

# Mécanismes Avancés

161

*Turning Client Vision into Results*

## Mécanismes Avancés

- **Extension de la Validation**
  - Prise en charge de nouveaux types
  - Composants utilisateur
  - I18N
  - Assets
  - Services
  - Javascript/Ajax
  - Test unitaire des pages/composants

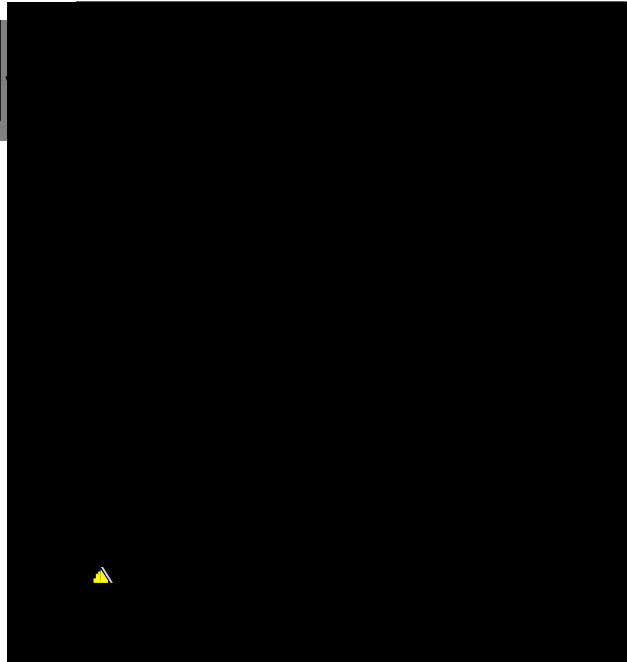
162

*Turning Client Vision into Results*

- **Utiliser les méthodes de type « contributeXxx » de la classe « AppModule »**
  - **Plusieurs types de contributions**
    - Ajout de filtre pour les requêtes
    - Ajout de « Dispatcher » pour le traitement des requêtes
    - Modification des paramètres par défaut (page de démarrage, langues supportées...)
    - Mais aussi ...
  - **Prise en charge de nouveaux types**
    - Validation
    - Transtypage
- <http://canalweb.atosworldline.com/FR/ART/5117811/How-To-enhance-to-Tapestry-5.htm>

Les méthodes contribute permet de contribuer aux fonctionnalités de Tapestry

1. Implémenter l'interface  
`org.apache.tapestry5.Validator`



```
public final class DateValidator implements Validator<String, String>
{
    public Class<String> getConstraintType() {
        return String.class;
    }

    public String getMessageKey() {
        return "date-validator";
    }

    public Class<String> getValueType() {
        return String.class;
    }

    public void validate(Field field, String constraintValue, MessageFormatter
formatter, String value) throws ValidationException {
        // Implémentation du contrôle de type
    }

    public void render(Field arg0, String arg1, MessageFormatter arg2,
MarkupWriter arg3, RenderSupport arg4) {
        // TODO Auto-generated method stub
    }

    public boolean isRequired() {
        return false;
    }
}
```

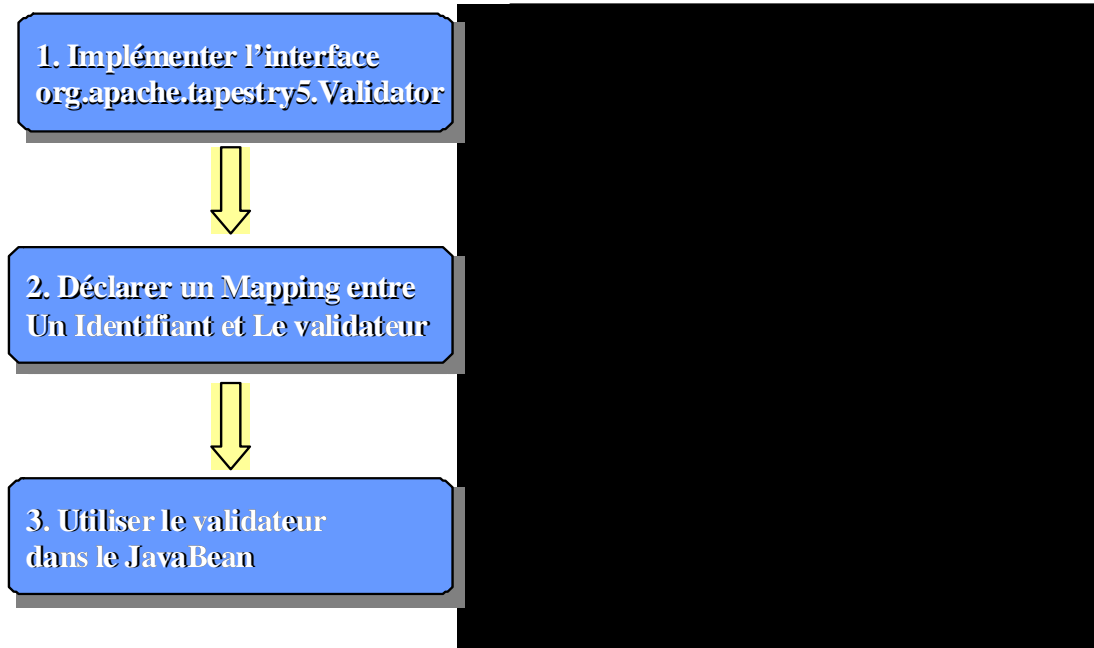
Les classes des Validator, que nous implémentons, seront le plus souvent placées dans le package Services.

