Bonne pratique de développement

Table des matières

[1 Guide d’utilisation 3](#_Toc447896433)

[1.1 Eclipse 3](#_Toc447896434)

[1.2 Tomcat 3](#_Toc447896435)

[1.2.1 Ajouter un serveur local à Eclipse 3](#_Toc447896436)

[1.2.2 Déployer un projet sur le serveur local 5](#_Toc447896437)

[1.3 GIT 6](#_Toc447896438)

[1.3.1 Générer une clé publique pour connecter au serveur GIT 6](#_Toc447896439)

[1.3.2 Importer un projet depuis le serveur GIT 8](#_Toc447896440)

[1.3.3 Tracer les modifications d’un nouveau fichier 11](#_Toc447896441)

[1.3.4 Ignorer un fichier 12](#_Toc447896442)

[1.3.5 Comparer deux versions d’un fichier 12](#_Toc447896443)

[1.3.6 Commit le projet dans le répertoire local 13](#_Toc447896444)

[1.3.7 Pousser le projet sur le serveur 14](#_Toc447896445)

[1.3.8 Résoudre des conflits 16](#_Toc447896446)

[2 Détail du projet 19](#_Toc447896447)

[2.1 Back-end 19](#_Toc447896448)

[2.1.1 Model 19](#_Toc447896449)

[2.1.2 Persistence 21](#_Toc447896450)

[2.1.3 Service 22](#_Toc447896451)

[2.2 Front-end 23](#_Toc447896452)

[2.2.1 Structure globale 23](#_Toc447896453)

[2.2.2 index.html 23](#_Toc447896454)

[2.2.3 app.js 23](#_Toc447896455)

[2.2.4 Structure Contrôleur – Template 23](#_Toc447896456)

[2.2.5 Librairie javascript 25](#_Toc447896457)

[3 Règle de développement 34](#_Toc447896458)

[3.1 Convention codage 34](#_Toc447896459)

[3.1.1 Commentaire 34](#_Toc447896460)

[3.2 Convention de nommage 36](#_Toc447896461)

[3.2.1 Code JAVA 36](#_Toc447896462)

[3.2.2 Code HTML et JavaScript 37](#_Toc447896463)

[3.2.3 La base de données 37](#_Toc447896464)

[4 Bonnes pratiques de développement 38](#_Toc447896465)

[4.1 Bonnes pratiques de programmation 38](#_Toc447896466)

[4.1.1 Les classes Model 38](#_Toc447896467)

[4.1.2 Programmer un Service 38](#_Toc447896468)

[4.1.3 Envoyer et récupérer une requête AJAX au serveur 38](#_Toc447896469)

[4.1.4 Construit une partie de la page en utilisant un template et les données reçus 41](#_Toc447896470)

[4.2 Bonnes pratiques d’utilisation de la base de données 41](#_Toc447896471)

[4.2.1 Recommandations d'implémentation du MDD de Prodige pour une BDD Oracle : 41](#_Toc447896472)

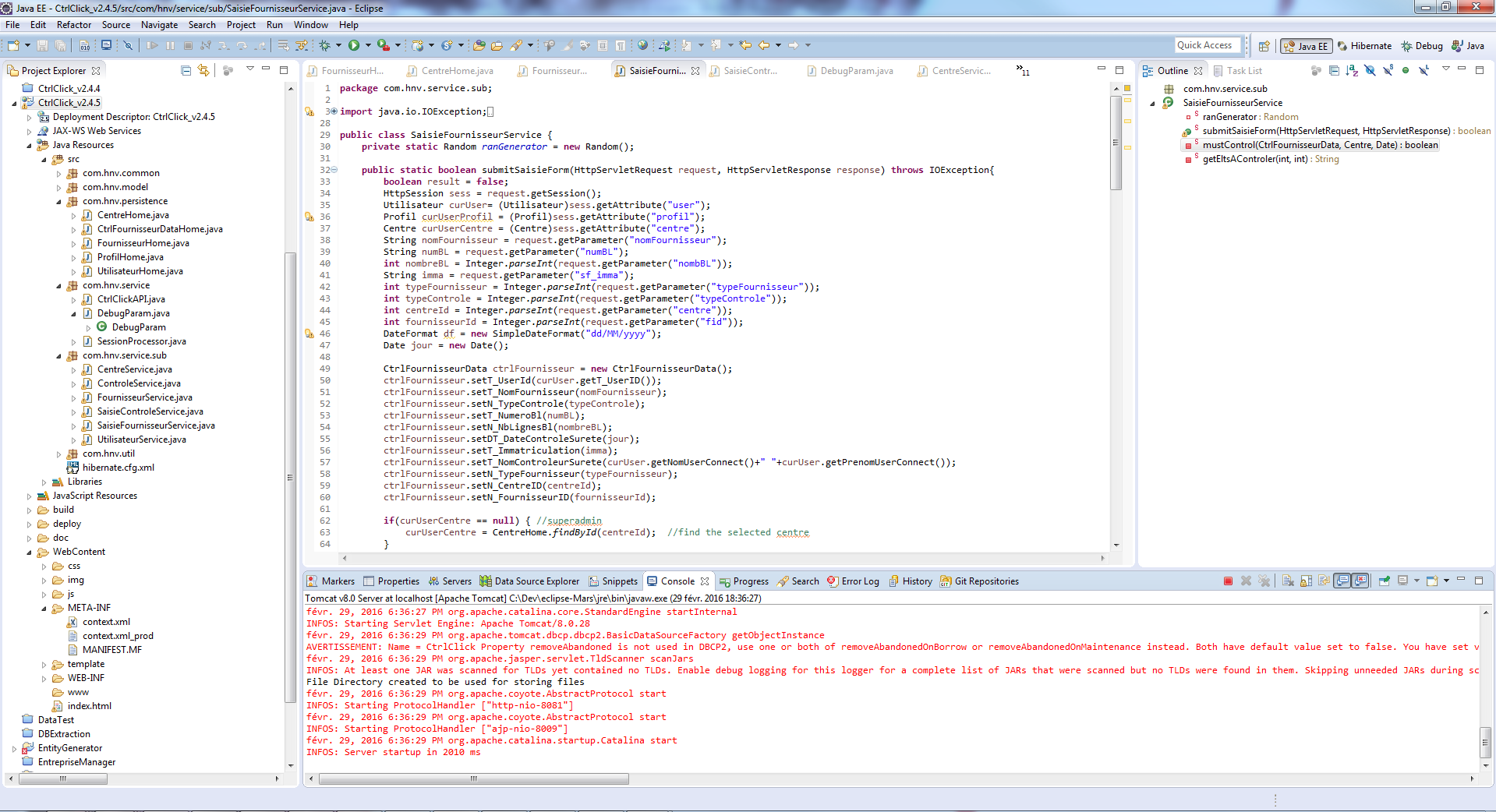
[4.2.2 Gestion des accès aux différents environnements : 42](#_Toc447896473)

# Guide d’utilisation

## Eclipse

Version utilisé : Eclipse Java EE IDE for Web Developers Mars.1 (4.5.1)

L’écran de développement d’Eclipse :



1. La fenêtre à gauche affiche la structure du projet dont l’arbre des fichiers java, javascript, …
2. La fenêtre au centre affiche le contenu du fichier actuellement ouvert, on peut aussi éditer ce contenu dans cette fenêtre.
3. La fenêtre en bas affiche les logs de l’application

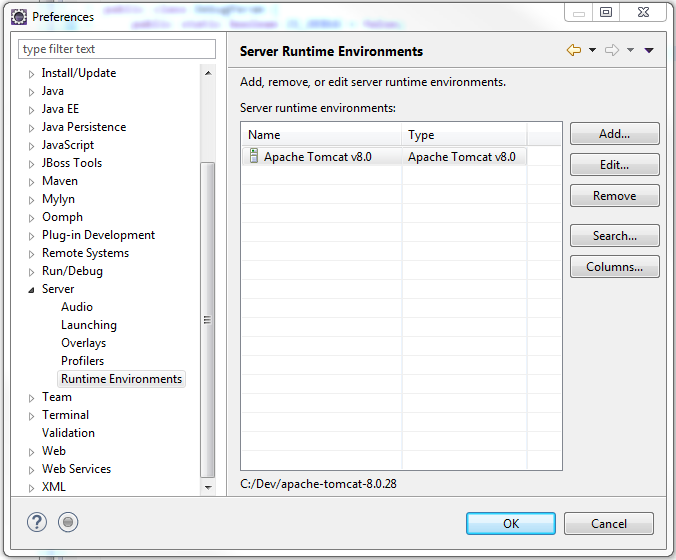
## Tomcat

Version utilisé : Tomcat 8

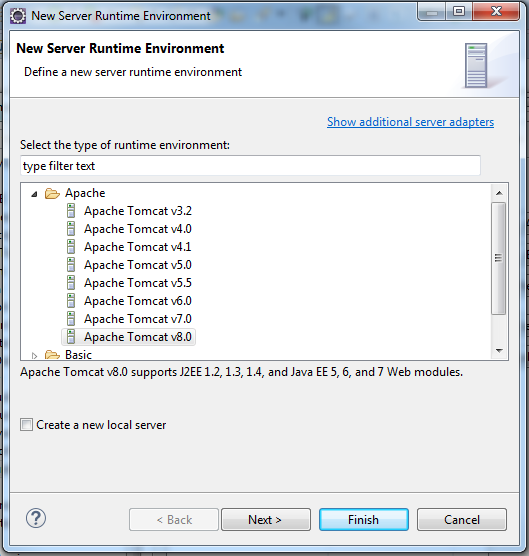
### Ajouter un serveur local à Eclipse

Ouvrir la fenêtre de configuration d’Eclipse : Window -> Preferences

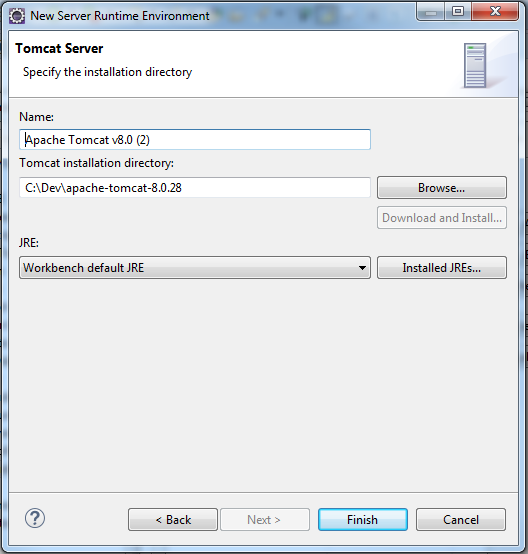
Puis choisir Server -> Runtime Environment



Ajouter un nouveau serveur en cliquant sur Add



Choisir Apache Tomcat v8.0 et cliquer sur Next

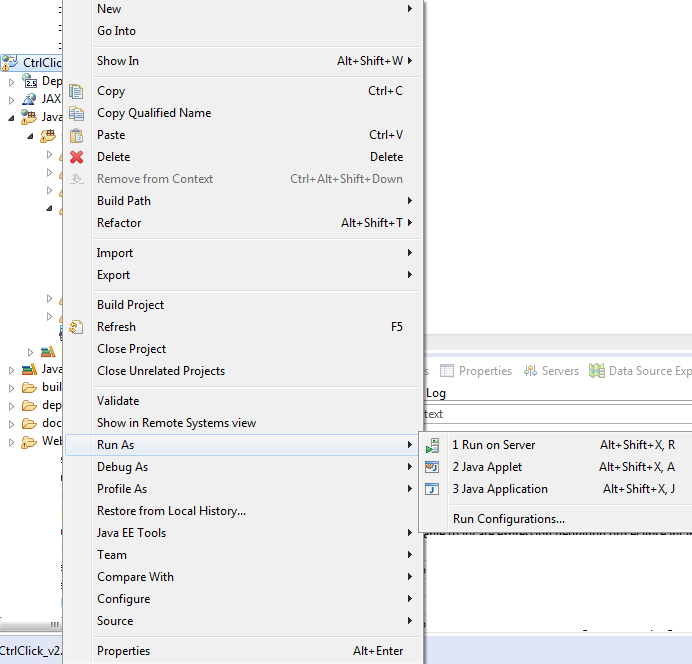


Entrer le nom du serveur et le chemin d’accès au répertoire d’installation du serveur Tomcat 8, par exemple : C:\Dev\apache-tomcat-8.0.xx. Enfin cliquer sur Finish pour terminer l’ajout d’un nouveau serveur.

