TP Mobilité et Domotique

EI5 AGI 2013/2014

Serge Tahé, Sébastien Lagrange
avril 2013

1 Introduction

1.1 Contenu du document

Le point de départ de ce document a été un projet étudiant de dernière année de l'école d'ingénieurs ISTIA de l'université d'Angers [istia.univ-angers.fr] : piloter des éléments de la maison (lampe, chauffage, porte de garage, ...) avec un mobile, smartphone ou tablette. Ce projet a été proposé par Sébastien Lagrange, enseignant-chercheur de l'ISTIA. Il a été réalisé par six étudiants : Thomas Ballandras, Raphaël Berthomé, Sylvain Blanchon, Richard Carrée, Jérémy Latorre et Ungur Ulu. Le projet a été décliné en trois sous-projets :

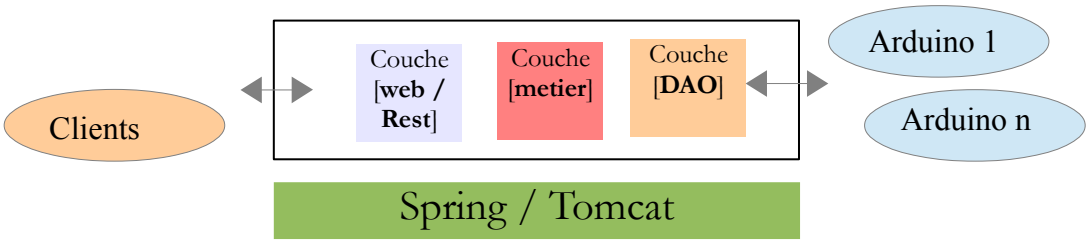
1. application web mobile avec JSF2 / Primefaces ;
2. application web mobile avec HTML5 / Javascript / Windows 8 ;
3. application native Android.

Le TP consiste à réaliser les sous-projets 3 et 1 dans cet ordre. Plusieurs documents sont référencés :

- [**ref1**] : " Introduction aux frameworks JSF2, Primefaces et Primefaces Mobile " disponible à l'URL [http://tahe.developpez.com/java/primefaces/]. Il vous servira pour construire le client web mobile ;
- [**ref2**] : " Android pour les développeurs J2EE : un modèle asynchrone pour les clients web " disponible à l'URL [http://tahe.developpez.com/android/avat]. Il vous servira pour construire le client Android ;
- [**ref3**] : " Introduction à Java EE avec Netbeans 6.8 " disponible à l'URL [http://tahe.developpez.com/java/javaee]. Il vous servira pour construire le serveur ;
- [**ref4**] : " Introduction au langage Java " disponible à l'URL [http://tahe.developpez.com/java/cours]. Il vous servira pour construire les serveurs et clients TCP-IP de la couche [DAO] du serveur.

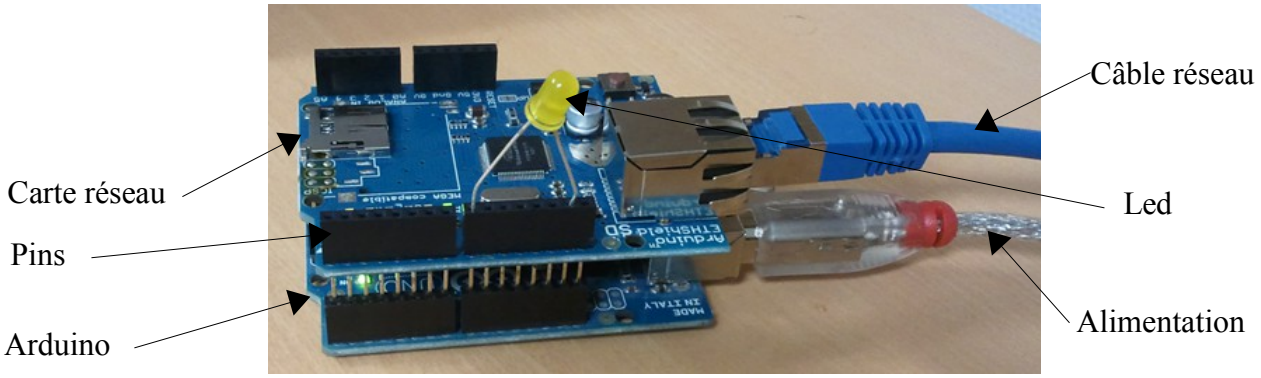
1.2 Le projet Android

L'architecture du projet Android est celle d'une application client / serveur. L'architecture du serveur est la suivante :



Wikipedia donne la définition suivante des Arduinos :

Arduino *est un circuit imprimé en matériel libre (dont les plans sont publiés en licence libre) sur lequel se trouve un microcontrôleur qui peut être programmé pour analyser et produire des signaux électriques, de manière à effectuer des tâches très diverses comme la charge de batteries, la domotique (le contrôle des appareils domestiques - éclairage, chauffage…), le pilotage d'un robot, etc. C'est une plateforme basée sur une interface entrée/sortie simple et sur un environnement de développement utilisant la technique du Processing/Wiring.*



La couche [DAO] est reliée aux arduinos via un réseau. Elle est constituée
- d'un serveur TCP/IP pour enregistrer les Arduinos qui se connectent ;
- d'un client TCP/IP pour envoyer les commandes à exécuter aux Arduinos connectés.

La couche [DAO] peut commander plusieurs Arduinos.

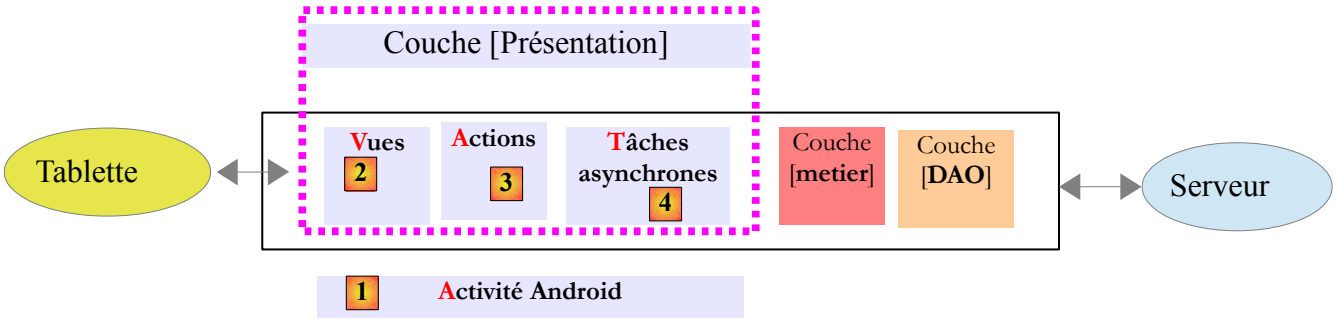
Côté Arduino, on trouve les mêmes composantes mais programmées en langage C :
- un client TCP/IP pour s'enregistrer auprès du serveur d'enregistrement de la couche [DAO] ;
- un serveur TCP/IP pour exécuter les commandes que la couche [DAO] envoie.

Un Arduino peut servir plusieurs clients. Ils sont alors servis séquentiellement.

Les échanges entre la couche [DAO] et un Arduino se font par lignes de texte :
- la couche [DAO] envoie une ligne de texte contenant une commande au format JSON (**J**ava**S**cript **O**bject **N**otation) ;
- l'Arduino interprète puis exécute cette commande et renvoie une ligne de texte contenant une réponse également au format JSON.

La couche métier du serveur est exposée au client via un service REST (**RE**presentational **S**tate **T**ransfer) implémenté par le framework Spring MVC.

L'architecture du client Android est elle la suivante :



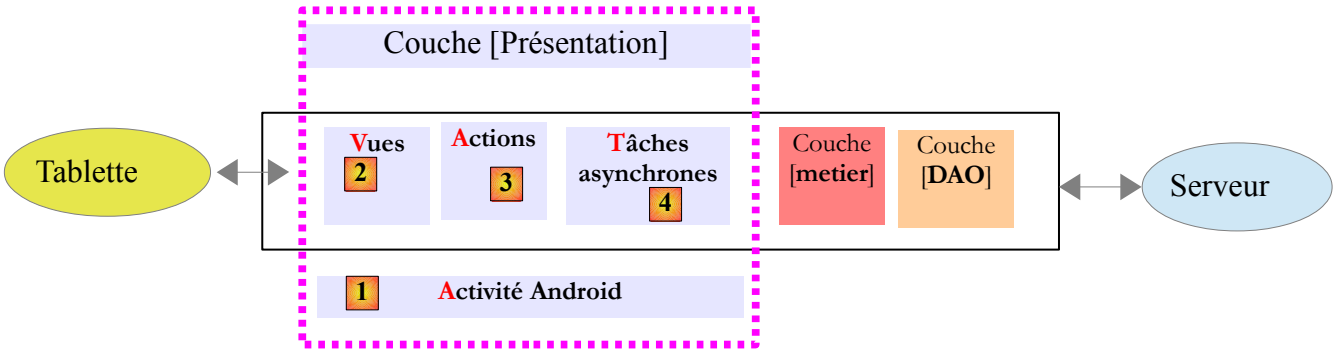
- la couche [DAO] communique avec le serveur REST. C'est un client REST implémenté par **Spring-Android** ;
  la couche [métier] reprend l'interface de la couche [métier] du serveur ;

Les versions actuelles d'Android exigent que les connexions réseau soient faites dans un autre thread que celui qui gère les interfaces visuelles. C'est ce qui explique la présence du bloc [4]. Les méthodes de la couche [métier] sont exécutées au sein d'un thread différent de celui de l'UI (**U**ser **I**nterface). Lorsqu'on fait exécuter une méthode de la couche[métier] dans un thread, on peut attendre ou non la réponse de la méthode. Dans le premier cas, synchrone, on bloque l'UI en attendant la réponse de la tâche. Dans le second cas, asynchrone, l'UI reste disponible pour l'utilisateur qui peut lancer d'autres actions. Ici, on a choisi la solution asynchrone pour deux raisons :

- le client Android doit pouvoir commander plusieurs Arduinos simultanément. Par exemple, on veut pouvoir faire clignoter une led placée sur deux Arduinos, en même temps et non pas l'une après l'autre. On ne peut donc attendre la fin de la première tâche pour commencer la seconde ;
- les connexions réseau peuvent être longues ou ne pas aboutir. Pour cette raison, on veut donner à l'utilisateur la possibilité d'interrompre une action qu'il a lancée avec un bouton [Annuler]. Pour cela, l'UI ne doit pas être bloquée.

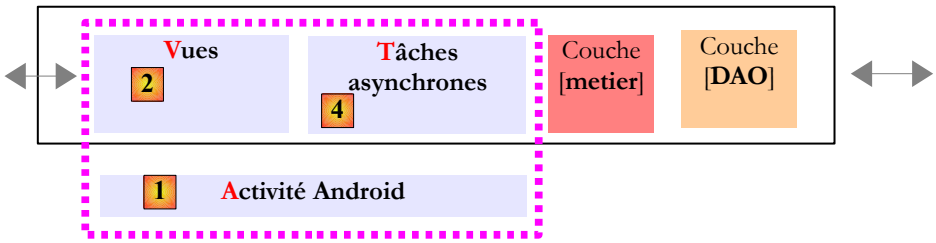
Les vues [2] sont les interfaces visuelles présentées à l'utilisateur. Sur une tablette, elles ressemblent à ceci :

A partir de cette vue, l'utilisateur lance une action avec le bouton [Exécuter].



Le bloc [3] regroupe les actions exécutées par les vues. La vue ne fait que saisir des données et contrôler leur validité. On part du principe qu'elle ne sait pas à quoi elles vont servir. Elle sait simplement à quelle action elle doit transmettre les données saisies. L'action lancera les tâches asynchrones nécessaires, mettra en forme leurs résultats et rendra à la vue un modèle pour que celle-ci se mette à jour. La vue n'est pas bloquée par l'action, afin de donner à l'utilisateur la possibilité de l'interrompre.

Les activités [1] sont le coeur d'une application Android. Ici on n'en a qu'une et elle ne fait quasiment rien si ce n'est d'assurer les changements de vues. On appellera ce modèle, **AVAT** (**A**ctivité – **V**ues – **A**ctions – **T**âches) et c'est ce modèle que nous allons décrire et illustrer dans ce document. On verra qu'il peut être allégé de la façon suivante :

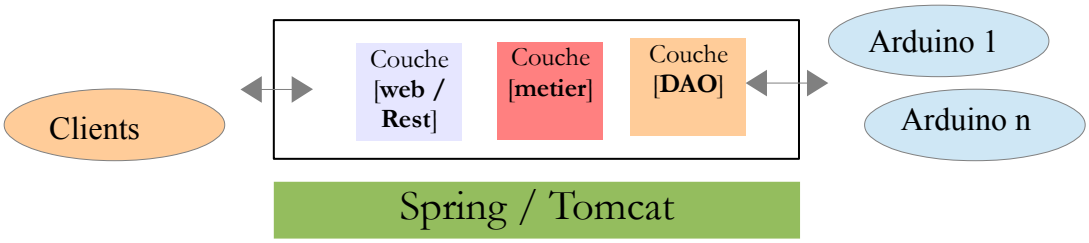


Dans le modèle AVAT, la vue est complètement ignorante de l'utilisation faite des données qu'elle a saisies. Dans le modèle **AVT** ci-dessus (**A**ctivité – **V**ues – **T**âches), la logique de l'action est transférée dans la vue. On trouve donc un peu de logique dans la vue. Celle-ci doit organiser les appels des tâches asynchrones elle-même. Cette logique est forcément faible. Si elle était importante, elle

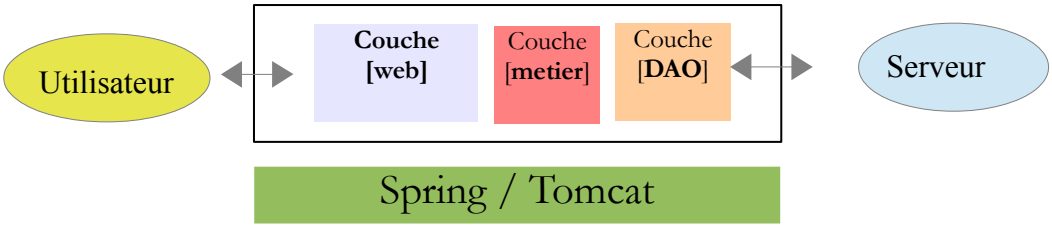
serait normalement transférée dans la couche [métier]. Ce modèle AVT est suffisant dans tous les cas. Le modèle AVAT est pour les puristes qui cherchent à limiter la vue à la saisie / affichage de données.

1.3 Le projet Web Mobile

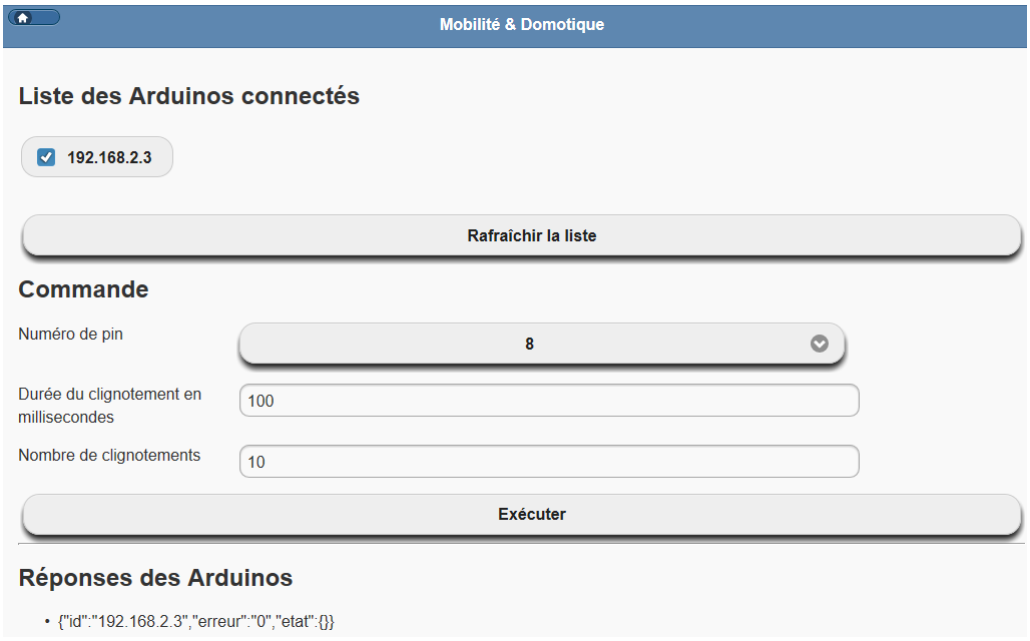
L'architecture du projet web mobile est également celle d'une application client / serveur. Le serveur reste ce qu'il était pour le client Android :



L'architecture du client web mobile est la suivante :



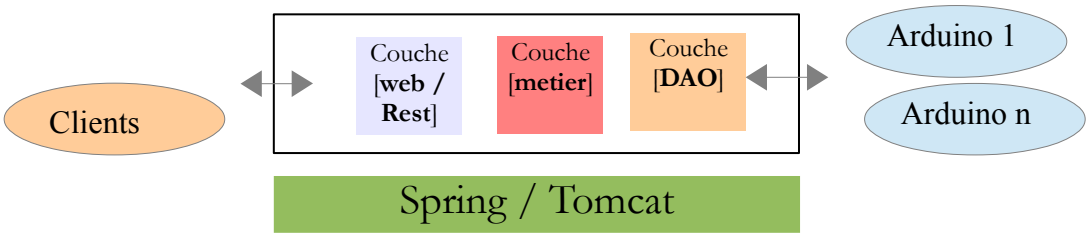
Sur une tablette, les vues ressemblent à ceci :



Le client web mobile aura les mêmes fonctionnalités que le client Android mais il aura l'avantage de fonctionner sur toutes les tablettes quelque soit l'OS (Android, IOS, Windows 8).

1.4 La progression dans le TP

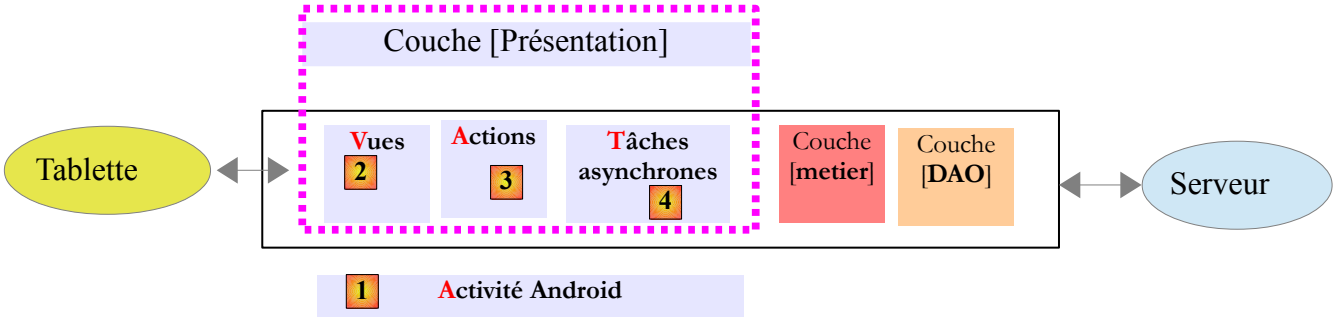
Nous allons tout d'abord écrire la partie serveur du projet qui est commune aux deux types de clients :



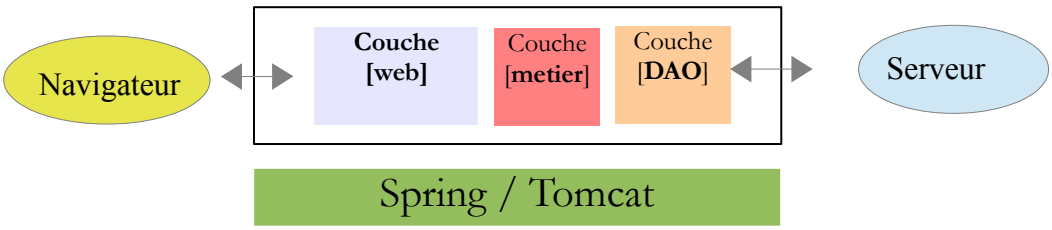
Nous allons réaliser dans l'ordre :

- la programmation des Arduinos ;
- l'implémentation de la couche [DAO] ;
- l'implémentation de la couche [métier] ;
- l'implémentation du service REST.

Puis ensuite, nous réaliserons le client Android :

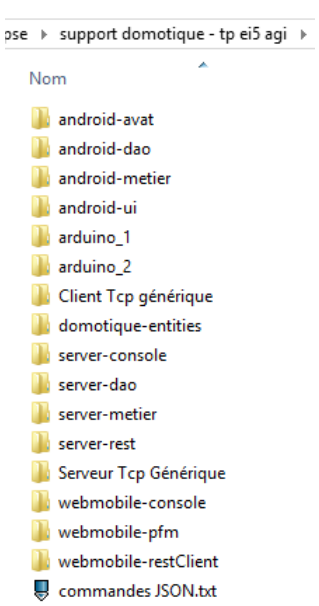


L'objectif principal du TP est de réaliser ce client Android. Ceux qui le peuvent continueront le TP par l'écriture du client web mobile :



1.5 Méthode pédagogique

Les projets " client Android " et " client web mobile " ont tous deux été donnés en projet l'an dernier. Chaque projet a été traité par un groupe de deux étudiants sur une période d'environ 150 h. A la fin du projet, chaque groupe arrivait à allumer / éteindre une lampe connectée à un Arduino. L'objectif est de faire plus en seulement 30 h. Pour y arriver, on vous propose un cadre très contraint dans lequel des squelettes de code vous sont donnés afin que vous puissiez avancer plus vite. Vous trouverez ces éléments dans un dossier [support] :



Par ailleurs, de nombreux codes sont expliqués dans ce document. Faites du copier / coller entre ce document et vos projets Eclipse.

Il est important que vous lisiez et compreniez le document avant de programmer. Avant chaque étape importante du TP, il y a des lectures recommandées. Il ne faut pas faire ces lectures pendant le TP mais chez vous avant le TP. Sinon, vous allez perdre un temps précieux pendant le TP.

Ce TP est également un cours sur la mobilité et plus particulièrement sur Android. Vous ne devez pas vous contenter de coder sans comprendre l'architecture générale de l'application. L'étape normale suivante de ce cours / TP est que ce soit vous qui dessiniez l'architecture générale d'un nouveau projet.

Pour certains, il peut être frustrant de suivre un parcours tracé à l'avance. Vous pouvez alors, si vous le souhaitez suivre votre propre chemin pour réaliser le projet. Vous devez avoir conscience des avantages / inconvénients de cette méthode. En avantage elle est plus créative. En inconvénients, vous aurez moins d'aide de la part des encadrants. Celui-ci n'a pas la capacité de suivre des projets différents alors qu'il le peut si les projets suivent tous la même trame. Il se concentrera donc sur ces derniers. Il y a ainsi le risque de rencontrer des blocages qui peuvent vous retarder.

1.6 Estimation de la durée des différentes étapes du TP

Voici une progression possible du TP :

Tâche	Durée
Lectures diverses (en dehors du TP)	20 h
Démarrage TP, Programmation des Arduinos	6 h
Couche [DAO] du serveur	6 h
Couche [métier] du serveur	3 h
Couche [REST] du serveur	3 h
Projet Android	12 h

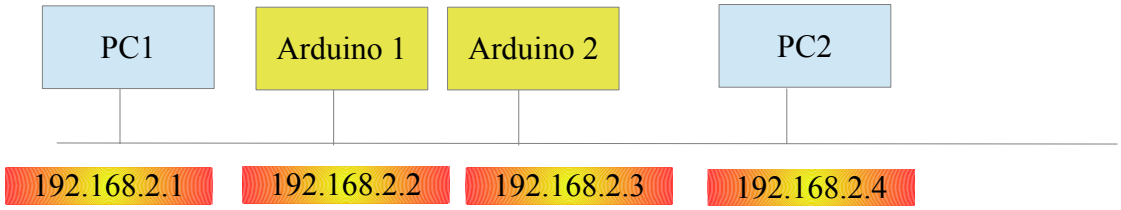
Pour arriver à l'objectif final qui est de réaliser le client Android de l'application, il est probable que vous aurez à travailler en-dehors des cours. Les 12 h affectées ci-dessus au projet Android semblent en effet insuffisantes.

1.7 Le matériel

Chaque étudiant aura à sa disposition :

- un Arduino avec une led et un capteur de température ;
- un minihub à partager avec un autre étudiant ;

- un câble USB pour alimenter l'Arduino ;
- deux câbles réseau pour connecter l'Arduino et le PC sur un même réseau privé :



- une tablette Android.