1. Implémenter l'interface  
`org.apache.tapestry5.Validator`



2. Déclarer un Mapping entre  
Un Identifiant et Le validateur

```
public static void contributeFieldValidatorSource(
    MappedConfiguration<String, Validator> configuration) {
    configuration.add("datevalidator", new DateValidator());
}
public static void contributeValidationMessagesSource(
    Configuration<String> configuration) {
    configuration.add("net/atos/mm/services/myMessages.properties"
);
}
```



## TP 16 : Ajout d'un validateur d'extension de fichiers

- Ecrire un validateur propriétaire
- Sécurisation de l'upload par un mécanisme simple de comptage

L'objectif de ce TP est de sécuriser la page d'upload réalisée au cours du TP précédent. Les objectifs sont les suivants :

1. Désactiver les champs de saisie après trois tentatives
2. Ajouter un validateur propriétaire qui contrôle les extensions des fichiers « uploadés »

### Enoncé

1. Créer une classe « FileExtensionValidator » qui implémente l'interface « Validator »
  - a. Type de la contrainte : String, liste des extensions séparés par le caractère « : »
  - b. Type des valeurs testées : UploadedFile
  - c. Clé du message d'erreur : « extension-not-supported »

```
public Class<String> getConstraintType() {  
    return String.class;  
}  
  
public String getMessageKey() {  
    return "extension-not-supported";  
}  
  
public Class<UploadedFile> getValueType() {  
    return UploadedFile.class;  
}
```

2. Déclarer la classe « FileExtensionValidator » dans la classe « AppModule » sous l'identifiant « extensionValidator »

```
public static void contributeFieldValidatorSource(  
    MappedConfiguration<String, Validator> configuration) {  
    configuration.add("extensionValidator", new FileExtensionValidator());  
}  
  
public static void contributeValidationMessagesSource(  
    OrderedConfiguration<String> configuration) {  
    configuration.add("extensionValidator", "net/atos/mm/formation/tapestry/services/FileExtensionValidator.properties");  
}
```

La méthode `contributeFieldValidatorSource` permet d'associer un libellé de `Validator` à sa classe java.

La méthode `contributeValidatorMessageSource` permet d'ajouter un fichier `properties`, réservé aux messages de validation.

### 3. Sécurisation de classe « UploadPhoto »

- a. Ajouter l'annotation sur la variable « `count` » qui nous permettra de comptabiliser le nombre d'erreurs de validation

```
/** Used to store in session the number of validation error */
@Persist
private int count;
```

- b. Ajouter l'annotation sur la variable « `uploadPhotoForm` » pour pouvoir manipuler le formulaire dans la suite du TP

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

- c. Annoter et implémenter la méthode « `countFailure` » pour compter le nombre d'échec de validation du formulaire

```
/**
 * Used by the form on failure to increment the failure counter and get back
 * to the input upload form if a validation error occurs
 *
 * @return The uploadPhoto page
 */
@OnEvent(value = EventConstants.FAILURE, component="uploadPhotoForm")
public Object countFailure() {
    this.count++;
    return UploadPhoto.class;
}
```

La méthode `countFailure` sera exécutée lorsque le formulaire `uploadPhotoForm` sera incorrect.



d. Annoter et implémenter la méthode « `verifySecurityException` » pour vérifier le nombre de tentatives d'upload échouées (méthode « `getMaxAttemptsExceeded` ») pendant la phase de l'activation de la page, si le nombre de tentatives est dépassé :

i. Vider la liste des erreurs du formulaire

ii. Ajouter un message d'erreur général qui indique que le formulaire est désactivé pour des raisons de sécurité

```

/**
 * Used to verify if the upload feature must be disabled
 *
 * @return true if the number of failed attempts exceeds 3, false otherwise
 */
public boolean getMaxAttemptsExceeded() {
    return count >= 3;
}

/**
 * Used to verify during page activation, if the upload feature is available
 */
@OnEvent(EventConstants.ACTIVATE)
private void verifySecurityException() {
    // Add security check here
    if(getMaxAttemptsExceeded()) {
        uploadPhotoForm.clearErrors();
        uploadPhotoForm.recordError("You have done more than three attempts to upload illegal files.");
        uploadPhotoForm.recordError("Upload feature is disabled for this session");
    }
}

