter » pour chaque attribut de la classe Portfolio
2. Paramétrer les différents blocs en utilisant les composants adéquats : « label », « textfield », « datefield »
3. Ajouter une clé dans le fichier « addPortfolio.properties » pour paramétrer la valeur du bouton de soumission
4. Paramétrer l'attribut « validate » de chacun des champs de saisie pour ajouter les contraintes suivantes :
    a. **label** : obligatoire
    b. **amount** : obligatoire avec un minimum de 100
    c. **bookingDate** : obligatoire
5. Modifier les différents styles CSS pour obtenir un rendu similaire au formulaire d'enregistrement

```
<form id="addPortolfioForm" t:type="beaneditform" t:object="portfolio" t:submitLabel="message:submit-label">
    <p><span t:type="errors" /></p>

    <t:parameter name="label">
        <p>
            <label t:type="label" t:for="label"/>
            <input t:type="textfield" t:id="label" t:value="portfolio.label" t:validate="required"/>
        </p>
    </t:parameter>

    <t:parameter name="amount">
        <p>
            <label t:type="label" t:for="amount"/>
            <input t:type="textfield" t:id="amount" t:value="portfolio.amount" t:validate="required,min=100"/>
        </p>
    </t:parameter>

    <t:parameter name="bookingDate">
        <p>
            <label t:type="label" t:for="bookingDate"/>
            <input t:type="datefield" t:format="MM/dd/yy" t:id="bookingDate" t:value="portfolio.bookingDate" t:validate="required"/>
        </p>
    </t:parameter>

</form>
```

Il est possible de modifier le comportement des champs BeanEditForm, en ajouter des blocs <t:parameter> qui fon référence à la variable de la classe. Ainsi, Tapestry ne générera plus automatique le champ de cette variable, mais utilisera le template que vous aurez déclaré.

Il est également possible de paramétrer lz label du bouton submit grâce à l'attribut t:submitLabel du composant BeanEditForm. Dans notre exemple, le préfixe "message:" indique que nous voulons récupérer la chaine de caractère associée à la clé (dans notre cas submit-label), dans le fichier properties associé à la page.

- **Contribuer à la prise en charge de nouveaux types :**
  - **Permet de généraliser l'utilisation d'un bloc pour un type donné dans une application**
  - **Ex : La prise en charge du type « java.util.Date » à été ajoutée via ce mécanisme d'extension à partir de la version 5.0.6 de Tapestry, l'interface générée fournit un champ de saisie manuelle accompagnée d'un calendrier graphique pour une saisie assistée**
  - **L'ajout d'un nouveau type se déroule en trois étapes**
    1. **Associer un identifiant au type de donnée en question**
    2. **Définir l'affichage associé au type : création d'un bloc d'édition et d'un bloc d'affichage si besoin**
    3. **Créer l'association entre le(s) bloc(s) et l'identifiant choisi initialement**

Atos Origin

Composants Avancés

- BeanEditForm
- **Grid**
- BeanModel
- Upload

Liens vers la documentation : http://tapestry.apache.org/tapestry5/tapestry-core/ref/org/apache/tapestry5/corelib/components/Grid.html

- **Le composant « Grid » permet de présenter une liste d'éléments sous la forme d'un tableau disposant des fonctionnalités suivantes**
  - **Tri des colonnes**
  - **Navigation dans les pages de résultat**
- **Le composant génère un modèle par défaut à partir du premier élément de la liste**
- **Tous les éléments de la liste doivent être du même type**

Nous avons précédemment créé un tableau grâce au composant Loop. Cependant, il est très probable que ce dernier n'ait pas les fonctionnalités suffisantes à vos besoins. L'ajout d'un mécanisme de pagination ou de tri des colonnes doivent être réalisés par l'ajout de code.

Tapestry propose un composant GRID, qui permet de façon très simple d'implémenter un tableau dynamique, avec notamment les fonctionnalités citées ci-dessus.

- **Un paramètre obligatoire**
  - **source :** indique la liste des éléments à afficher, doit être mappé sur un getter de la classe Java associée
- **Paramètres de mise en forme**
  - **pagerPosition :** Modifier la position des liens de navigation, vaut « top », « bottom », « both » ou « none »
  - **rowsPerPage :** Nombre de lignes par page de résultats
  - **rowClass :** style « css » à utiliser pour la ligne courante
  - **empty :** bloc à utiliser lorsqu'il n'y a aucun résultat à présenter

Voici un exemple de déclaration d'un composant GRID :

```
<t:grid t:source="maSource" t:pagerPosition="top" t:rowsPerPage="5" />
```