2 Les vues du projet Android

Le client Android permet de gérer les Arduinos à distance. Il présente à l'utilisateur les écrans suivants :

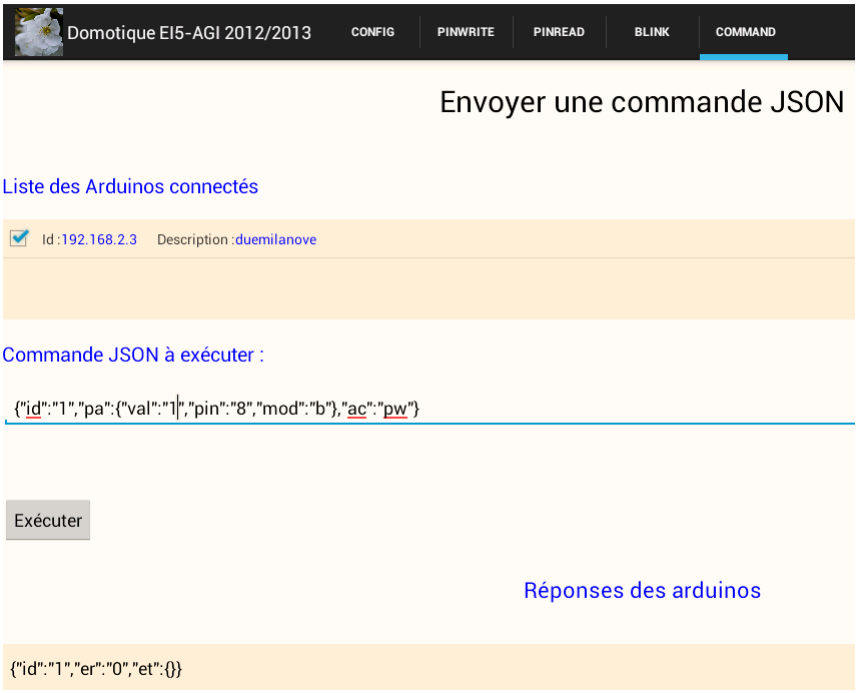
L'onglet [CONFIG] permet de se connecter au serveur et de récupérer la liste des Arduinos connectés :

L'onglet [PINWRITE] permet d'écrire une valeur sur une pin d'un Arduino :

L'onglet [PINREAD] permet de lire la valeur d'une pin d'un Arduino :

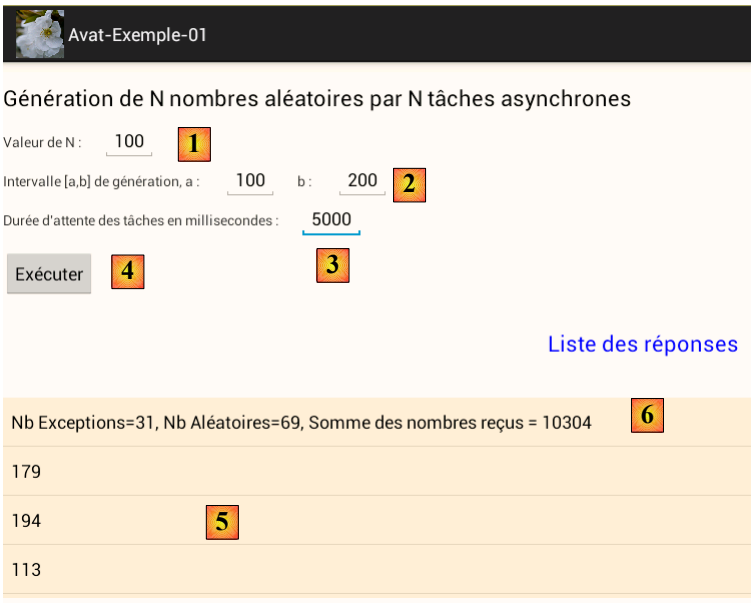
L'onglet [BLINK] permet de faire clignoter une led d'un Arduino :

L'onglet [COMMAND] permet d'envoyer une commande JSON à un Arduino :


Domotique EI5-AGI 2012/2013
CONFIG
PINWRITE
PINREAD
BLINK
COMMAND
Envoyer une commande JSON
Liste des Arduinos connectés
Id :192.168.2.3
Description :duemilanove
Commande JSON à exécuter :
{"id":"1","pa":{"val":"1","pin":"8","mod":"b"},"ac":"pw"}
Exécuter
Réponses des arduinos
{"id":"1","er":"0","et":{}}

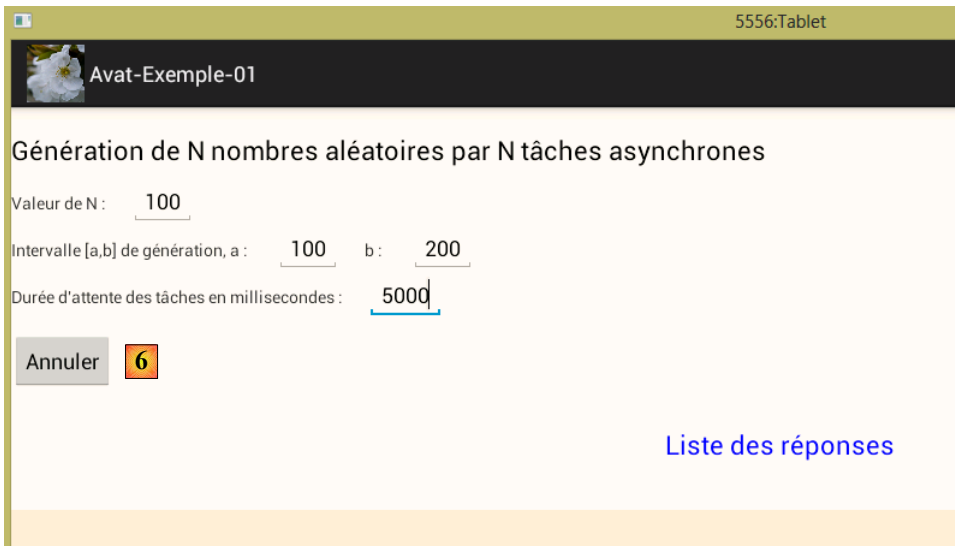
3 L'exemple à suivre pour le projet Android

Le document [**ref2**] : *Android pour les développeurs J2EE : un modèle asynchrone pour les clients web* disponible à l'URL [http://tahe.developpez.com/android/avat] est le document de référence pour ce TP, en particulier l'exemple 2. Vous devez lire ce document jusqu'à l'exemple 2 qui décrit l'application suivante :

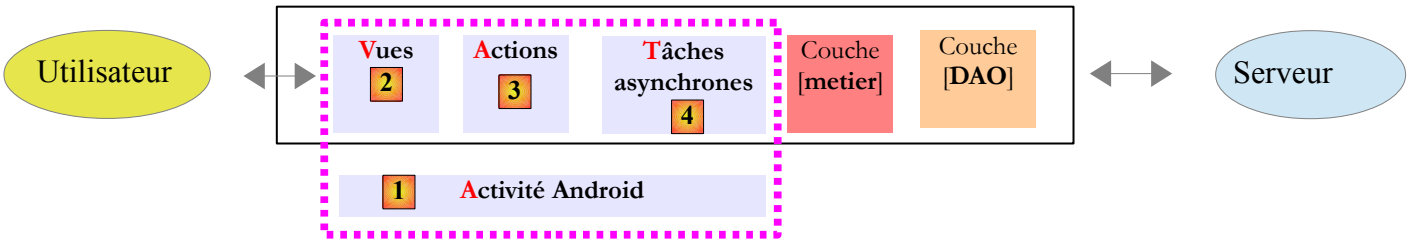


- l'application lance N [1] tâches asynchrones qui génèrent chacune un nombre aléatoire dans un intervalle [a,b] [2]. Avec une probabilité d'1/3, la tâche peut générer une exception.
- le bouton [4] lance une action unique qui va à son tour lancer les N tâches asynchrones ;
- afin de pouvoir annuler les N tâches lancées, on leur impose un délai d'attente avant de rendre leur réponse, exprimé en millisecondes [3]. Sur l'exemple, le délai de 5 secondes fait que les N tâches sont lancées et toutes attendent 5 secondes avant de faire le travail qui leur est demandé ;
- en [5] remontent les informations produites par les tâches, nombre ou exception ;
- la ligne [6] récapitule ce qui a été reçu. Il faut vérifier qu'on a bien reçu les N réponses des N tâches asynchrones et accessoirement on pourra vérifier la somme des nombres reçus.

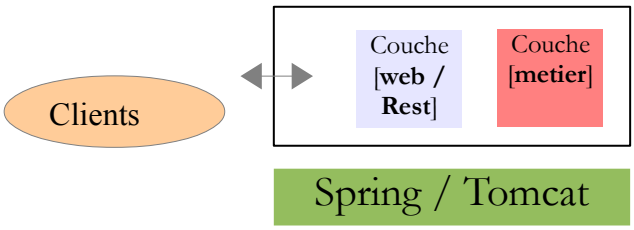
Dès que les N tâches ont été lancées, un bouton [Annuler] [6] remplace le bouton [Exécuter]. Il permet d'annuler les tâches lancées :



Cette application a l'architecture client / serveur suivante :



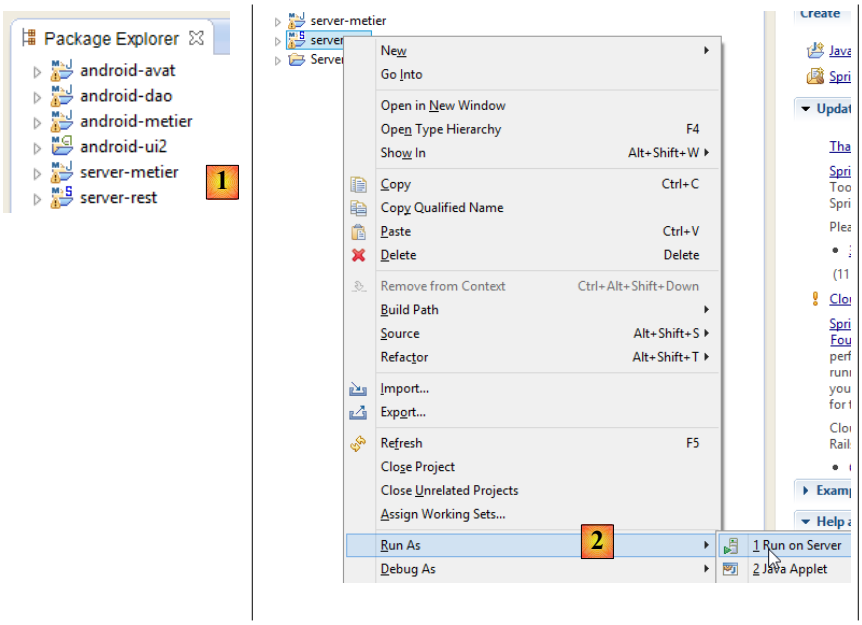
C'est le serveur qui génère les nombres aléatoires affichés par la tablette Android :



La couche [métier] a une méthode de génération de nombres aléatoires exposée au monde web via un service REST.

Pour tester cette application, on importera sous Eclipse les projets Maven des dossiers [android-avat] et [android-avat-exemple2] des exemples de [**ref2**] :

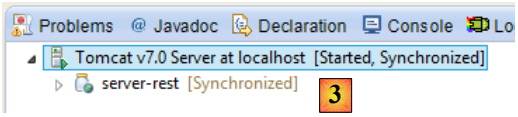
Note : ceux qui veulent installer une version d'Eclipse pour Android trouveront dans les annexes de [**ref2**] la façon de le faire.



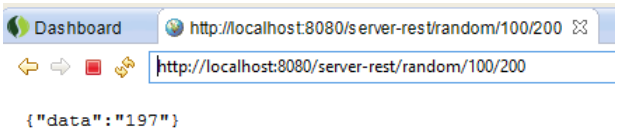
En [1] : les projets à importer :

- [android-avat] : le projet de base pour le client Android ;
- [android-dao] : la couche [DAO] du client Android ;
- [android-metier] : la couche [métier] du client Android ;
- [server-metier] : la couche [métier] du serveur ;
- [server-rest] : le service REST du serveur ;

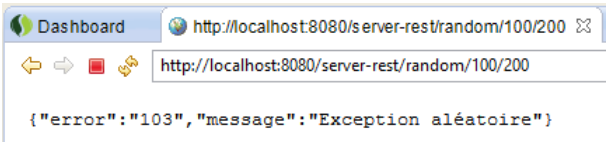
En [2], on exécute le projet web [server-rest] sur le serveur Tomcat [3] :



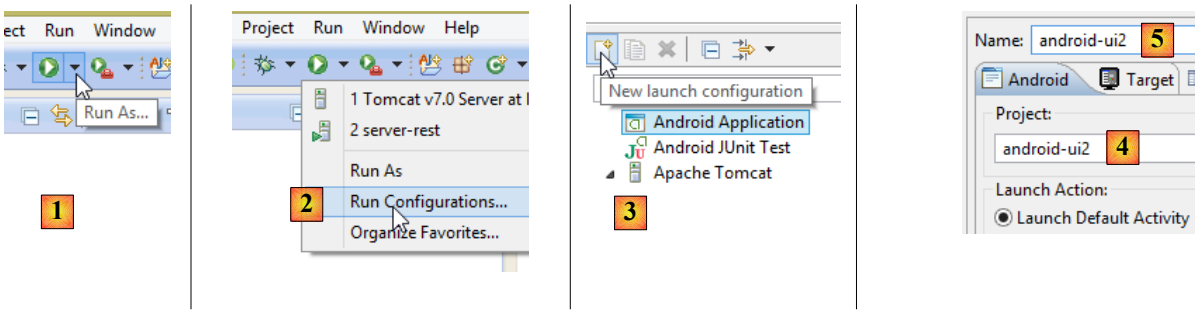
On peut tester le service REST en tapant dans un navigateur l'URL suivante :



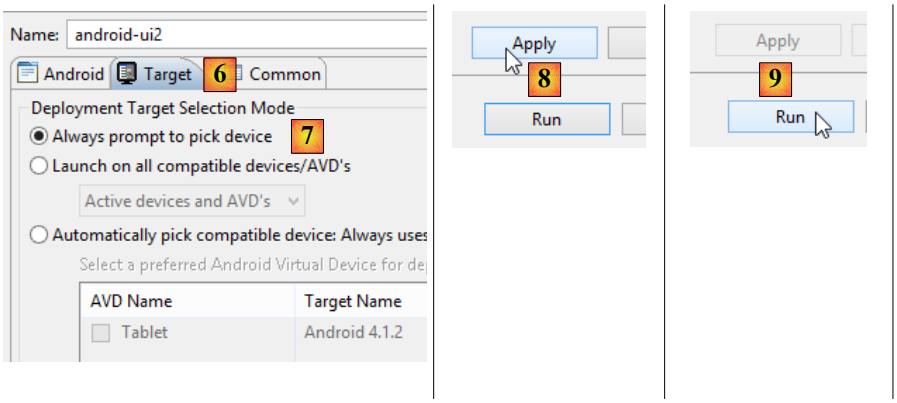
Deux fois sur 3 en moyenne on aura un nombre aléatoire entre 100 et 200 et une fois sur trois en moyenne on aura une exception :



Ensuite on crée une configuration d'exécution pour le projet Android [android-ui2] :

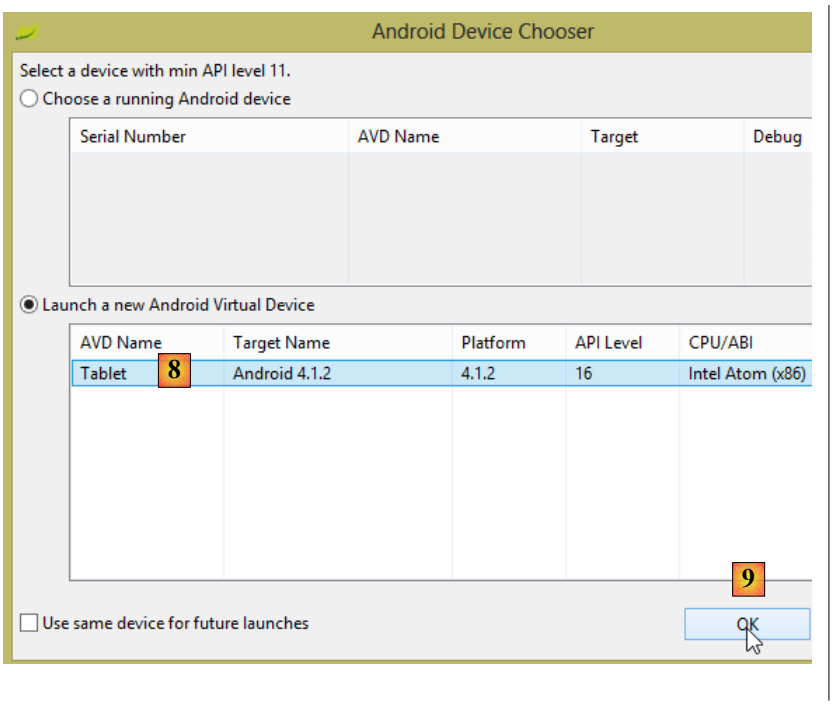


- en [1], sélectionnez l'icône [Run as...] ;
- en [2], sélectionnez l'option [Run Configurations...] ;
- en [3], sélectionnez le type [Android Application] puis l'icône [New launch configuration] ;
- en [4], indiquez le projet qui sera exécuté par cette configuration ;
- en [5], donnez un nom à cette configuration. Peut être quelconque ;



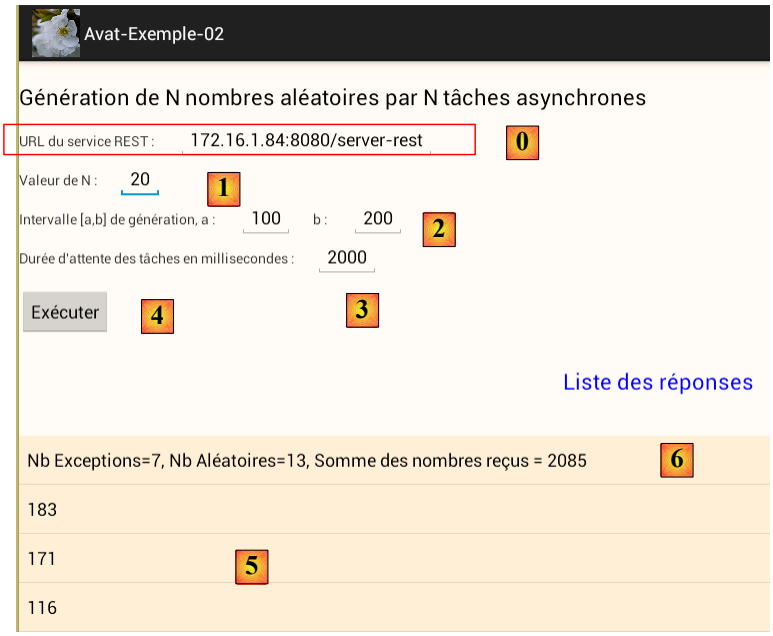
- dans l'onglet [Target] [6], sélectionnez l'option [7]. Elle permet de choisir le mode d'exécution : en mode émulation avec une tablette logicielle ou en mode réel avec une tablette Android ;

- en [8], validez cette configuration ;
- en [9], exécutez-la ;



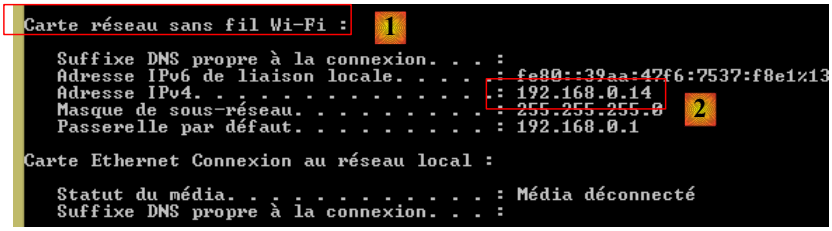
- en [8], sélectionnez l'émulateur de tablette et en [9] exécutez l'application ;

L'émulateur logiciel est alors lancé et affiche la vue suivante :

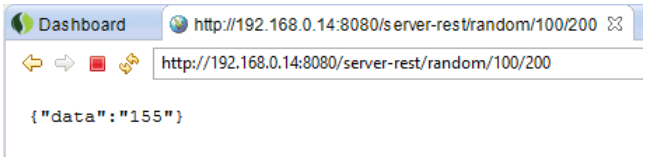


Pour mettre la bonne URL en [0], on peut procéder ainsi :

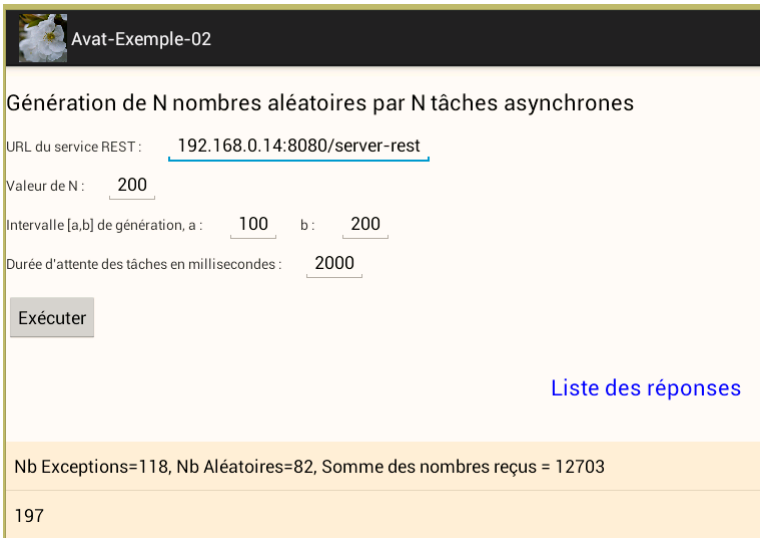
- taper la commande [ipconfig] dans une fenêtre [DOS] :



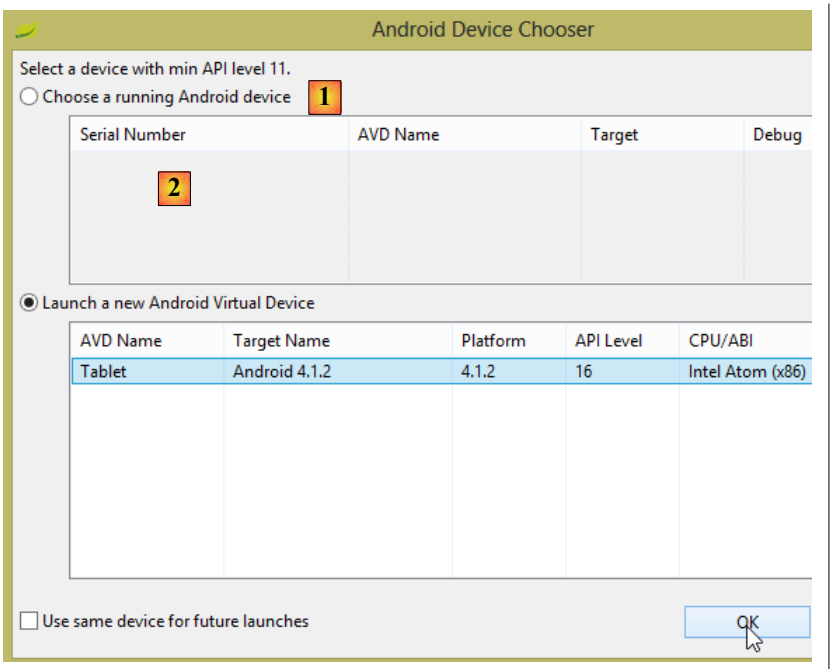
- prendre l'adresse IP [2] de la carte Wifi [1] ;
- déployer le serveur REST sous Eclipse et demander l'URL du service REST avec un navigateur. Utiliser l'adresse IP précédente ;



- l'URL du service REST à taper sur la tablette est alors [192.168.0.14:8080/server-rest]. Il faudra peut-être désactiver le pare-feu du PC voire inhiber un éventuel logiciel antivirus qui pourrait bloquer les connexions entrantes. C'est par exemple le cas de Mcafee.



Branchez maintenant une tablette Android sur un port USB du PC serveur et exécutez l'application sur celle-ci :

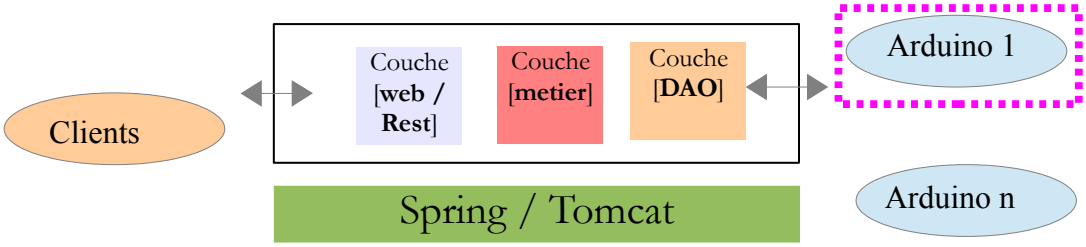


- en [1] et [2], sélectionnez la tablette Android ;

Cet exemple sera le modèle à suivre pour le TP à suivre. Etudiez-le et reproduisez-le.

4 Programmation des Arduinos

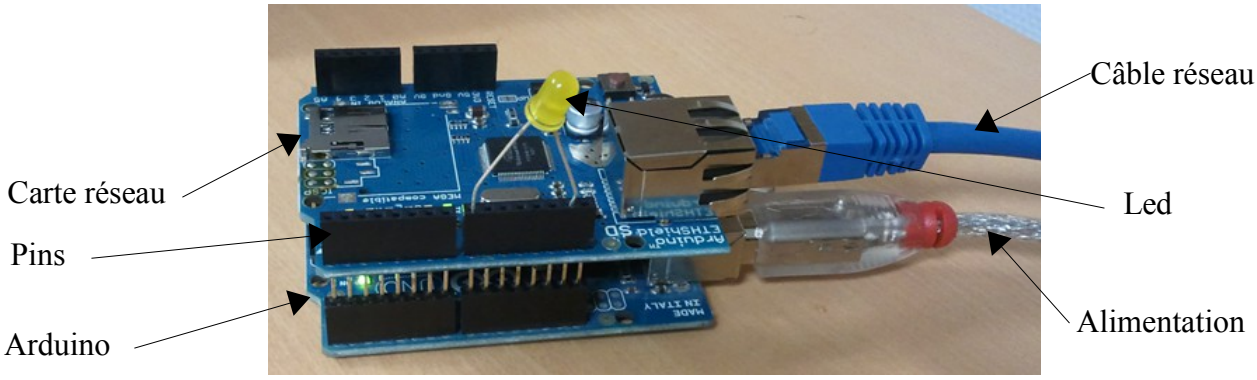
Nous nous intéressons ici à l'écriture du code C des Arduinos :



A lire

- dans [**ref2**] : l'utilisation de bibliothèques JSON (Annexes, paragraphes 11.1 et 11.2) ;
- dans l'IDE de l'Arduino, tester l'exemple d'un serveur TCP (le serveur web par exemple) et celui d'un client TCP (le client Telnet par exemple) ;
- les annexes sur l'environnement de programmation des Arduinos au paragraphe 13, page 145.

4.1 Les spécifications de la couche [Arduino]



Un Arduino est un ensemble de pins reliées à du matériel. Ces pins sont des entrées ou des sorties. Leur valeur est binaire ou analogique. Pour commander l'Arduino, il y aura deux opérations de base :

- **écrire une valeur** binaire / analogique sur une pin désignée par son numéro ;
- **lire une valeur** binaire / analogique sur une pin désignée par son numéro ;

A ces deux opérations de base, nous en ajouterons une troisième :

- **faire clignoter une led** pendant une certaine durée et avec une certaine fréquence. Cette opération peut être réalisée en appelant de façon répétée les deux opérations de base précédentes. Mais nous verrons aux tests que les échanges de la couche [DAO] avec un Arduino sont de l'ordre de la seconde. Il n'est alors pas possible de faire clignoter une led toutes les 100 millisecondes. Aussi implanterons-nous sur l'Arduino lui-même cette fonction de clignotement.