```

4. Modifier le Template HTML « UploadPhoto » pour

a. Ajouter une instance de composant « errors »

b. Ajouter les contraintes suivantes sur le champ de type « upload »

i. Obligatoire

ii. Autoriser « `.png:.gif:.jpeg:.jpg` »

c. Désactiver les champs du formulaire si le nombre de tentatives échouées est supérieur à 3

```

<fieldset id="fsGeneral">
    <p><label for="photoPath">Photo path</label>
        <input t:type="upload"
            t:id="file"
            type="file"
            id="photoPath"
            t:disabled="maxAttemptsExceeded"
            t:validate="required,extensionValidator=.png:.gif:.jpeg:.jpg" />
    </p>
</fieldset>

```

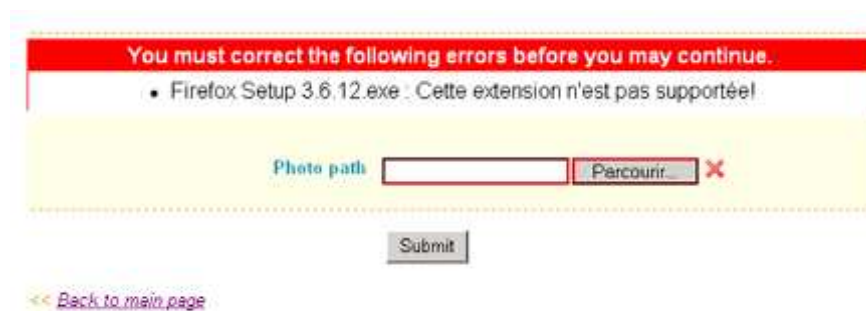
Le paramètre t:disabled prend comme valeur le retour de la fonction getMaxAttemptsExceed.

Le paramètre t:valide prend comme valeur notre Validator.

5. Tester l'application avec des fichiers ne respectant pas les extensions ci-dessus

⚠ La vérification de ce TP peut nécessiter un redémarrage du serveur Tomcat

Voici le message d'erreur lorsque le fichier n'a pas une extension correcte.



The screenshot shows a web application interface with a red error banner at the top that reads "You must correct the following errors before you may continue.". Below the banner, a list of errors is displayed: "• Firefox Setup 3.6.12.exe : Cette extension n'est pas supportée!". The main form area has a yellow background and contains a label "Photo path" next to an empty text input field. To the right of the input field is a "Parcourir..." button with a red "X" icon. Below the input field is a "Submit" button. At the bottom left of the form, there is a link "<< Back to main page".

## Mécanismes Avancés

- Extension de la Validation
- **Prise en charge de nouveaux types**
- Composants utilisateur
- I18N
- Assets
- Services
- Javascript/Ajax
- Test unitaire des pages/composants

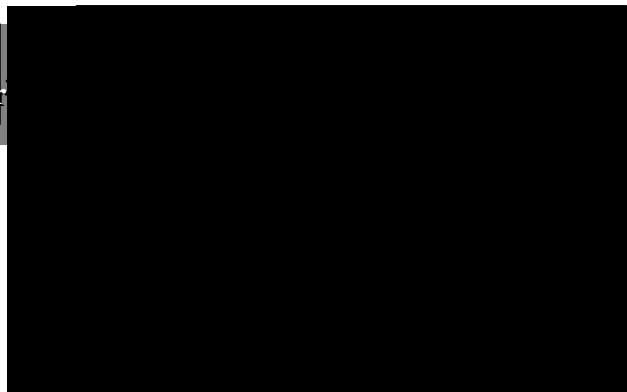
170

*Turning Client Vision into Results*

## Mécanismes Avancés

*Prise en charge de nouveaux types*

1. Implémenter l'interface  
`org.apache.tapestry5.Translator`



171

*Turning Client Vision into Results*

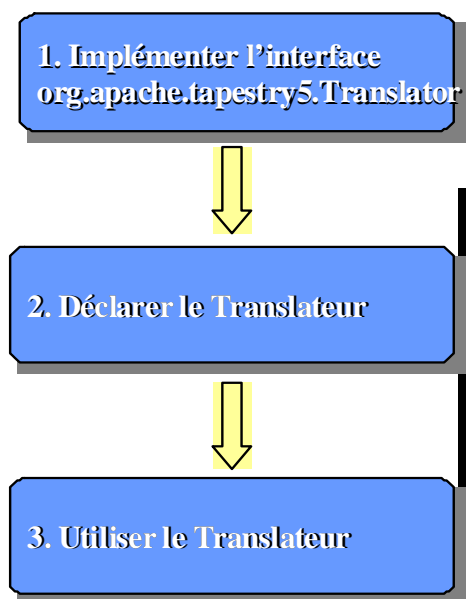
```
public final class IntegerTranslator extends IntegerNumberTranslator<Integer>
{
    // CF Tapestry source code
}
```

1. Implémenter l'interface  
`org.apache.tapestry5.Translator`



2. Déclarer le Translateur

```
public static void contributeTranslatorSource(
    MappedConfiguration<Translator> configuration) {
    configuration.add(new IntegerTranslator());
    // ...
}
```