TP 12 et 13 : Utilisation simple du composant Grid

Concepts abordés

- Utilisation des composant de type « block » (@Inject)
- Composant Tapestry « delegate »
- Définition de l'affichage dynamique

L'objectif de ce TP est de préparer le déroulement du prochain TP et permettre à l'utilisateur de sélectionner l'affichage de la liste des portfolios via le composant « loop » où à l'aide du composant « Grid »

Enoncé

1. Ajouter un lien de type « actionlink » identifié par « advanced » sur le libellé « Advanced grid »

```
<p><a t:type="actionlink" t:id="advanced">Advanced Grid</a></p>
```

2. Ajouter l'annotation adéquate sur la variable d'instance « mode » pour stocker en session le mode d'affichage choisi par l'utilisateur

```
/**
 * Used to store the grid display in user session
 */
@Persist
private String mode;
```

Etat une variable propre à la page Main, l'annotation à utiliser pour sauvegarder la variable en session est @Persist.

3. Annoter et implémenter la méthode « selectAdvancedMode » pour affecter le mode d'affichage à la constante « GRID »

```
/**
 * This method is used to setup the advanced view mode
 *
 */
@OnEvent(value = EventConstants.ACTION, component = "advanced")
public void selectAdvancedMode() {
    this.mode = GRID;
}
```

Ainsi, la méthode selectAdvancedMode sera exécutée lorsque l'utilisateur actionnera la composant advanced.

4. Dans le Template, englober le lien et la table actuelle par une balise Tapestry « block » identifié par « simpleList »

```
<!-- Define blocks for view and edit mode -->
<t:block id="simpleList">

    <p><a t:type="actionlink" t:id="advanced">Advanced Grid</a></p>

    <table border="1" cellpadding="0" cellspacing="0" class="t-data-grid">

        ...

    </table>
</t:block>
```

5. Ajouter dans le Template un deuxième « block » contenant le code suivant

```
<t:block id="advancedList">
    <p><a t:type="actionlink" t:id="simple">Simple Grid</a></p>


</t:block>
```

L'utilisation des composants t:block permettra d'afficher l'un des deux en fonction du mode d'affichage de l'utilisateur.

6. Annoter et implémenter la méthode « selectSimpleMode » pour affecter le mode d'affichage à la constante « LOOP »

```
/**
 * This method is used to setup the simple view mode
 *
 */
@OnEvent(value = EventConstants.ACTION, component = "simple")
public void selectSimpleMode() {
    this.mode = LOOP;
}
```

7. Ajouter les annotations adéquates sur les variables d'instance « simpleList » et « advancedList »

```
/**
 * Used to store a reference on the simple grid view
 */
@Inject
private Block simpleList;

/**
 * Used to store a reference on the advanced grid component view
 */
@Inject
private Block advancedList;

...
```

Pour les Block, nous ne pouvons pas utiliser l'annotation @Component. Nous devons absolument utiliser @Inject.

8. Implémenter le corps de la méthode « getPorfolioGrid » pour renvoyer l'une des deux variables citées ci-dessus en fonction du mode d'affichage choisi par l'utilisateur. Par défaut, on choisira d'afficher le format simple

```
/**
 * This method is used by the delegate component to get the block to display
 *
 */
public Block getPorfoliosGrid() {

    // Check for the selected mode and return the appropriate block
    if (GRID.equals(this.mode)) {
        return this.advancedList;
    } else {
        return simpleList;
    }

}
```

9. Utiliser le composant Tapestry « delegate » pour afficher le « block » correspondant à l'affichage souhaité par l'utilisateur

```
<div t:type="delegate" t:to="porfoliosGrid"></div>
```

Concepts abordés

- Utilisation simple du composant « Grid »

L'objectif est ici d'implémenter le corps du « block » advancedList. Nous proposons ici une première utilisation du composant pour afficher la liste des portfolios de l'utilisateur.

Enoncé

1. Dans le block « advancedList » de la page « Main » créé au cours du TP précédent, ajouter une balise « table » dont le type Tapestry sera « grid », voici le paramétrage souhaité pour cette grille
    a. Utiliser comme source la liste des portfolios de l'utilisateur
    b. Afficher le « pager » au dessus de la liste
    c. Afficher trois éléments par page

```
<table t:type="grid" t:source="loggedUser.portfolios" t:row="currentPortfolio" t:pagerPosition="top" t:rowsPerPage="3">
</table>
```