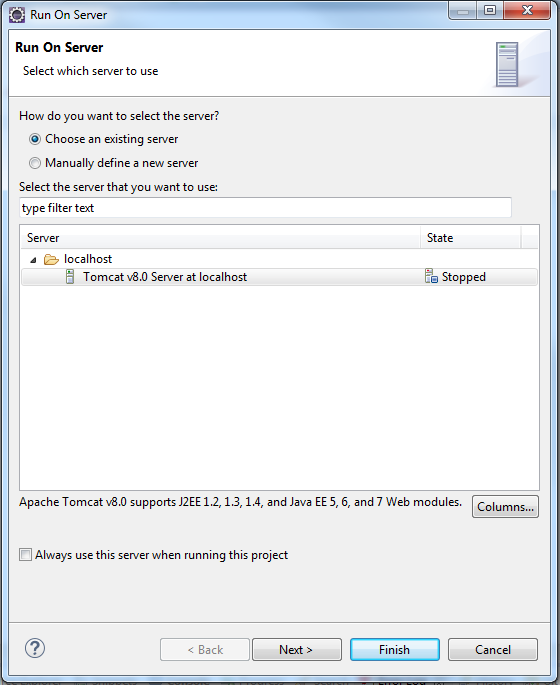
### Déployer un projet sur le serveur local

Un projet d’application web doit être déployé sur un serveur local avant d’être testé.

Cliquer droit sur le projet et choisir Run as -> Run on server



Puis choisir un serveur local (ajouté auparavant) pour déployer le projet.

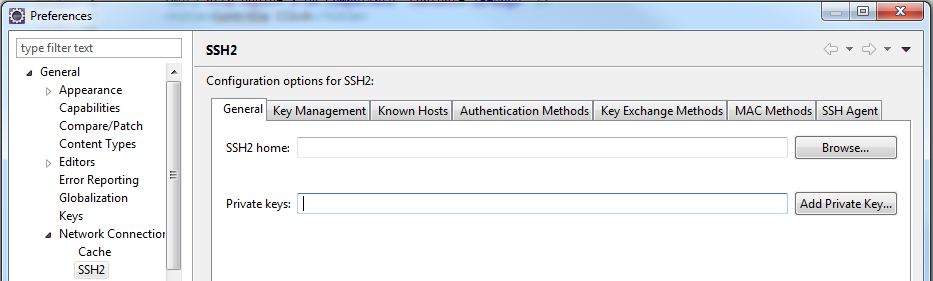


## GIT

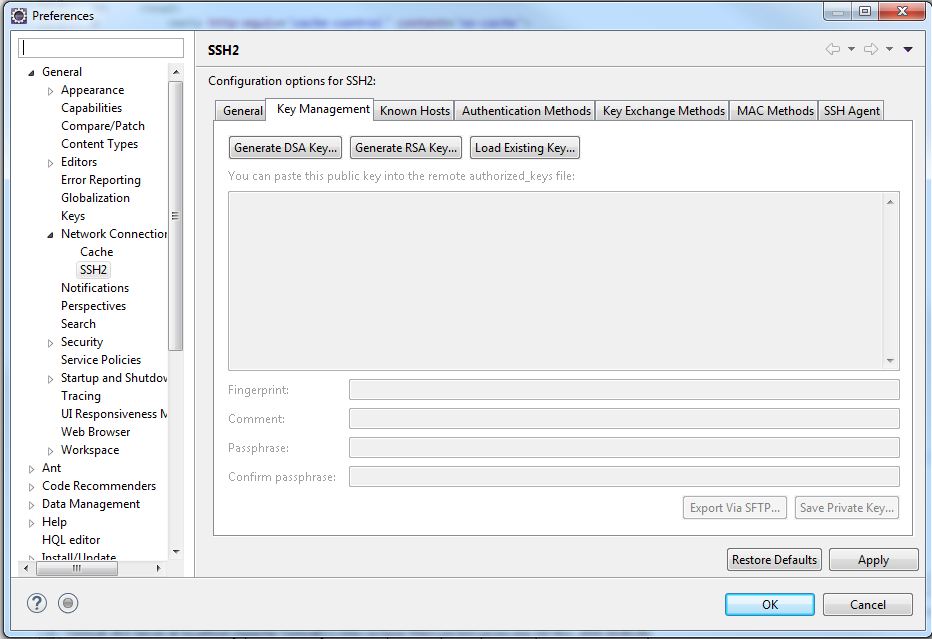
On utilise le plugin EGit sur Eclipse en tant que client GIT sur le poste de travail.

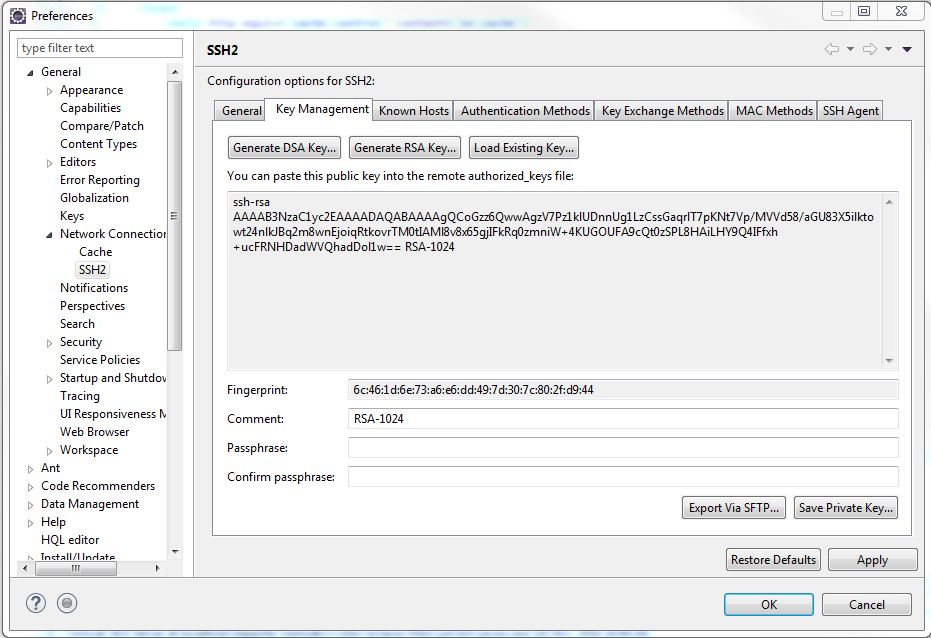
### Générer une clé publique pour connecter au serveur GIT

Dans le l’écran Preferences de l’Eclipse (Window -> Preference)

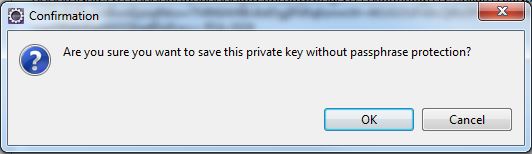


Générer une clé publique





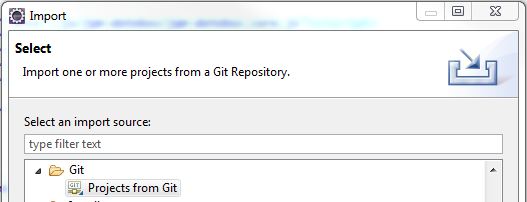
Sauvegarder la clé privée sur l’ordinateur



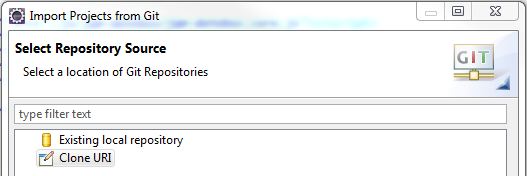
Envoyer la clé publique à l’administrateur du serveur pour qu’il puisse ajouter la clé dans la liste des clés sur le serveur.

### Importer un projet depuis le serveur GIT

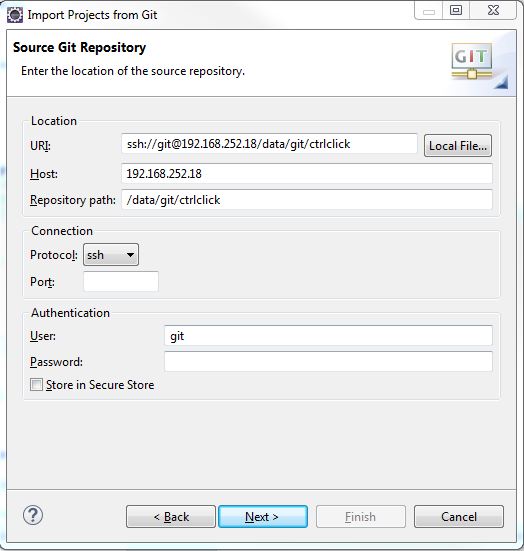
Accéder à l’écran d’importer de Eclipse (File -> Import…)



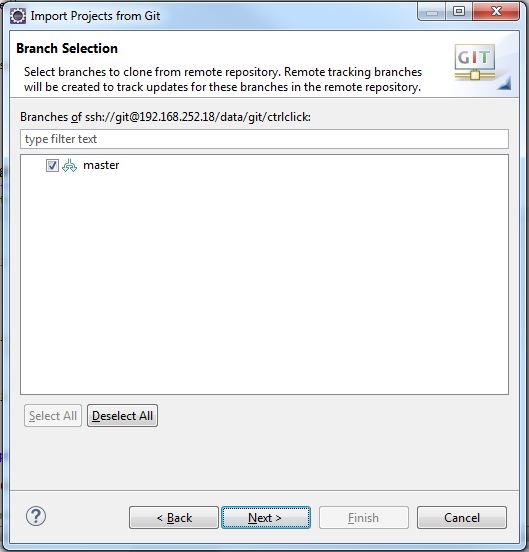
Choisir le type de projet : Git/Projets from Git. Puis cliquer sur Next



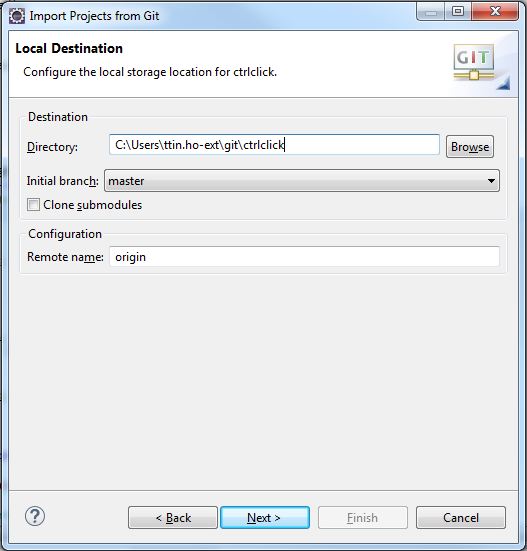
Choisir le type de répertoire : Clone URI. Puis cliquer sur Next



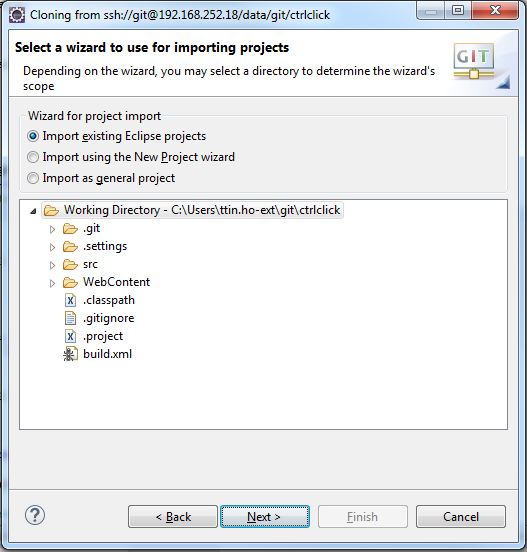
1. Copier l’URI du répertoire sur le serveur dans le champ URI :
2. Choisir le protocole de connexion : ssh
3. L’utilisateur pour authentification est : git
4. Laisser le champ mot de passe vide.
5. Choisir Store in Secure Store et cliquer sur Next



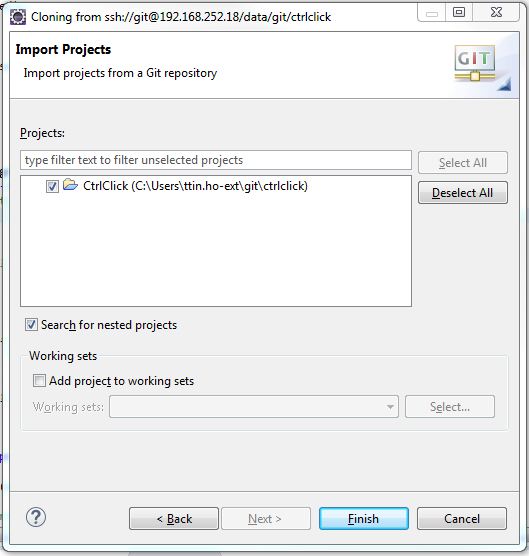
Choisir une branche : exemple : master. Cliquer sur Next



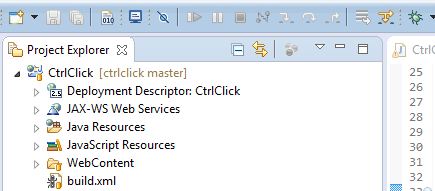
Cliquer sur Next, EGit va commencer à copier le projet de puis le serveur GIT



Choisir Import existing Eclipse projects pour importer le projet dans le répertoire copié depuis le serveur dans l’espace de travail d’Eclipse. Cliquer sur Next

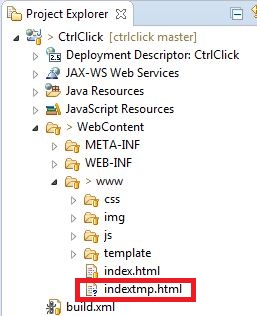


Cliquer sur Finish pour finaliser l’importation d’un projet sur le serveur GIT dans l’espace de travail de l’Eclipse. Le projet apparait dans l’écran Project Explorer de l’Eclipse.