Le fonctionnement de l'Arduino sera le suivant :

- les communications entre la couche [DAO] et un Arduino se font via un réseau TCP-IP par échanges de lignes de texte au format JSON (**J**ava**S**cript **O**bject **N**otation) ;
- au démarrage, l'Arduino vient se connecter au port 100 d'un serveur d'enregistrement présent dans la couche [DAO]. Il envoie au serveur une unique ligne de texte :

```
{"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
```

C'est une chaîne JSON caractérisant l'Arduino qui se connecte :

- **id** : un identifiant de l'Arduino, ici son adresse IP ;
- **desc** : une description de ce que sait faire l'Arduino. Ici on a simplement mis le type de l'Arduino ;
- **mac** : adresse Mac de l'Arduino ;
- **port** : le numéro du port sur lequel l'Arduino va attendre les commandes de la couche [DAO].

Toutes ces informations sont de type chaînes de caractères sauf le port qui est un nombre entier.

- une fois que l'Arduino s'est inscrit auprès du serveur d'enregistrement, il se met à l'écoute sur le port qu'il a indiqué au serveur. Il attend des commandes JSON de la forme suivante :

```
{"id":"identifiant","ac":"une_action","pa":{"param1":"valeur1","param2":"valeur2",...}}
```

C'est une chaîne JSON avec les éléments suivants :

- **id** : un identifiant de la commande. Peut être quelconque ;
- **ac** : une action. Il y en a trois :
  - **pw** (**pin write**) pour écrire une valeur sur une pin,
  - **pr** (**pin read**) pour lire la valeur d'une pin,
  - **cl** (**clignoter**) pour faire clignoter une led ;
- **pa** : les paramètres de l'action. Ils dépendent de l'action.

- l'Arduino renvoie **systématiquement** une réponse à son client. Celle-ci est une chaîne JSON de la forme suivante :

```
{"id":"1","er":"0","et":{"pinx":"valx"}}
```

où

- **id** : l'identifiant de la commande à laquelle on répond ;
- **er (erreur)** : un code d'erreur s'il y a eu une erreur, 0 sinon ;
- **et (état)** : un dictionnaire toujours vide sauf pour la commande de lecture **pr**. Le dictionnaire contient alors la valeur de la pin n° **x** demandée.

Voici des exemples destinés à clarifier les scpécifications précédentes :

Faire clignoter la led n° 8 10 fois avec une période de 100 millisecondes :

Commande	`{"id":"1","ac":"cl","pa":{"pin":"8","dur":"100","nb":"10"}}`
Réponse	`{"id":"1","er":"0","et":{}}`

Les paramètres **pa** de la commande **cl** sont : la durée **dur** en millisecondes d'un clignotement, le nombre **nb** de clignotements, le n° **pin** de la pin de la led.

Ecrire la valeur binaire 1 sur la pin n° 7 :

Commande	`{"id":"2","ac":"pw","pa":{"pin":"7","mod":"b","val":"1"}}`
Réponse	`{"id":"2","er":"0","et":{}}`

Les paramètres **pa** de la commande **pw** sont : le mode **mod** ***b*** (binaire) ou ***a*** (analogique) de l'écriture, la valeur **val** à écrire, le n° **pin** de la pin. Pour une écriture binaire, **val** est 0 ou 1. Pour une écriture analogique, **val** est dans l'intervalle [0,255].

Ecrire la valeur analogique 120 sur la pin n° 2 :

Commande	`{"id":"3","ac":"pw","pa":{"pin":"2","mod":"a","val":"120"}}`
Réponse	`{"id":"3","er":"0","et":{}}`

Lire la valeur analogique de la pin 0 :

Commande	`{"id":"4","ac":"pr","pa":{"pin":"0","mod":"a"}}`
Réponse	`{"id":"4","er":"0","et":{"pin0":"1023"}}`

Les paramètres **pa** de la commande **pr** sont : le mode **mod b** (binaire) ou **a** (analogique) de la lecture, le n° **pin** de la pin. S'il n'y a pas d'erreur, l'Arduino met dans le dictionnaire **"et"** de sa réponse, la valeur de la pin demandée. Ici **pin0** indique que c'est la valeur de la pin n° 0 qui a été demandée et **1023** est cette valeur. En lecture, une valeur analogique sera dans l'intervalle [0, 1024].

Nous avons présenté les trois commandes **cl**, **pw** et **pr**. On peut se demander pourquoi on n'a pas utilisé des champs plus explicites dans les chaînes JSON, **action** au lieu de **ac**, **pinwrite** au lieu de **pw**, **parametres** au lieu de **pa**, ... On va découvrir qu'un Arduino a une mémoire très réduite. Or les chaînes JSON échangées avec l'Arduino participent à l'occupation mémoire. On a donc choisi de raccourcir celles-ci au maximum.

Voyons maintenant quelques cas d'erreur :

Commande	`xx`
Réponse	`{"id":"","er":"100","et":{}}`

On a envoyé une commande qui n'est pas au format JSON. L'Arduino a renvoyé le code d'erreur 100.

Commande	`{"id":"4","ac":"pr","pa":{"mod":"a"}}`
Réponse	`{"id":"4","er":"302","et":{}}`

On a envoyé une commande **pr** en oubliant le paramètre **pin**. L'Arduino a renvoyé le code d'erreur 302.

Commande	`{"id":"4","ac":"pinread","pa":{"pin":"0","mod":"a"}}`
Réponse	`{"id":"4","er":"104","et":{}}`

On a envoyé une commande **pinread** inconnue (c'est pr). L'Arduino a renvoyé le code d'erreur 104.

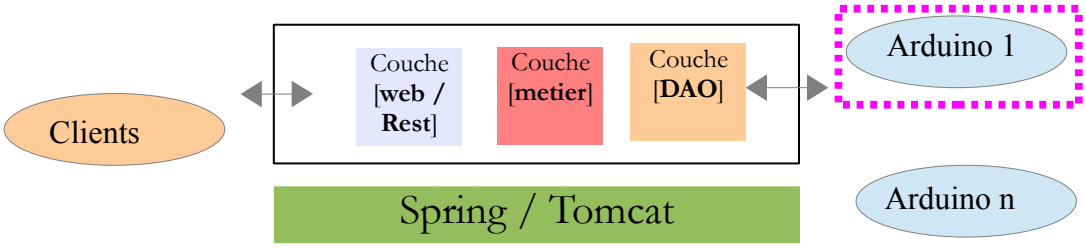
On ne continuera pas les exemples. La règle est simple. L'Arduino **ne doit pas planter**, quelque soit la commande qu'on lui envoie. Avant d'exécuter une commande JSON, il doit s'assurer que celle-ci est correcte. Dès qu'une erreur apparaît, l'Arduino arrête l'exécution de la commande et renvoie à son client la chaîne JSON d'erreur. Là encore, parce qu'on est contraint en espace mémoire, on renverra un code d'erreur plutôt qu'un message complet.

4.2 Enregistrement de l'Arduino

A lire

- dans [**ref2**] : l'utilisation de bibliothèques JSON (Annexes, paragraphes 11.1 et 11.2) ;
- dans l'IDE de l'Arduino, tester l'exemple d'un serveur TCP (le serveur web par exemple) et celui d'un client TCP (le client Telnet par exemple) ;
- les annexes sur l'environnement de programmation des Arduinos au paragraphe 13, page 145.

4.2.1 Le code



Au démarrage, l'Arduino doit s'enregistrer auprès d'un serveur de la couche [DAO] sur le port 102 en lui envoyant une chaîne JSON de la forme :

```
{"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
```

Le code de démarrage de l'Arduino pourrait être le suivant :

```
#include <SPI.h>
#include <Ethernet.h>
#include <aJSON.h>

// ---------------------------------- CONFIGURATION DE L'ARDUINO DUEMILANOVE
// adresse MAC de l'Arduino DUEMILANOVE
byte macArduino[] = {
  0x90, 0xA2, 0xDA, 0x0D, 0xEE, 0xC6 };
char * strMacArduino="90:A2:DA:0D:EE:C6";
// l'adresse IP de l'Arduino
IPAddress ipArduino(192,168,2,3);
// son identifiant=IP
char * idArduino="192.168.2.3";
// port du serveur Arduino
int portArduino=102;
// description de l'Arduino
char * descriptionArduino="duemilanove";
// le serveur Arduino travaillera sur le port 102
EthernetServer server(portArduino);
// IP du serveur d'enregistrement
IPAddress ipServeurEnregistrement(192,168,2,1);
// port du serveur d'enregistrement
int portServeurEnregistrement=100;
// le client Arduino du serveur d'enregistrement
EthernetClient clientArduino;
// la commande du client
char commande[100];
// la réponse de l'Arduino
char message[100];

// initialisation
void setup() {
  // Le moniteur série permettra de suivre les échanges
  Serial.begin(9600);
  // démarrage de la connection Ethernet
  Ethernet.begin(macArduino,ipArduino);
  // mémoire disponible
  Serial.print(F("Memoire disponible : "));
  Serial.println(freeRam());
}

// boucle infinie
void loop()
{
  boolean enregistrementFait=false;
  // tant que l'arduino ne s'est pas enregistré auprès du serveur, on ne peut rien faire
  while(! enregistrementFait){
    // on se connecte au serveur d'enregistrement
    Serial.println(F("Connexion au serveur d'enregistrement..."));
    if (connecte(&clientArduino,macArduino,ipArduino,ipServeurEnregistrement,portServeurEnregistrement)) {
      // suivi
      Serial.println(F("Connecte..."));
      // on envoie l'adresse Mac de l'Arduino, une description de ses fonctionnalités, son port de service
      sprintf(message,"{\"id\":\"%s\",\"desc\":\"%s\",\"mac\":\"%s\",\"port\":
%d}",idArduino,descriptionArduino,strMacArduino,portArduino);
      clientArduino.println(message);
      Serial.print(F("enregistrement : "));
      Serial.println(message);
      // enregistrement fait
      enregistrementFait=true;
      // on ferme la connexion
      delay(1);
      clientArduino.stop();
      // suivi
      Serial.println(F("Enregistrement termine..."));
    }
    else {
      // suivi
      Serial.println(F("Echec de la connexion..."));
    }
  }
  // suivi
```

```
72.   Serial.print(F("Demarrage du serveur sur l'adresse IP "));
73.   Serial.print(Ethernet.localIP());
74.   Serial.print(F(" et le port "));
75.   Serial.println(portArduino);
76.
77.   // on lance le serveur
78.   server.begin();
79.   // on attend indéfiniment les clients
80.   // on les traite un par un
81.   while(true){
82.     // placer votre code ici
83.     ....
84.     }
85. ...
86.   }
87. }
88.
89. // connexion au serveur
90. int connecte(EthernetClient *client, byte macClient[], IPAddress ipClient, IPAddress serveurIP, int
    serveurPort) {
91.   // on se connecte au serveur après 1 seconde d'attente
92.   delay(1000);
93.   return client->connect(serveurIP, serveurPort);
94. }
95.
96. // mémoire libre
97. int freeRam ()
98. {
99.   extern int __heap_start, *__brkval;
100.  int v;
101.  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
102.}
```

- lignes 1-3 : les fichiers headers des fonctions utilisées dans le code ;
- ligne 1 : pour les fonction de base de l'Arduino ;
- ligne 2 : pour les fonction réseau ;
- ligne 3 : pour la bibliothèque JSON d'écriture / lecture des chaînes JSON ;
- lignes 7-8 : adresse Mac de l'Arduino. Mettez votre propre adresse Mac ;
- ligne 9 : idem sous forme de chaîne de caractères ;
- ligne 11 : adresse IP de l'Arduino. Mettez votre propre adresse IP ;
- ligne 13 : idem sous forme de chaîne de caractères ;
- ligne 15 : le port du serveur de l'Arduino ;
- ligne 17 : description de l'Arduino ;
- ligne 19 : le serveur de l'Arduino ;
- ligne 21 : l'adresse IP du serveur d'enregistrement. C'est l'adresse IP locale du PC auquel vous avez relié l'Arduino, normalement 192.168.2.1 ;
- ligne 23 : port du service d'enregistrement des Arduinos ;
- ligne 25 : le client Arduino du service d'enregistrement ;
- ligne 27 : la chaîne JSON de commande du client ;
- ligne 29 : la chaîne JSON de réponse au client ;
- ligne 32 : la fonction [setup] d'initialisation de l'Arduino ;
- ligne 36 : la carte réseau de l'Arduino est initialisée ;
- lignes 38-39 : on logue la mémoire disponible au démarrage de l'application ;
- ligne 38 : on notera la notation **F(chaîne)**. Cela met la chaîne de caractères dans la mémoire Flash de l'Arduino. Ainsi celle-ci n'est pas placée dans la même mémoire que les autres variables du programme. Cela contribue à libérer de la mémoire pour ces dernières ;
- lignes 97-102 : la fonction **freeRam** rend le nombre d'octets disponibles dans la mémoire. Cette fonction a été trouvée sur internet. Au départ, on a environ 1100 octets disponibles sur un Arduino de type " uno " ou " duemilanove ". En-deça de 600 octets le programme " plante ". Il faudra appeler régulièrement cette fonction dans votre code pour voir comment évolue la mémoire ;
- ligne 43 : la boucle infinie exécutée par l'Arduino ;
- lignes 47-70 : ce code est exécuté tant que l'Arduino n'a pas réussi à s'enregistrer auprès du serveur d'enregistrement de la couche [DAO] ;
- ligne 50 : on essaie de se connecter. Les paramètres sont les suivants :
    - l'adresse du client Arduino de la ligne 25,
    - l'adresse IP du serveur d'enregistrement,
    - le port du service d'enregistrement ;
- ligne 93 : notez la façon dont on se connecte à un serveur ;

- ligne 51 : lorsqu'on est là, c'est qu'on a réussi à se connecter au serveur d'enregistrement. On rappelle que l'Arduino doit alors envoyer une ligne JSON du genre :

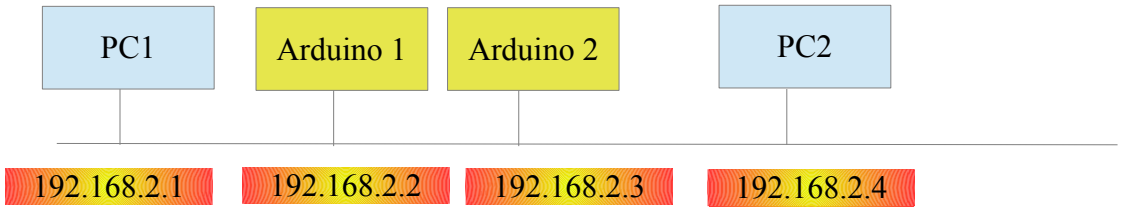
```
{"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
```

- ligne 54 : cette ligne est préparée ;
- ligne 55 : elle est envoyée au serveur d'enregistrement ;
- ligne 62 : on ferme la connexion avec le serveur d'enregistrement ;
- ligne 71 : l'enregistrement est terminé. Maintenant l'Arduino va exécuter les commandes que va lui envoyer la couche [DAO]. Pour cela, on crée un serveur ;
- ligne 78 : un serveur est lancé sur le port 102 de l'Arduino ;
- lignes 81-84 : dans une boucle infinie, vous allez traiter les commandes envoyées au serveur.

4.2.2 Les tests

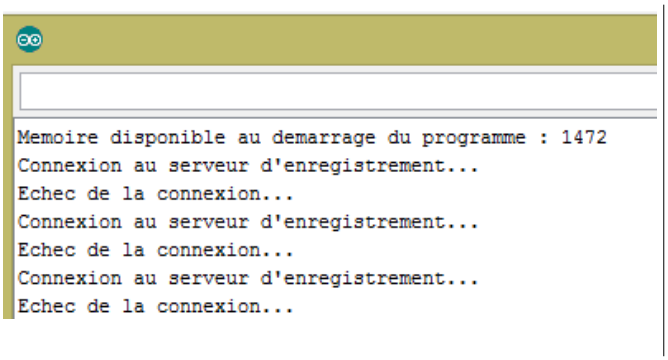
Nous allons tester cette première mouture du code de l'Arduino. On va vérifier qu'il s'enregistre bien auprès du serveur d'enregistrement qui opère sur le port 100 de la machine hôte.

Tout d'abord, vérifiez votre réseau local. Il doit être le suivant :



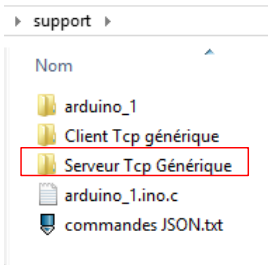
Ensuite, chargez dans l'IDE Arduino, le programme [arduino_1.ino] que vous trouverez dans le dossier [support] du TP :

Téléversez-le sur l'Arduino, puis faites afficher la console de logs (Ctrl-M) :



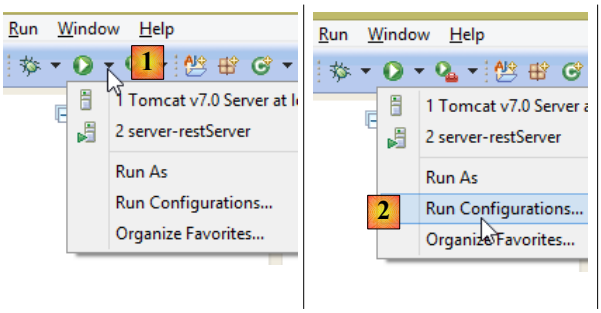
Toutes les secondes, l'Arduino essaie de se connecter au serveur d'enregistrement qui n'existe pas encore. Nous allons en créer un.

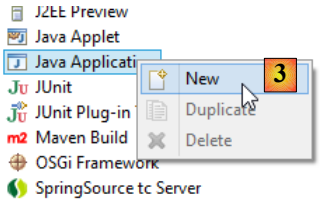
Sous Eclipse, importez le projet Maven [Serveur Tcp générique] du dossier [support] :



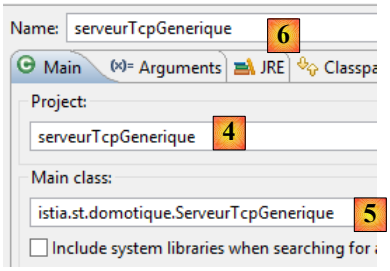
Ce projet Maven crée un serveur générique sur un port de la machine hôte qu'on lui passe en paramètre. Ce serveur affiche sur la console toutes les commandes qu'il reçoit. Nous allons le lancer sur le port 100 de la machine hôte, là où doit opérer le serveur d'enregistrement des Arduinos. On va voir ainsi si l'Arduino arrive à se connecter et à envoyer la ligne de texte qui l'identifie.

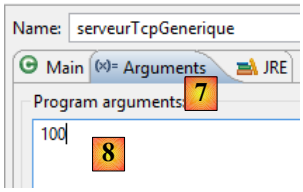
Il nous faut configurer le projet Eclipse :

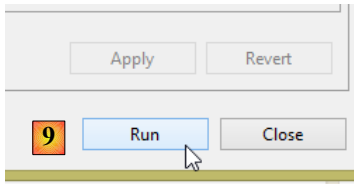




- en [1,2], créez une nouvelle configuration d'exécution de type [Java Application] [3] ;







- en [4], sélectionnez le projet [serveurTcpGenerique] ;
- en [5], sélectionnez sa classe principale, celle qui a la méthode [main] ;
- en [6], donnez un nom à la configuration d'exécution que vous êtes en train de créer ;
- dans l'onglet [Arguments] [7], mettez 100 [8] pour indiquer que le serveur doit s'exécuter sur le port 100 de la machine hôte ;

Exécutez la configuration [9]. Le serveur s'exécute et affiche des logs dans la console :

```
1. Serveur générique lancé sur le port 100
2. Thread de lecture des réponses du serveur au client 1 lancé
3. Thread de lecture des demandes du client 1 lancé
4. --> 1 : <-- {"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
5. [fin du Thread de lecture des demandes du client 1]
```

- ligne 1 : le serveur a été lancé ;
- entre les lignes 1 et 2, un client a été détecté : l'Arduino ;

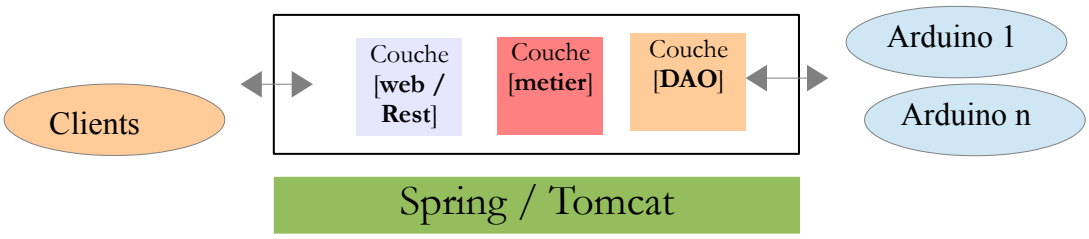
- lignes 2-3 : deux threads sont lancés : un pour lire les demandes du client (ligne 3), l'autre pour lire les lignes tapées par l'utilisateur au clavier (ligne 2). Ici, le client n'attend pas de réponse, donc nous n'aurons pas à taper une réponse au clavier ;
- ligne 4 : le serveur affiche la commande que l'Arduino lui a envoyée. C'est bien la ligne attendue ;
- ligne 5 : le client (l'Arduino) a fermé sa connexion. Ceci a pour conséquence de fermer le thread lancé ligne 3 ;

Maintenant consultons la console de logs de l'Arduino :

```
1. Connexion au serveur d'enregistrement...
2. Connecte...
3. enregistrement :
   {"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
4. Enregistrement termine...
5. Demarrage du serveur sur l'adresse IP 192.168.2.3 et le port 102
6. Memoire disponible avant service : 1286
```

- ligne 1 : toutes les secondes l'Arduino essaie de se connecter au serveur d'enregistrement ;
- ligne 2 : il y arrive ;
- ligne 3 : la ligne qu'il a envoyée au serveur d'enregistrement ;
- ligne 4 : l'enregistrement est terminé. L'Arduino va fermer sa connexion avec le serveur d'enregistrement ;
- ligne 5 : le serveur qui opère sur le port 102 de l'Arduino est alors lancé. Il va exécuter toutes les commandes que la couche [DAO] va lui envoyer ;
- ligne 6 : la mémoire disponible. On a déjà perdu 1472-1286=186 octets ;

4.3 Exécution des commandes de la couche [DAO]



Le serveur qui vient d'être lancé sur le port 102 de l'Arduino va exécuter les commandes JSON que la couche [DAO] va lui envoyer.

A lire

Dans les exemples de l'IDE Arduino :
- le code du serveur web [Ethernet / Webserver] ;
- le code de l'exemple [aJson / Examples / MultiLevelParsing] ;

4.3.1 Le code

Le traitement des commandes envoyées à l'Arduino pourrait ressembler à ce qui suit (cf *while* ligne 81 page 22):

```
1. ..............
2.   // on attend indéfiniment les clients
3.   // on les traite un par un
4.   while(true){
5.     // mémoire disponible
6.     Serial.print(F("Memoire disponible avant service : "));
7.     Serial.println(freeRam());
8.     // on attend un client
9.     EthernetClient client=server.available();
10.    while(! client  ){
11.      delay(1);
12.      client=server.available();
13.    }
```

```
14.     // si et tant que ce client est connecté
15.     while(client.connected()){
16.       Serial.println(F("client connecte..."));
17.       // mémoire disponible
18.       Serial.print(F("Memoire disponible debut boucle : "));
19.       Serial.println(freeRam());
20.
21.       // on lit la commande de ce client
22.       lireCommande(&client,commande);
23.       // suivi
24.       Serial.print(F("commande : ["));
25.       Serial.print(commande);
26.       Serial.println(F("]"));
27.
28.       // si la commande est vide, on ne la traite pas
29.       if(strlen(commande)==0){
30.         continue;
31.       }
32.       // sinon on la traite
33.       traiterCommande(&client,commande);
34.       // mémoire disponible
35.       Serial.print(F("Memoire disponible fin boucle : "));
36.       Serial.println(freeRam());
37.
38.       // on passe à la commande suivante
39.       delay(1);
40.     }
41.     // on est déconnecté - on passe au client suivant
42.     Serial.println(F("client deconnecte..."));
43.     client.stop();
44.     // mémoire disponible
45.     Serial.print(F("Memoire disponible fin service : "));
46.     Serial.println(freeRam());
47.   }
48.   // on ne rebouclera jamais sur le loop (normalement)
49.   Serial.println(F("loop"));
50. }
51.
52. // traitement d'une commande
53. void traiterCommande(EthernetClient *client, char * commande){
54.   // on décode la commande pour voir quelle action est demandée
55.   ...
56.   // pb ?
57.   if(json==NULL){
58.     sendReponse(client,reponse(NULL,"100",NULL, message));
59.     return;
60.   }
61.   // attribut id
62.   ...
63.   if(id==NULL){
64.     // suppression json
65.     aJson.deleteItem(json);
66.     // on envoie la réponse
67.     sendReponse(client,reponse(NULL,"101",NULL, message));
68.     // retour
69.     return;
70.   }
71.   // on mémorise l'id
72.   ...
73.   // attribut action
74.   ...
75.   if(action==NULL){
76.     // suppression json
```

```
77.     aJson.deleteItem(json);
78.     // on envoie la réponse
79.     sendReponse(client,reponse(NULL,"102",NULL,message));
80.     // retour
81.     return;
82.   }
83.   // on mémorise l'action
84.   ...
85.   // on récupère les parametres
86.   ...
87.   if(parametres==NULL){
88.     // suppression json
89.     aJson.deleteItem(json);
90.     // on envoie la réponse
91.     sendReponse(client,reponse(NULL,"103",NULL,message));
92.     // retour
93.     return;
94.   }
95.   // c'est bon - on traite l'action
96.   // echo
97.   if(strcmp("ec",strAction)==0){
98.     // traitement
99.     doEcho(client, strId);
100.      // suppression json
101.      aJson.deleteItem(json);
102.      // retour
103.      return;
104.    }
105.    // clignotement
106.    if(strcmp("cl",strAction)==0){
107.      // traitement
108.      doClignoter(client,strId,parametres);
109.      // suppression json
110.      aJson.deleteItem(json);
111.      // retour
112.      return;
113.    }
114.    // pinWrite
115.    if(strcmp("pw",strAction)==0){
116.      // traitement
117.      doPinWrite(client,strId,parametres);
118.      // suppression json
119.      aJson.deleteItem(json);
120.      // retour
121.      return;
122.    }
123.    // pinRead
124.    if(strcmp("pr",strAction)==0){
125.      // traitement
126.      doPinRead(client,strId,parametres);
127.      // suppression json
128.      aJson.deleteItem(json);
129.      // retour
130.      return;
131.    }
132.
133.    // ici, l'action n'est pas reconnue
134.    // suppression json
135.    aJson.deleteItem(json);
136.    // réponse
137.    sendReponse(client, reponse(strId,"104",NULL,message));
138.    // retour
139.    return;
```

```
140.  }
141.
142.  // connexion au serveur
143.  int connecte(EthernetClient *client, byte macClient[], IPAddress ipClient, IPAddress
    serveurIP, int serveurPort) {
144.    // on se connecte au serveur après 1 seconde d'attente
145.    delay(1000);
146.    return client->connect(serveurIP, serveurPort);
147.  }
148.
149.  // lecture d'une commande du serveur
150.  void lireCommande(EthernetClient *client, char *commande){
151.    // les caractères reçus sont cumulés dans [commande]
152.    ...
153.  }
154.
155.  // formatage json de la réponse au client
156.  char * reponse(char * id, char *  erreur, char * etat, char * message){
157.    // attribut id
158.    char *id2="";
159.    if(id!=NULL){
160.      id2=id;
161.    }
162.    // attribut erreur
163.    char *erreur2="";
164.    if(erreur!=NULL){
165.      erreur2=erreur;
166.    }
167.    // attribut etat - pour l'instant dictionnaire vide
168.    char *etat2="{}";
169.    if(etat!=NULL){
170.      etat2=etat;
171.    }
172.    // construction de la réponse json au client
173.    sprintf(message,"{\"id\":\"%s\",\"er\":\"%s\",\"et\":%s}",id2,erreur2,etat2);
174.    // résultat
175.    return message;
176.  }
177.
178.  // echo
179.  void doEcho(EthernetClient *client, char * strId){
180.    // on fait l'écho
181.    sendReponse(client, reponse(strId,"0",NULL,message));
182.  }
183.
184.  //clignoter
185.  void doClignoter(EthernetClient *client,char * strId, aJsonObject* parametres){
186.    // exploitation des parametres
187.    // il faut une pin
188.    ...
189.    if(pin==NULL){
190.      // réponse d'erreur
191.      sendReponse(client, reponse(strId,"202",NULL,message));
192.      return;
193.    }
194.    // valeur de la pin à faire clignoter
195.    ...
196.    Serial.print(F("clignoter led="));
197.    Serial.println(led);
198.    // il faut la durée d'un clignotement
199.    ...
200.    if(dur==NULL){
201.      // réponse d'erreur
```