- **Supprimer ou modifier l'ordre d'affichage des colonnes**
  - **Par défaut c'est l'ordre de déclaration des attributs dans le JavaBean qui définit l'ordre d'affichage des colonnes**
  - **Il est préférable d'utiliser les paramètres d'affichage du composant**
    - **reorder : Modifier l'ordre d'affichage des colonnes**
      - **Liste des propriétés du Bean séparées par des virgules**
      - **Non sensibles à la casse**
      - **Les attributs non présents sont placés en fin de liste**
    - **Include/exclude : Supprimer une colonne de la grille**
      - **Liste des propriétés du Bean séparées par des virgules**
      - **Non sensibles à la casse**
  - **Attention : L'utilisation de ces attributs modifie le modèle**

Atos Origin

Composants Avancés
Grid - Personnalisation

- **Modifier l'apparence d'une ligne (Ex : effet zébra)**
  - **Ajouter l'attribut rowClass : donner l'identifiant de la propriété de classe Java permettant d'obtenir la classe**

```
<table t:type="Grid" t:id="gridmain" t:pagerPosition="none"
t:rowsPerPage="10" t:source="elements"t:rowClass="laClasse" >
…
</table>
```

  - **Ajouter le getter pour la propriété de sélection de la classe**

```
public String getLaClasse(){
  // Choix de la classe CSS
  return "laClasse";
}
```

  - **L'attribut « rowClass » est recalculé pour chaque ligne, ce qui permet de dynamiser le choix du style d'une ligne**

- **Modifier l'apparence d'une ligne (Suite)**
  - **Si le style dépend du contenu de la ligne courante, utiliser l'attribut « row » pour avoir accès à la ligne courante dans la méthode de sélection**

```
<table t:type="Grid" t:id="gridmain" t:pagerPosition="none" t:rowsPerPage="10" t:source="elements"
t:row="currentUser" t:rowClass="laClasse" >
…
</table>
```

Et dans la classe Java :

```
private User currentUser;

public void setCurrentUser(User aUser) {
  currentUser = aUser;
}

public User getCurrentUser() {
  return currentUser;
}

public String getLaClasse() {
  if(currentUser.getFirstName()…
}
```


Atos Origin

Composants Avancés
Grid - Personnalisation

- **Modifier le titre des colonnes**
  - **Le titre d'une colonne est créé de la même manière que les labels du « BeanEditForm »**
  - **Ajouter une clé « property-label » dans le fichier de propriétés associé à la page (où « property » est une propriété du JavaBean)**
  - **Ex : pour l'attribut « firstName » on ajoutera la clé « firstname-label »**

- **Modifier l'apparence du tableau**
  - **table.t-data-grid thead tr th :** Permet de personnaliser l'ensemble des cellules correspondant aux headers de colonnes
  - **<attibute_name>-header :** classe CSS définie pour spécifier le style de chaque header indépendamment
  - **t-sort-icon :** Style de l'image pour trier (ascendant/descendant…)
- **Modifier l'apparence du « Pager »**
  - **table.t-data-grid-pager :** classe affectée au « div » contenant les éléments du « Pager »
  - **table.t-data-grid-pager span.current :** classe affectée au « span » contenant l'indice de la page courante

Atos Origin

Composants Avancés
Grid - Personnalisation

- **Personnaliser le rendu des cellules**
  - **Obligatoire : définir l'attribut « row » pour stocker la ligne en cours de traitement dans la classe Java associée**
  - **Utiliser un bloc « parameter » pour redéfinir l'apparence de la cellule de votre choix**
  - **Pour associer un bloc à une propriété du JavaBean, utiliser le nom de la propriété suffixée par « Cell »**
  - **Ex : pour la propriété « firstName », afficher le nom de l'utilisateur courant**

```
<table t:type="Grid" t:row="currentUser" … >
  <t:parameter name="firstNameCell">
    ${currentUser.firstName}
  </t:parameter>
</table>
```

- **Possibilité de redéfinir un Header de colonne (depuis la 5.0.11)**

```
<table t:type="Grid" t:row="currentUser" … >
  <t:parameter name="firstNameHeader">
 …
  </t:parameter>
</table>
```

- **Possibilité de redéfinir le style de la grid**

```
<table t:type="Grid" t:class="myClass" … >…</table>
```

- **Possibilité de redéfinir son propre « Pager »**

```
Etendre org.apache.tapestry.corelib.components.GridPager
Utiliser le « BeanModel » du composant Grid
```

TP 14 : Personnalisation de la Grille

Concepts abordés

- Paramétrage dynamique
- Bloc « parameter »
- Composant simple « output »
- Modification des styles « css » par défaut

Nous avons vu au cours du TP précédent comment inclure une grille dans un Template Tapestry, l'objectif de ce TP est de personnaliser l'apparence de cette grille et le contenu des cellules.