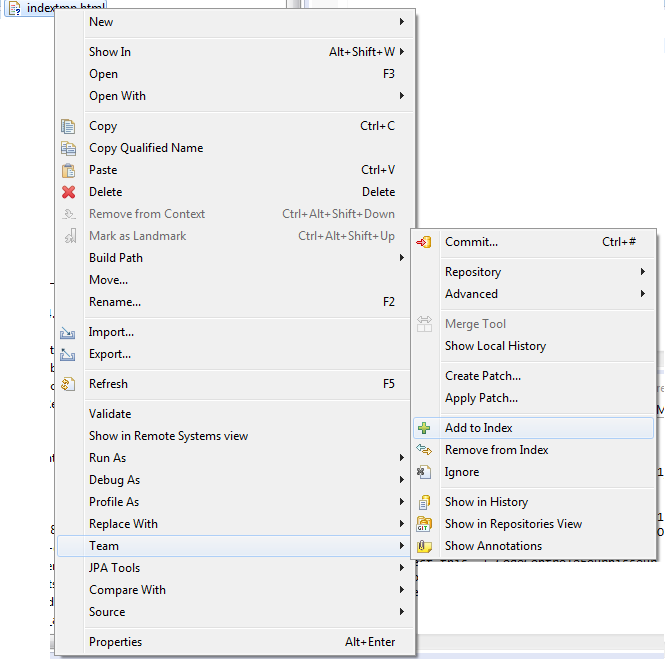


### Tracer les modifications d’un nouveau fichier

Après d’ajouter un nouveau fichier qui n’existe pas dans le projet copié depuis le serveur, ce fichier va être marqué par un point d’interrogation :



Cela signifie que le nouveau fichier n’est pas tracé par la gestion de version de Git. Pour tracer ce fichier, il faut indexer ce fichier : Cliquer droit sur le nouveau fichier et choisir Team -> Add to Index



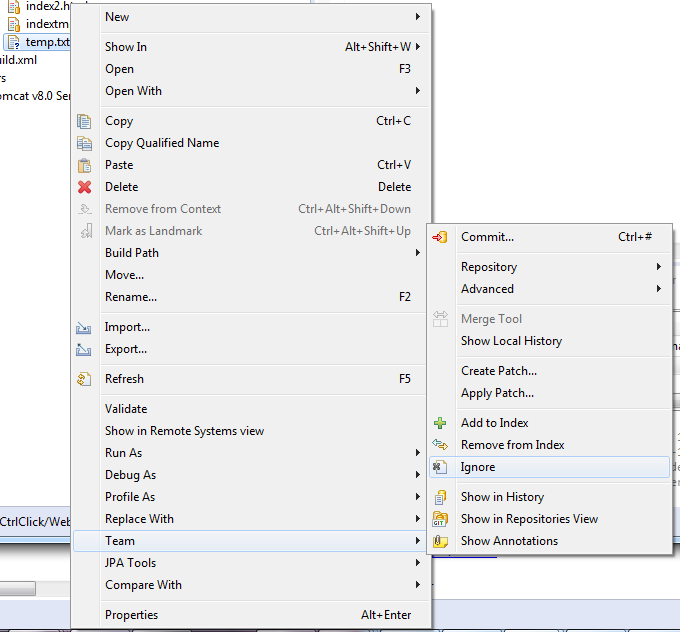
Le nouveau fichier est maintenant marqué par un +, cela signifie qu’il est un nouveau fichier et qu’il est tracé.



### Ignorer un fichier

Si un fichier ne contribue pas dans le projet, ou il n’est pas nécessaire de tracer les modifications sur ce fichier, par exemple un fichier temporaire, fichier de log ou un fichier généré automatiquement pendant l’exécution de l’application, on peut marquer ce fichier comme ignoré.

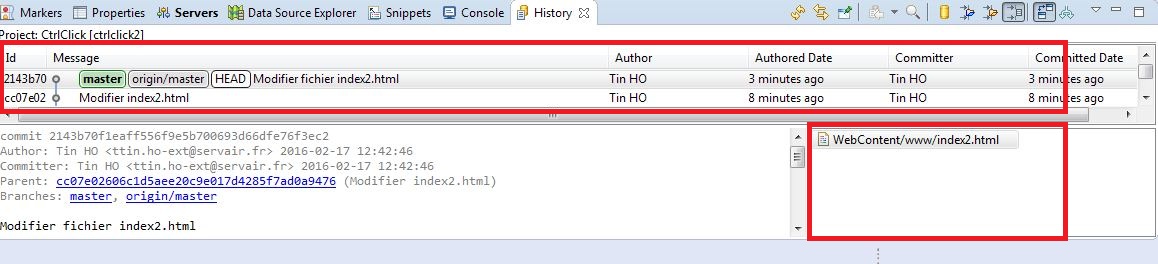
Cliquer droit sur le fichier et choisir Team -> Ignore



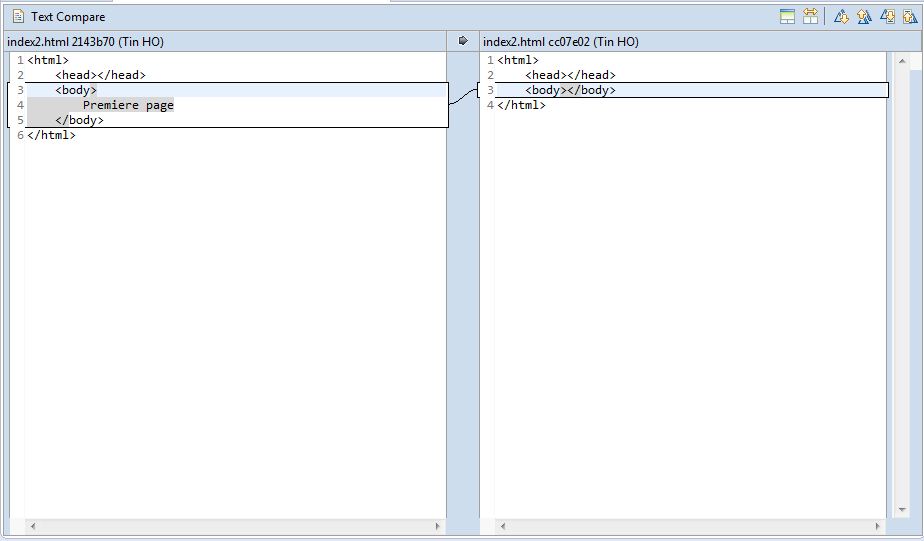
### Comparer deux versions d’un fichier

Une fois les modifications d’un fichier est commis, ces modification sont tracées, et on peut comparer la nouvelle version avec l’ancienne version.

Activer le mode de comparaison en cliquant sur l’icône Compare Mode de History View. 

Puis sélectionner une version dans la liste des commises. Et double cliquer sur le fichier qu’on veut comparer avec l’ancienne version dans la liste des ressources. 

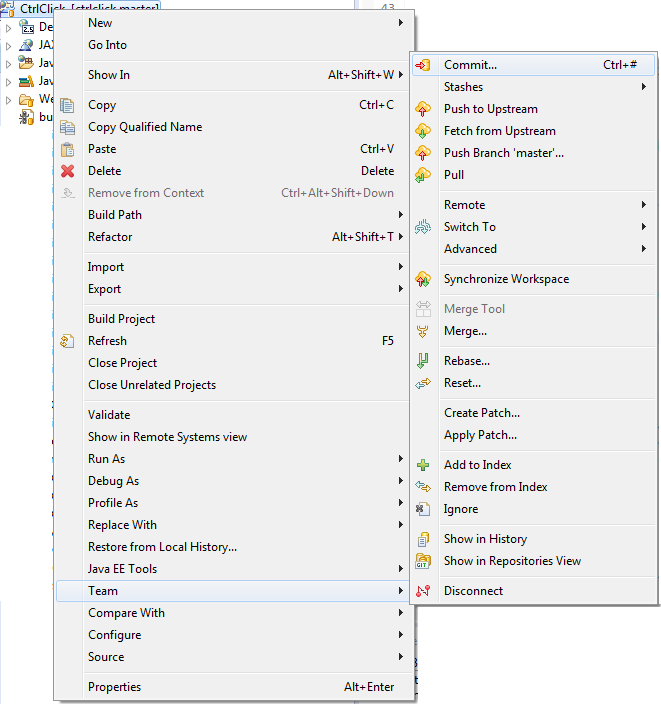
Ca va ouvrir l’écran de comparaison, dont la version à gauche est la nouvelle version, et la version à droite est l’ancienne version.



### Commit le projet dans le répertoire local

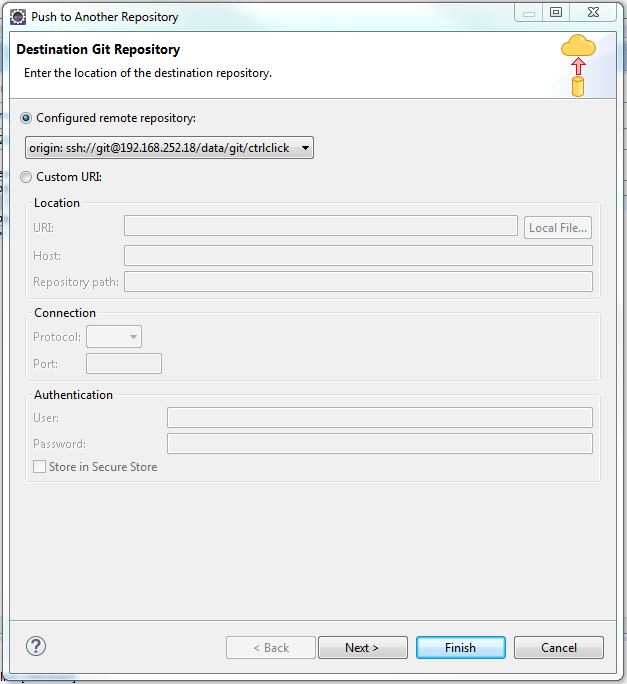
Avant de pousser les modifications sur le projet sur le serveur, il faut commettre ces modifications sur le répertoire local avant.

Cliquer droit sur le nom du projet et choisir Team -> Commit ou utiliser le raccourci Ctrl+# pour commettre les modifications dans le répertoire local.