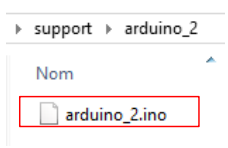
```
202.      sendReponse(client, reponse(strId,"211",NULL,message));
203.      return;
204.    }
205.    // valeur de la durée
206.    ...
207.    Serial.print(F("duree="));
208.    Serial.println(duree);
209.    // il faut le nombre de clignotements
210.    ...
211.    if(nb==NULL){
212.      // réponse d'erreur
213.      sendReponse(client, reponse(strId,"212",NULL,message));
214.      return;
215.    }
216.    // valeur du nombre de clignotements
217.    ...
218.    Serial.print(F("nb="));
219.    Serial.println(nbClignotements);
220.
221.    // on rend la réponse tout de suite
222.    sendReponse(client, reponse(strId,"0",NULL,message));
223.
224.    // on opère le clignotement
225.    ...
226.    }
227.  }
228.
229.  // pinWrite
230.  void doPinWrite(EthernetClient *client, char * strId, aJsonObject* parametres){
231.    // il faut une pin
232.    ...
233.    if(pin==NULL){
234.      // réponse d'erreur
235.      sendReponse(client, reponse(strId,"201",NULL,message));
236.      return;
237.    }
238.    // numéro de la pin à écrire
239.    ...
240.    // suivi
241.    Serial.print(F("pw pin="));
242.    Serial.println(pin2);
243.    // il faut une valeur
244.    ...
245.    if(val==NULL){
246.      // réponse d'erreur
247.      sendReponse(client, reponse(strId,"202",NULL,message));
248.      return;
249.    }
250.    // valeur à écrire
251.    ...
252.    // suivi
253.    Serial.print(F("pw val="));
254.    Serial.println(val2);
255.    // il faut un mode d'écriture
256.    ...
257.    if(mod==NULL){
258.      // réponse d'erreur
259.      sendReponse(client, reponse(strId,"203",NULL,message));
260.      return;
261.    }
262.    ...
263.    // ce doit être a (analogique) ou b (binaire)
264.    if(...){
```

```
265.      // réponse d'erreur
266.      sendReponse(client, reponse(strId,"204",NULL,message));
267.      return;
268.    }
269.    // c'est bon pas d'erreur
270.    sendReponse(client,reponse(strId,"0",NULL,message));
271.    // suivi
272.    Serial.print(F("pw mod="));
273.    Serial.println(mod2);
274.    // on écrit sur la pin
275.    ...
276.  }
277.
278.  //pinRead
279.  void doPinRead(EthernetClient *client,char * strId, aJsonObject* parametres){
280.    // exploitation des parametres
281.    // il faut une pin
282.    ...
283.    if(pin==NULL){
284.      // réponse d'erreur
285.      sendReponse(client, reponse(strId,"302",NULL,message));
286.      return;
287.    }
288.    // numéro de la pin à lire
289.    ...
290.    // suivi
291.    Serial.print(F("pr pin="));
292.    Serial.println(pin2);
293.    // il faut un mode d'écriture
294.    ...
295.    if(mod==NULL){
296.      // réponse d'erreur
297.      sendReponse(client, reponse(strId,"303",NULL,message));
298.      return;
299.    }
300.    ...
301.    // ce doit être a (analogique) ou b (binaire)
302.    if(...){
303.      // réponse d'erreur
304.      sendReponse(client, reponse(strId,"304",NULL,message));
305.      return;
306.    }
307.    // c'est bon pas d'erreur
308.    // suivi
309.    Serial.print(F("pw mod="));
310.    Serial.println(mod2);
311.    // on lit la pin
312.  ...
313.    // on rend le résultat
314.    char json[20];
315.    sprintf(json,"{\"pin%d\":\"%d\"}",pin2,valeur);
316.    sendReponse(client, reponse(strId,"0",json,message));
317.  }
318.
319.  // envoi réponse
320.  void sendReponse(EthernetClient *client, char * message){
321.    // envoi de la réponse
322.    client->println(message);
323.    // suivi
324.    Serial.print(F("reponse="));
325.    Serial.println(message);
326.    // mémoire disponible
327.    Serial.print(F("Memoire disponible en fond de pile : "));
```

```
328.     Serial.println(freeRam());
329.   }
330.
331.   // mémoire libre
332.   int freeRam ()
333.   {
334.     extern int __heap_start, *__brkval;
335.     int v;
336.     return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
337.   }
```

Question 1 : Ecrire le code complet de l'Arduino. Vous pouvez suivre l'ossature proposée ou bien opter pour votre propre solution qui devra respecter les spécifications.

Note : vous trouverez le code précédent dans le dossier [support du TP] :

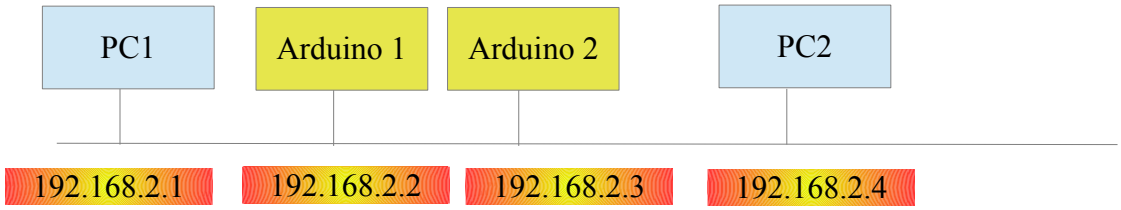


4.3.2 Tests de la couche [Arduino]

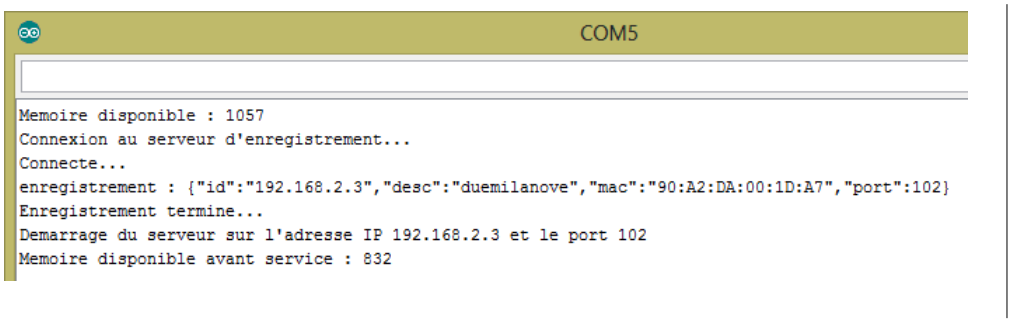
Pour tester la couche [Arduino], il faut lui envoyer des commandes JSON et voir comment elles sont traitées. On pourra procéder comme suit :

- mettez en place le serveur Java qui va permettre l'enregistrement de l'Arduino (voir page 23, paragraphe 4.2.2) ;

Vérifiez votre réseau local. Il doit être le suivant :

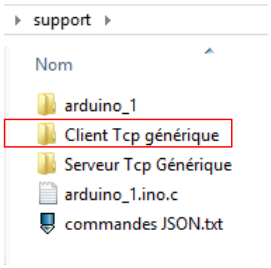


Téléversez votre programme sur l'Arduino, puis faites afficher la console de logs (Ctrl-M) :



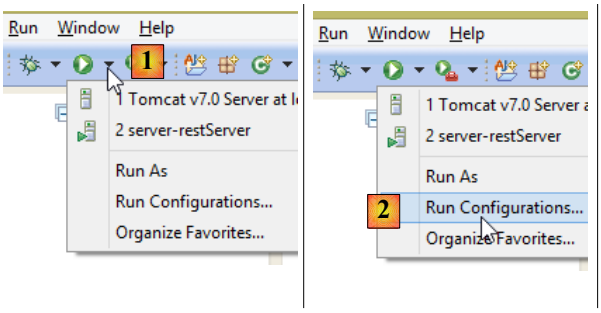
Ci-dessus, l'Arduino s'est enregistré auprès du serveur d'enregistrement. Il attend désormais des commandes JSON sur le port 102 de l'Arduino.

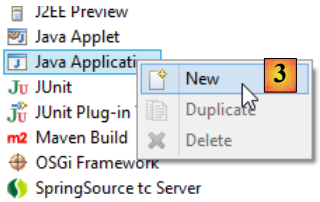
Sous Eclipse, importez le projet Maven [Client Tcp générique] du dossier [support] :



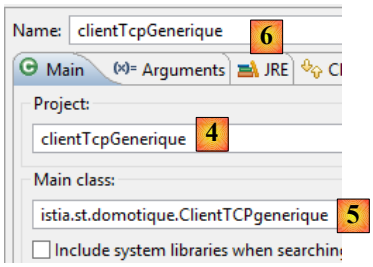
Ce projet Maven crée un client générique TCP pour une machine distante et un port qu'on lui passe en paramètres. Ce client permet d'envoyer des lignes de texte à la machine distante et affiche ce que celle-ci renvoie. Nous allons le connecter au port 102 de l'Arduino.

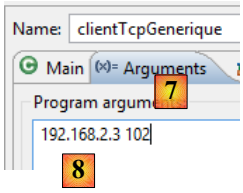
Il nous faut configurer le projet Eclipse :

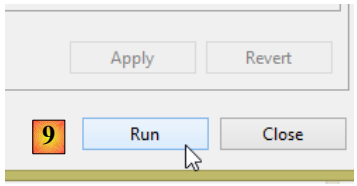




- en [1,2], créez une nouvelle configuration d'exécution de type [Java Application] [3] ;

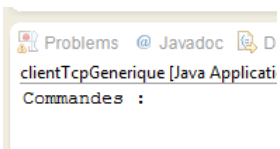






- en [4], sélectionnez le projet [clientTcpGenerique] ;
- en [5], sélectionnez sa classe principale, celle qui a la méthode [main] ;
- en [6], donnez un nom à la configuration d'exécution que vous êtes en train de créer ;
- dans l'onglet [Arguments] [7], mettez *IPArduino 102* [8] où *IPArduino* est l'adresse IP de votre Arduino. Cela va connecter le client générique au port 102 de l'Arduino. C'est là qu'il attend les commandes JSON ;

Exécutez la configuration [9]. Si le client générique arrive à se connecter à l'Arduino, il affiche la chose suivante sur la console [1] :



Maintenant dans la console, tapez la commande suivante qui fait clignoter la led de la pin n° 8 :

```
{"id":"1","ac":"cl","pa":{"pin":"8","dur":"100","nb":"10"}}
```

La led doit clignoter 10 fois et l'Arduino doit renvoyer la réponse JSON suivante :

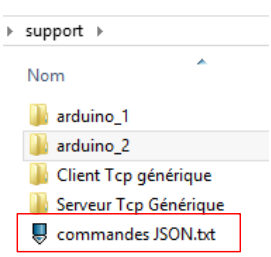
```
<-- {"id":"1","er":"0","et":{}}
```

qui indique que la commande a été exécutée correctement.

Sur la console de l'Arduino, des logs affichent l'échange qui vient d'avoir lieu :

```
1.  client connecte...
2.  Memoire disponible debut boucle : 832
3.  commande : [{"id":"1","ac":"cl","pa":{"pin":"8","dur":"100","nb":"10"}}]
4.  clignoter led=8
5.  duree=100
6.  nb=10
7.  reponse={"id":"1","er":"0","et":{}}
8.  Memoire disponible fin boucle : 617
9.  client connecte...
10. Memoire disponible debut boucle : 617
```

Exécutez les commandes JSON que vous trouverez dans le dossier [support] du TP :

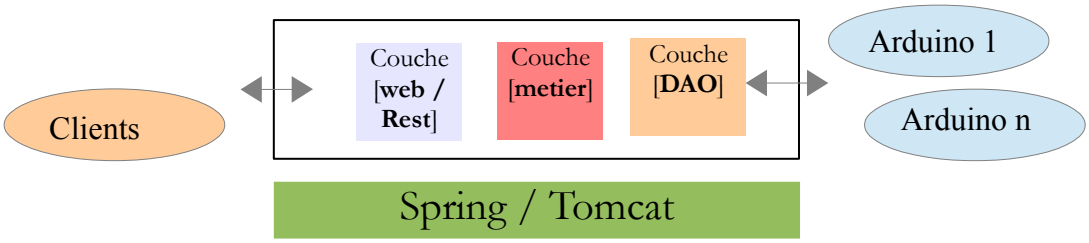


Vous y trouverez les trois commandes attendues par l'Arduino ainsi qu'un lot de commandes erronées. Pour ces dernières, l'Arduino doit renvoyer un code d'erreur significatif de l'erreur faite. Pour chaque commande, consultez la console de logs de l'Arduino.

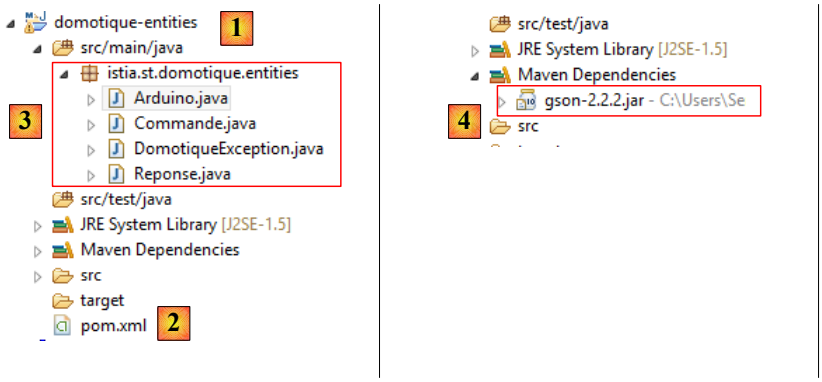
Nous avons terminé la programmation des Arduinos. Nous n'y reviendrons plus. Nous allons passer maintenant à la construction du serveur de l'application.

5 Les entités du serveur

Revenons à l'architecture de notre application client / serveur. L'architecture du serveur est la suivante :



La couche [DAO] va communiquer avec les Arduinos. Elle va échanger avec elle des lignes de texte JSON. Côté serveur, ces lignes de texte vont être transformées en objets que nous présentons maintenant. Par ailleurs, le serveur échange également avec ses clients (Android ou web) des objets. Nous les appellerons des entités. Elles sont présentées maintenant et vous seront fournies :



- en [1], le projet Eclipse des entités ;
- en [2], c'est un projet Maven. Il n'y a qu'une dépendance, celle sur la bibliothèque JSON Gson de Google [4] ;
- en [3], les entités ;

La couche [DAO] envoie à l'Arduino des commandes et en reçoit des réponses. Ces deux entités sont représentées par les classes [Commande] et [Reponse].

5.1 La classe [Commande]

C'est la suivante :

```
1.  package istia.st.domotique.entities;
2.
3.  ...
4.
5.  public class Commande implements Serializable {
6.     // données
7.     private static final long serialVersionUID = 1L;
8.     private String id;
9.     private String action;
10.    private Map<String, Object> parametres;
11.    private Map<String, Object> map = new HashMap<String, Object>();
12.
13.    // constructeurs
14.    public Commande() {
15.    }
16.
17.    public Commande(String id, String action, Map<String, Object> parametres) {
18.       // initialisation
```

```
19.       this.id = id;
20.       this.action = action;
21.       this.parametres = parametres;
22.       // construction de la map
23.       map.put("id", id);
24.       map.put("ac", action);
25.       map.put("pa", parametres);
26.    }
27.
28.    // construction d'une commande à partir d'une chaîne Json
29.    @SuppressWarnings("unchecked")
30.   public Commande(String json) throws Exception {
31.       // parsing de la chaîne json
32.       Map<String, Object> map = new Gson().fromJson(json, new
    TypeToken<Map<String,Object>>() {
33.      }.getType());
34.       // on récupère les valeurs du dictionnaire
35.       this.id = (String) map.get("id");
36.       this.action = (String) map.get("ac");
37.       this.parametres = (Map<String, Object>) map.get("pa");
38.    }
39.
40.    @Override
41.    public String toString() {
42.       // on rend la commande au format JSON
43.       try {
44.          // on rend la commande au format JSON
45.          return new Gson().toJson(this);
46.       } catch (Exception e) {
47.          return "Problème de conversion en JSON : " + e;
48.       }
49.    }
50.
51.    // conversion en Json
52.    public String toJson() throws IOException {
53.       // on rend la commande au format JSON
54.       return new Gson().toJson(map);
55.    }
56.
57.    // getters et setters
58.    ...
59. }
```

On se rappelle que les lignes JSON envoyées à l'Arduino sont de la forme :

```
{"id":"...","ac":"...","pa":{...}}
```

- [id] est l'identifiant de la commande ;
- [ac] est le nom de l'action à exécuter ;
- [pa] est un dictionnaire listant les paramètres de l'action ;

Les trois paramètres [id, ac, pa] sont définis aux lignes 8-10 de la classe. Si on sérialise en JSON cette classe on aura une chaîne comme suit :

```
{"id":"...","action":"...","parametres":{...}}
```

Parce que l'Arduino a très peu de mémoire, on ne peut envoyer cette chaîne trop longue. On décide donc d'envoyer celle montrée plus haut. Pour cela, on crée dans la classe un dictionnaire avec les attributs [id, ac, pa]. Il est défini ligne 11. Lorsqu'on voudra envoyer une commande, c'est la chaîne JSON de ce dictionnaire qui sera envoyée.

- ligne 30 : la classe possède un constructeur qui accepte pour paramètre la chaîne JSON d'un dictionnaire ayant les attributs [id, ac, pa] ;

- ligne 52 : lorsqu'on veut envoyer la chaîne JSON de la commande, c'est celle de son dictionnaire interne qui sera envoyée (ligne 54) ;
- ligne 41 : la méthode [toString] rend la chaîne JSON de la classe ;

5.2 La classe [Reponse]

C'est la suivante :

```
1.  package istia.st.domotique.entities;
2.
3.  ...
4.
5.  public class Reponse implements Serializable {
6.    // données
7.
8.    private static final long serialVersionUID = 1L;
9.    private String json;
10.   private String id;
11.   private String erreur;
12.   private Map<String, Object> etat;
13.
14.   // constructeurs
15.   public Reponse() {
16.   }
17.
18.   public Reponse(String json, String id, String erreur, Map<String, Object> etat) {
19.     this.json = json;
20.     this.id = id;
21.     this.erreur = erreur;
22.     this.etat = etat;
23.   }
24.
25.   @SuppressWarnings("unchecked")
26.   public Reponse(String json) {
27.     Map<String, Object> map = null;
28.     try {
29.       // parsing de la chaîne json
30.       map = new Gson().fromJson(json, new TypeToken<Map<String, Object>>() {
31.       }.getType());
32.       this.id = (String) map.get("id");
33.       this.erreur = (String) map.get("er");
34.       this.etat = (Map<String, Object>) map.get("et");
35.     } catch (Exception ex) {
36.       this.json = json;
37.       this.erreur = "json invalide";
38.     }
39.   }
40.
41.   // signature
42.
43.   @Override
44.   public String toString() {
45.     try {
46.       // on rend la commande au format JSON
47.       return new Gson().toJson(this);
48.     } catch (Exception e) {
49.       return "Problème de conversion en JSON : " + e;
50.     }
51.   }
52.
53.   // getters et setters
54. ...
```

```
55. }
```

L'Arduino renvoie des lignes JSON de la forme :

```
{"id":"2","er":"303","et":{}}
```

ou

```
{"id":"4","er":"0","et":{"pin0":"1023"}}
```

- [id] est l'identifiant de la commande à laquelle on répond ;
- [er] est un code d'erreur ;
- [et] est un dictionnaire listant les résultats de l'action ;

Les attributs [id, er, et] sont définis lignes 10-12 de la classe.

- ligne 26 : l'objet [Reponse] est construit à partir de la chaîne JSON envoyée par l'Arduino ;
- lignes 27, 30 : on construit d'abord un dictionnaire ;
- lignes 32-34 : l'objet [Reponse] est construit ensuite à partir de ce dictionnaire ;
- ligne 35 : si la chaîne JSON envoyée par l'Arduino n'a pas pu être parsée, on aura une exception ;
- ligne 36 : on mémorise alors la chaîne JSON invalide dans le champ de la ligne [9] ;
- ligne 44 : la méthode [toString] affiche la chaîne JSON de l'objet. On aura donc ainsi la chaîne JSON invalide si ce cas se produit ;

5.3 La classe [Arduino]

On se rappelle que l'Arduino, en s'enregistrant auprès de la couche [DAO] envoie la chaîne d'identification suivante :

```
{"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
```

On enregistre ces informations dans la classe [Arduino] suivante :

```
1.  package istia.st.domotique.entities;
2.
3.  import java.io.Serializable;
4.
5.  public class Arduino implements Serializable{
6.    // données
7.
8.     private static final long serialVersionUID = 1L;
9.     private String id;
10.   private String description;
11.   private String mac;
12.   private String ip;
13.   private int port;
14.
15.   // identité
16.   public String toString() {
17.     return String.format("id=[%s], description=[%s], mac=[%s], ip=[%s], port=[%s]", id,
    description, mac, ip, port);
18.   }
19.   // getters et setters
20. ...
21. }
```

- les informations des lignes [9-11,13] sont tirées de la ligne envoyée par l'Arduino ;
- la ligne 12 qui représente l'IP de l'Arduino sera renseignée par le serveur d'enregistrement qui traite la connexion de l'Arduino ;

5.4 La classe [DomotiqueException]

Les différentes couches du serveur lanceront des exceptions non contrôlées par le compilateur de type [DomotiqueException] :

```
1.  package istia.st.domotique.entities;
2.
3.  import java.io.Serializable;
4.
5.  public class DomotiqueException extends RuntimeException implements Serializable{
6.
7.    private static final long serialVersionUID = 1L;
8.  // champs privés
9.    private int code = 0;
10. // constructeurs
11.
12.   public DomotiqueException() {
13.     super();
14.   }
15.
16.   public DomotiqueException(String message) {
17.     super(message);
18.   }
19.
20.   public DomotiqueException(String message, Throwable cause) {
21.     super(message, cause);
22.   }
23.
24.   public DomotiqueException(Throwable cause) {
25.     super(cause);
26.   }
27.
28.   public DomotiqueException(String message, int code) {
29.     super(message);
30.     setCode(code);
31.   }
32.
33.   public DomotiqueException(Throwable cause, int code) {
34.     super(cause);
35.     setCode(code);
36.   }
37.
38.   public DomotiqueException(String message, Throwable cause, int code) {
39.     super(message, cause);
40.     setCode(code);
41.   }
42. // getters - setters
43. ...
44.   }
```

C'est une classe d'exception standard si ce n'est qu'elle encapsule un code d'erreur, ligne 9.

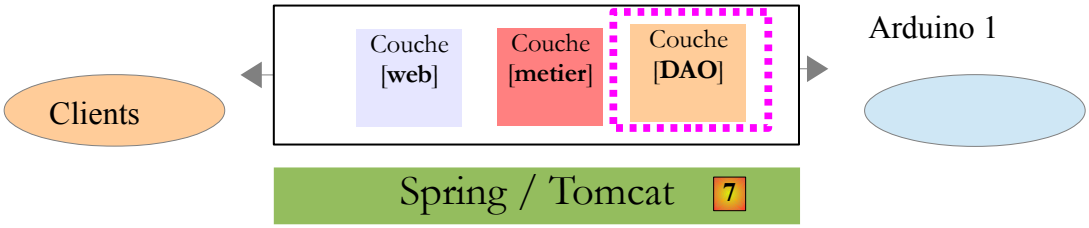
6 Le serveur d'enregistrement des Arduinos

A lire

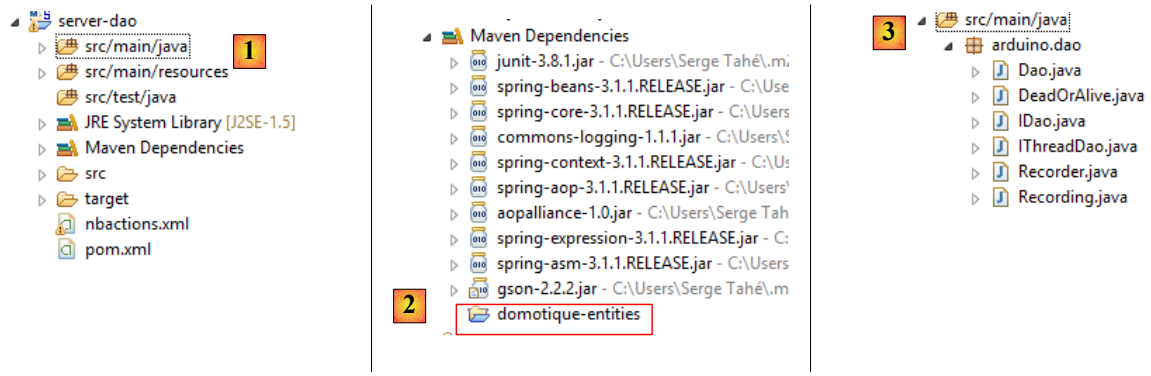
- dans [**ref4**] : chapitre 6 [**Les threads d'exécution**], chapitre 7 [**Programmation TCP / IP**] ;
- le code du serveur TCP générique présenté page 23, paragraphe 4.2.2 ;
- le code du client TCP générique présenté page 31, paragraphe 4.3.2.