## Mécanismes Avancés

- Extension de la Validation
- Prise en charge de nouveaux types
- **Composants utilisateur**
- I18N
- Assets
- Services
- Javascript/Ajax
- Test unitaire des pages/composants

## Mécanismes Avancés

*Ecrire un composant utilisateur*

### Qu'est-ce qu'un composant ?

- Présentation de données et Interaction avec l'utilisateur
- Réutilisable
- Paramétrable
- Ne contacte pas directement le Middle
- Possède sa propre gestion d'évènements et son propre cycle de vie

### Méthode

1. Structurer le projet Tapestry
2. Définir les paramètres du composant
3. Écrire le Template d'affichage
4. Écrire la gestion des évènements et implémenter certaines méthodes de rendu

#### 1. Créer le sous-package « components »

- On placera les sources Java des composants dans un répertoire dédié
- On placera les Templates au même niveau de package mais dans un répertoire racine dédié aux ressources

#### 2. La classe du composant

- Publique et possède un constructeur public sans paramètres (les autres sont ignorés)
- Variables d'instance privées (dû à un mécanisme interne de transformation des classes)
- Variables d'instance réinitialisées à la fin de chaque requête
- L'annotation « @Retain » évite la suppression de l'information à la fin de la requête mais n'assure pas qu'un utilisateur retrouve la valeur qu'il avait fixé

#### 3. Un composant peut en contenir d'autres (dits « embedded »)

- Si un composant « embedded » est déclaré dans la classe alors il doit être utilisé dans le Template

Attention l'annotation « @Retain » n'assure pas que l'utilisateur retrouvera la valeur à laquelle il l'avait affectée, elle signifie simplement que l'information n'est pas supprimée à la fin du traitement de la requête. Autrement dit, c'est Tapestry qui gère les instances des composants, même si l'utilisateur utilise deux fois un composant, il peut très bien retourner une instance dont la variable vaut encore « null »

- **Les paramètres**
  - Variables d'instance privées associées à l'annotation `@Parameter`
  - Par défaut, un paramètre est optionnel. Pour le rendre obligatoire : `@Parameter(required=true)`
  - Donner une valeur par défaut pour les paramètres optionnels
- **Tapestry gère**
  - Le Transtypage
  - Le nommage des attributs
  - Différents types de « binding » pour associer un paramètre à sa valeur
  - Les paramètres sont « Bidirectionnels » (Ex : attribut « row » du composant « Grid »)
  - Le cache des paramètres : `@Parameter(cache=true)`

La notion de paramètre « Bidirectionnel » permet d'associer une propriété du composant à une variable de la page qui va contenir le composant. C'est-à-dire que si la valeur du paramètre change dans le composant, ceci aura pour effet de modifier la valeur de l'attribut de la page.

Lors de l'héritage d'un composant, il est possible de surcharger la valeur d'un attribut du composant père en



- L'annotation « **@Parameter** » possède un attribut de type chaîne « **defaultPrefix** » permettant de spécifier le type de Binding par défaut pour le paramètre en question, parmi la liste suivante
  - « **prop:** » (utilisé par défaut) : une propriété de la page
  - « **literal:** » : une chaîne de caractère littérale
  - « **message:** » : permet d'extraire une valeur dans un catalogue de messages
  - « **inherit:** » : cf. Agrégation de composant

**Ex :**

```
@Parameter(defaultPrefix = "literal")  
private String _remove;
```

« **block:** » : Identifiant d'un « block » dans le Template  
« **component:** » : Identifiant d'un composant dans le Template  
« **translate:** » : Le nom d'un « Translator » configuré  
« **validate:** » : La spécification d'une liste de validateurs

- On reprend les règles d'écriture des Templates de page
- Les Templates doivent être dans le classpath de l'application et ne peuvent pas être placés dans le contexte contrairement aux Pages
- Héritage : si un composant étend un autre composant et qu'il ne redéfinit pas le Template, alors il dispose par défaut du Template de son parent

- La balise « `<t:body>` » permet de faire référence dans le Template d'un composant au contenu de la balise qui utilise le composant
- Ex :  
Un composant hypothétique « *decorate* » ci-dessous prend le contenu et ajoute une balise de type « *bold* »  
`<p t:type="decorate">`  
    Mon Contenu  
`</p>`  
Pour faire référence à « Mon Contenu » dans le Template du composant  
`<b><t:body/></b>`

- Les phases de rendu intermédiaires permettent d'influer sur l'affichage du corps
- Pour l'écriture de composants réutilisables et génériques, il pourra être intéressant d'implémenter d'autres méthodes de rendu
  - @BeginRender
  - @BeforeRenderTemplate
  - @BeforeRenderBody
  - @AfterRenderBody
  - @AfterRender
  - <http://tapestry.apache.org/tapestry5.1/guide/rendering.html>

- **Valeur de retour possible**
  - Booléen : permet de naviguer dans le diagramme d'état présenté dans la première partie de l'exposé
  - Un composant : Dans ce cas, il faut injecter au préalable le composant avec l'annotation « @Component », ou le passer en paramètre
- **Renvoyer un composant l'ajoute à la queue des composants dont il faut traiter le rendu**
- **Ce rendu suspend le rendu du composant actif**
- **Le traitement récursif de rendu de composants est interdit**