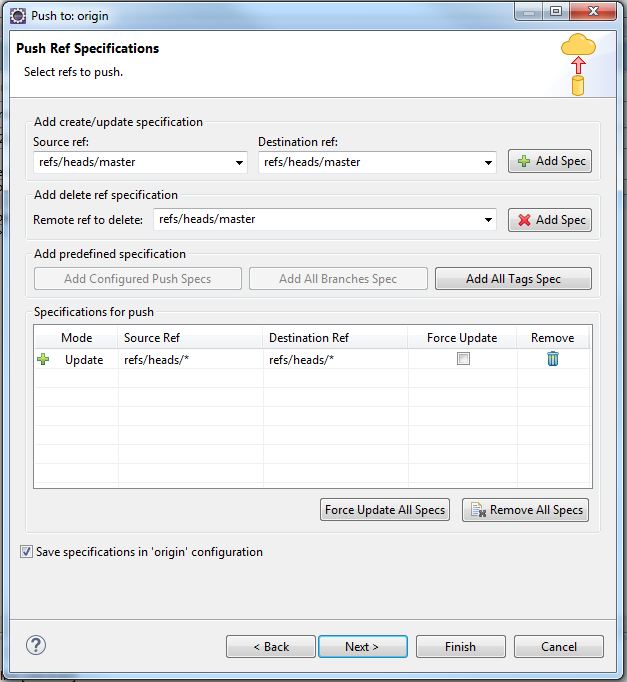


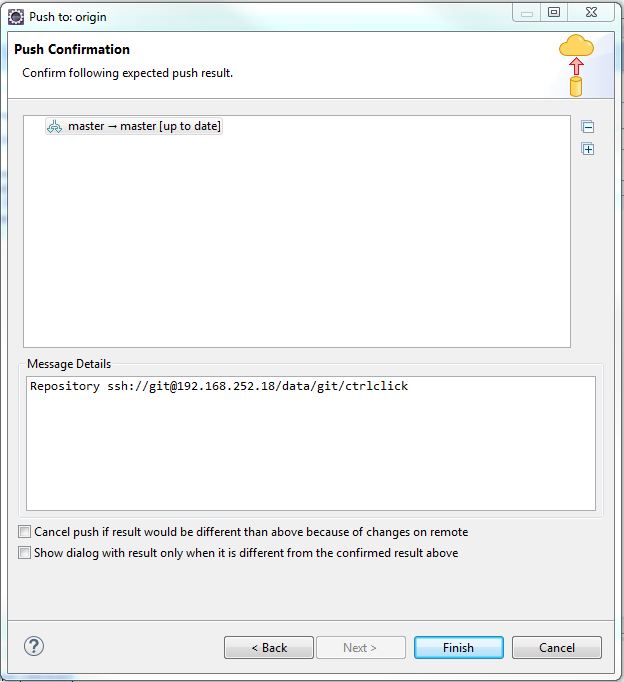
### Pousser le projet sur le serveur

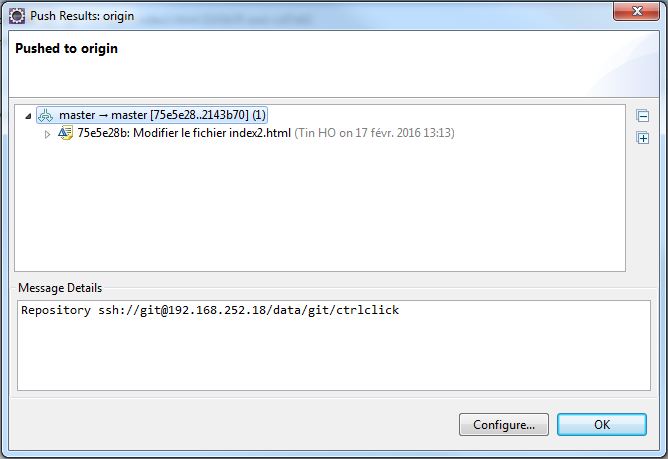
Pour pousser les modifications qu’on a commis dans le répertoire local sur le serveur, il faut cliquer droit sur le projet et choisir Team -> Remote -> Push…

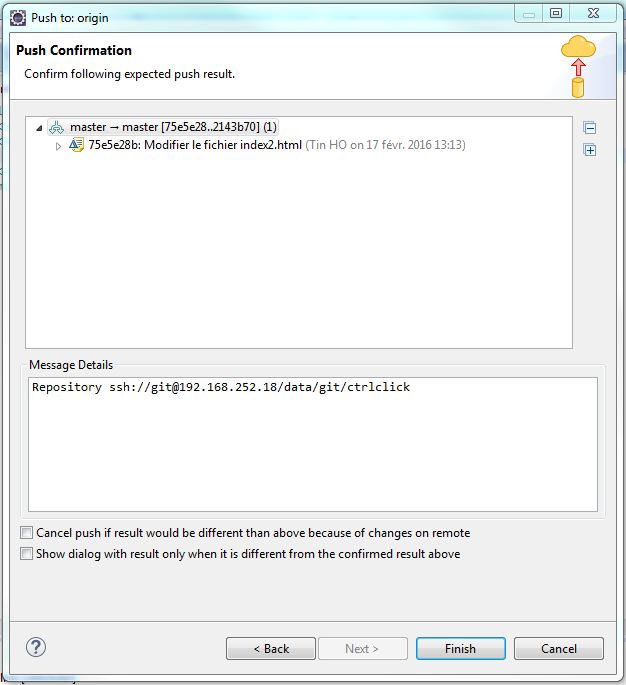
Choisir un répertoire à distance configuré.

Choisir le Source ref, le Destination ref et le Remote ref to delete. Puis appuyer sur Add All Branches Spec. Cliquer sur Next pour aller à l’écran Confirmation de pousse.



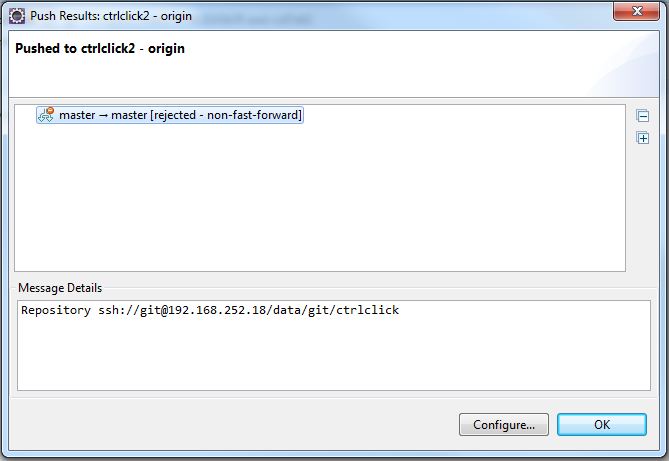
Enfin cliquer sur Finish pour pousser les modifications sur le serveur.



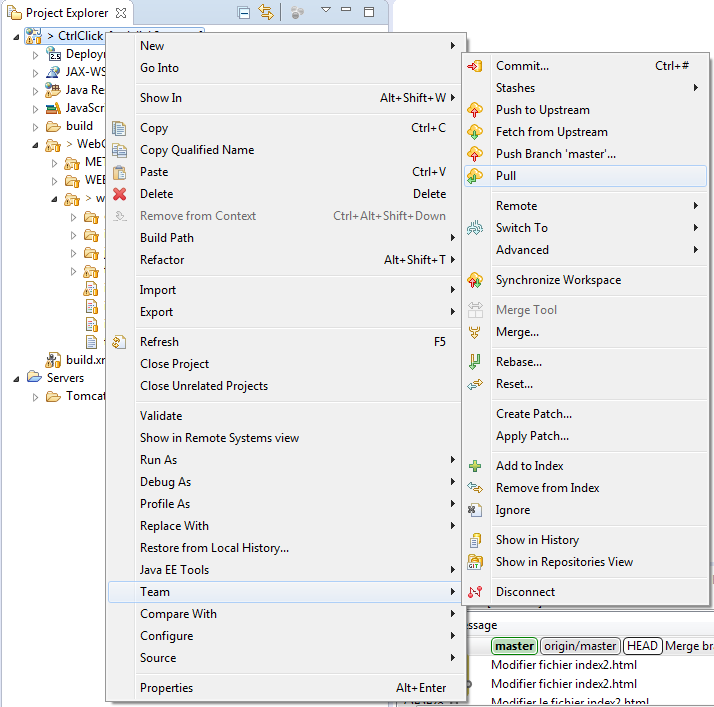


### Résoudre des conflits

Si il y a un conflit entre la version poussé et la dernière version sur le serveur, le message de résultat de pousser va indiquer [rejected – non-fast-foward]

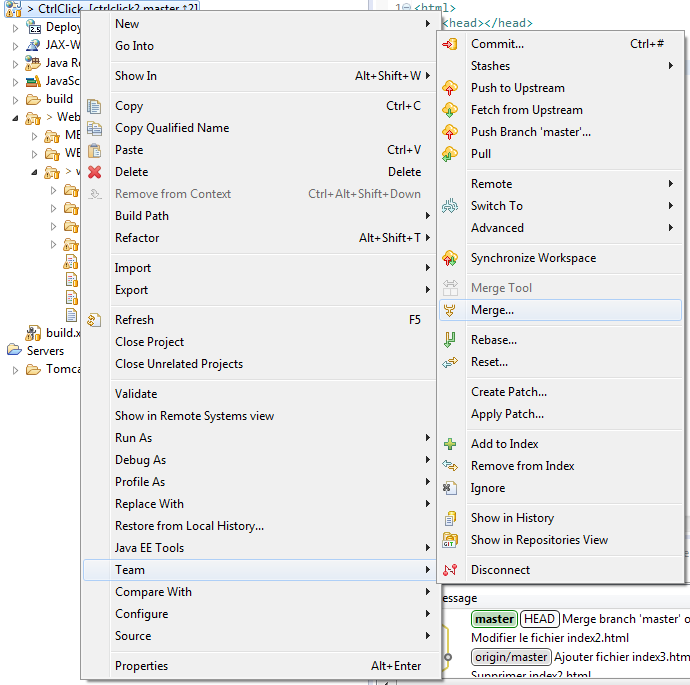


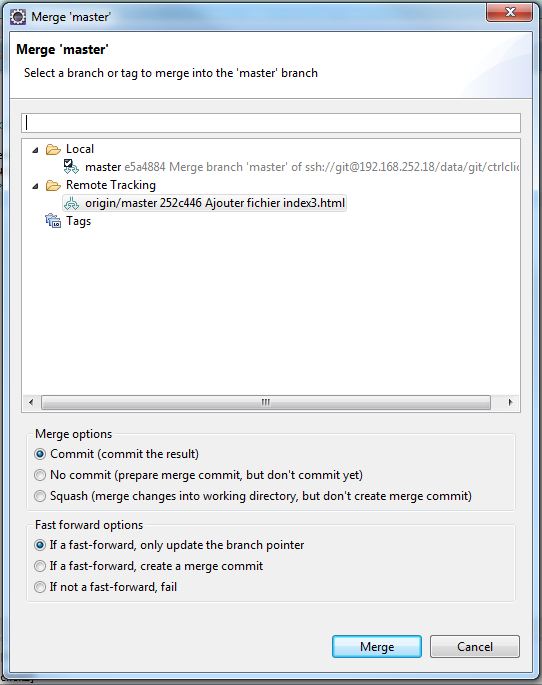
Dans ce cas-là, il faut tirer la dernière version depuis le serveur en choisissant Team -> Pull. Puis il faut fusionner les deux versions ou résoudre des conflits.

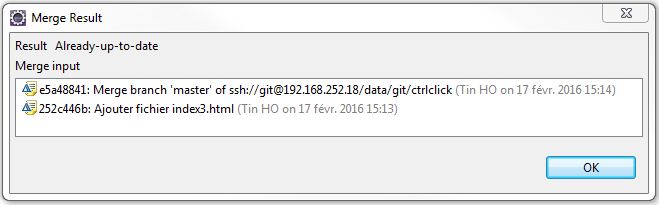


#### Fusionner deux versions

Après tirer la dernière version depuis le serveur, et il n’y a pas de conflit, on peut fusionner les deux versions en choisissant Team -> Merge



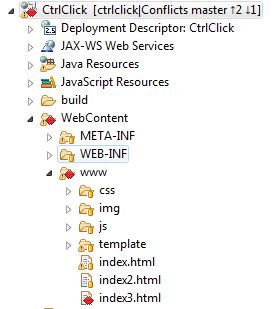




Puis on peut Commettre et Pousser sur le serveur.

#### Résoudre le conflit sur le même fichier

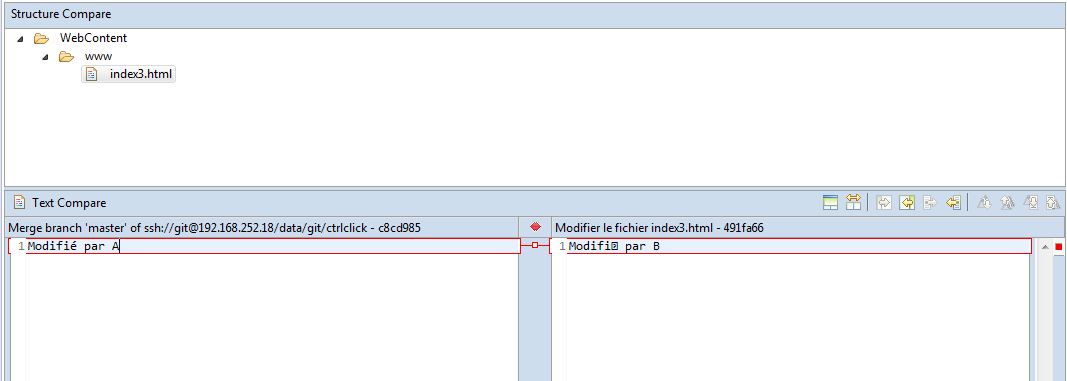
S’il y a un conflit dans un fichier modifié par plusieurs personnes, le fichier va être marqué par une icône rouge :



Et dans le fichier, EGit a maqué aussi les modifications en conflits



Puis on peut utiliser l’outil de fusionner de EGit pour comparer les deux versions



Ensuite on résoudre les conflits en fusionner les modifications. Après de résoudre les conflits, il faut cliquer droit sur le fichier en conflit et choisir Team -> Add index pour dire que le conflit est résolu. Enfin on peut Commettre et Pousser sur le serveur.