6.1 Le projet Eclipse

Revenons à l'architecture du serveur :



Le projet Eclipse de la couche [DAO] est le suivant :



- en [1] : le projet Eclipse ;
- en [2] : les dépendances Maven du projet. On notera les dépendances sur les entités du serveur ;
- en [3] : les éléments de la couche [DAO] ;

6.2 Les dépendances Maven

Le fichier [pom.xml] du projet est le suivant :

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.    <modelVersion>4.0.0</modelVersion>
4.
5.    <groupId>istia.st.domotique.server</groupId>
6.    <artifactId>server-dao</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>server-dao</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <properties>
```

```
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.   <dependencies>
18.     <dependency>
19.       <groupId>org.springframework</groupId>
20.       <artifactId>spring-beans</artifactId>
21.       <version>3.1.1.RELEASE</version>
22.       <type>jar</type>
23.     </dependency>
24.     <dependency>
25.       <groupId>org.springframework</groupId>
26.       <artifactId>spring-context</artifactId>
27.       <version>3.1.1.RELEASE</version>
28.       <type>jar</type>
29.     </dependency>
30.     <dependency>
31.       <groupId>istia.st.domotique.common</groupId>
32.       <artifactId>domotique-entities</artifactId>
33.       <version>1.0-SNAPSHOT</version>
34.     </dependency>
35.   </dependencies>
36.
37.   <repositories>
38.     <repository>
39.       <id>codehaus</id>
40.       <url>http://repository.codehaus.org/org/codehaus</url>
41.     </repository>
42.   </repositories>
43. </project>
```

- lignes 5-8 : l'identité Maven du projet ;
- lignes 30-34 : la dépendance sur le projet Maven des entités ;
- lignes 18-29 : deux dépendances sur le framework Spring. Celui-ci est utilisé pour configurer et instancier la couche [DAO] ;

6.3 L'interface de la couche [DAO]

L'interface [IDao] de la couche [DAO] sera la suivante :

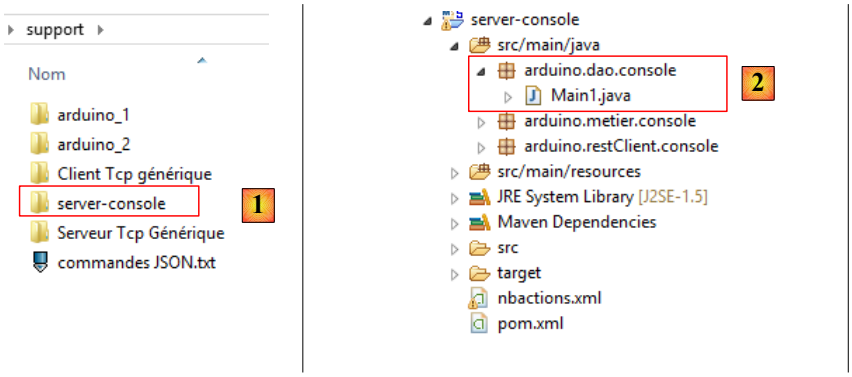
```
1.  package arduino.dao;
2.
3.  ...
4.
5.  public interface IDao {
6.    // liste des Arduinos
7.    public Collection<Arduino> getArduinos();
8.    // ajout d'un arduino
9.    public void addArduino(Arduino arduino);
10.   // suppression d'un arduino
11.   public void removeArduino(String idArduino);
12.   // envoyer une suite de commandes à un Arduino
13.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes);
14.   // envoyer une suite de commandes Json à un Arduino
15.   public List<String> sendCommandesJson(String idArduino, List<String> commandes);
16. }
```

- ligne 13 : la couche [DAO] peut envoyer une liste de commandes de type [Commande] à un Arduino identifié par son *id*. Elle reçoit une liste de réponses de type [Reponse] ;
- ligne 15 : la couche [DAO] peut envoyer une liste de commandes au format texte JSON à un Arduino identifié par son *id*. Elle reçoit une liste de réponses texte au format JSON ;
- ligne 7 : la couche [DAO] gère la liste des Arduinos qui se sont connectés. Cette méthode en donne la liste ;

- ligne 9 : pour ajouter un Arduino à la liste ;
- ligne 11 : pour enlever un Arduino de la liste ;

6.4 Tests de la couche [DAO]

Dans le dossier [support] du TP, vous trouverez un projet Eclipse qui va vous permettre de tester les différents éléments du serveur :



- en [1] : le projet console de tests ;
- en [2] : la classe [Main1] permet de tester la couche [DAO] ;

La classe [Main1] est la suivante :

```
1.  package arduino.dao.console;
2.
3.  ...
4.
5.  import arduino.dao.IDao;
6.
7.  public class Main1 {
8.
9.     public static void main(String[] args) throws InterruptedException {
10.
11.       // création de la couche [dao]
12.       IDao dao = (IDao) new
    ClassPathXmlApplicationContext("applicationContext.xml").getBean("dao");
13.
14.       // allumer la led
15.       List<Commande> allumer = new ArrayList<Commande>();
16.       // allumage pin 13
17.       Map<String, Object> paramètres = new HashMap<String, Object>();
18.       paramètres.put("pin", "8");
19.       paramètres.put("val", "1");
20.       paramètres.put("mod", "b");
21.       allumer.add(new Commande("0", "pw", paramètres));
22.
23.       // éteindre la led
24.       List<Commande> eteindre = new ArrayList<Commande>();
25.       // extinction pin 13
26.       paramètres = new HashMap<String, Object>();
27.       paramètres.put("pin", "8");
28.       paramètres.put("val", "0");
29.       paramètres.put("mod", "b");
30.       eteindre.add(new Commande("1", "pw", paramètres));
31.
32.       // état d'une pin
33.       List<Commande> getPin = new ArrayList<Commande>();
34.       // état pin 7
```

```
35.       paramètres = new HashMap<String, Object>();
36.       paramètres.put("pin", "0");
37.       paramètres.put("mod", "a");
38.       getPin.add(new Commande("2", "pr", paramètres));
39.
40.       // boucle infinie de dialogue avec les arduinos connectés
41.       boolean éteinte = true;
42.       List<Commande> commandes;
43.       while (true) {
44.          // attente
45.          System.out.println("Main : attente...");
46.          Thread.sleep(1000);
47.          // allumage / extinction
48.          if (éteinte) {
49.             commandes = allumer;
50.          } else {
51.             commandes = eteindre;
52.          }
53.          // on bascule
54.          éteinte = !éteinte;
55.
56.          // on envoie les commandes aux arduinos enregistrés
57.          for (Arduino arduino : dao.getArduinos()) {
58.             // action sur la pin
59.             sendCommandes(dao, arduino.getId(), commandes);
60.             // état de la pin
61.             sendCommandes(dao, arduino.getId(), getPin);
62.          }
63.       }
64.    }
65.
66.    private static void sendCommandes(IDao dao, String idarduino, List<Commande> commandes) {
67.       try {
68.          // envoie les commandes à l'Arduino
69.          dao.sendCommandes(idarduino, commandes);
70.       } catch (Exception e) {
71.       }
72.    }
73. }
```

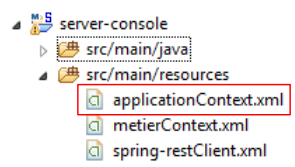
Nous vous laissons comprendre ce code. Il fait clignoter la led n° 8 de tous les Arduinos connectés. Pour cela, il envoie à intervalles réguliers la commande *PinWrite* en écrivant alternativement 0 et 1 sur la pin n° 8.

6.5 La configuration de Spring

Le programme de test précédent instancie la couche [DAO] à l'aide de Spring :

```
// création de la couche [dao]
      IDao dao = (IDao) new ClassPathXmlApplicationContext("applicationContext.xml").getBean("dao");
```

La configuration de Spring dans [applicationContext.xml] est la suivante :




```
1.  <?xml version="1.0" encoding="UTF-8"?>
```

```
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
5.
6.    <!-- thread d'enregistrement des arduinos -->
7.    <bean id="recorder" class="arduino.dao.Recorder" init-method="init">
8.      <property name="port" value="100"/>
9.      <property name="ip" value="192.168.2.1"/>
10.     <property name="nbConnexions" value="5"/>
11.     <property name="traced" value="true"/>
12.   </bean>
13.   <!-- couche dao -->
14.   <bean id="dao" class="arduino.dao.Dao" init-method="init">
15.     <property name="recorder" ref="recorder"/>
16.     <property name="timeoutConnection" value="1000"/>
17.     <property name="timeoutRead" value="1000"/>
18.     <property name="traced" value="false"/>
19.   </bean>
20. </beans>
```

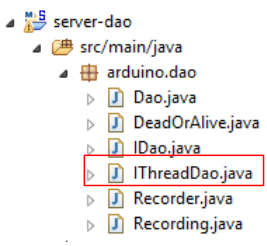
La couche [DAO] est configurée par les lignes 14-19 :

- ligne 14 : après instanciation de la classe [arduino.dao.Dao], Spring va exécuter la méthode *init* de cette classe. C'est dans cette méthode que nous créerons et lancerons le serveur d'enregistrement des Arduinos ;
- ligne 15 : la couche [DAO] aura une référence sur le serveur d'enregistrement appelé ici *recorder*. Il est défini lignes 7-12 ;
- ligne 16 : le délai d'attente maximal qu'attendra la couche [DAO] lorsqu'elle se connecte à un Arduino. Au-delà, elle lancera une exception [DomotiqueException] ;
- ligne 17 : le délai d'attente maximal qu'attendra la couche [DAO] lorsqu'elle atend une réponse d'un Arduino. Au-delà, elle lancera une exception [DomotiqueException] ;
- ligne 18 : un booléen qui indique si la couche [DAO] logue ou pas sur la console ce qu'elle fait ;

Le serveur d'enregistrement est configuré par les lignes 7-12 :

- le port d'écoute ;
- l'IP du serveur d'écoute. Une machine peut avoir plusieurs adresses IP si elle a plusieurs cartes réseau. Il est alors nécessaire de dire si le serveur écoute sur toutes les adresses IP ou seulement sur certaines comme c'est le cas ici ;
- ligne 10 : les demandes à un serveur sont mises en file d'attente si le serveur est occupé lorsque la demande arrive. Ici, on fixe à 5 la taille maximale de cette file. Si une sixième demande arrive alors elle reçoit une exception de la part du serveur ;
- ligne 11 : on peut demander au serveur d'enregistrement d'écrire ou non des logs sur la console ;

6.6 L'interface [IThreadDao] du serveur d'enregistrement

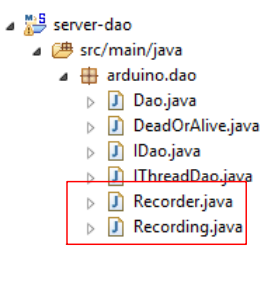


Le serveur d'enregistrement implémente l'interface [IThreadDao] suivante :

```
1.  package arduino.dao;
2.
3.  // thread lancé par la couche [dao]
4.  public interface IThreadDao extends Runnable {
5.    // le thread doit connaître la couche [dao]
```

```
6.    public void setDao(IDao dao);
7.  }
```

6.7 Le serveur d'enregistrement [Recorder]



Le squelette du serveur d'enregistrement est le suivant :

```
1.  package arduino.dao;
2.
3.  // paquetages
4.  ...
5.
6.  public class Recorder implements IThreadDao {
7.
8.    // injections Spring
9.    private int nbConnexions;
10.   private int port;
11.   private String ip;
12.   private IDao dao;
13.   private Boolean traced;
14.
15.   // données locales
16.   private InetAddress inetAddress;
17.
18.   @SuppressWarnings("unused")
19.    private void init() {
20.     try {
21.       // on initialise l'Ip du serveur
22.       inetAddress = InetAddress.getByName(ip);
23.     } catch (UnknownHostException e) {
24.       throw new DomotiqueException("Erreur à l'initialisation de l'IP du serveur", e, 4);
25.     }
26.   }
27.
28.   // ------------------ méthode run
29.   @SuppressWarnings("resource")
30.    public void run() {
31. ...
32.   }
33.
34.   // logs
35.   private void trace(Level level, String message, Exception e) {
36.     String msg = String.format("Recorder : [%s] : [%s]", new
    SimpleDateFormat("HH:mm:ss:SS").format(new Date()), message);
37.     if (e == null) {
38.       Logger.getLogger(Recorder.class.getName()).log(level, msg);
39.     } else {
40.       Logger.getLogger(Recorder.class.getName()).log(level, msg, e);
41.     }
42.   }
```

```
43.
44.   // getters et setters
45.   ...
46. }
```

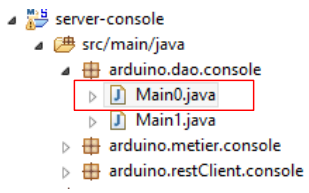
- lignes 9-13 : proviennent du fichier de configuration de Spring ;

Question : complétez le code du serveur d'enregistrement. Pour cela, aidez-vous du code du serveur générique en éliminant de celui-ci tout ce qui est inutile pour le serveur d'enregistrement. Pour assurer son service, le serveur [Recorder] lance un thread [Recording].

Conseils :

- lorsque le serveur reçoit une connexion d'un Arduino, il peut servir cette connexion lui-même ou lancer un autre thread pour le faire. La première solution est suffisante ici. La deuxième est à préférer dans le cas où le serveur a beaucoup de clients ;
- lorsque le serveur a lu la ligne envoyée par l'Arduino, il doit en faire un objet [Arduino] et appeler la méthode [addArduino] de la couche [DAO] pour que celle-ci enregistre le nouvel Arduino. La construction de l'objet [Arduino] est réalisée grâce à une bibliothèque JSON ;

6.8 Tests du serveur d'enregistrement



Dans le projet de tests [server-console], on exécutera le programme [Main0] suivant :

```
1.  package arduino.dao.console;
2.
3.  ...
4.  import arduino.dao.IDao;
5.  import arduino.dao.IThreadDao;
6.
7.  @SuppressWarnings("deprecation")
8.  public class Main0 implements IDao {
9.
10.   public static void main(String[] args) throws InterruptedException {
11.      // on instancie la classe [Main0]
12.      new Main0();
13.   }
14.
15.   // constructeur
16.   public Main0() {
17.      // on instancie le serveur d'enregistrement
18.      IThreadDao recorder = (IThreadDao) new XmlBeanFactory(new
    ClassPathResource("applicationContext.xml"))
19.            .getBean("recorder");
20.      // on lui injecte une couche [DAO] fictive
21.      recorder.setDao(this);
22.      // on le lance
23.      new Thread(recorder).start();
24.   }
25.
26.   public Collection<Arduino> getArduinos() {
27.      return null;
```

```
28.    }
29.
30.    public void addArduino(Arduino arduino) {
31.       // suivi
32.       System.out.println(String.format("L'Arduino [%s] a été ajouté", arduino));
33.    }
34.
35.    public void removeArduino(String idArduino) {
36.    }
37.
38.    public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes) {
39.       return null;
40.    }
41.
42.    public List<String> sendCommandesJson(String idArduino, List<String> commandes) {
43.       return null;
44.    }
45. }
```

- ligne 8 : la classe [Main0] implémente l'interface [IDao] de la couche [DAO]. Une seule méthode de l'interface sera implémentée, celles des lignes 30-33 ;
- ligne 16 : le constructeur ;
- ligne 18 : on instancie le serveur d'enregistrement défini comme suit dans le fichier [applicationContext.xml] :

```
1.    <!-- thread d'enregistrement des arduinos -->
2.    <bean id="recorder" class="arduino.dao.Recorder" init-method="init">
3.      <property name="port" value="100"/>
4.      <property name="ip" value="192.168.2.1"/>
5.      <property name="nbConnexions" value="5"/>
6.      <property name="traced" value="true"/>
7.  </bean>
```

Adaptez ce fichier à vos besoins.

- le serveur d'enregistrement doit remonter à la couche [DAO], l'Arduino qui s'enregistre. C'est en effet la couche [DAO] qui gère la liste des arduinos connectés. Pour cela, le serveur d'enregistrement doit avoir une référence sur la couche [DAO]. C'est fait en ligne 21 ;
- ligne 23 : le thread du serveur est lancé. A partir de ce moment, il va vivre tout seul, en parallèle des activités de la couche [DAO]. Ici, la classe [Main0] et le serveur d'enregistrement s'exécutent dans des threads différents ;
- lorsqu'un Arduino se connecte, le serveur appelle la méthode de la ligne 30 pour l'enregistrer. On se contente de loguer l'enregistrement sur la console.

Lorsqu'on exécute la classe [Main0], on doit vérifier que l'Arduino arrive à se connecter. Cela peut se voir dans les logs du serveur si vous les avez prévus :

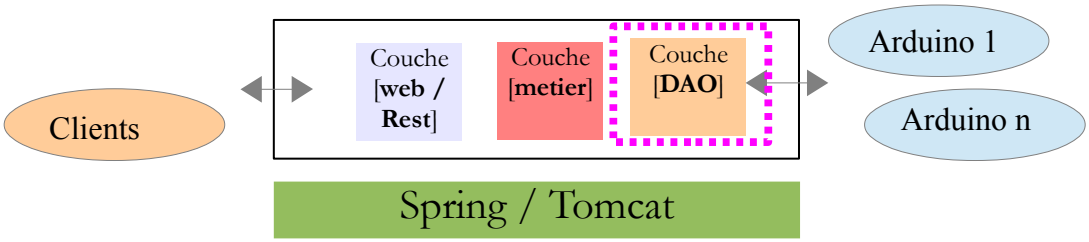
```
1.  Serveur d'enregistrement lancé sur 192.168.2.1:100
2.  mars 30, 2013 9:22:29 AM arduino.dao.Recorder trace
3.  Infos: Recorder : [09:22:29:265] : [Serveur d'enregistrement : attente d'un client]
4.  mars 30, 2013 9:22:34 AM arduino.dao.Recorder trace
5.  Infos: Recorder : [09:22:34:728] : [Serveur d'enregistrement : client détecté]
6.  mars 30, 2013 9:22:34 AM arduino.dao.Recorder trace
7.  Infos: Recorder : [09:22:34:733] : [Serveur d'enregistrement : attente d'un client]
8.  mars 30, 2013 9:22:34 AM arduino.dao.Recording trace
9.  Infos: Recording : [09:22:34:734] : [Service d'enregistrement d'un arduino lancé]
10. L'Arduino [id=[192.168.2.3], description=[duemilanove], mac=[90:A2:DA:00:1D:A7],
    ip=[192.168.2.3], port=[102]] a été ajouté
11. mars 30, 2013 9:22:34 AM arduino.dao.Recording trace
12. Infos: Recording : [09:22:34:850] : [Service d'enregistrement d'un arduino terminé]
```

- lignes 1 et 10 : logs de [Main0] ;
- les autres lignes sont des logs du serveur d'enregistrement.

Les logs de la console de l'Arduino montrent également cette connexion :

```
1. Connexion au serveur d'enregistrement...
2. Connecte...
3. enregistrement : {"id":"192.168.2.3","desc":"duemilanove","mac":"90:A2:DA:00:1D:A7","port":102}
4. Enregistrement termine...
5. Demarrage du serveur sur l'adresse IP 192.168.2.3 et le port 102
6. Memoire disponible avant service : 832
```

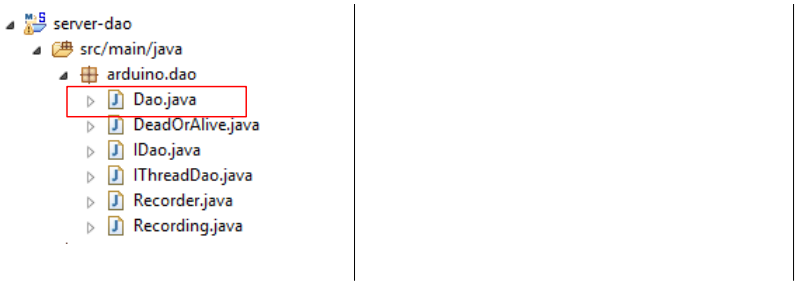
7 Implémentation de la couche [Dao] du serveur



A lire

- le code du client TCP générique présenté page 31, paragraphe 4.3.2.

7.1 Le code



Le squelette de la classe [Dao] est le suivant :

```
1.  package arduino.dao;
2.
3.  ...
4.
5.  import com.google.gson.Gson;
6.
7.  public class Dao implements IDao {
8.
9.    // injections Spring
10.   private IThreadDao recorder;
11.   private int timeoutConnection;
12.   private int timeoutRead;
13.   private Boolean traced;
14.   // données locales
15.   // dictionnaire des arduinos
16.   private Map<String, Arduino> arduinos = new HashMap<String, Arduino>();
17.   // verrou d'accès à la liste
18.   final private Object verrou = new Object();
19.   // mapper Json
20.   Gson jsonMapper = new Gson();
21.
22.   // constructeur
23.   public Dao() {
24.   }
25.
26.   // initialisation de la couche [dao]
27.   @SuppressWarnings("unused")
28.    private void init() {
29.     // lance le service d'enregistrement dans un thread à part
30.     recorder.setDao(this);
```

```
31.     new Thread(recorder).start();
32.   }
33.
34.   // liste des Arduinos
35.   public Collection<Arduino> getArduinos() {
36.     // accès contrôlé
37.     synchronized (verrou) {
38.       return arduinos.values();
39.     }
40.   }
41.
42.   // ajouter un arduino à la liste
43.   public void addArduino(Arduino arduino) {
44.     // accès contrôlé
45.     synchronized (verrou) {
46.         // on enregistre le nouvel arduino
47.         arduinos.put(arduino.getId(), arduino);
48.     }
49.   }
50.
51.   // supprimer un arduino de la liste
52.   public void removeArduino(String id) {
53.     // accès contrôlé
54.     synchronized (verrou) {
55.       arduinos.remove(id);
56.     }
57.   }
58.
59.   // envoyer des commandes [Commande] à un Arduino
60.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes) {
61.     // on récupère l'arduino
62.     Arduino arduino = arduinos.get(idArduino);
63.     // a-t-on trouvé ?
64.     if (arduino == null) {
65.       throw new DomotiqueException(String.format("L'arduino [%s] n'existe pas", idArduino),
    44);
66.     }
67.     // on exécute la liste des commandes
68. ...
69.     // on rend la liste des réponses [Reponse]
70.     return réponses;
71.   }
72.
73.   // envoyer des commandes Json à un arduino
74.   public List<String> sendCommandesJson(String idArduino, List<String> commandes) {
75.     // on récupère l'arduino
76.     Arduino arduino = arduinos.get(idArduino);
77.     // a-t-on trouvé ?
78.     if (arduino == null) {
79.       throw new DomotiqueException(String.format("L'arduino [%s] n'existe pas",
    idArduino));
80.     }
81.     // on exécute la liste des commandes
82. ...
83.     // on rend la liste des réponses JSON
84.     return réponses;
85.   }
86.
87.   // pour tracer l'échange avec l'arduino
88.   private void trace(String message) {
89.     trace(Level.INFO, String.format("RestServer Dao SendCommandesJson : [%s] : [%s]", new
    SimpleDateFormat("HH:mm:ss:SS").format(new Date()), message), null);
90.   }
```

```
91.
92.   // logs
93.   private void trace(Level level, String message, Exception e) {
94.     String msg = String.format("Dao : [%s] : [%s]", new
   SimpleDateFormat("HH:mm:ss:SS").format(new Date()), message);
95.     if (e == null) {
96.       Logger.getLogger(Dao.class.getName()).log(level, msg);
97.     } else {
98.       Logger.getLogger(Dao.class.getName()).log(level, msg, e);
99.     }
100.    }
101.
102.    // setters pour Spring
103.    ...
104.  }
```

- ligne 10 : la couche [DAO] a une référence sur le serveur d'enregistrement ;
- ligne 11 : le délai d'attente maximal qu'attendra la couche [DAO] lorsqu'elle se connecte à un Arduino. Au-delà, elle lancera une exception [DomotiqueException] ;
- ligne 12 : le délai d'attente maximal qu'attendra la couche [DAO] lorsqu'elle atend une réponse d'un Arduino. Au-delà, elle lancera une exception [DomotiqueException] ;
- ligne 13 : un booléen qui indique si la couche [DAO] logue ou pas sur la console ce qu'elle fait ;
- ligne 16 : les Arduinos connectés seront maintenus dans un dictionnaire plutôt qu'une liste. En effet, les clients envoient des commandes à un Arduino identifié par son *id*. A partir de celui-ci, l'Arduino sera retrouvé plus facilement dans un dictionnaire que dans une liste ;
- ligne 18 : un serveur peut faire assurer le service d'un client par un thread dit de " service ". Ainsi si à un moment donné, il y a *n* clients, il y a *n* threads de service. Si dans notre exemple, le service des Arduinos était assuré par des threads de service, alors il serait possible que la méthode [addArduino] de la couche [DAO] soit appelée en même temps par deux ou plusieurs threads de service. On dit alors que la méthode [addArduino] est une ressource critique. L'accès à celle-ci doit être contrôlé afin qu'un seul thread n'y ait accès à la fois. Ici, nous n'avons pas choisi la solution des threads de service parce qu'elle n'était pas indispensable. Nous allons quand même, par souci de pédagogie et parce que c'est simple, synchroniser l'accès aux ressources critiques ;
- ligne 18 : un verrou contrôlera l'accès aux ressources critiques. C'est un simple objet ;
- ligne 45 : un exemple d'utilisation d'un verrou. La syntaxe :