**« org.apache.tapestry5.ComponentResources »**

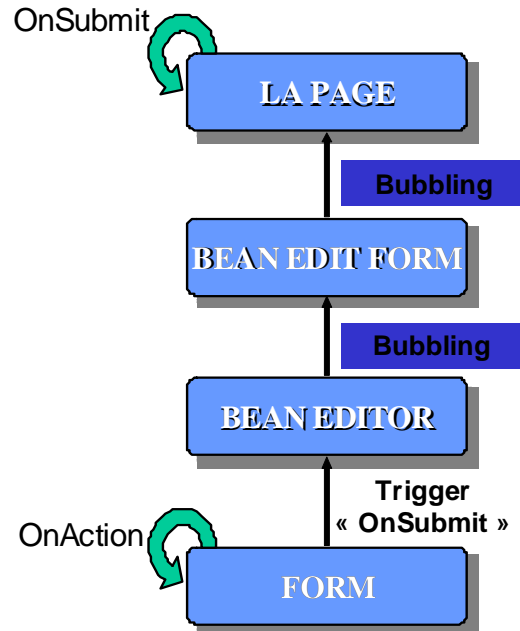
- **Permet de faire le lien entre ce qui est fourni par le Framework Tapestry, l'environnement et le composant**
- **Voici une liste non exhaustive des méthodes proposées par cet objet**
  - **getPage()** : obtenir la page contenant le composant
  - **getContainerMessages()** : obtenir le catalogue de messages de la page qui contient le composant
  - **getEmbeddedComponent(String id)** : obtenir une référence sur un composant « embedded »
  - **isBound(String parameterName)** : permet de savoir si un paramètre a été configuré
  - **triggerEvent(String, Object[], ComponentEventHandler)** : permet d'exécuter un évènement qui se propage du composant à la page

Pour un paramètre de type primitif, il peut être difficile de détecter si le paramètre a été valué ou non puisqu'ils ont des valeurs par défaut.

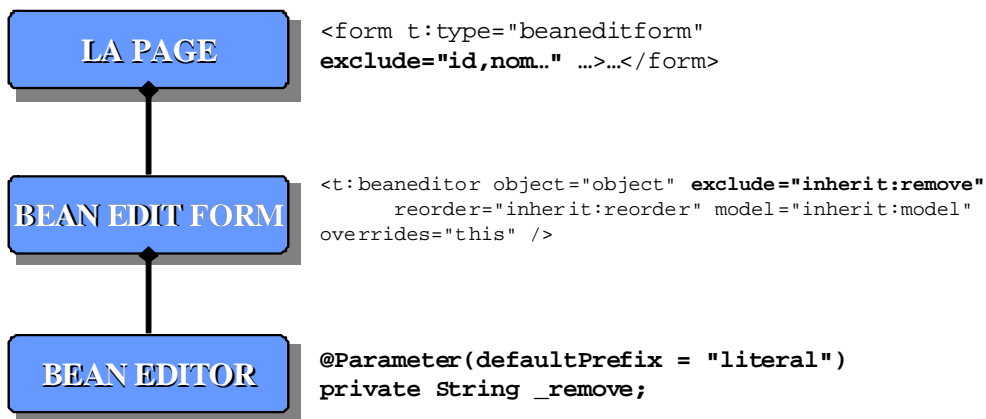
Dans ce contexte, comment savoir si le paramètre a été fixé par l'utilisateur ou s'il s'agit de sa valeur par défaut.

L'utilisation de la méthode « isBound » permet de savoir de manière explicite si l'attribut a été fixé ou pas.

- Tapestry gère un cycle de vie pour les composants similaires à celui des pages
- La propagation des événements se fait du composant jusque la page en passant par les composants intermédiaires jusqu'à ce qu'une réponse soit donnée
- Ex: Lorsque l'utilisateur appuie sur le bouton « Submit », ceci émet une requête de type « action » sur le composant Form qui propage à son tour un événement « OnSubmit »



- Le préfixe « inherit: » permet d'associer un paramètre du composant à un des paramètres d'un composant inclus
- Ceci permet de ne valoriser qu'une seule fois le paramètre au niveau de la page principale



### Concepts abordés

- Créer un composant utilisateur simple

L'objectif de ce TP est de créer un composant simple qui affiche un pied de page sur la page « Main »

### Enoncé

1. Dans le package « components », créer une classe Footer qui étend la classe « BaseComponent »

```
package net.atos.mm.formation.tapestry.components;

import net.atos.mm.formation.tapestry.base.BaseComponent;

public class Footer extends BaseComponent {

}
```

2. Analyser le code de classe « BaseComponent », cette classe permet d'inclure une feuille de style dans le document qui sera généré. Il s'agit d'un exemple d'utilisation des ressources embarquées
3. Créer le fichier Template associé dans le package « components » du répertoire « ressources » et y copier le code ci-dessous

```
<fieldset class="footer" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <legend class="overtitle">
    Copyright ©#169; 2008 Atos Worldline
  </legend>
</fieldset>
```

4. Modifier le Template de la page « Main » pour utiliser le composant « Footer » nouvellement créé et observer le contenu du fichier HTML généré



ADD A NEW PORTFOLIO

UPLOAD PHOTO

Bonjour Mr **tapestry**.