# Détail du projet

## Back-end

Notre back-end est divisé en trois parties principales

1. Model : où se trouve tous les classes qui correspondent avec les tables dans la base de données.
2. Persistance : Ce package qui gère tous les actions concernés la base de données comme ajouter, mettre à jour, supprimer et rechercher.
3. Service : Le package où se trouvent les services de l’application fourni à l’utilisateur

### Model

Création une nouvelle classe model :

Par exemple, on va créer une classe User

#### Mapper avec la base de données utilisant Hibernate

Une classe model dois être correspondre avec une et seulement une table dans la base de données. On peut déclarer la table à qui la classe correspond avec l’annotation de Hibernate suivant :

@Entity

@Table(name = "Nom de la table", schema = "schéma de la base")

**public** **class** User {

Dont :

1. @Entity : sert à déclarer que cette classe est un entité dans la base de données.
2. @Table(name = "Nom de la table", schema = "schéma de la base") : pour spécifier le schéma de la base et nom de la table avec qui la classe est mappée.

Ces déclarations doivent être impérativement au-dessus de la déclaration de la classe.

Puis lister tous les noms de colonne de la table en utilisant les constants :

**public** **static** **final** String ***COL\_T\_USE\_ID*** = "T\_UserID";

**public** **static** **final** String ***COL\_T\_USE\_PASS*** = "T\_UserPassword";

**public** **static** **final** String ***COL\_I\_USE\_PROFIL\_ID*** = "N\_IdProfil";

Ensuite déclarer les attributs en utilisant les annotations de Hibernate. Les noms des attributs doit être identiques avec les noms des colonnes de la table. Les annotations doivent être impérativement au-dessus de la déclaration de chaque attribut.

@Id

@Column(name=***COL\_T\_USE\_ID***)

**private** String T\_UserID;

@Column(name=***COL\_T\_USE\_PASS***)

**private** String T\_UserPassword;

@Column(name="T\_UserName")

**private** String T\_UserName;

@Column(name=***COL\_I\_USE\_PROFIL\_ID***)

**private** **int** I\_IdProfil;

Dont :

1. @Id : pour déclarer que cette colonne est la clé primaire de la table.
2. @Column(name= ***COL\_T\_USE\_ID***) : sert à définir à quel colonne le variable est correspond.
3. @GeneratedValue(strategy=GenerationType.***AUTO***) : utiliser pour déclarer que cette colonne est augmenter automatiquement par la base de données.

#### Gestion des relations entre les tables

Des relations entre les tables de la base de données, dont : One to One, One to Many ou Many to Many, doivent être gérer manuellement dans les classes models. Par exemple : La classe User qu’on vient de créer précédemment a une clé étranger I\_IdProfil qui est une relation One to One avec la table Profil.

Premièrement, on va créer un attribut de type Profile (une autre classe model qui correspond avec la table Profil dans la base de donées). Et ajouter l’annotation @Transient au-dessus de cette attribut.

@Transient

**private** Profil profil;

Deuxièmement, rédiger des fonctions de récupérer et de modifier cette attribut.

#### Manipuler les attributs

Les fonctions « getter » et « setter » génériques sont fourni pour récupérer ou modifier les attributs dans la classe modèle.

On peut utiliser les fonctions get et set pour récupérer ou mettre à jour un attribut.

**public** **final** Object get(String attrName) {

Cette fonction qui prend comme paramètre le nom de l’attribut qu’on veut récupérer sa valeur. Exemple :

u.get(User.***COL\_T\_USE\_ID***);

Cet appel de la fonction *get* récupère la valeur de l’attribut ***COL\_T\_USE\_ID*** de *u* (un instant de la classe User). Cette fonction retourne un objet de type Object, il faut le transformer en type de l’attribut.

**public** **final** **void** set(String attrName, Object value) **throws** Exception {

Cette fonction *set* met à jour l’attribut dont son nom est attrName par la nouvelle valeur value. Exemple :

u.set(User.***COL\_T\_USE\_ID***,"abc");

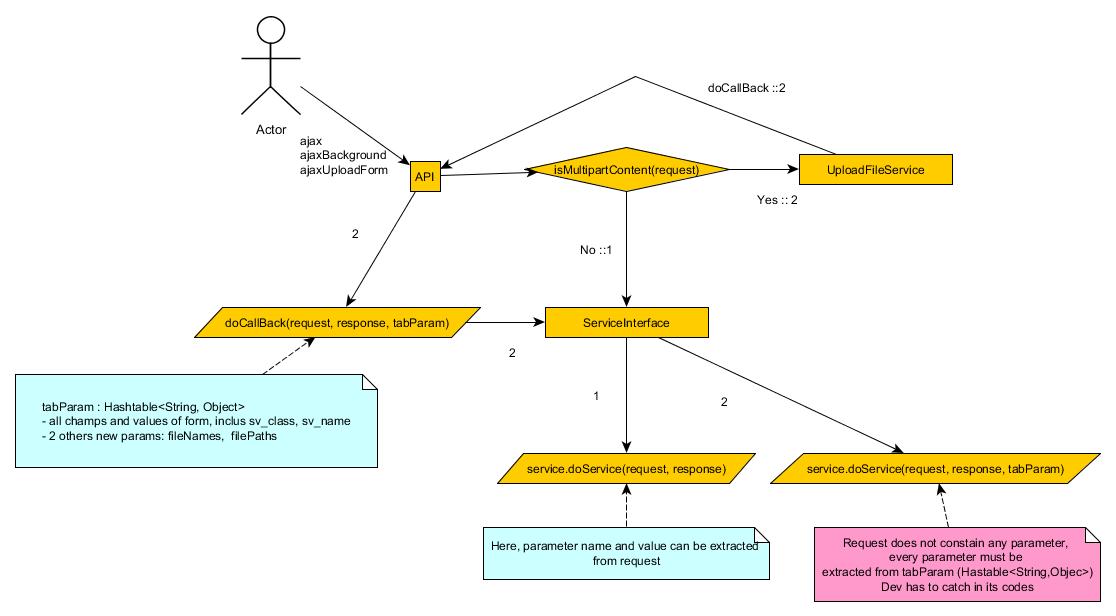
Cet appel de la fonction set met à jour l’attribut ***COL\_T\_USE\_ID*** de *u* (un instant de la classe User) par la valeur"abc".

### Persistence

Persistence est le package qui affecte tous les actions sur la base de données comme ajouter, modifier, supprimer et chercher. Tous les classes Model ont un objet DAO, qui fournit les fonctions de manipuler la base de données (Ajouter, Mettre à jour, Supprimer, Chercher dans la base de données).

Les fonctions fournis sont :

### Service



#### Service central

Le service central (API) est le seul point d’accès de l’application au côté serveur. Ici avant de traiter la requête de l’utilisateur, le service central vérifie l’authentification et l’autorisation de l’utilisateur sur le service demandé dans la requête. Puis il va rediriger la requête au service correspondant.

Le service central est le seul servlet qui a un lien d’accessible de l’application.

#### Service

Les services sont les fonctionnalités qu’application fournit aux utilisateurs et ou aux autres applications en tant que l’interface de communication inter-application.

Les services peuvent être regroupés dans les classes différentes.

Chaque service a un code de service unique au sein de l’application et doit être déclaré dans le service central.

Un service est une méthode de la classe qui prend une HttpServletRequest request, et une HttpServletResponse response de la requête envoyé au serveur (et qui sont passés par le Service central) comme paramètre d’entrer.

Le service va récupérer les paramètres attachés dans request, les traiter, récupérer les données de la base de données, et envoyer la réponse au client via response.

## Front-end

### Structure globale

La structure globale de l’application au côté client est :

* **WebContent**
  + **www** : répertoire de tous les ressources
  + **css** : répertoire des fiches de styles
  + **img** : répertoire des images
  + **font** : répertoire des fiches de polices
  + **js** : répertoire des scripts et des templates
    - **app** : répertoire des scripts et des templates de l’application
      * **common** : répertoire des scripts commun qui peuvent être utilisés par des autres scripts.
      * **controler** : répertoire des scripts qui contrôlent le construction des templates HTML
        + RouteController.js
      * **template** : répertoire des template HTML
    - **lib** : répertoire des bibliothèques utilisées dans l’application
  + app.js
  + require.js
* index.html

### index.html

index.html est le seul point d’entrer de l’application au côté client. Il faut impérativement aller à index.html pour accéder à l’application. Donc sur index.html, il faut défini tous les paramètres et des fichiers nécessaire. Il contient aussi les emplacements de base de l’affichage, par exemple : où se trouve l’entête, où se trouve le menu principal, où se trouve le contenu de la page, … Les contenus de l’application doivent être mis dans ces emplacements correspondants.

### app.js

Le premier script qui est téléchargé et exécuté quand un utilisateur ouvre la page index.html de l’application. *app.js* défini les configurations de RequireJS (les scripts directories) et des premiers actions de l’application (i.e charger la page de connexion).

### Structure Contrôleur – Template

Structure Contrôleur – Template est défini pour moduler la gestion des templates en utilisant la gestion des modules de RequireJS. Chaque contrôleur est un module qui gère le téléchargement de ses fichiers templates et le remplissage les données dans ces templates en utilisant Handlebars.

#### Contrôleur

Un contrôleur est un module dans le notion de RequireJS, il est défini par la fonction define() de RequireJS.

define(["dependencie1","dependencie2"],

**function**(param1, param2){

**return** object;

}

);

Le premier paramètre ["dependencie1","dependencie2"] est un tableau des dépendances de ce module. Le deuxième paramètre **function**(param1, param2) est la fonction qui définit le module. Cette fonction doit retourner un objet qui définit ce module. Elle n’est exécuté que tous les dépendances sont complètement téléchargés. Les dépendances seront passées comme des paramètres de la fonction de définition, listé dans le même ordre que l'ordre dans le tableau des dépendances. C’est-à-dire param1 correspond avec "dependencie1" et param2 correspond avec "dependencie2".

Pour chaque module, déclarer un variable qui représente le contrôleur, puis définir les attributs et les fonctionnalités dans cette variable, et enfin retourner ce variable comme la définition du module.

define(["dependencie1","dependencie2"],

**function**(param1, param2){

**var** sampleController = **function**(controllerParam) {

//Do something

}

**return** sampleController;

}

);

Des fonctions dont tous les contrôleurs doivent définir sont :

**this**.show = **function**(){

Cette fonction est pour l’objectif d’afficher la page gérée par ce contrôleur. Elle va d’abord vérifier le langage actuellement choisi par l’utilisateur et ensuite exécute la fonction buidPage().

**var** buildPage = **function**() {

Cette fonction sert à construire la page à partir du/des template(s) et les données reçu du serveur.

**var** bindingEventsPage = **function**() {

Cette fonction sert à ajouter les événements (des actions) à la page construit. Par exemple ajouter un événement clique à un bouton qui va exécuter une fonction quand l’utilisateur clique sur ce bouton.

#### Template

Les templates sont des modèles d’une partie de la page HTML affiché au côté client.

#### RouteController

RouteController est le contrôleur central, qui gère accès à tous les autres contrôleurs de l’application. Pour qu’un contrôleur puisse être accédé, il faut le déclarer dans RouteController et lui donner un chemin d’accès.

**this**.init = **function**() {

**this**.router = **new** PathParser();

**this**.router.add("/home/:varName", **function** () {

**var** sampleController = **new** SampleController(**this**.varName);

sampleController.showHomePage();

});

};

**this**.run = **function** (path) {

**this**.router.run(path);

};

Pour exécuter une fonction qui est liée à un chemin déclaré dans RouteController, il faut juste exécuter la fonction :

RouteController.run(<le chemin>);

### Librairie javascript

#### JQuery

jQuery est une bibliothèque JavaScript rapide, petit, et riche en fonctionnalités. Il fait des choses comme le document HTML traversée et la manipulation, la gestion des événements, de l’animation, et de l'Ajax beaucoup plus simple avec une API facile à utiliser qui fonctionne à travers une multitude de navigateurs.