```
1.      synchronized (verrou) {
2.          ...
3.  }
```

assure que le code de la ligne 2 ne sera exécuté que par un seul thread à la fois même si celui-ci est interrompu au milieu du code de la ligne 2. Le principe est le suivant :

- un premier thread arrivé en ligne 1. Le verrou est libre. Il le prend et exécute le code de la ligne 2. Il possède le verrou tant qu'il n'a pas passé la ligne 3,
- un second thread arrive. Si le verrou n'a pas été libéré par le thread précédent, il est bloqué. Il va attendre que le verrou se libère ;

On peut faire la même chose avec d'autres méthodes de synchronisation non présentées ici.

- lignes 74-85 : cette méthode est le coeur de la couche [DAO]. Elle envoie à l'Arduino identifié par le 1er paramètre, la liste des commandes JSON identifiée par le second paramètre. Pour chaque commande, elle reçoit une réponse JSON. La méthode rend la liste des réponses JSON qu'elle a reçues ;
- lignes 60-71 : une méthode analogue mais avec des paramètres différents. Elle transforme sa liste de commande [Commande] en liste de commandes JSON et appelle la méthode [sendCommandesJson]. Elle reçoit une liste de réponses JSON qu'elle transforme en liste de réponses [Reponse].

Question 2 : Implémenter la classe [DAO]. Il s'agit d'écrire un client TCP-IP classique. On pourra s'aider du code du client TCP générique déjà utilisé pour des tests.

7.2 Tests de la couche [DAO]

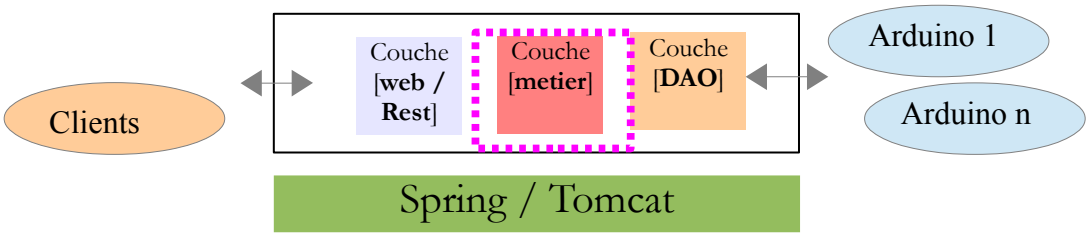
Exécutez les tests présentés au paragraphe 6.4, page 41.

7.3 Améliorations

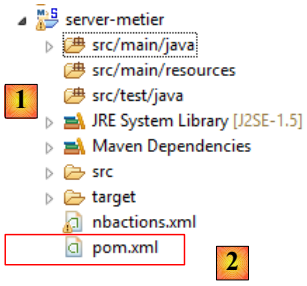
La couche [DAO] peut être améliorée de diverses façons. En voici deux :

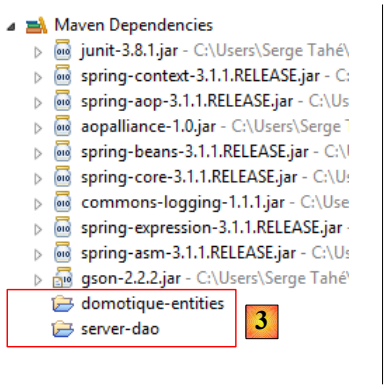
- un Arduino qui se connecte au serveur d'enregistrement est mémorisé. Si un utilisateur déconnecte son câble réseau ou son alimentation, l'Arduino n'est plus accessible mais reste mémorisé dans la couche [DAO]. On peut prévoir un thread qui à intervalles réguliers (10 secondes) essaierait de se connecter au port 102 de tous les Arduinos mémorisés. Lorsque l'un ne répond pas, il est enlevé de la liste des Arduinos mémorisés ;
- les adresses IP et Mac sont fixées par le développeur dans le code des Arduinos. Ainsi il n'est pas impossible de vouloir connecter deux Arduinos qui par erreur auraient la même adresse IP. La méthode [addArduino] de la couche [DAO] peut faire des tests. Si l'Arduino qui arrive, a la même adresse IP ou la même adresse Mac ou la même description qu'un Arduino déjà enregistré, son enregistrement est refusé et une exception [DomotiqueException] est lancée.

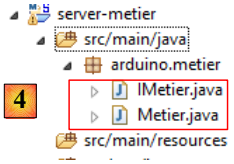
8 Implémentation de la couche [métier] du serveur



8.1 Le projet Eclipse







- en [1] le projet Eclipse est un projet Maven [2] ;
- en [3], les dépendances Maven. On remarque la dépendance sur les deux projets précédents [domotique-entities] et [server-dao] ;
- en [4], le code source de la couche [métier].

8.2 Les dépendances Maven

Le fichier [pom.xml] du projet Maven est le suivant :

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    4.0.0.xsd">
3.    <modelVersion>4.0.0</modelVersion>
4.
5.    <groupId>istia.st.domotique.server</groupId>
6.    <artifactId>server-metier</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>server-metier</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <properties>
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.   <dependencies>
18.     <dependency>
19.       <groupId>istia.st.domotique.server</groupId>
```

```
20.       <artifactId>server-dao</artifactId>
21.       <version>1.0-SNAPSHOT</version>
22.     </dependency>
23.   </dependencies>
24. </project>
```

- lignes 5-8 : l'identité Maven du projet ;
- lignes 18-22 : il n'y a qu'une dépendance, celle sur le projet [server-dao] de la couche [DAO]. Les autres dépendances en découlent en cascade.

8.3 Le code

L'interface [IMetier] de la couche [metier] est la suivante :

```
1.  package arduino.metier;
2.
3.  import arduino.entites.Arduino;
4.  ...
5.
6.  public interface IMetier {
7.    // liste des arduinos
8.    public List<Arduino> getArduinos();
9.    // lecture d'une pin
10.   public Reponse pinRead(String idCommande, String idArduino, int pin, String mode);
11.   // écriture d'une pin
12.   public Reponse pinWrite(String idCommande, String idArduino, int pin, String mode,int val);
13.   // faire clignoter une led
14.   public void faireClignoterLed(String idCommande, String idArduino, int pin, int millis, int nbIter);
15.   // envoyer une suite de commandes Json à un Arduino
16.   public List<String> sendCommandesJson(String idArduino, List<String> commandes);
17.   // envoyer une suite de commandes à un Arduino
18.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes);
19. }
```

- ligne 10 : pour lire la valeur d'une pin d'un Arduino :
  - **idCommande** : identifiant de la commande – une chaîne de caractères quelconque qui permettra de retrouver la réponse dans des logs. La réponse reprend en effet l'identifiant de la commande,
  - **idArduino** : l'identifiant de l'Arduino à qui est destinée la commande,
  - **pin** : le n° de la pin à lire,
  - **mode** : le mode de lecture : " a " pour analogique, " b " pour binaire ;
- ligne 12 : pour écrire une valeur sur la pin d'un Arduino :
  - **idCommande** : identifiant de la commande,
  - **idArduino** : l'identifiant de l'Arduino à qui est destinée la commande,
  - **pin** : le n° de la pin à écrire,
  - **mode** : le mode d'écriture : " a " pour analogique, " b " pour binaire,
  - **val** : la valeur à écrire sur la pin ;
- ligne 14 : pour faire clignoter une led d'un Arduino :
  - **idCommande** : identifiant de la commande,
  - **idArduino** : l'identifiant de l'Arduino à qui est destinée la commande,
  - **pin** : n° de la pin à laquelle est reliée la led,
  - **millis** : durée en millisecondes d'un clignotement,
  - **nbIter** : le nombre de clignotements à opérer ;
- lignes 8, 16, 18 : correspondent aux méthodes de mêmes noms dans la couche [DAO].

Question 3 : écrire la classe [Metier] qui implémente l'interface [IMetier]. On lira auparavant les informations ci-dessous.

Pour les différentes méthodes, les chaînes JSON à envoyer à l'Arduino sont les suivantes :

Méthode	commande JSON envoyée à l'Arduino	Signification
pinRead	{"id":"0","parametres":{"pin":"3","mod":"b"},"action":"pr"}	lire (action) la pin 3 (pin) en mode binaire (mod)
pinWrite	{"id":"0","parametres": {"val":"200","pin":"8","mod":"a"},"action":"pw"}	écrire (action) la valeur 200 (val) sur la pin 8 (pin) en mode analogique (mod)
faireClignoter	{"id":"1","parametres":	faire clignoter (action) la pin 8 (pin) 100 fois (nb)

{"nb":"100","dur":"200","pin":"8"},"action":"cl"}	avec une période de 200 ms (dur)

L'objet [Commande] correspondant à la chaîne JSON de la méthode [pinRead] de l'exemple ci-dessus est le suivant :

Champ	JSON
String id	"id":"0"
String action	"action":"pr"
Map<String,String> parametres	"parametres":{"pin":"3","mod":"b"}

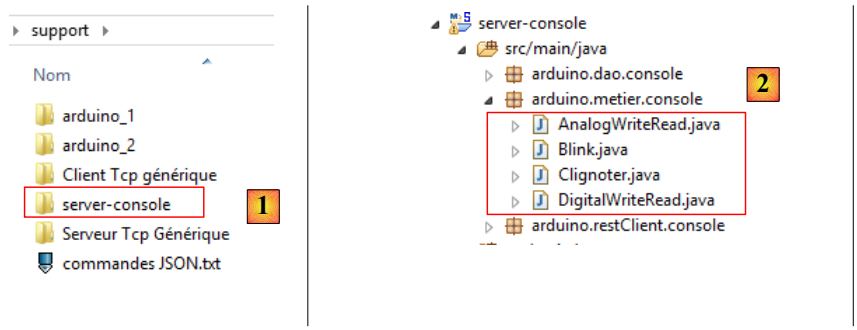
On respectera les contraintes suivantes :

Méthode	Contraintes
pinRead	• le mode doit être " a " (analogique) ou " b " (binaire) ; • en mode analogique, le N° de pin doit être dans l'intervalle [0,5] ; • en mode binaire, le N° de pin doit être dans l'intervalle [1,13].
pinWrite	• le mode doit être " a " (analogique) ou " b " (binaire) ; • le N° de pin doit être dans l'intervalle [1,13] ; • en mode binaire, la valeur doit être 0 ou 1 ; • en mode analogique, la valeur doit être dans l'intervalle [0,255].
faireClignoter	• le N° de pin doit être dans l'intervalle [1,13] ; • la durée du clignotement doit être un entier >0

Si une contrainte n'est pas vérifiée, la méthode lancera une exception de type [DomotiqueException].

8.4 Tests de la couche [métier]

Dans le dossier [support] du TP, vous trouverez un projet Eclipse qui va vous permettre de tester les différents éléments du serveur :



- en [1] : le projet console de tests ;
- en [2] : les classes de test de la couche [métier].

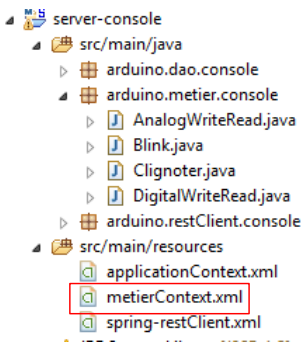
Examinons le programme [Clignoter] qui fait clignoter une led :

```
1.  package arduino.metier.console;
2.
3.  ...
4.
5.  public class Clignoter {
6.
7.    public static void main(String[] args) throws InterruptedException {
8.
9.      String syntaxe = "Syntaxe : pg pin duréeClignotement nbClignotements";
10.
```

```
11.     // il faut 3 arguments : le n° de pin, la durée du clignotement en millisecondes et le
    nombre de clignotements
12.     if (args.length != 3) {
13.       System.out.println(syntaxe);
14.       System.exit(0);
15.     }
16.     // le n° de pin doit être un entier dans l'intervalle 0-13
17.     int pin = 0;
18.     Boolean erreur = false;
19.     try {
20.       pin = Integer.parseInt(args[0]);
21.       if (pin < 1 || pin > 13) {
22.         erreur = true;
23.       }
24.     } catch (NumberFormatException e) {
25.       erreur = true;
26.     }
27.     // erreur ?
28.     if (erreur) {
29.       System.out.println("Le n° de pin doit être dans l'intervalle [1,13]");
30.       System.exit(0);
31.     }
32.     // la durée d'un clignotement
33.     erreur = false;
34.     int durée = 0;
35.     try {
36.       durée = Integer.parseInt(args[1]);
37.       if (durée < 100 || durée > 2000) {
38.         erreur = true;
39.       }
40.     } catch (NumberFormatException e) {
41.       erreur = true;
42.     }
43.     // erreur ?
44.     if (erreur) {
45.       System.out.println("La durée du clignotement en millisecondes doit être dans
    l'intervalle [100,2000]");
46.       System.exit(0);
47.     }
48.     // le nbre de clignotements
49.     erreur = false;
50.     int nbIter = 0;
51.     try {
52.       nbIter = Integer.parseInt(args[2]);
53.       if (nbIter < 2) {
54.         erreur = true;
55.       }
56.     } catch (NumberFormatException e) {
57.       erreur = true;
58.     }
59.     // erreur ?
60.     if (erreur) {
61.       System.out.println("Le nombre de clignotements doit être au moins égal à 2");
62.       System.exit(0);
63.     }
64.
65.     // création de la couche [métier]
66.     IMetier métier = (IMetier) new
    ClassPathXmlApplicationContext("metierContext.xml").getBean("metier");
67.     // liste des arduinos
68.     System.out.println("Attente d'arduinos connectés...");
69.     Collection<Arduino> arduinos = métier.getArduinos();
70.     while (arduinos.isEmpty()) {
```

```
71.       // attente d'1 seconde
72.       Thread.sleep(1000);
73.       // liste des arduinos
74.       arduinos = métier.getArduinos();
75.     }
76.     // on fait clignoter la pin pin de tous les arduinos nbIter fois avec un intervalle de
    millis millisecondes
77.     for (Arduino arduino : arduinos) {
78.         métier.faireClignoterLed(arduino.getIp(), arduino.getId(), pin, durée, nbIter);
79.     }
80.     System.out.println("fin de main");
81.   }
82. }
```

- ligne 9 : le programme se lance avec trois arguments. Ceux-ci sont expliqués ligne 11 ;
- lignes 11-63 : la validité des trois arguments est vérifiée ;
- ligne 66 : la couche [métier] est instanciée à l'aide du fichier de configuration [metierContext.xml] suivant :




```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
5.
6.    <!-- thread d'enregistrement des arduinos -->
7.    <bean id="recorder" class="arduino.dao.Recorder" init-method="init">
8.      <property name="port" value="100"/>
9.      <property name="ip" value="192.168.2.1"/>
10.     <property name="nbConnexions" value="5"/>
11.     <property name="traced" value="true"/>
12.   </bean>
13.
14.   <!-- couche dao -->
15.   <bean id="dao" class="arduino.dao.Dao" init-method="init">
16.     <property name="recorder" ref="recorder"/>
17.     <property name="timeoutConnection" value="1000"/>
18.     <property name="timeoutRead" value="1000"/>
19.     <property name="traced" value="false"/>
20.   </bean>
21.   <!-- couche métier -->
22.   <bean id="metier" class="arduino.metier.Metier">
23.     <property name="dao" ref="dao"/>
24.   </bean>
25. </beans>
```

- lignes 6-20 : on retrouve la configuration de la couche [DAO] déjà expliquée ;
- lignes 22-24 : configuration de la couche [métier]. Celle-ci a une référence sur la couche [DAO] (ligne 23) ;

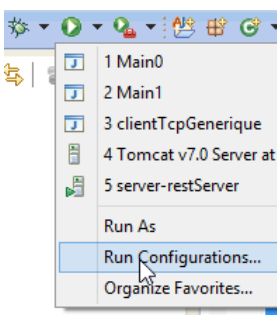
Retour au code du test :

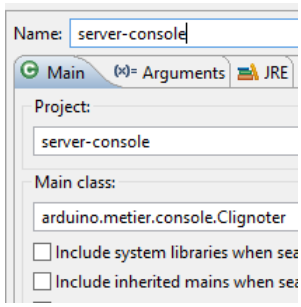
- ligne 69 : la liste des Arduinos connectés est demandée à la couche [métier] ;
- lignes 70-75 : si celle-ci est vide, elle est redemandée toutes les secondes ;
- lignes 77-79 : on envoie la commande de clignotement à tous les Arduinos connectés à l'aide de la méthode [faireClignoter] de la couche [métier].

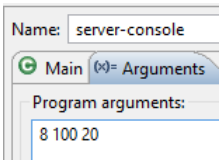
Les autres tests ont un code analogue.

Travail à faire : Assurez-vous que votre couche [métier] passe tous les tests.

Note : Pour passer des arguments à un programme, il faut créer une configuration d'exécution. Voici un exemple :

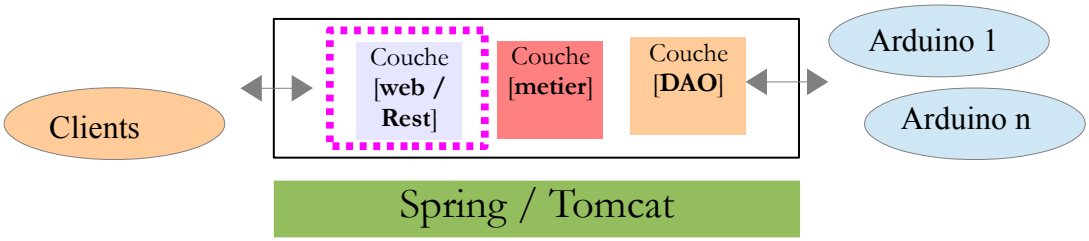






Ce processus a été décrit au paragraphe 4.2.2, page 23.

9 Implémentation du service REST du serveur

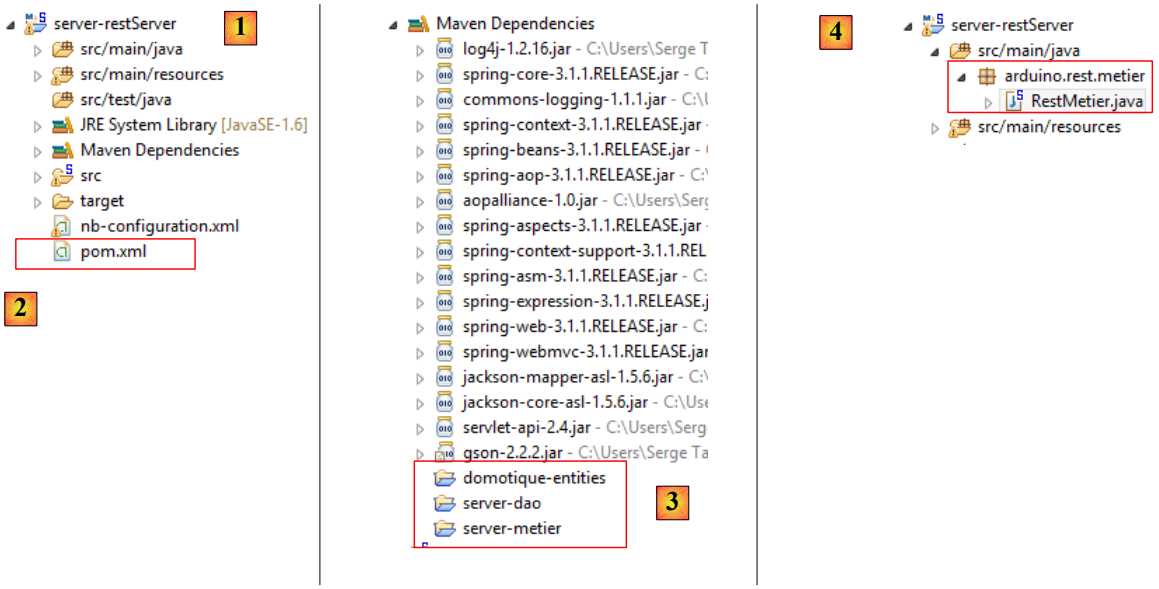


La couche web expose la couche [métier] aux clients web via un service REST.

A lire

- paragraphes 4.3.2 et 9.6 du document [**ref2**] ;

9.1 Le projet Eclipse



- en [1], le projet Eclipse est un projet Maven [2] ;
- en [3], les dépendances Maven du projet. Celui-ci dépend du projet de la couche [métier]. Les autres dépendances en découlent en cascade ;
- en [4], la classe [RestMetier] implémente la couche [métier].

9.2 Les dépendances Maven

Le fichier [pom.xml] du projet Maven est le suivant :

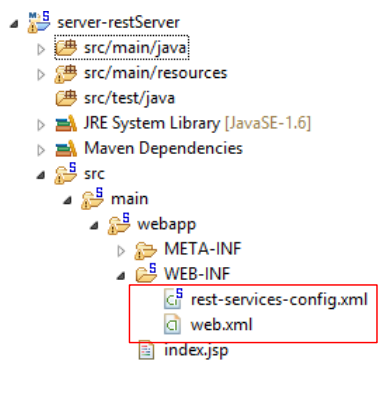
```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.   <modelVersion>4.0.0</modelVersion>
4.
5.     <groupId>istia.st.avat.server</groupId>
6.   <artifactId>server-rest</artifactId>
7.   <version>1.0-SNAPSHOT</version>
```

```
8.    <packaging>war</packaging>
9.
10.   <name>server-rest</name>
11.
12.   <properties>
13.     <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.     <releaseCandidate>1</releaseCandidate>
16.     <spring.version>3.1.1.RELEASE</spring.version>
17.     <jackson.mapper.version>1.5.6</jackson.mapper.version>
18.   </properties>
19.
20.   <repositories>
21.     <repository>
22.       <id>com.springsource.repository.bundles.release</id>
23.       <name>SpringSource Enterprise Bundle Repository - Release</name>
24.       <url>http://repository.springsource.com/maven/bundles/release</url>
25.     </repository>
26.   </repositories>
27.
28.   <dependencies>
29.     <dependency>
30.       <groupId>org.springframework</groupId>
31.       <artifactId>spring-core</artifactId>
32.       <version>${spring.version}</version>
33.     </dependency>
34.     <dependency>
35.       <groupId>org.springframework</groupId>
36.       <artifactId>spring-beans</artifactId>
37.       <version>${spring.version}</version>
38.     </dependency>
39.     <dependency>
40.       <groupId>org.springframework</groupId>
41.       <artifactId>spring-web</artifactId>
42.       <version>${spring.version}</version>
43.     </dependency>
44.     <dependency>
45.       <groupId>org.springframework</groupId>
46.       <artifactId>spring-webmvc</artifactId>
47.       <version>${spring.version}</version>
48.     </dependency>
49.     <dependency>
50.       <groupId>org.codehaus.jackson</groupId>
51.       <artifactId>jackson-mapper-asl</artifactId>
52.       <version>${jackson.mapper.version}</version>
53.     </dependency>
54.     <dependency>
55.         <groupId>istia.st.domotique.server</groupId>
56.         <artifactId>server-metier</artifactId>
57.         <version>1.0-SNAPSHOT</version>
58.     </dependency>
59.   </dependencies>
60.
61.   <build>
62.     <plugins>
63.       ...
64.     </plugins>
65.   </build>
66.
67. </project>
```

- le service REST est implémenté à l'aide du framework SpringMVC. On trouve donc un certain nombre de dépendances sur ce framework (lignes 29-48) ;
- lignes 49-53 : le serveur REST va échanger avec ses clients des objets au format JSON (JavaScript Object Notation). C'est une représentation texte des objets. Un objet JAVA peut être sérialisé en un texte JSON. Inversement, ce dernier peut être désérialisé pour créer un objet JAVA. Côté serveur, la bibliothèque Jackson est utilisée pour faire ces opérations ;
- lignes 54-58 : le serveur REST a une dépendance sur la couche [métier] que nous avons décrite.

9.3 La configuration du service REST

Un service REST est une application web et à ce titre est configurée par un fichier [web.xml] classique :



Le fichier [web.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
    app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
3.    <display-name>REST for AVAT</display-name>
4.    <servlet>
5.      <servlet-name>restservices</servlet-name>
6.      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
7.      <init-param>
8.        <param-name>contextConfigLocation</param-name>
9.        <param-value>/WEB-INF/rest-services-config.xml</param-value>
10.     </init-param>
11.     <load-on-startup>1</load-on-startup>
12.   </servlet>
13.   <servlet-mapping>
14.     <servlet-name>restservices</servlet-name>
15.     <url-pattern>/</url-pattern>
16.   </servlet-mapping>
17. </web-app>
```

- lignes 4-12 : définition d'une servlet, une classe capable de traiter les requêtes HTTP des clients web ;
- ligne 5 : son nom *restservices*. On peut l'appeler comme on veut ;
- ligne 6 : la servlet qui gère les requêtes HTTP des clients web. C'est la classe [DispatcherServlet] de SpringMVC ;
- lignes 7-10 : indiquent le fichier de configuration de la servlet, ici [WEB-INF/rest-services-config.xml] ;
- ligne 11 : la servlet sera chargée au démarrage du serveur, ici un serveur Tomcat 7 ;
- lignes 13-16 : les URL (ligne 15) traitées par la servlet *restservices* (ligne 14).

Au final, ce fichier indique que toutes les URL (ligne 15) seront gérées par la servlet *restservices* (ligne 14) définie lignes 4-12.

La classe [DispatcherServlet] est configurée par le fichier [rest-services-config.xml] suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
```

```
3.          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.          xmlns:oxm="http://www.springframework.org/schema/oxm"
5.          xmlns:util="http://www.springframework.org/schema/util"
6.          xmlns:mvc="http://www.springframework.org/schema/mvc"
7.          xmlns:context="http://www.springframework.org/schema/context"
8.          xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
9.                      http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
10.                     http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
11.                     http://www.springframework.org/schema/oxm
12.                     http://www.springframework.org/schema/oxm/spring-oxm-3.0.xsd
13.                     http://www.springframework.org/schema/util
14.                     http://www.springframework.org/schema/util/spring-util-3.0.xsd">
15.
16.
17.   <!-- thread d'enregistrement des arduinos -->
18.   <bean id="recorder" class="arduino.dao.Recorder" init-method="init">
19.     <property name="port" value="100"/>
20.     <property name="ip" value="192.168.2.1"/>
21.     <property name="nbConnexions" value="5"/>
22.     <property name="traced" value="true"/>
23.   </bean>
24.   <!-- couche dao -->
25.   <bean id="dao" class="arduino.dao.Dao" init-method="init">
26.     <property name="recorder" ref="recorder"/>
27.     <property name="timeoutConnection" value="1000"/>
28.     <property name="timeoutRead" value="1000"/>
29.     <property name="traced" value="true"/>
30.   </bean>
31.   <!-- couche métier -->
32.   <bean id="metier" class="arduino.metier.Metier">
33.     <property name="dao" ref="dao"/>
34.   </bean>
35.
36.    <!-- Recherche des annotations SpringMVC dans les classes du package nommé -->
37.    <context:component-scan base-package="arduino.rest.metier" />
38.
39.    <!-- L'unique View utilisée. Elle génère du JSON -->
40.    <bean
41.       class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
42.       <property name="contentType" value="text/plain" />
43.    </bean>
44.
45.    <!-- Le convertisseur JSON -->
46.    <bean id="jsonMessageConverter"
47.       class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter" />
48.
49.       <!-- la liste des convertisseurs de messages. Il n'y en qu'un : le JSON ci-dessus -->
50.    <bean
51.       class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
52.       <property name="messageConverters">
53.          <util:list id="beanList">
54.             <ref bean="jsonMessageConverter" />
55.          </util:list>
56.       </property>
57.    </bean>
58.
59. </beans>
```

- lignes 17-34 : configuration des couches [métier] et [DAO] du serveur ;

- lignes 36-37 : il est possible de configurer Spring à l'aide d'annotations dans le code Java. C'est ce qui a été fait ici. La ligne 37 dit à Spring d'exploiter les annotations qu'il trouvera dans le package [*arduino.rest.metier*] ;

Il y trouvera trois genres d'annotations :

```
1. @Controller
2. public class RestMetier {
3.
4.    // couche métier
5.    @Autowired
6.    private IMetier metier;
7.    @Autowired
8.    private View view;
9.
10.   // liste des arduinos
11.   @RequestMapping(value = "/arduinos/", method = RequestMethod.GET)
12.   public ModelAndView getArduinos() {
13. ...
14.   }
```

- ligne 1 : l'annotation **@Controller** fait de la classe [RestMetier] un contrôleur SpringMVC, c-à-d une classe capable de traiter des requêtes HTTP. Celles-ci sont définies par l'annotation **@RequestMapping** qu'on voit en ligne 11 ;
- lignes 5 et 7 : l'annotation **@Autowired** tague des champs qui sont des références sur des beans définis dans le fichier de configuration de SpringMVC. Dans notre cas, ce sont les beans des lignes 2 et 10 du fichier [rest-services-config.xml] ci-dessous. Les champs des lignes ainsi taguées sont automatiquement initialisés par Spring.