Voici la liste de vos portefeuilles :

[Advanced Grid](#)

Label	Amount	Booking date
Portfolio n°0	100.0	Mon Dec 13 09:04:29 CET 2010
Portfolio n°1	200.0	Mon Dec 13 09:04:39 CET 2010
Portfolio n°2	300.0	Mon Dec 13 09:04:49 CET 2010
Portfolio n°3	400.0	Mon Dec 13 09:04:59 CET 2010
Portfolio n°4	500.0	Mon Dec 13 09:05:09 CET 2010
Portfolio n°5	600.0	Mon Dec 13 09:05:19 CET 2010
Portfolio n°6	700.0	Mon Dec 13 09:05:29 CET 2010
Portfolio n°7	800.0	Mon Dec 13 09:05:39 CET 2010
Portfolio n°8	900.0	Mon Dec 13 09:05:49 CET 2010
Portfolio n°9	1000.0	Mon Dec 13 09:05:59 CET 2010

Copyright © 2008 Atos Worldline



# Modification d'un composant utilisateur complexe

## Concepts abordés

- Agrégation de composants
- Passage de paramètres
- Héritage de paramètre
- Bubbling

L'objectif ce TP est d'appréhender l'écriture de composants plus complexes, ce qui nous permettra de préparer le TP suivant et d'analyser les concepts liés au composant. Nous allons modifier le composant « Menu » que nous utilisons depuis le début du TP pour lui ajouter les fonctionnalités suivantes :

- Choix de la langue de l'application : Le composant « country » va nous permettre d'implémenter le choix de la langue dans le TP suivant
- Logout : Le composant « logout » propose un lien permettant de quitter l'application et de rediriger l'utilisateur vers la page de Login (paramètre « indexPage »). Il propage aussi un événement « logout » que nous utiliserons pour supprimer l'utilisateur en session

## Enoncé

1. Analyser le contenu du composant « Menu » pour identifier comment le composant affiche la liste des éléments du menu
2. Annoter la variable d'instance « menuIndexPage » pour que celle-ci devienne un paramètre obligatoire du composant et ayant pour préfixe par défaut « literal »

```
@Parameter(required = true, defaultPrefix="literal")
private String menuIndexPage;
```

3. Modifier le Template du Menu pour
  - a. Ajouter le composant « logout » : ce composant dispose d'un paramètre obligatoire « indexPage » qui doit être associé au paramètre « menuIndexPage » du composant « Menu »
  - b. Ajouter le composant « country »

```

<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">

  <div id="logout" t:type="logout" t:indexPage="inherit:menuIndexPage"/>

  <div id="header">

    <p id="logo"></p>
    <h1>Tapestry 5 Training Application</h1>

  </div>

  <div id="menu">
    <p t:type="country" />
    <br />
    <t:loop t:type="loop" t:source="verifiedList" t:value="currentAction">
      <p>
        <a class="btlink" t:type="pagelink" t:page="prop:currentAction.url">${currentAction.libelle}</a>
      </p>
    </t:loop>
  </div>

</t:container>

```

4. Annoter et implémenter la méthode « logout » pour que celle-ci réponde à l'évènement « logout » propagé par le composant « logout » inclus précédemment. L'objectif est ici de mettre en évidence la notion de Bubbling. Une fois la méthode annotée, vous pourrez voir s'afficher les traces sur la sortie standard.

```

/**
 * This method must be called on logout component action using the bubbling
 * process.
 */
@OnEvent(value = "logout", component="logout")
public void logout() {

    System.out
        .println("Menu Component : Logged User has been removed from session...");
    System.out.println("Menu Component : Logout Event Completed...");

}

```

5. Modifier la page Main pour préciser le paramètre « indexPage » et tester la fonctionnalité de logout

```

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

```

Le paramètre t:menuIndexPage permet d'indiquer le nom de la page vers laquelle nous voulons être redirigés.

## Mécanismes Avancés

- Extension de la Validation
- Prise en charge de nouveaux types
- Composants utilisateur
- **I18N**
- Assets
- Services
- Javascript/Ajax
- Test unitaire des pages/composants

- **Rendre l'application indépendante de la langue de l'utilisateur**
  - Adapter la vue pour la langue de l'utilisateur
    - Affichage des dates
    - Affichage des formats de devises
    - ...
  - Adapter l'ergonomie générale en fonction de critères associés au pays en question
- **Extraire les données textuelles du code Java**
- **Faciliter la maintenance et la prise en charge de nouvelles langues**

L'I18N s'applique aux Bundle, aux templates et aux assets.