Enoncé

1. Utiliser la variable d'instance « currentPortfolio » pour stocker le portfolio courant
2. Utiliser la variable d'instance « rowClass » implémentée pendant le « TP Loop » pour modifier dynamiquement des lignes de la table

```
<table t:type="grid" t:source="loggedUser.portfolios"
                     t:row="currentPortfolio"
                     t:pagerPosition="top"
                     t:rowsPerPage="3"
                     t:rowClass="rowClass">



</table>
```

3. Ajouter un bloc « parameter » pour l'attribut « bookingDate »
4. Ajouter dans ce bloc un composant « output » qui permettra d'afficher la date en utilisant la variable d'instance « dateFormat »

```
<t:parameter name="bookingDateCell">
    <t:output t:value="currentPortfolio.bookingDate" t:format="dateFormat" />
</t:parameter>
```

Pour modifier un champ du composant Grid, il suffit d'ajouter un block <t:parameter> qui fait référence à la variable de la classe, grâce à l'attribut name. La valeur de cet attribut doit contenir le nom de la variable, suffixé par Cell.

5. Modifier les styles CSS en vous appuyant sur les styles utilisés dans les Templates HTML du « TP Loop » pour vous rapprocher d'un visuel similaire

Bonjour Mr **tapestry**.
photo
Voici la liste de vos portefeuilles :
Simple Grid

1 2 3 4

Label	Amount	Booking Date
Portfolio n°0	100.0	10/12/2010
Portfolio n°1	200.0	10/12/2010
Portfolio n°2	300.0	10/12/2010

Footer

Atos Origin

Composants Avancés

- BeanEditForm
- Grid
- **BeanModel**
- Upload

- **Le BeanModel est utilisé par les composants Grid et BeanEditForm pour**
  - **Identifier les propriétés d'un JavaBean**
  - **Obtenir leur ordre d'affichage, leur label**
  - **Obtenir les objets permettant de valider et décoder les données saisies**
- **Redéfinir le modèle si**
  - **Le type de certains attributs du JavaBean ne peut être résolu**
  - **Et que la généralisation de la prise en charge de ces types est inutile**

Vue

```
<table t:type="grid"  t:source="bean" t:model="model"  …>
</table>
```

Classe : Déclaration des outils pour la personnalisation des colonnes

```
@Retain
private BeanModel model;

@Inject
private ComponentResources resources;

@Inject
private BeanModelSource beanModelSource;
```

Classe : Chargement de la page et création du modèle

```
@PageLoaded
void pageLoaded() {
  model = beanModelSource.create(User.class, true, resources);

  // Ajout d'une colonne via le
  model.add("deleteUser" , null);

  // Modifier l'apparence de la colonne
  model.get("deleteUser").dataType("String");
  model.get("deleteUser").sortable(false);
}

public BeanModel getModel() {
  return model;
}
```


Bilan des composants BeanEditForm et Grid

- Idéals pour des développements rapides, des interfaces d'administration ou des applications « jetables » (maquette, première version…) …
- Offre des fonctionnalités avancées (« lazy loading » et filtres pour la Grid)

Atos Origin

Composants Avancés

- BeanEditForm
- Grid
- BeanModel
- **Upload**

Atos Origin

Composants Avancés
Upload - Présentation

- **Tapestry propose un composant d'upload sous la forme d'un module optionnel : « tapestry-upload »**
- **Ce composant permet de prendre en charge les champs de type « file » dans les formulaires HTML**
- **Ce projet s'appuie sur les librairies**
  - **commons-fileupload-1.2**
  - **commons-io-1.3**
  - **tapestry-core-5**
- **Le paramétrage de l'outil se réduit donc à obtenir les dépendances vers les projets ci-dessus**
- **Tapestry supprime les fichiers temporaires après chaque requête**

Voici quelques paramètrages possibles concernant le composant Upload. (Paramètres modifiables dans AppModule.java)
* upload.repository-location : Le repertoire dans lequel les fichiers trop volumineux seront stockés.

* upload.repository-threshold : La taille maximale des fichiers qui seront gardés en mémoire. Si la taille est supérieure, ils seront stockés dans le repertoire paramètré par <upload.repository-location.

* upload.filesize-max : Taille maximale d'un fichier.