Dans notre application, jQuery est la bibliothèque de base qu’on utilise travers nos codes javascript. Il est important de maîtriser l’utilisation de jQuery.

Utilisation :

Manipulation le document HTML :

Pour chercher un/des élément(s) de la page HTML (des <div></div>, un <input/>,…). On peut utiliser la commande :

$("critère")

jQuery utilise les CSS3 sélecteurs comme critère de recherche. Par exemple : on peut utiliser #idachercher pour chercher les éléments HTML qui a attribut id = idachercher.

Puis on peut utiliser les fonctionnalités fournis par jQuery pour manipuler cet élément HTML. Par exemple : on peut utiliser la fonction .html() pour récupérer ou mettre à jour le contenu HTML du élément cherché.

$("critère").html("contenu");

$("critère").html();

Pour connaître plus sur l’utilisation de jQuery, il faut aller au site https://jquery.com/

#### Handlebars

Handlebars fournit la puissance nécessaire pour vous permettre de créer des modèles sémantiques efficacement sans frustration. Handlebars intègre les données dans un fichier template HTML dynamiquement. En utilisant Handlebars, on peut facilement remplir le template HTML par les données afin d’avoir un affichage complet.

Utilisation

Construire un template HTML avec Handlebars :

Les commandes de Handlebars est intégré dans le code HTML, mais identifié par « {{ » et « }} ».

<div class=*"entry"*>

<h1>{{title}}</h1>

<div class=*"body"*>

<h1>{{body.head}}</h1>

<p>{{body.content}}</p>

</div>

</div>

Les données pour remplir le template doit être un objet javascript, appelé le contexte. Par exmple :

**var** context = {

title:"First Handlebars template",

body:{

head:"Body header",

content:"Body content"

}

}

Après remplir le template par le contexte on va avoir :

<div class=*"entry"*>

<h1>Premier Handlebars template</h1>

<div class=*"body"*>

<h1>Body header</h1>

<p>Body content</p>

</div>

</div>

Handlebars fournit plusieurs « helper » afin de faciliter le remplissage d’un template :

Pour vérifier si une variable existe dans le contexte ou s’il n’est pas null ou undefined, on utilise

{{#if variable}}

{{else}}

{{/if}}

Ou si on veut parcourir tous les elements d’un variable array, on utilize

{{#each lstCtrl}}

{{/each}}

On peut aussi enregistrer notre helper s’il n’est pas encore fourni par Handlebars en exécutant la commande :

Handlebars.registerHelper('if\_eq', **function**(a, b, opts) {

**if**(a == b)

**return** opts.fn(**this**);

**else**

**return** opts.inverse(**this**);

});

Cette commande enregistre le helper if\_eq qui compare si a est égale à b ou non.

Pour avoir plus information sur Handlebars, veuillez consulter le site <http://handlebarsjs.com/>

Remplir le template par les données :

Pour remplir un template HTML qui est intégré ou non les commandes Handlebars, on utilise la librairie Template.js fourni. L’utilisation de cette librairie est spécifiée dans la partie Template.

Les commandes de Handlebars intégrées dans un template sont strictement liées à un contexte dont son format est fixe. C’est-à-dire, on ne peut pas remplir le template avec un autre contexte dont le format est différent. On reprend le premier exemple où le template est rempli par le contexte :

**var** context = {

title:"First Handlebars template",

body:{

head:"Body header",

content:"Body content"

}

}

Il ne peut pas remplir proprement avec ce contexte :

**var** context = {

title2:"First Handlebars template",

body2:{

head:"Body header"

}

}

#### JQuery Mobile

#### JQuery Form Validator

JQuery Form Validator est une bibliothèque qui sert à vérifier les champs de saisie selon les restrictions et afficher les messages d’erreur correspondants. Elle valide seulement les champs de saisie (<input>) qui sont dans un <form>. Ça veut dire qu’il faut mettre tous les champs de saisie dans un <form>.

Utilisation

Valider un champ de saisie :

Pour valider un champ de saisie, il faut y ajouter les attributs de Form Validator. Par exemple :

<input type=*"text"* id=*"login"* name=*"login"*

data-validation=*"required alphanumeric"*

data-validation-error-msg-required=*"Identifiant est obligatoire!"*

data-validation-error-msg-alphanumeric=*"Identifiant ne peut contenir que des chiffres ou des lettres!"*

data-validation-allowing=*"-\_"*/>

* Attribut data-validation détermine le type de validation appliqué, on peut appliquer plusieurs type de validation pour un champ de saisie. Les types de validations les plus utilisés sont :
* *required :* Le champ de saisie est obligatoire, il ne peut pas être vide.
* *alphanumeric :* Le champ de saisie ne peut contenir que des lettres et des chiffres.
* *length :* Le champ de saisie doit respecter un longueur défini.
* *number :* Le champ de saisie doit être un numéro.
* Attribut data-validation-error-msg indique le message par défaut qui va apparaître quand le champ de saisie n’est pas valide. On peut ajouter –type\_de\_validation à la fin de l’attribut data-validation-error-msg pour indiquer le message d’erreur uniquement pour ce type de validation. Par exemple data-validation-error-msg-required est le message d’erreur pour le type de validation required seulement.
* Attribut data-validation-allowing indique les valeurs exceptionnellement acceptées pour ce champ de saisie.

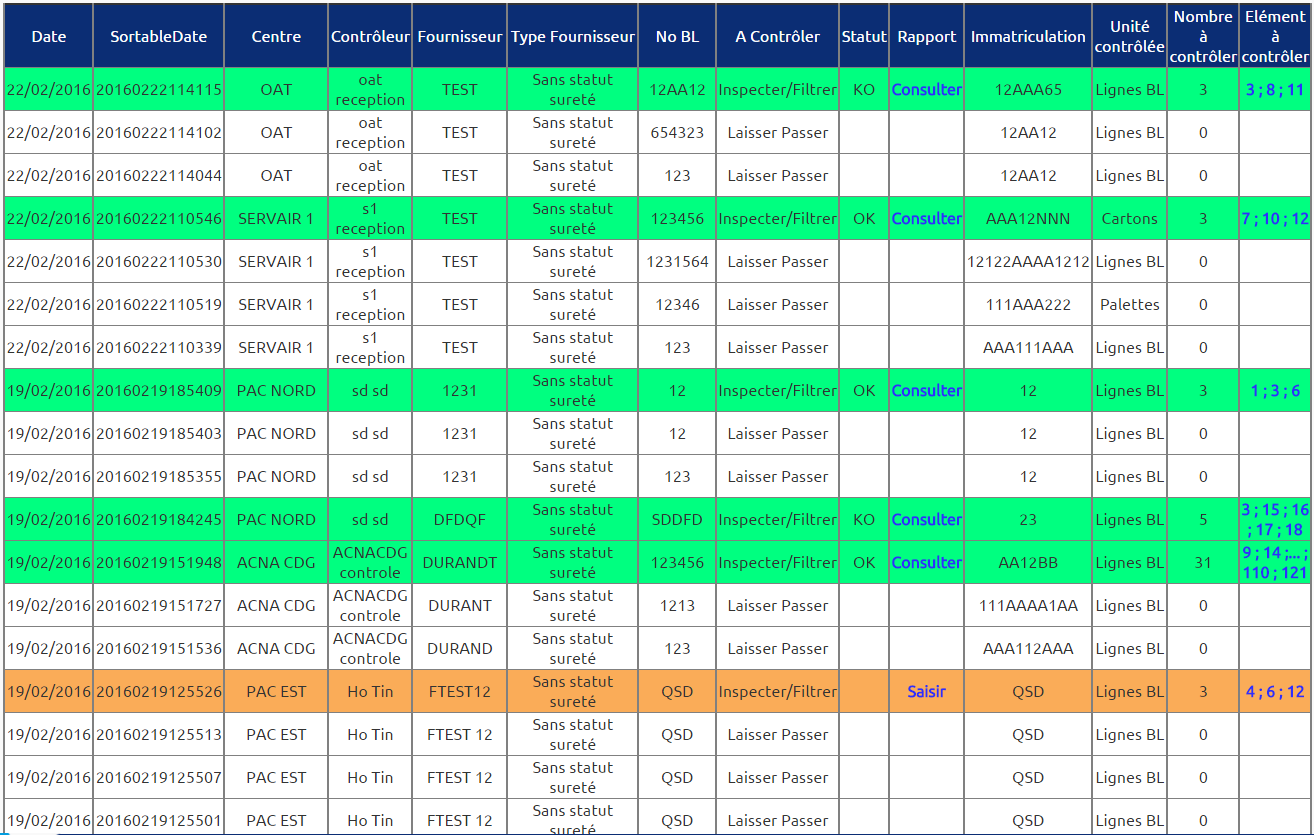
Pour avoir plus d’information sur l’utilisation de jQuery Form Validator, veuillez consulter le site <http://www.formvalidator.net/>

#### JQuery Datatable

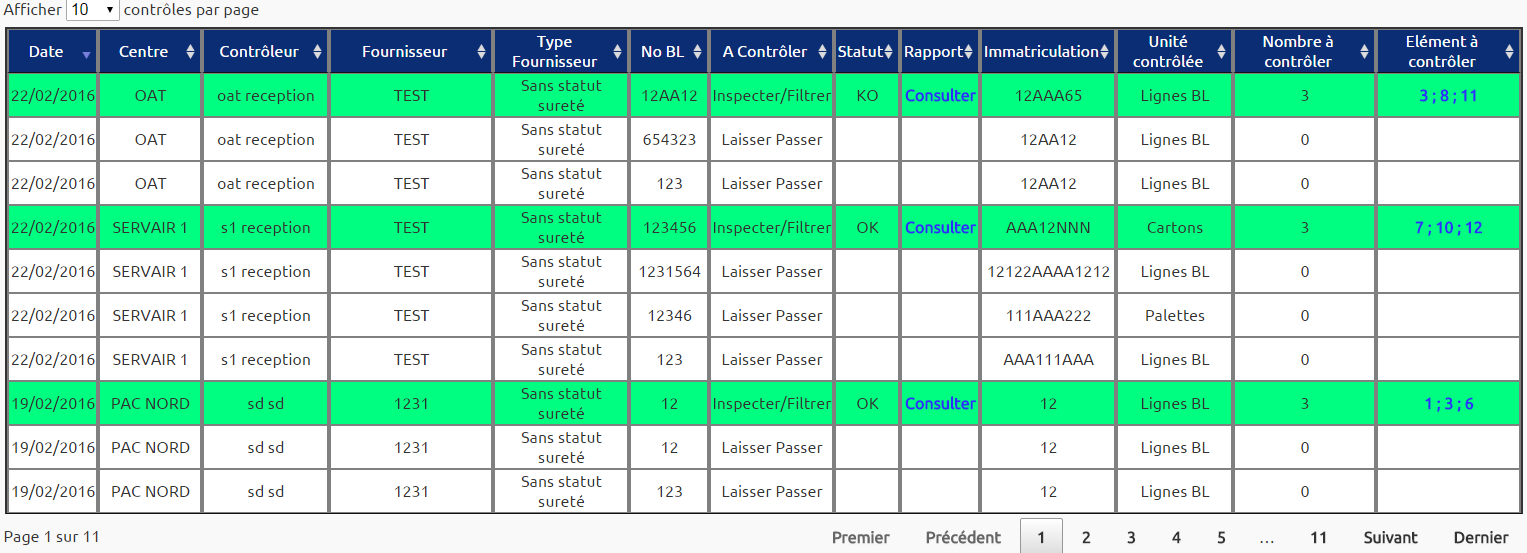
Cette bibliothèque est utilisée pour mettre en forme l’affichage d’un tableau de données. Elle permet aussi des fonctionnalités de trier par colonne et de la pagination.

Par exemple :

Avant d’utiliser jQueryDatatable



Après d’utiliser jQueryDatatable



Des traitements de triage et de changement de page sont exécutés au côté client, aucune requête envoyée au serveur. Et il peut traiter une grande quantité de données rapidement.

Utilisation

jQueryDatatable fourni plusieurs façon de l’utiliser, ici on utilise une seul façon qui est compatible avec notre façon de construire le page HTML en utilisant des Templates HTML : transformer l’affichage un table HTML construit en utilisant jQueryDatatable.