- **La Localisation**
  - Quelle est l'information pertinente à présenter ?
  - Quelle langue utiliser pour présenter cette information ?
- **Avec FW3**
  - Soit « dynamique » : utilisation des « Resource Bundle » Java
  - Soit « statique » : génère autant de pages JSP que de langues supportées
- **Tapestry est plus souple**
  - Catalogues hiérarchisés : équivalent « Ressource Bundle »
  - Et/Ou Internationalisation de Template
    - Il faut suffixer une ressources par une locale pour lui ajouter le support de l'i18n dans la locale courante

Tapestry utilise le terme de « localisation » pour englober l'internationalisation. De part le modèle utiliser pour la gestion du catalogue de message, Tapestry doit identifier pour une clé donnée la bonne information à présenter en fonction du contexte. La notion d'héritage implique d'une information peut se trouver dans toute la hiérarchie du composant ou de la page.

Tapestry permet d'utiliser des catalogues de messages ou d'internationaliser les Templates directement, mais ce mécanisme n'est pas exclusif.

- **Catalogue global :**
  - WEB-INF/AppName.properties
  - WEB-INF/AppName\_fr.properties
  - ...
- **Un catalogue spécifique par Page**
  - Pour une page org.exemple.pages.SamplePage
    - org/exemple/pages/SamplePage.properties
    - org/exemple/pages/SamplePage\_en.properties
    - org/exemple/pages/SamplePage\_en\_GB.properties
    - org/exemple/pages/SamplePage\_en\_US.properties
    - org/exemple/pages/SamplePage\_fr.properties
    - ...

**Rappel :** « AppName » est équivalent au nom donné au filtre Tapestry dans le fichier « web.xml »

- **Spécialiser un message du catalogue global**
  - Une page peut redéfinir un message du catalogue global
- **Héritage et surcharge**
  - Si une page étend un composant, alors elle hérite du catalogue associé au composant
  - Elle peut redéfinir les messages ou utiliser les valeurs par défaut

- **Dans un Template : utiliser le préfixe « message: »**
  - Dans un attribut : `att="message:xxx"`
  - Dans les expansions : `${message:xxx}`
- **Exemple :**
  - Soit un fichier de ressources contenant les informations suivantes

```
page-title=Your Account
greeting=Welcome back
```

- Voici la page exploitant le préfixe dans un attribut et via une expansion

```
<html t:type="layout" t:title="message:page-title">
  ${message:greeting}, ${user.name}! . . .
</html>
```

- Dans la classe associée au Template : Injecter le service de messages

```
@Inject private Messages messages;
```

- Ex: soit un fichier de ressources contenant les informations suivantes

```
no-items=Your shopping cart is empty.  
item-summary=You have %d items in your cart.
```

- Obtenir un message et le formater
  - Obtenir un message : « `get(String key)` »
  - Formater un message : « `format(String key, Object... args)` » (eq. `printf C`)

```
public String getCartSummary() {  
    if (items.isEmpty())  
        return messages.get("no-items");  
    return messages.format("item-summary", items.size());  
}
```

- Fonctionnalités
  - Une clé manquante génère un message par défaut : [missing key: key-not-found] ou « key-not-found » est le nom de la clé manquante
  - Rechargement automatique
  - L'internationalisation fonctionne aussi pour les « Assets »



- Un Template HTML associé à une page peut lui-même être internationalisé
- La même norme de nommage peut être appliquée aux Templates
- Ex : MaPage.tml peut exister sous la forme
  - MaPage\_fr.tml
  - MaPage\_en.tml
  - MaPage\_en\_GB.tml
  - ...

- **Pour une application internationalisée, paramétrer le symbole « `tapestry.supported-locales` »**
  - Tapestry lui affecte une valeur par défaut.
  - Définit la liste des locales supportées par l'application
  - Contient une liste des locales séparées par des virgules
- **Tapestry utilise ce symbole et détermine en fonction de la locale de la requête quelle langue s'approche le plus de ce que l'application peut servir**
- **Si l'application ne supporte pas la langue demandée, Tapestry vous redirigera vers la page principale.**
- **Ex :**
  - `tapestry.supported-locales=en,fr`
  - La requête de l'utilisateur demande à être servie en `en_US`
  - Ce sera le catalogue « `en` » qui sera choisi

Il existe trois moyens pour renseigner un symbole :

1. « `context-param` » dans le fichier `web.xml`
2. « `contributeApplicationDefaults` » dans `AppModule.java`
3. Utilisation des variables système de la JVM « `-D` »

- Header HTTP « Accept-Language »
- Priorité à la locale présente dans l'URL
- Comment modifier la locale programmatiquement ?
  - Utiliser le service « PersistentLocale » et sa méthode set(Locale \_locale)

```
@Inject
private PersistentLocale persistentLocaleService;

@OnEvent(value="action",component="FrLocaleLink")
void selectFrLang() {
    persistentLocaleService.set(Locale.FRANCE);
}
```

Dans le Template :

```
<a t:type="ActionLink" t:id="frLocaleLink" href="#">Francais</a>
```

- Dans l'exemple ci-dessus un lien permet d'activer la locale Française
- Ceci aura pour effet de modifier les URLs pour y ajouter la locale choisie