- **Ajouter un champ de type « upload » dans votre formulaire**
- **Définir un identifiant pour le champ**
  - **Utiliser soit l'attribut « value »**
  - **A défaut c'est l'attribut « id » qui est utilisé**

```
<form t:type="form" t:id="uploadForm">
  <div t:type="errors" />
  <input t:type="upload" t:id="file" t:validate="required"/>
  <br/>
  <input t:type="submit" value="upload"/>
</form>
```


Atos Origin

Composants Avancés
Upload - Utilisation

- **Tapestry crée au moment de la soumission un objet du type « UploadedFile »**

```
public class UploadExemple {

  private UploadedFile file;

  public UploadedFile getFile(){
    return this.file;
  }

  public void setFile(UploadedFile aFile){
    this.file = aFile;
  }

  @OnEvent(value=EventConstants.SUCCESS,component="uploadForm")
  public void uploadFile(){
    File copied = new File("/path/to/file/"+file.getFileName());
    file.write(copied);
  }
}
```

- **Le projet dépend du projet « commons-fileupload » pour le traitement et le stockage des fichiers uploadés**
- **Le nettoyage des fichiers temporaires est réalisé par un Thread contenu dans la librairie « commons-io »**
  - ⚠ **Peut poser des problèmes si le JAR est partagé par plusieurs applications**
  - ⚠ **Actuellement, il n'est pas possible de désactiver ce Thread**
  - ⚠ **Il n'existe pas non plus de moyen d'étendre ou de modifier la politique de suppression des fichiers**

TP 16 : Utilisation du composant Tapestry Upload

Concepts abordés

- Utilisation du composant « upload »
- Utilisation du composant utilisateur « photo »
- Analyse du code d'un objet de type « StreamResponse »

Dans ce TP, nous abordons l'utilisation du composant « upload » pour affecter une photo à l'utilisateur. L'utilisation d'un composant fourni avec le TP vous permettra d'analyser comment utiliser les objets de type « StreamResponse » pour générer du contenu binaire pour l'utilisateur.

⚠ Ce composant est fourni uniquement à titre d'exemple et ne peut être utilisé dans l'état dans un environnement de production.

Enoncé

1. Annoter les variables d'instance de la classe « UploadPhoto » pour
    a. Récupérer l'instance de l'utilisateur loggé
    b. Manipuler l'instance du composant Formulaire identifié par « uploadPhotoForm »

```
/**
 * Used to stored the authenticated user after authentication or
 * registration
 */
@SessionState
private User loggedUser;

/** Created by Tapestry to check if the loggedUser has been set in session */
private boolean loggedUserExists;

/** Used to add global error message on security checks */
@Component(id = "uploadPhotoForm")
private Form uploadPhotoForm;
```

2. Annoter et implémenter la méthode « uploadUserPhoto » pour
    a. Copier le fichier uploadé dans un répertoire de votre choix
    b. En cas de succès de l'upload
        i. Modifier l'utilisateur actuellement loggé pour lui affecter le chemin d'accès à la photo uploadé (attribut « photoPath » de la classe « User »)

ii. Rediriger l'utilisateur vers la page « Main »

c. En cas d'échec, si l'utilisateur n'est pas authentifié par exemple, le rediriger vers la page « Login »

```
/**
 * Used to process upload on form submission and modify logged user if
 * exists.
 *
 * @return Main page on success, Index on failure
 */
@OnEvent(value = EventConstants.SUCCESS, component = "uploadPhotoForm")
public Object uploadUserPhoto() {

    File copied = new File(globals.getContext().getRealFile("/static/images").getAbsolutePath()+ File.separator +  file.getFileName());
    file.write(copied);

    if (loggedUserExists) {
        // Set photo path for the logged user
        loggedUser.setPhotoPath(copied.getPath());
        return Main.class;
    } else {
        return Login.class;
    }

}
```

3. Compléter le Template « UploadPhoto » pour

   a. Déclarer un composant « form » identifié par « uploadPhotoForm »

   b. Modifier les champs du formulaire et utiliser le composant « upload »

   c. Utiliser la variable d'instance « file » pour stocker le fichier uploadé

```
<form t:type="form" t:id="uploadPhotoForm" id="uploadPhotoForm" method="post">

    <p><span>_errors</span></p>

    <fieldset id="fsGeneral">
        <p><label for="photoPath">Photo path</label> <input t:type="upload" t:id="file" type="file" id="photoPath" /></p>
    </fieldset>

    <fieldset id="fsSubmit">
        <p><input t:clientId="btSubmit" t:type="submit" value="Submit" /></p>
    </fieldset>

    <p><span class="commts">&lt;&lt; <a t:type="pagelink" t:page="main">Back to main page</a></span></p>

</form>
```

4. Modifier le Template « Main », utiliser le composant « photo » pour afficher la photo de l'utilisateur. Ce composant contient un paramètre obligatoire :

   a. filepath : chemin vers le fichier. Si ce paramètre est erroné ou que la photo n'existe pas, rien ne s'affiche

```
<p t:type="photo" t:filepath="loggedUser.photoPath"></p>
```