Premièrement, il faut construire une table HTML en utilisant Handlebars et les données JSON reçus du serveur.

templateTable.html

<table id=*"tb\_table"* class=*"lstTable"*>

<thead>

<tr style="color:*White*; background-color:*DarkBlue*;">

<th>Header 1</th>

<th>Header 2</th>

<th>Header 3</th>

<th>Header 4</th>

</tr>

</thead>

<tbody>

{{#each liste}}

<tr class=*"{{#if cond}}class1{{else}}class2{{/if}}"* >

<td>{{content1}}</td>

<td>{{content2}}</td>

<td>{{content3}}</td>

<td>{{content4}}</td>

</tr>

</tbody>

</table>

Note :

La table HTML doit respecter les conditions suivantes :

1. Il faut avoir le <thead> et le <tbody> pour distinguer l’entête et le corps de la table.
2. Le nombre de colonne de l’entête doit être égale au nombre de colonne du corps de la table.

Table HTML construit à partir des données reçues

<table id=*"tb\_table"* class=*"lstTable"*>

<thead>

<tr style="color:*White*; background-color:*DarkBlue*;">

<th>Header 1</th>

<th>Header 2</th>

<th>Header 3</th>

<th>Header 4</th>

</tr>

</thead>

<tbody>

<tr class=*"class1"* >

<td>contentA1</td>

<td>contentA2</td>

<td>contentA3</td>

<td>contentA4</td>

</tr>

<tr class=*"class2"* >

<td>contentB1</td>

<td>contentB2</td>

<td>contentB3</td>

<td>contentB4</td>

</tr>

<tr class=*"class1"* >

<td>contentC1</td>

<td>contentC2</td>

<td>contentC3</td>

<td>contentC4</td>

</tr>

</tbody>

</table>

Puis on peut utiliser jQueryDatatable pour transformer l’affichage de la table HTML construit en utilisant la fonction :

$(*"#tb\_table"*).DataTable({

"pagingType": "full\_numbers",

"order": [[ 0, "desc" ]],

"language": {

"lengthMenu":"Afficher \_MENU\_ contrôles par page",

"info":"Page \_PAGE\_ sur \_PAGES\_",

"paginate": {

"first": "Premier",

"last": "Dernier",

"next": "Suivant",

"previous": "Précédent"

}

}

});

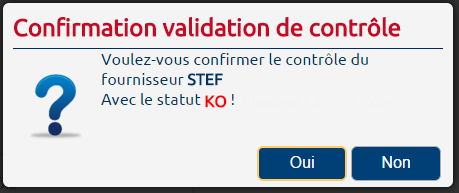
La fonction DataTable applique sur l’élément HTML sélecté par jQuery ($(*"#tb\_table"*)) et elle prend un objet Javascript comme paramètre, dont les options principales sont :

* "pagingType" : Type de pagination de la table
* "order": L’ordre initial de la table
* "language": personnaliser les textes d’utiliser par DataTable

Pour avoir plus d’information sur l’utilisation de jQueryDatatable, veuillez consulter le site <https://www.datatables.net/>

#### JQuery MsgBox

Cette bibliothèque sert à afficher un Pop-up Message.



Utilisation

Pour utiliser cette bibliothèque, il faut juste appeler la fonction :

$.msgBox({

title:"Message Title",

content:"<span style='color:#5858FA'>Message Content</span>"

});

Dont les paramètres supportés par jQuery MsgBox sont :

|  |  |  |
| --- | --- | --- |
| **Option** | **Valeur accepté** | **Description** |
| content | tout string | Le contenu du message. Par défaut : 'Message' |
| title | tout string | Le titre du message, par défaut: 'Warning' |
| type | 'alert','confirm','error','info','prompt' | Le type du message qui sera affiché. Par default: 'alert' |
| autoClose | valeur de type boolean | true pour activer auto-closing, ou false. Par défaut : false |
| timeOut | milliseconds | Le délai de auto-close. Par défaut : content.length \* 70 |
| showButtons | valeur de type boolean | true pour afficher les boutons sur le message, ou false. Par défaut : true |
| buttons | array | format:[{value:"Button1"},{value:"Button2"}]. Par défaut:[{value:"Ok"}] |
| success | callback | Une callback fonction qui est passée comme paramètre la valeur du bouton cliqué. |
| beforeShow | callback | Une callback avant que le message serait affiché. |
| afterShow | callback | Une callback après que le message est affiché. |
| beforeClose | callback | Une callback avant que le message serait fermé. |
| afterClose | callback | Une callback après que le message est fermé. |
| opacity | une valeur entre 1 et 0 | la propriété css du panel derrière du message. Par default:0.1 |

#### RequireJS

RequireJS est un chargeur Javascript de fichiers et de modules. L'utilisation d'un chargeur de scripts modulaire comme RequireJS permettra d'améliorer la vitesse et la qualité de votre code.

Utilisation

Configurer RequireJs

requirejs.config({

// Base dir for all js

baseUrl: 'www/js/lib',

// paths and aliases

paths: {

jquery : 'jquery-1.9.1.min',

handlebars : 'handlebars',

pathparser : 'pathparser',

app : '../app',

common : '../app/common',

controller : '../app/main/controller',

template : '../app/main/template'

}

});

1. baseUrl : est le chemin de base des librairies javascript
2. paths : défini le chemin d’accès pour chaque librairie javascript basé sur le baseUrl.
   1. RequireJs va ajouter automatiquement « .js » à la fin du chemin
   2. Chaque chemin est représenté par un nom

Définir un module

Un module est défini par la fonction define() de RequireJS.

define(["dependencie1","dependencie2"],

**function**(param1, param2){

**return** object;

}

);

Le premier paramètre ["dependencie1","dependencie2"] est un tableau des dépendances de ce module, les dépendances peuvent être des librairies, des fichiers ou des autres modules. Le deuxième paramètre **function**(param1, param2) est la fonction qui définit le module. Cette fonction doit retourner un objet qui définit ce module. Elle n’est exécuté que tous les dépendances sont complètement téléchargés. Les dépendances seront passées comme des paramètres de la fonction de définition, listé dans le même ordre que l'ordre dans le tableau des dépendances. C’est-à-dire param1 correspond avec "dependencie1" et param2 correspond avec "dependencie2".

Pour chaque module, déclarer un variable qui représente le contrôleur, puis définir les attributs et les fonctionnalités dans cette variable, et enfin retourner ce variable comme la définition du module.

define(["dependencie1","dependencie2"],

**function**(param1, param2){

**var** sampleController = **function**(controllerParam) {

//Do something

}

**return** sampleController;

}

);

Pour avoir plus information sur l’utilisation de RequireJS, veuillez consulter le site <http://requirejs.org/>

#### PathParser.js

pathparser.js est une bibliothèque de JavaScript URL parseur/routeur simple et facile à utiliser sans dépendances. Elle est utilisée par RouteController pour réaliser la correspondante entre un lien abstrait défini dans app.js avec une fonction qui faire appel à un contrôleur.

Utilisation

Déclarer une router qui gère tous les liens :

router = **new** PathParser();

Enregistrer un pair lien/fonction :

router.add("path/:param", **function** () {

console.log("Loading page...");

sampleController.login(**this**.param);

});

On a enregistré le lien path avec un paramètre :param dans le router qui correspond avec une fonction. Pour accéder à un paramètre, utiliser **this.**<nom de paramètre> dans la fonction.

Pour exécuter un lien enregistré, on peut utiliser la fonction :

router.run('path/12345');

Le dernier paramètre n’est pas obligatoire :

router.run('path');

#### Network

Network est une bibliothèque développé par Servair qui sert à gérer les requêtes envoyés au serveur et les réponses reçus du serveur.

#### Template

Template est une bibliothèque développé par Servair qui sert à gérer et à compiler les templates en utilisant Handlebars.

# Règle de développement

Note important :

Tous les commentaires et des noms de packages, de classes, de méthodes, de variable doivent être en anglais.

## Convention codage

### Commentaire

#### Java et JavaScript

Ajouter une ligne de commentaire :

//Commentaire

Ajouter un bloc commentaire :

Taper :

/\*\* [Enter]

Eclipse va ajouter un bloc de commentaire comme ci-dessous :

/\*\*

\* Commentaire

\*/

Commenter une fonction (Obligatoire) :

Après d’ajouter un bloc de commentaire au-dessus de chaque méthode, l’éclipse va insérer automatiquement les champs :

1. **@param** Le paramètre d’entré de la fonction, le nombre de champs **@param** correspond au nombre de paramètre d’entrés de la fonction.
2. **@return** La valeur obtenue après l’exécution de la fonction
3. **@throws** Le message d’erreur

Exemple :

/\*\*

\* Objectif de la fonction

\* **@param** param1

\* **@param** param2

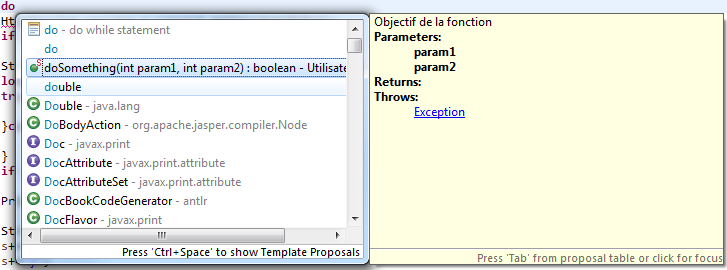
\* **@return**

\* **@throws** Exception if error

\*/

**public** **static** **boolean** doSomething(**int** param1, **int** param2) **throws** Exception {

Ce commentaire va apparaître dans la description de la fonction:



Commenter une variable

Ajouter un bloc ou une ligne de commentaire au-dessus de la déclaration de la variable.

/\*\*

\* Commentaire

\*/

**int** variable;

#### HTML

Ajouter des commentaires avant les éléments HTML pour faciliter la compréhension est recommandé. Pour ajouter un commentaire, utiliser :

<!-- This is a comment -->

<!--   
  This is a long comment example. This is a long comment example. This is a long comment example.  
  This is a long comment example. This is a long comment example. This is a long comment example.  
-->

## Convention de nommage

### Code JAVA

#### Les packages

1. Tout en minuscule.
2. Utiliser seulement [a-z], [0-9] et le point '.': Ne pas utiliser de tiret '-', d'underscore '\_', d'espace, ou d'autres caractères ($, \*, accents, ...).
3. Généralement les packages sont nommés avec un nom de domaine en commençant par la fin et en ajoutant ensuite le nom du projet. Cela donnerait pour le package Services de l’application de Servair : com.servair.prodige.services.

#### Les classes

1. 1ère lettre en majuscule
2. Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
3. Donner des noms simples et descriptifs
4. Ne pas commencer les noms avec '$' ou '\_' bien que ce soit possible.
5. Les interfaces se nomment de la même manière que les classes.

#### Les Méthodes

1. 1ère lettre en minuscule
2. Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
3. Donner des noms simples et descriptifs
4. Comporter généralement un verbe d'action
5. Ne pas commencer les noms avec '$' ou '\_' bien que ce soit possible.

Cela donne par exemple : closeWindow(), eat(), etc.  
En plus de cela, il faut distinguer les fonctions de services qui exposent aux autres applications et les fonctions qui sont utilisées uniquement à l’intérieur de l’application en utilisant le mot clé **request** pour les fonctions de services. Par exemple requestUserInfo().

#### Les variables

1. 1ère lettre en minuscule
2. Mélange de minuscule, majuscule avec la première lettre de chaque mot en majuscule
3. Donner des noms simples et descriptifs
4. Ne pas commencer les noms avec '$' ou '\_' bien que ce soit possible.
5. Variable d'une seule lettre (pour un usage local)
   1. int : i, j, k, m, et n
   2. char : c, d, et e
   3. boolean : b
6. Les attributs des classes model qui correspondent avec des tables dans la base de données doivent être nommé impérativement identiques avec les noms des colonnes des tables.

#### Les constantes

1. Tout en majuscule
2. Séparer les mots par underscore '\_'
3. Donner des noms simples et descriptifs
4. N'utiliser que les lettres [A-Z], [0-9] et '\_' : Ne pas utiliser de tiret '-' ou d'autres caractères ($, \*, accents, ...).

### Code HTML et JavaScript

#### Id des éléments HTML

1. Tout en minuscule
2. Commencer par type de l’élément, puis le nom choisi :
   1. div\_ : <div>
   2. inp\_ : <input type=’text’>
   3. sel\_ : <input type=’select’>
   4. btn\_ : <input type=’button’>
   5. tab\_ : <table>
   6. ul\_  : <ul>
   7. spa\_ : <span>
   8. img\_ : <img>
   9. frm\_ : <form>
3. Suivre par le nom du template html
4. Séparer les mots par underscore '\_'

#### Clés des éléments traduction

#### Fonction JavaScript

Fonction globale :

Les fonctions globales sont les fonctions qui peuvent être accédées partout.