### Concepts abordés

- Mécanisme de sélection programmatique de la locale
- Utilisation des ressources Bundle simple
- Messages paramétriques

L'objectif de ce TP est de démontrer le fonctionnement de l'internationalisation via les fichiers de ressources de type « Bundle »

### Enoncé

1. Modifier la classe du composant « Country » pour modifier la locale de l'utilisateur grâce au « PersistentLocale »

```
@OnEvent(EventConstants.ACTION)
public void changeLocale(String country){

    // Implement here the mechanism to change the user local
    System.out.println("Country Component : Modify local to "+country);
    if("fr".equals(country) ) {
        persistentLocaleService.set(Locale.FRENCH);
    }else{
        if("en".equals(country)) {
            persistentLocaleService.set(Locale.ENGLISH);
        }else{
            System.out.println("Country Component : Locale not supported");
        }
    }
}
```

2. Créer trois fichiers de ressources dans le package « pages »
  - a. Main.properties
  - b. Main\_fr.properties
  - c. Main\_en.properties
3. Ajouter deux clés dans ces fichiers
  - a. hello : message de bienvenue
  - b. datePattern : contient le pattern d'affichage de l'attribut « bookingDate »
4. Modifier le Template de la page « Main » pour afficher le message de bienvenue

```
<p>${message:hello} <b>${loggedUser.login}</b>.</p>
```

5. Modifier la classe de la page « Main » pour paramétrer la valeur de l'objet « dateFormat »

- a. Injecter le catalogue de message dans la page
- b. Pendant la phase d'activation de la page, créer une nouvelle instance de l'objet « SimpleDateFormat » avec pour format la valeur associée à la clé « datePattern », si le format préciser dans le fichier de propriété est erroné, utiliser un pattern par défaut « dd/MM/yyyy »

```
@OnEvent(EventConstants.ACTIVATE)
public Object assertUserExists() {

    // Create date formatter
    if (messages.contains("dateFormat")) {
        try {
            dateFormat = new SimpleDateFormat(messages.get("dateFormat"));
        } catch (Exception ex) {
            dateFormat = new SimpleDateFormat(DEFAULT_DATE_FORMAT);
        }
    } else {
        dateFormat = new SimpleDateFormat(DEFAULT_DATE_FORMAT);
    }

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

    return null;
}
```

6. Modifier la classe « AppModule » pour préciser la liste des langues supportées par l'application

```
...

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("tapestry.supported-locales", "en,fr");

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

7. Tester l'application et vérifier que le changement de langue fonctionne correctement
8. Optionnel
  - a. Créer un message de bienvenue ayant pour paramètre une chaîne de caractère (login de l'utilisateur)

- b. Créer une méthode permettant d'instrumenter le paramètre du message avec le login de l'utilisateur connecté
- c. Afficher ce message dans le Template

### Aller plus loin

Dynamiser le composant « Country » pour afficher autant de drapeaux (ou de lien simples) que préciser dans la variable système « `tapestry.supported-locales` ».

Indications :

- Obtenir la valeur de cette variable via les annotations « `@Inject` » et « `@Symbol` »
- Parser la valeur de cette variable
- Parcourir la liste des langues supportées pour afficher autant de lien que nécessaire
- Le service `AssetSource` permet de créer des assets dynamiquement

Ex : pour obtenir la valeur d'un symbole dans une page ou un composant de l'application

```
@Inject
@Symbol("tapestry.supported-locales")
private String supportedLocales;
```

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <t:loop t:source="supportedLocales" t:value="locale">
    <a t:type="actionlink" t:id="flag" t:context="locale" class="flag">
      
    </a>
  </t:loop>
</t:container>
```

```
public class Country extends BaseComponent {

    @Inject
    private PersistentLocale persistentLocaleService;

    @Inject
    @Symbol("tapestry.supported-locales")
    private String supportedLocales;

    @Inject
    private AssetSource assetSource;

    @Property
    private String locale;

    public Asset getFlag() {
        return assetSource.getAsset(null, "context:static/images/flag.png", new Locale(locale));
    }

    @OnEvent("action")
    public void changeLocale(String country) {

        // Implement here the mechanism to change the user local
        System.out.println("Country Component : Modify local to "+country);
        if("fr".equals(country)) {
            persistentLocaleService.set(Locale.FRENCH);
        } else {
            if("en".equals(country)) {
                persistentLocaleService.set(Locale.ENGLISH);
            } else {
                System.out.println("Country Component : Locale not supported");
            }
        }
    }

    public String[] getSupportedLocales() {
        return supportedLocales.split(",");
    }
}
```

Nous récupérons la liste des locales pris en charge par l'application, grâce à la variable `supportedLocales`. Pour pouvoir alimenter le composant `Loop`, nous devons transformer cette variable en un tableau. Cette transformation est réalisée grâce à la fonction `getSupportedLocales`.

Ensuite, pour chaque locale, nous affichons le drapeau correspondant, grâce au getter `getFlag`.