Revenons au code du fichier [rest-services-config.xml] :

```
1.  <!-- couche métier -->
2.    <bean id="metier" class="arduino.metier.Metier">
3.      <property name="dao" ref="dao"/>
4.    </bean>
5.
6.     <!-- Recherche des annotations SpringMVC dans les classes du package nommé -->
7.     <context:component-scan base-package="arduino.rest.metier" />
8.
9.     <!-- L'unique View utilisée. Elle génère du JSON -->
10.    <bean
11.       class="org.springframework.web.servlet.view.json.MappingJacksonJsonView">
12.       <property name="contentType" value="text/plain" />
13.    </bean>
14.
15.    <!-- Le convertisseur JSON -->
16.    <bean id="jsonMessageConverter"
17.       class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"
    />
18.
19.       <!-- la liste des convertisseurs de messages. Il n'y en qu'un : le JSON ci-dessus
    -->
20.    <bean
21.
       class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
22.       <property name="messageConverters">
23.          <util:list id="beanList">
24.             <ref bean="jsonMessageConverter" />
25.          </util:list>
26.       </property>
27.    </bean>
28.
29. </beans>
```

- lignes 10-13 : les vues générées par l'unique contrôleur seront du type [*MappingJacksonJsonView*] (ligne 11). Cette vue va transformer son modèle en un unique objet JSON qu'elle va transmettre à son client. La ligne 12 configure un entête HTTP envoyé par le serveur à son client : elle dit que le serveur envoie du texte, ici l'objet JSON ;
- lignes 16-17 : un « convertisseur de messages ». Je ne sais pas bien ce que signifie cette expression. Il semble que cela définisse une classe capable de créer un objet à partir d'une requête ou d'une réponse HTTP et inversement. Ici la classe [*MappingJacksonHttpMessageConverter*] va créer un objet JSON à partir du modèle de la vue. Inversement, elle est capable de produire un objet Java à partir de données JSON postées par le client ;
- lignes 20-27 : il peut y avoir plusieurs « convertisseurs de messages » pour différents cas de figure. Entre les lignes 21-23, on met les beans chargés de « convertir des messages ». On y met donc le bean défini ligne 16. C'est le seul.

9.4 Le code du service REST

Le serveur REST est implémenté par la classe [RestMetier] suivante :

```
15. package arduino.rest.metier;
16.
17. ...
18.
19. import arduino.metier.IMetier;
20.
21. @Controller
22. public class RestMetier {
23.
24.    // couche métier
25.    @Autowired
26.    private IMetier metier;
27.    @Autowired
28.    private View view;
29.
30.    // liste des arduinos
31.    @RequestMapping(value = "/arduinos/", method = RequestMethod.GET)
32.    public ModelAndView getArduinos() {
33. ...
34.    }
35.
36.    // clignotement
37.    @RequestMapping(value = "/arduinos/blink/{idCommande}/{idArduino}/{pin}/{duree}/
   {nombre}", method = RequestMethod.GET)
38.    public ModelAndView faireClignoterLed(@PathVariable("idCommande") String idCommande,
39.          @PathVariable("idArduino") String idArduino, @PathVariable("pin") String pin,
40.          @PathVariable("duree") String duree, @PathVariable("nombre") String nombre) {
41. ...
42.    }
43.
44.    // envoi de commandes JSON
45.    @SuppressWarnings("unchecked")
46.    @RequestMapping(value = "/arduinos/commands/{idArduino}", method = RequestMethod.POST,
   consumes = "application/json")
47.    public ModelAndView sendCommandesJson(@PathVariable("idArduino") String idArduino,
48.          @RequestBody List<Map<String, String>> commandes) {
49. ...
50.    }
51.
52.    // lecture pin
53.    @RequestMapping(value = "/arduinos/pinRead/{idCommande}/{idArduino}/{pin}/{mode}", method
   = RequestMethod.GET)
54.    public ModelAndView pinRead(@PathVariable("idCommande") String idCommande,
55.          @PathVariable("idArduino") String idArduino, @PathVariable("pin") String pin,
   @PathVariable("mode") String mode) {
56. ...
57.    }
58.
```

```
59.    // écriture pin
60.    @RequestMapping(value = "/arduinos/pinWrite/{idCommande}/{idArduino}/{pin}/{mode}/
   {valeur}", method = RequestMethod.GET)
61.    public ModelAndView pinWrite(@PathVariable("idCommande") String idCommande,
62.          @PathVariable("idArduino") String idArduino, @PathVariable("pin") String pin,
   @PathVariable("mode") String mode,
63.          @PathVariable("valeur") String valeur) {
64. ...
65.    }
66.
67.    // crée une réponse d'erreur
68.    private ModelAndView createResponseError(String numero, String message) {
69.       Map<String, Object> modèle = new HashMap<String, Object>();
70.       modèle.put("error", numero);
71.       modèle.put("message", message);
72.       return new ModelAndView(view, modèle);
73.    }
74. }
```

Question 4 : en suivant l'exemple du paragraphe 4.3.2.5 de [**ref2**], écrire la classe [RestMetier].

9.5 Les tests du service REST

Travail à faire : en suivant l'exemple du paragraphe 4.3.2.6 de [**ref2**], déployer et tester le service REST. **Toutes les méthodes** de ce service doivent être testées.

Voici quelques exemples :

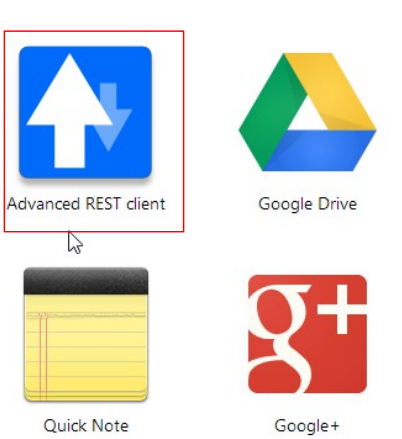
URL	rôle
http://localhost:8080/server-restServer/arduinos/	rend la liste des Arduinos connectés
http://localhost:8080/server-restServer/arduinos/blink/1/192.168.2.3/8/100/20/	fait clignoter la led de la pin n° 8 de l'Arduino identifié par 192.168.2.3
http://localhost:8080/server-restServer/arduinos/pinRead/1/192.168.2.3/0/a/	lecture analogique de la pin n° 8 de l'Arduino identifié par 192.168.2.3
http://localhost:8080/server-restServer/arduinos/pinRead/1/192.168.2.3/5/b/	lecture binaire de la pin n° 5 de l'Arduino identifié par 192.168.2.3
http://localhost:8080/server-restServer/arduinos/pinWrite/1/192.168.2.3/8/b/1/	écriture binaire de la valeur 1 sur la pin n° 8 de l'Arduino identifié par 192.168.2.3
http://localhost:8080/server-restServer/arduinos/pinWrite/1/192.168.2.3/4/a/100/	écriture analogique de la valeur 100 sur la pin n° 4 de l'Arduino identifié par 192.168.2.3

Le test de la méthode [sendCommandesJson] est plus délicat :

```
1.    // envoi de commandes JSON
2.    @SuppressWarnings("unchecked")
3.    @RequestMapping(value = "/arduinos/commands/{idArduino}", method = RequestMethod.POST,
   consumes = "application/json")
4.    public ModelAndView sendCommandesJson(@PathVariable("idArduino") String idArduino,
5.          @RequestBody List<Map<String, String>> commandes) {
6.  ...
7.  }
```

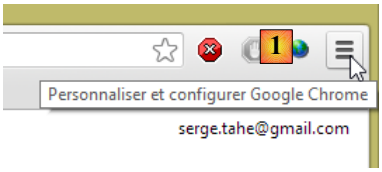
- ligne 3 : la méthode attend une requête POST qu'on ne peut pas simuler simplement avec un navigateur. Il faut des outils spéciaux ;
- ligne 3 : la requête POST doit demander une URL de la forme [/arduinos/commands/192.168.2.3] pour envoyer à l'Arduino désigné une liste de commandes JSON ;
- ligne 5 : cette liste de commandes JSON est le corps du POST (@RequestBody), c-à-d la valeur postée.

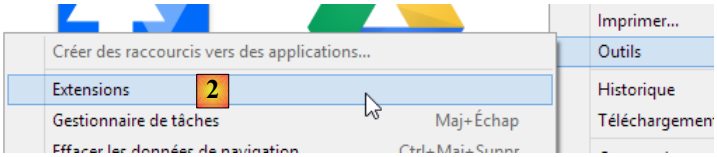
Pour tester cette méthode, on pourra utiliser un navigateur Chrome avec l'extension [Advanced REST Client] :



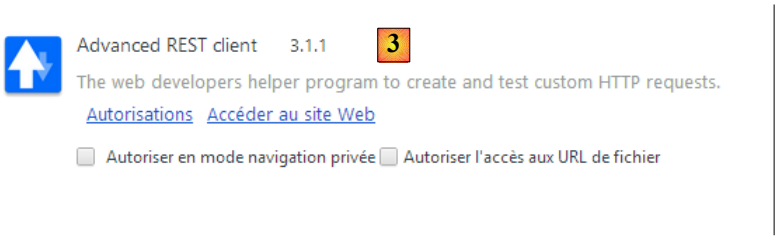
Pour installer cette extension, on pourra procéder de la façon suivante :

- lancer le navigateur Chrome ;



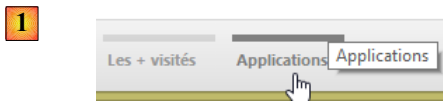


- en [1], personnaliser le navigateur ;
- en [2], sélectionner l'option [Outils / Extensions] ;



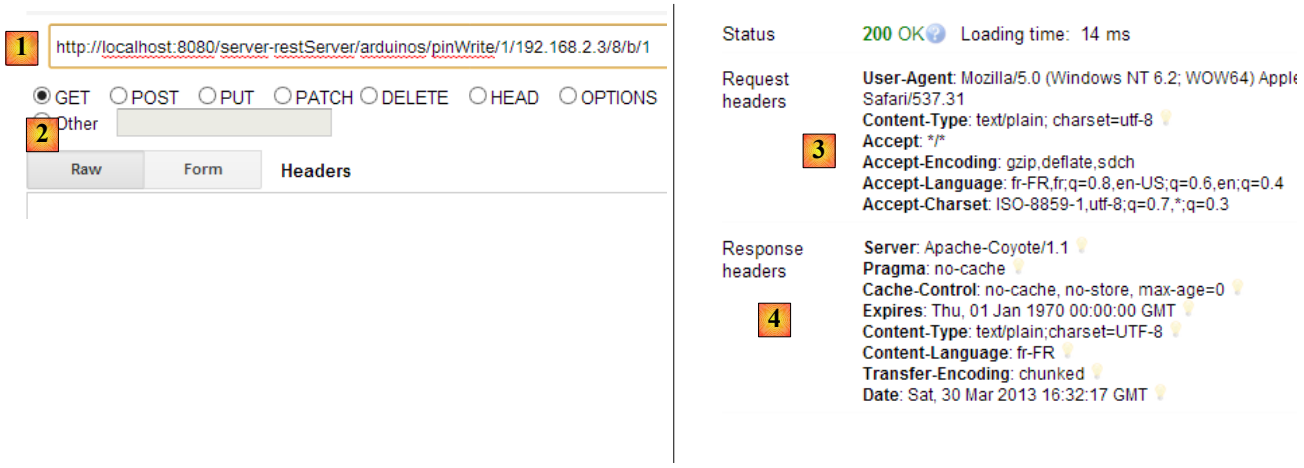
- en [3], installer l'extension [Advanced REST Client].

Une fois cette extension installée dans Chrome, on peut l'utiliser de la façon suivante :



- en [1], en bas du navigateur, on sélectionne la vue [Applications]. Il faut parfois créer un nouvel onglet pour avoir cette barre de sélection ;

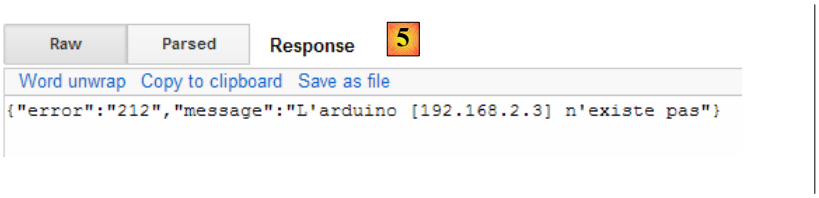
- en [2], on lance l'extension désirée ;



- en [1], l'URL de la méthode REST à tester ;
- en [2], la méthode GET ou POST pour envoyer la requête ;

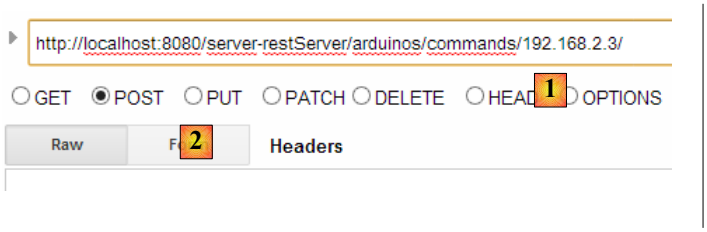
La requête est envoyée en la validant ou via le bouton [Send].

- en [3], les entêtes HTTP de la demande ;
- en [4], les entêtes HTTP de la réponse ;



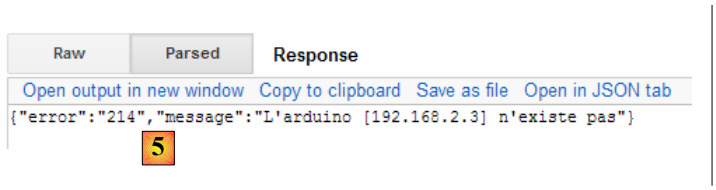
- en [5], la réponse JSON renvoyée par le serveur REST ;

Toutes les méthodes GET du service REST peuvent être testées ainsi. Pour la méthode POST [sendCommandesJson], on procèdera ainsi :



- en [1], l'URL de la méthode [sendCommandsJson] ;
- en [2], la requête se fait via la commande HTTP POST ;

- en [3], on indique que l'objet envoyé par le POST est du texte JSON ;
- en [4], la liste des commandes JSON. On notera bien les crochets qui commencent et terminent la liste. Ici, dans la liste il n'y a qu'une commande JSON qui écrit la valeur binaire 0 sur la pin 8 de l'Arduino ;



- en [5], la réponse JSON envoyée par le serveur ;

Le serveur est désormais écrit. Il servira aussi bien aux client natifs Android qu'aux client web mobiles.

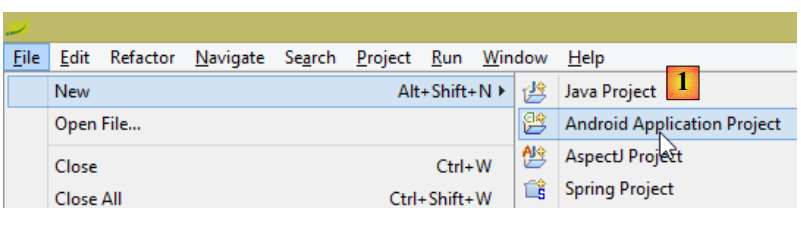
10 Introduction au développement d'une application Android

Nous allons introduire quelques concepts d'Android au travers d'une application simple. Créons avec Eclipse un projet Android :

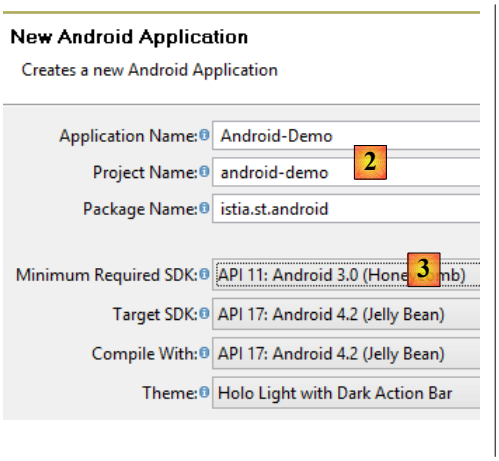
10.1 Les outils de développement

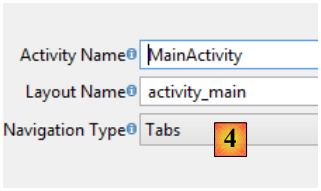
Les outils nécessaires pour développer une application Android sont décrits dans les annexes de [**ref2**], paragraphe 11.

10.2 Génération du projet



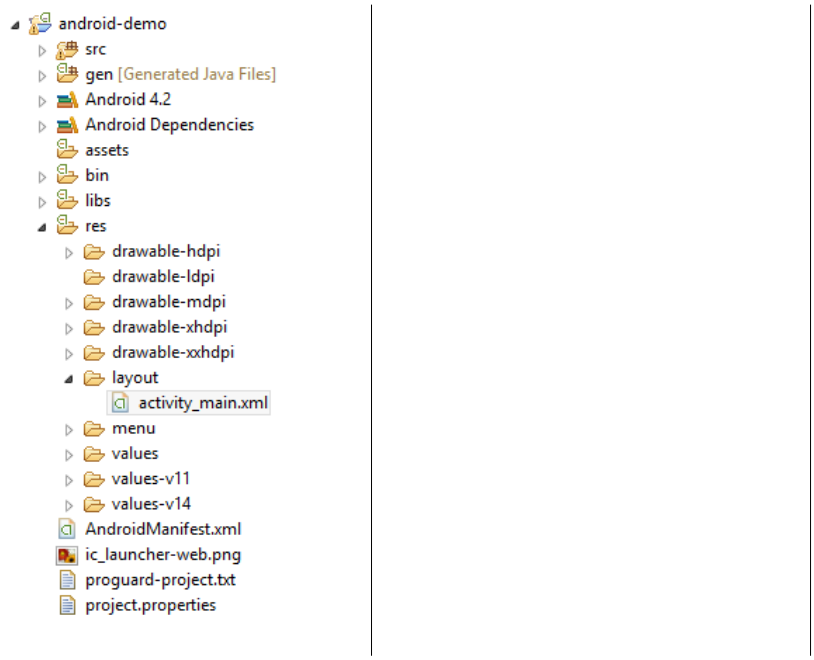
- en [1], on crée un projet de type [Android Application Project] ;



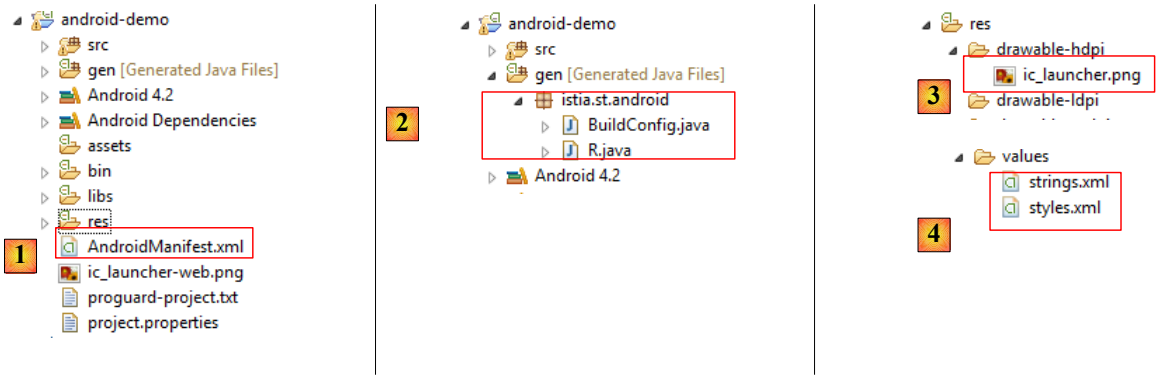


- en [2], on remplit les champs d'identité de l'application ;
- en [3], on garde les valeurs proposées par défaut sauf pour le champ [Minimum Required SDK] qui fixe la version la plus ancienne d'Android sur laquelle l'application peut être exécutée. On va utiliser des onglets. Ils ne sont supportés qu'à partir de la version 11 ;
- on valide les valeurs par défaut de l'assistant jusqu'à la dernière page. Là il y a quelque chose à changer : en [4], choisir la navigation par onglets ;

Le projet créé est le suivant :



10.3 Le manifeste de l'application



Le fichier [AndroidManifest.xml] [1] fixe les caractéristiques de l'application Android. Son contenu est ici le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.      package="istia.st.android"
4.      android:versionCode="1"
5.      android:versionName="1.0" >
6.
7.      <uses-sdk
8.          android:minSdkVersion="11"
9.          android:targetSdkVersion="17" />
10.
11.     <application
12.         android:allowBackup="true"
13.         android:icon="@drawable/ic_launcher"
14.         android:label="@string/app_name"
15.         android:theme="@style/AppTheme" >
16.         <activity
17.             android:name="istia.st.android.MainActivity"
18.             android:label="@string/app_name" >
```

```
19.             <intent-filter>
20.                 <action android:name="android.intent.action.MAIN" />
21.
22.                 <category android:name="android.intent.category.LAUNCHER" />
23.             </intent-filter>
24.         </activity>
25.     </application>
26.
27. </manifest>
```

- ligne 3 : le paquetage du projet Android. Un certain nombre de classes seront automatiquement générées dans ce paquetage [2] ;
- ligne 8 : la version minimale d'Android pouvant exécuter l'application. Ici la version 11, une version récente est nécessaire pour disposer des onglets, des fragments [Fragment] et d'une activité de type [FragmentActivity] ;
- ligne 9 : la version maximale d'Android. Mettre la dernière version de cet OS ;
- ligne 13 : l'icône [3] de l'application ;
- ligne 14 : le libellé de l'aplication. Il se trouve dans le fichier [strings.xml] [4] :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <resources>
3.
4.      <string name="app_name">Android-Demo</string>
5.      <string name="menu_settings">Settings</string>
6.      <string name="hello_world">Hello world!</string>
7.      <string name="title_section1">Section 1</string>
8.      <string name="title_section2">Section 2</string>
9.      <string name="title_section3">Section 3</string>
10.
11. </resources>
```

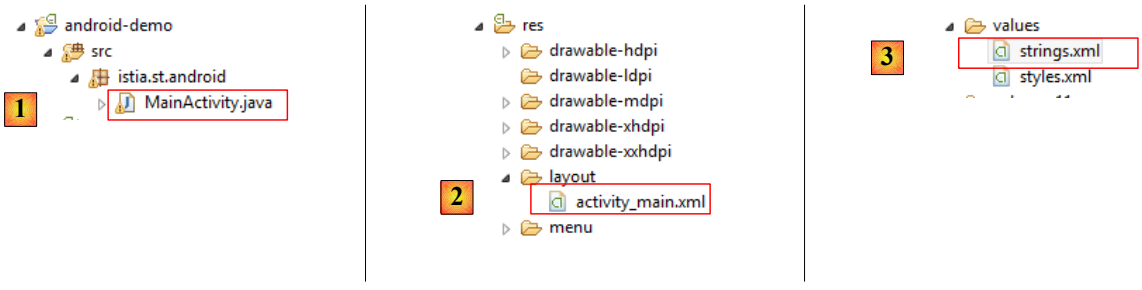
Le fichier [strings.xml] contient les chaînes de caractères utilisées par l'application.

- ligne 15 : le style de l'interface visuelle. Elle est définie dans le fichier [styles.xml] [4] :

```
1.      <style name="AppBaseTheme" parent="android:Theme.Light"/>
2.      <!-- Application theme. -->
3.  <style name="AppTheme" parent="AppBaseTheme"/>
```

- ligne 18 : une balise d'activité. Une application Android peut avoir plusieurs activités ;
- ligne 17 : le nom complet de la classe de l'activité ;
- ligne 18 : son libellé ;
- ligne 20 : l'activité est désignée comme étant l'activité principale ;
- ligne 22 : et elle doit apparaître dans la liste des applications qu'il est possible de lancer sur l'appareil Android.