### La base de données

#### Les tables

1. Séparer les mots par underscore '\_'
2. Commencés par « TA\_<Groupe>\_ » puis nom de la table.

#### Les colonnes

1. Séparer les mots par underscore '\_'
2. Commencer par « <Type>\_ ». <Type> est le type des valeurs de cette colonne, dont les types utilisés sont :
3. I\_ : nombre entière (int)
4. F\_ : nombre décimal (double)
5. D\_ : date et l’heure (datetime)
6. T\_ : texte (nvarchar)

# Bonnes pratiques de développement

## Bonnes pratiques de programmation

### Les classes Model

Ajouter les attributs qui représentent les relations avec les autres classes et ses fonctions de récupération et de modification.

### Programmer un Service

La classe du service doit avoir cet appel dans son constructeur par défaut :

ServiceLoader.*addService*(***SV\_CLASS***, **this**);

Pour envoyer une réponse au client, il faut obligatoirement utiliser la fonction doResponse du API dont l’utilisation est fixée comme suivant :

API.*doResponse*(response,JSONTool.*reqJSonString*(

filter,

JSONDef.***JS\_OBJ\_LOGGED*** , **true**,

JSONDef.***JS\_OBJ\_SV\_CODE*** , APICode.*SV\_CODE\_API\_YES*,

JSONDef.***JS\_OBJ\_RES\_DATA*** , response

));

JSONDef.***JS\_OBJ\_LOGGED***: ce paramètre sert à signaler que cet utilisateur est déjà connecté.

JSONDef.***JS\_OBJ\_SV\_CODE***: ce paramètre est le code pour s’identifier le type de la réponse.

JSONDef.***JS\_OBJ\_RES\_DATA*** : ce paramètre est les données de la réponse. Il est un l’objet qu’on veut envoyer au client par exemple un objet TaUser. Cet objet peut être une liste des objets différents ou une chaine de caractères.

Filter : est un filtre qui limite les attributs de l’objet réponse qu’on ne veut pas envoyer au client. Par exemple, l’objet TaUser contient le mot de passe de l’utilisateur, mais on ne veut pas l’envoyer au client, alors qu’il faut le filtrer avant d’envoyer la réponse.

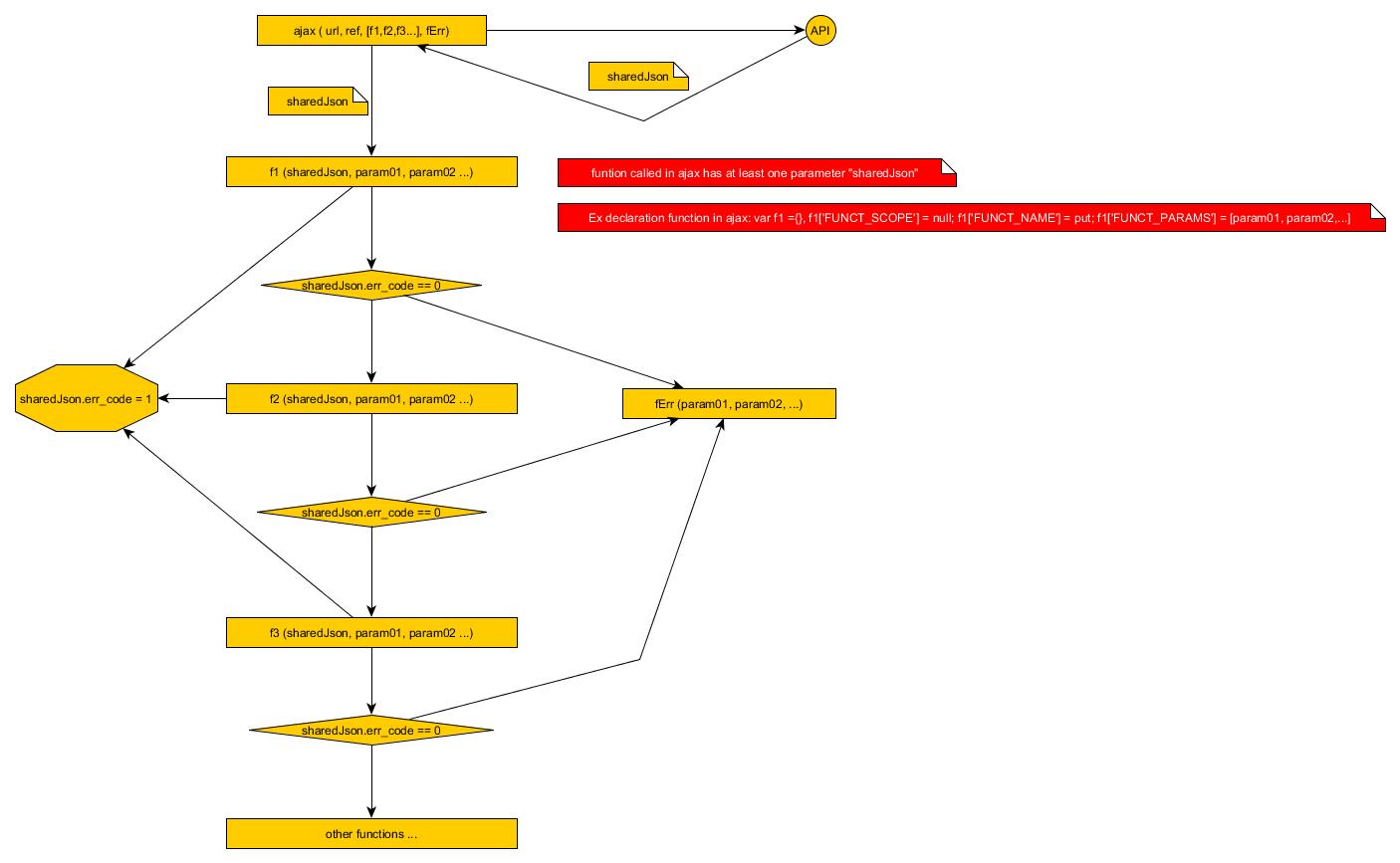
### Envoyer et récupérer une requête AJAX au serveur

#### Envoyer une requête au serveur :

Il faut utiliser la bibliothèque Network. Cette bibliothèque est décrire dans la partie Network. Avant d’envoyer une requête au serveur, il faut déterminer :

1. Le service sera demandé sur le serveur
2. Les paramètres seront envoyés au serveur
3. Les fonctions qui seront exécutés après la réception de la réponse du serveur
4. La fonction qui sera exécuté une fois une erreur rencontrée

Cercle de vie d’une requête AJAX :



Les fonctions Javascript fourni par la bibliothèque Network qu’il faut utiliser pour envoyer une requête AJAX au serveur sont :

1. ajax           = **function**(urlAPI, header, ref, timeWaitMax, fSucces, fError)
2. ajaxBackground = **function**(urlAPI, header, ref, timeWaitMax, fSucces, fError)

Toutes ces 4 fonctions envoient une requête au serveur avec ces paramètres :

1. urlAPI : lien de l’API sur le serveur
2. header : l’identifiant de l’utilisateur pour identifier l’utilisateur sur le serveur
3. ref : les paramètres envoyés au serveur
4. timeWaitMax : le temps d’attend maximale la réponse du serveur
5. fSucces : une liste des fonctions qui seraient exécutés consécutivement après la réception de la réponse du serveur. Une fonction dans fSucces doit procéder les paramètres suivant :
   1. fVar : le variable dont cette fonction appartient à (le scope de la fonction)
   2. fName : le nom de la fonction
   3. fParam : les paramètres d’entrer de la fonction.
6. fError : une seule fonction qui serait exécuté en cas d’erreur

Note : Une fonction de fSucces ou fError doit avoir impérativement le premier paramètre d’entrer qui est l’objet JSON reçu du serveur.

La fonction ajax qui envoie une requête en mode synchronisation, c’est-à-dire en attendant la réponse, le site est bloqué par une animation de chargement.

La fonction ajaxBackground qui envoie une requête en mode asynchronisation, c’est-à-dire il n’affiche pas l’animation de chargement et l’utilisateur peut continuer à travailler en attendant la réponse du serveur.

Une requête serait envoyé au serveur se compose des paramètres obligatoire (regroupant dans la variable ref) suivant :

1. sv\_class : La classe du service sur le serveur (e.g UserService)
2. sv\_name : Nom du service (e.g Login)
3. session\_id : une clé d’identité générer par le serveur une fois l’utilisateur accède à l’application. Cette clé doit être sauvegardé localement et envoyé dans tous les requêtes pour s’identifier l’utilisateur.
4. liste des paramètres : une liste de paramètre demandée par le service
5. req\_data\_nb : nb ligne max (maximal 20 000)
6. req\_data\_begin : la première ligne du bloc

Exemple d’une requête d’ajax :

//Préparer les paramètres de la requête

var fVar = "fVar";

var fName = "fName";

var fParam = "fParams";

**var** ref = {};

ref["sv\_class"] = "ServiceClass";

ref["sv\_name" ] = "doSomething";

ref["session\_id"] = App.data.session\_id;

ref["param1" ] = param1;

ref["param2" ] = param2;

//Définir les fonctions succès et erreur

**var** fSucces = [];

**var** f01 = {}; f01[fVar]= App ;f01[fName] = App.funct.put;f01[fParam]=[varname];

**var** f02 = {}; f02[fVar]= **this**;f02[fName] = loadView;f02[fParam]=[varname];

**var** f03 = {}; f03[fVar]= **this**;f03[fName] = bindingEventsPage;f03[fParam]=['ok'];

fSucces.push(f01);

fSucces.push(f02);

fSucces.push(f03);

**var** fError = {};

fError[fVar] = **null**; fError[fName] = showMsg; fError[fParam]=['Something wrong'];

App.network.ajax (App.path.BASE\_URL\_API, **null**, ref,100000, fSucces, fError) ;

Dont les étapes d’envoyer une requête sont :

1. Préparer les paramètres de la requête
   1. Nom de la classe du service souhaité
   2. Nom de la fonction du service demandé
   3. L’identifiant de la session
   4. Liste des paramètres complémentaires : chaque paramètre est identifier par un nom identique, et le nombre de paramètre n’est pas limité.
2. Définir la liste des fonctions exécutées en cas réussi et en cas d’échoué
   1. Chaque fonction est définie comme un objet javascript qui contient ses trois paramètres fVar, fName et fParam.
   2. Puis ses fonctions sont ajoutées dans la liste fSucces
   3. fError est seulement un objet javascript (ne pas être une liste comme fSucces) qui contient ses trois paramètres fVar, fName et fParam.
3. Recevoir la requête en utilisant la fonction App.network .ajax (\_background)

#### Recevoir la réponse du serveur

Après d’envoyer la requête au serveur, le serveur traite la requête et une réponse sera renvoyée au client. Cette réponse se compose une partie de des codes obligatoires qui se présentent dans tous les réponses reçu du serveur, une partie des données de réponse qui sont des résultats du traitement de la requête.

La partie des codes obligatoires seraient traité automatiquement par la bibliothèque Network :

1. logged : true si l’utilisateur est connecté
2. app\_code : le code de la réponse (une liste des codes)
3. session\_id :
4. res\_nb\_total : nombre total de ligne de données
5. res\_begin : le début des données reçu
6. res\_nb : nombre de lignes de données reçu
7. res\_data : données reçu en format Objet Javascript, va être passers aux fonctions succès

### Construit une partie de la page en utilisant un template et les données reçus

Avec les données reçues du serveur, on peut maintenant utiliser la bibliothèque Template pour construire un page HTML en utilisant Handlbars.

La bibliothèque Template supporte deux fonctions principales :

1. putTemplate(varName, templateFile);
2. getCompiled(varName, data);

## Bonnes pratiques d’utilisation de la base de données

### Recommandations d'implémentation du MDD de Prodige pour une BDD Oracle :

1. Utiliser un nombre réduit de schéma (idéalement 1) (rappel : sous Oracle : 1 schéma = 1 user)
2. Un seul utilisateur  accédera au schéma avec profil et rôle adequat. Les utilisateurs applicatifs seront gérés ainsi que leurs droits dans l'application
3. Si une table contient des mots de passe, faire en sorte que ceux-ci soient cryptés
4. S'assurer que chaque table ait au moins une clé primaire (quand le développement des requêtes sera suffisamment avancé, il faudra procéder certainement à la création d’indexes secondaires)
5. Faire en sorte que les dépendances fonctionnelles entre les tables soient concrétisées par des contraintes cela garantira un niveau minimal de cohérence des données

### Gestion des accès aux différents environnements :

1. Pas de modification directe sur les BDD hors environnement de développement, les livraisons auront lieu sous forme de PRI et seront réalisées par un DBA