10.4 L'activité principale



La classe générée est la suivante :

```
1.  package istia.st.android;
2.
3.  ...
```

```
4.  public class MainActivity extends FragmentActivity implements ActionBar.TabListener {
5.
6.      @Override
7.      protected void onCreate(Bundle savedInstanceState) {
8.          super.onCreate(savedInstanceState);
9.          setContentView(R.layout.activity_main);
10.
11.         // Set up the action bar to show tabs.
12.         final ActionBar actionBar = getActionBar();
13.         actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
14.
15.         // For each of the sections in the app, add a tab to the action bar.
16.         actionBar.addTab(actionBar.newTab().setText(R.string.title_section1).setTabListener(this));
17.         actionBar.addTab(actionBar.newTab().setText(R.string.title_section2).setTabListener(this));
18.         actionBar.addTab(actionBar.newTab().setText(R.string.title_section3).setTabListener(this));
19.     }
20.
21.     @Override
22.     public void onTabSelected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction) {
23.         // When the given tab is selected, show the tab contents in the
24.         // container view.
25.         Fragment fragment = new DummySectionFragment();
26.         Bundle args = new Bundle();
27.         args.putInt(DummySectionFragment.ARG_SECTION_NUMBER, tab.getPosition() + 1);
28.         fragment.setArguments(args);
29.         getSupportFragmentManager().beginTransaction().replace(R.id.container, fragment).commit();
30.     }
31.
32.     @Override
33.     public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction) {
34.     }
35.
36.     @Override
37.     public void onTabReselected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction) {
38.     }
39.
40.     /**
41.      * A dummy fragment representing a section of the app, but that simply
42.      * displays dummy text.
43.      */
44.     public static class DummySectionFragment extends Fragment {
45.         /**
46.          * The fragment argument representing the section number for this fragment.
47.          */
48.         public static final String ARG_SECTION_NUMBER = "section_number";
49.
50.         public DummySectionFragment() {
51.         }
52.
53.         @Override
54.         public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
55.             // Create a new TextView and set its text to the fragment's section
56.             // number argument value.
57.             TextView textView = new TextView(getActivity());
58.             textView.setGravity(Gravity.CENTER);
59.             textView.setText(Integer.toString(getArguments().getInt(ARG_SECTION_NUMBER)));
60.             return textView;
61.         }
62.     }
63.
64. }
```

- ligne 4 : la classe [MainActivity] étend la classe [FragmentActivity] et implémente l'interface [TabListener] qui permet de gérer des onglets. Les méthodes de cette interface sont les méthodes des lignes 22, 33 et 37;
- ligne 7 : la méthode [onCreate] est exécutée lorsque l'activité est créée. C'est avant l'affichage de la vue associée à l'activité ;
- ligne 8 : la méthode [onCreate] de la classe parente est appelée ;
- ligne 9 : le fichier [activity_main.xml] [2] est la vue associée à l'activité. La définition XML de cette vue est la suivante :

```
1.  <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      xmlns:tools="http://schemas.android.com/tools"
3.      android:id="@+id/container"
4.      android:layout_width="match_parent"
```

```
5.      android:layout_height="match_parent"
6.      tools:context=".MainActivity"
7.  tools:ignore="MergeRootFrame" />
```

La vue ne contient qu'un unique composant, un conteneur de type [FrameLayout] (ligne 1) nommé [container] (ligne 3).

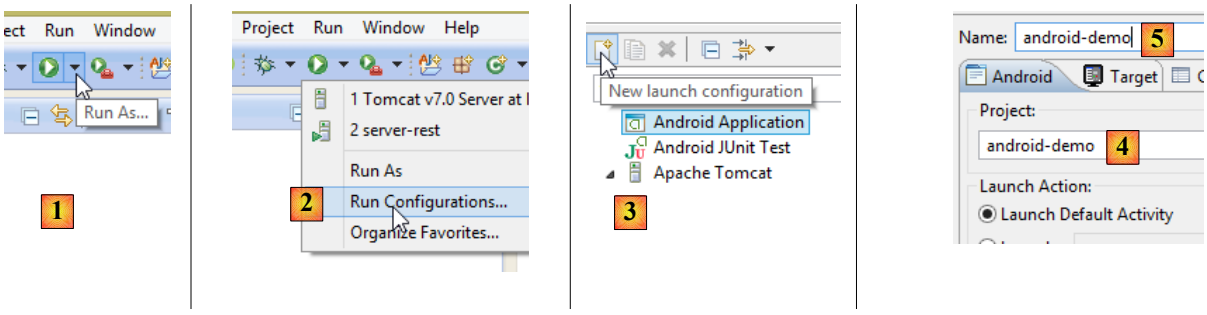
- ligne 12 : on récupère la barre d'action de l'activités. C'est elle qui va contenir les onglets ;
- ligne 13 : le mode de navigation est celui d'onglets qu'on clique ;
- lignes 16-18 : trois onglets sont construits. Un onglet [Tab] est construit comme suit :
    - *actionBar.newTab()* : construit un onglet [Tab] et rend sa référence,
    - *[Tab].setText(libellé)* : fixe un libellé à l'onglet et rend [Tab]. Le libellé est défini dans le fichier [strings.xml] [3] :

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <resources>
3.
4.     <string name="app_name">Android-Demo</string>
5.     <string name="menu_settings">Settings</string>
6.     <string name="hello_world">Hello world!</string>
7.     <string name="title_section1">Section 1</string>
8.     <string name="title_section2">Section 2</string>
9.     <string name="title_section3">Section 3</string>
10.
11.  </resources>
```

    - *[Tab].setTabListener* : fixe la classe qui gère les onglets et qui donc implémente l'interface [TabListener], ici *this*,
- ligne 22 : la méthode exécutée lorsque l'utilisateur clique sur un onglet. Le 1er paramètre est l'onglet cliqué. Le second paramètre est une transaction qui permet de changer le fragment affiché dans la vue principale [activity_main.xml] ;
- ligne 25 : on crée un fragment du type [DummySectionFragment] de la ligne 44 ;
- ligne 26 : on va passer des arguments au fragment dans un type [Bundle] ;
- ligne 27 : on met dans le bunble, une entrée de clé [section_number] et de valeur le n° de l'onglet plus 1 ;
- ligne 28 : cet argument est passé au fragment ;
- ligne 29 : le fragment est placé dans la vue principale [activity_main.xml] à la place du composant nommé [container]. Comme la vue principale n'avait que ce composant, le fragment remplace donc totalement la vue précédente ;
- ligne 44 : la classe [DummyFragment] étend la classe [Fragment] ;
- ligne 54 : la méthode exécutée juste avant l'affichage du fragment. Elle doit rendre l'objet [View] affiché par le fragment ;
- ligne 57 : un objet [TextView] est créé. C'est un composant texte. Son paramètre est l'activité principale obtenue avec *getActivity*() ;
- ligne 58 : le composant [TextView] s'affichera centré dans le fragment ;
- ligne 59 : fixe le texte du composant [TextView]. Ce texte sera le n° de l'onglet qu'on a mis dans les arguments passés au fragment ;
- ligne 60 : l'objet [TextView] est rendu. Le fragment se limite donc à ce composant.

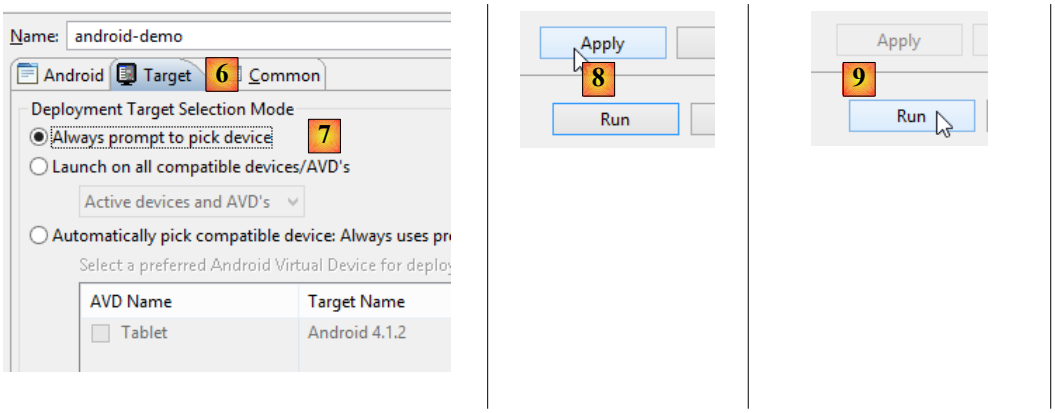
10.5 Exécution de l'application

Pour exécuter une application Android, il nous faut créer une configuration d'exécution :

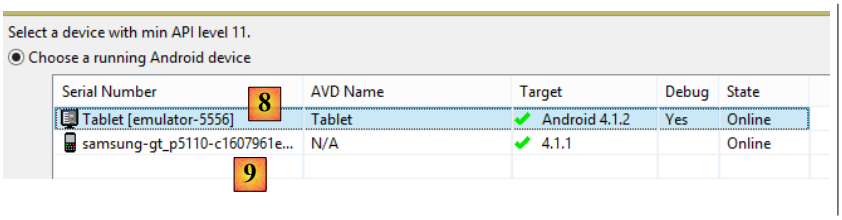


- en [1], sélectionnez l'icône [Run as...] ;
- en [2], sélectionnez l'option [Run Configurations...] ;
- en [3], sélectionnez le type [Android Application] puis l'icône [New launch configuration] ;

- en [4], indiquez le projet qui sera exécuté par cette configuration ;
- en [5], donnez un nom à cette configuration. Peut être quelconque ;

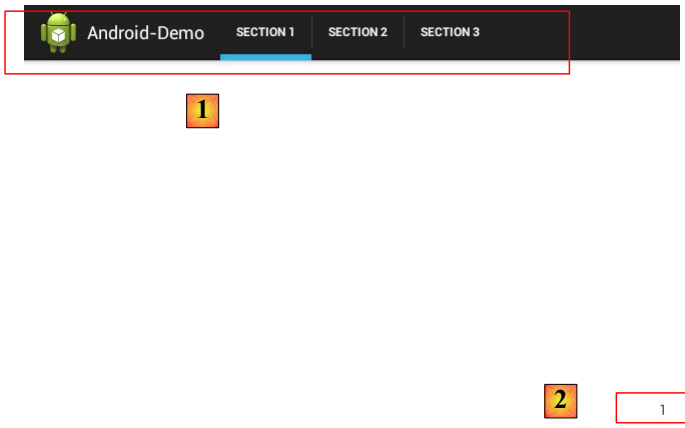


- dans l'onglet [Target] [6], sélectionnez l'option [7]. Elle permet de choisir le mode d'exécution : en mode émulation avec une tablette logicielle ou en mode réel avec une tablette Android ;
- en [8], validez cette configuration ;
- en [9], exécutez-la ;



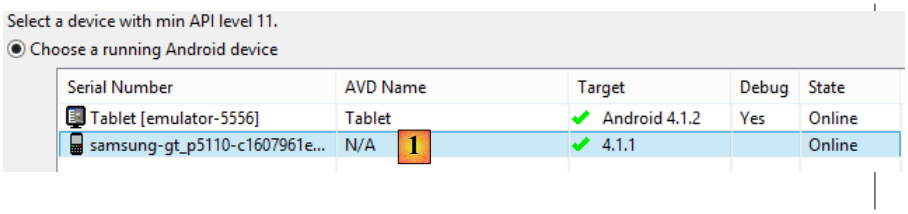
- en [8], un émulateur de tablette ;
- en [9], une tablette Android ;
- sélectionnez l'émulateur de tablette et exécutez l'application ;

L'émulateur logiciel est alors lancé et affiche la vue suivante :



- en [1], les trois onglets générés par l'application. L'onglet [SECTION 1] est sélectionné ;
- il affiche en [2] son composant [TextView] avec le n° de l'onglet. Cliquez sur les différents onglets : le n° affiché doit changer ;

Branchez maintenant une tablette Android sur un port USB du PC serveur et exécutez l'application sur celle-ci :



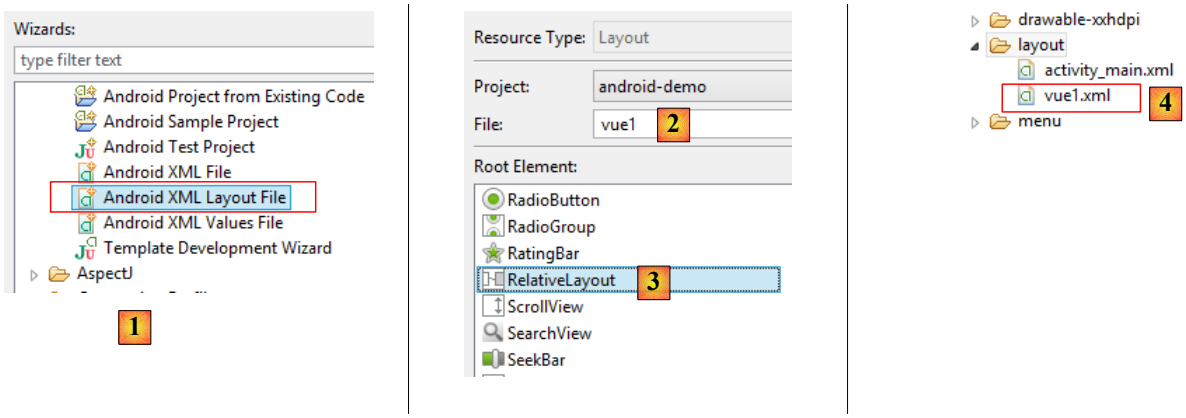
- en [1], sélectionnez la tablette Android et testez l'application.

10.6 Construire une vue

L'exemple précédent construisait avec du code la vue à afficher par le fragment :

```
1.      @Override
2.      public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
3.          // Create a new TextView and set its text to the fragment's section
4.          // number argument value.
5.          TextView textView = new TextView(getActivity());
6.          textView.setGravity(Gravity.CENTER);
7.          textView.setText(Integer.toString(getArguments().getInt(ARG_SECTION_NUMBER)));
8.          return textView;
9. }
```

Nous allons maintenant construire une vue avec l'éditeur graphique d'Eclipse ADT.

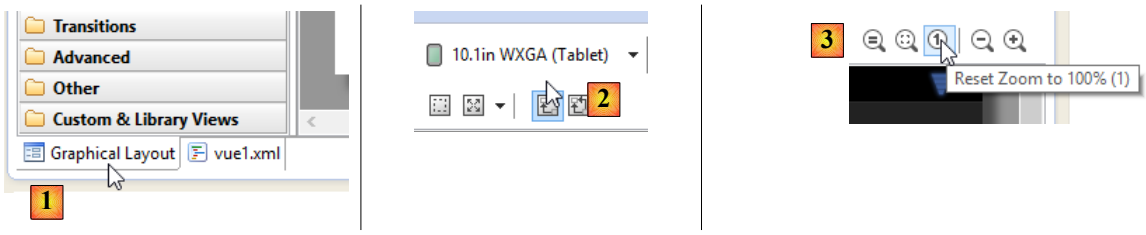


- en [1], créez un nouveau projet de type [Android XML Layout File] qui sert à décrire une vue à l'aide d'un fichier XML ;
- en [2], nommez la vue ;
- en [3], indiquez la balise racine de la vue. Ici, nous choisissons un conteneur [RelativeLayout]. Dans ce conteneur de composants, ceux-ci sont placés les uns par rapport aux autres : " à droite de ", " à gauche de ", " au-dessous de ", " au-dessus de " ;

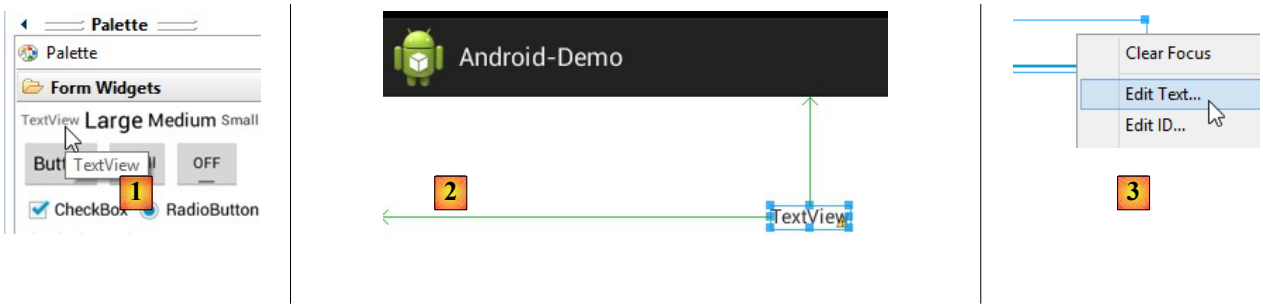
Le fichier [vue1.xml] généré [4] est le suivant :

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.     android:layout_width="match_parent"
4.     android:layout_height="match_parent" >
5. </RelativeLayout>
```

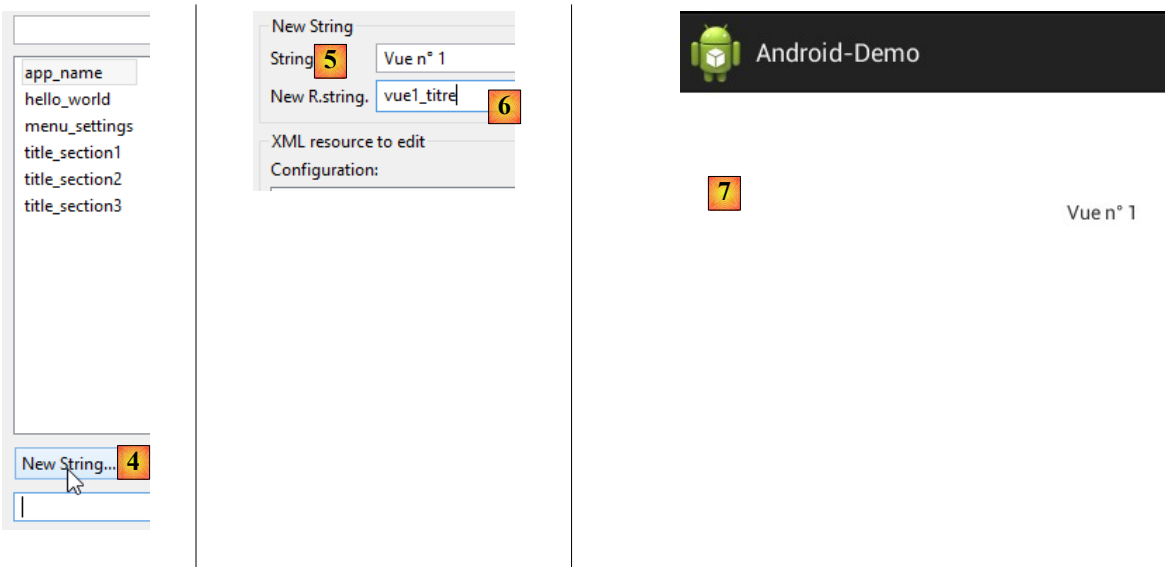
- ligne 2 : un conteneur [RelativeLayout] vide qui occupera toute la largeur de la tablette (ligne 3) et toute sa hauteur (ligne 4) ;



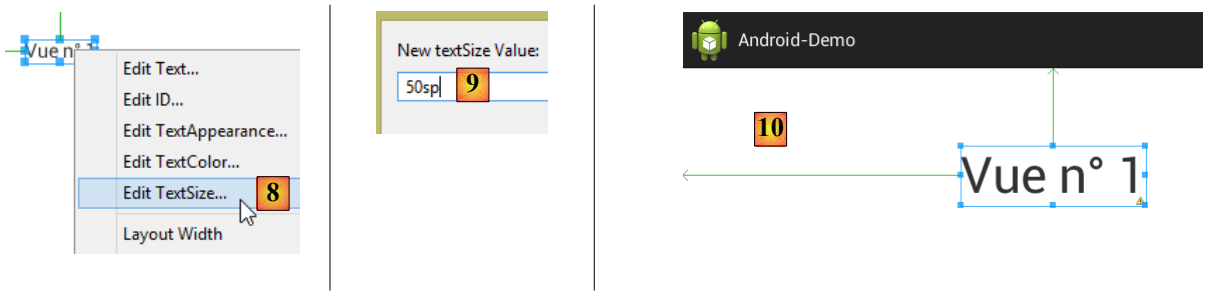
- en [1], sélectionnez l'onglet [Graphical Layout] ;
- en [2], mettez-vous en mode tablette ;
- en [3], mettez-vous à l'échelle 1 de la tablette ;



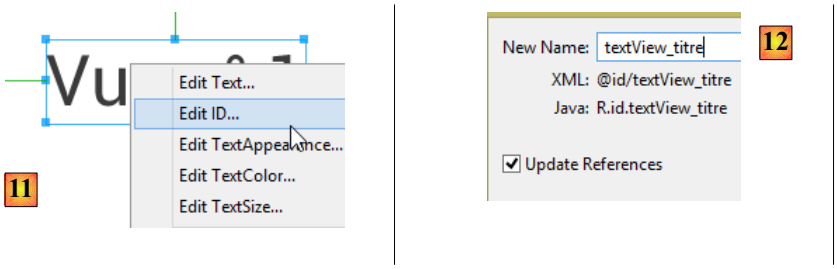
- en [1], prendre un [TextView] et le tirer sur la vue [2] ;
- en [3], fixer le texte du composant ;



- en [4], on veut créer une nouvelle chaîne de caractères dans le fichier [strings.xml] ;
- en [5], le texte de la chaîne de caractères créée ;
- en [6], l'identifiant de la chaîne de caractères créée ;
- en [7], l'interface visuelle se met à jour ;



- en [8], modifier la taille du texte ;
- en [9], mettre une taille, ici 50 pixels ;
- en [10], la nouvelle vue ;



- en [11] et [12], modifier lidentifiant du composant ;

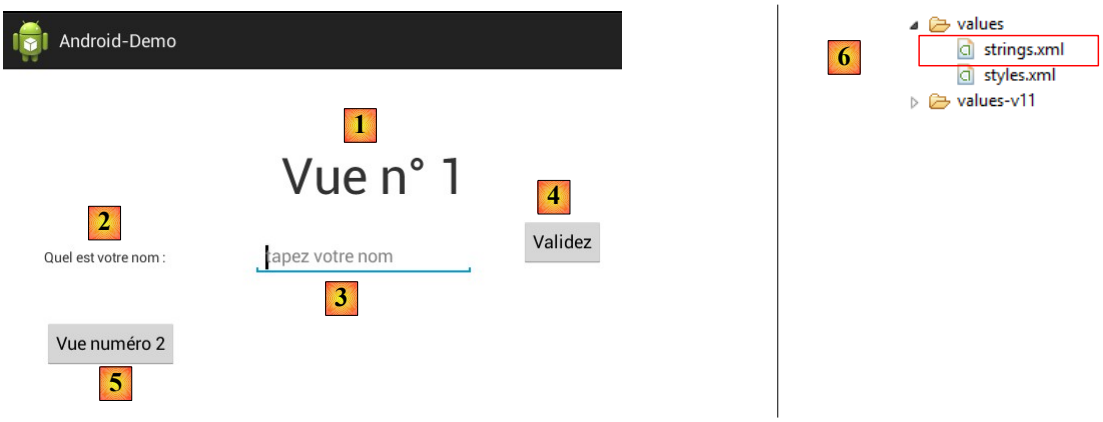
Le fichier [vue1.xml] a évolué comme suit :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:layout_width="match_parent"
4.      android:layout_height="match_parent" >
5.
6.      <TextView
7.          android:id="@+id/textView_titre"
8.          android:layout_width="wrap_content"
9.          android:layout_height="wrap_content"
10.         android:layout_alignParentLeft="true"
11.         android:layout_alignParentTop="true"
12.         android:layout_marginLeft="278dp"
13.         android:layout_marginTop="77dp"
14.         android:text="@string/vue1_titre"
15.         android:textSize="50sp" />
16.
17. </RelativeLayout>
```

- les modifications faites dans l'interface graphique sont aux lignes 7, 14 et 15. Les autres attributs du [TextView] sont des valeurs par défaut ou bien découlent du positionnement du composant dans la vue ;
- lignes 8-9 : la taille du composant est celle du texte qu'elle contient (wrap_content) en hauteur et largeur ;
- lignes 11, 13 : le haut du composant est aligné avec le haut de la vue (ligne 11), 77 pixels dessous (ligne 13) ;
- lignes 10, 12 : le côté gauche du composant est aligné avec la gauche de la vue (ligne 10), 278 pixels à droite (ligne 12) ;

En général, les tailles exactes des marges gauche, droite, haute et basse seront fixées directement dans le XML.

En procédant de la même façon, créez la vue suivante [1] :



Les composants sont les suivants :

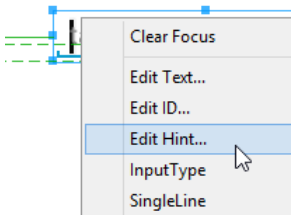
N°	Id	Type	Rôle
1	textView_titre	TextView	Titre de la vue
2	textView_nom	TextView	un texte
3	editText_Nom	EditText	saisie d'un nom
4	button_valider	Button	pour valider la saisie
5	button_vue2	Button	pour passer à la vue n° 2

Le fichier XML est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:layout_width="match_parent"
4.      android:layout_height="match_parent" >
5.
6.      <TextView
7.          android:id="@+id/textView_titre"
8.          android:layout_width="wrap_content"
9.          android:layout_height="wrap_content"
10.         android:layout_alignParentLeft="true"
11.         android:layout_alignParentTop="true"
12.         android:layout_marginLeft="278dp"
13.         android:layout_marginTop="77dp"
14.         android:text="@string/vue1_titre"
15.         android:textSize="50sp" />
16.
17.     <TextView
18.         android:id="@+id/textView_nom"
19.         android:layout_width="wrap_content"
20.         android:layout_height="wrap_content"
21.         android:layout_alignParentLeft="true"
22.         android:layout_below="@+id/textView_titre"
23.         android:layout_marginLeft="41dp"
24.         android:layout_marginTop="42dp"
25.         android:text="@string/vue1_txt_nom" />
26.
27.     <EditText
28.         android:id="@+id/editText_Nom"
29.         android:layout_width="wrap_content"
30.         android:layout_height="wrap_content"
31.         android:layout_alignBaseline="@+id/textView_nom"
32.         android:layout_alignBottom="@+id/textView_nom"
33.         android:layout_marginLeft="87dp"
34.         android:layout_toRightOf="@+id/textView_nom"
35.         android:ems="10"
36.         android:hint="@string/vue1_hint_nom" >
```

```
37.
38.         <requestFocus />
39.     </EditText>
40.
41.     <Button
42.         android:id="@+id/button_valider"
43.         android:layout_width="wrap_content"
44.         android:layout_height="wrap_content"
45.         android:layout_alignBottom="@+id/editText_Nom"
46.         android:layout_marginLeft="47dp"
47.         android:layout_toRightOf="@+id/editText_Nom"
48.         android:text="@string/btn_valider" />
49.
50.     <Button
51.         android:id="@+id/button_vue2"
52.         android:layout_width="wrap_content"
53.         android:layout_height="wrap_content"
54.         android:layout_alignLeft="@+id/textView_nom"
55.         android:layout_below="@+id/editText_Nom"
56.         android:layout_marginTop="53dp"
57.         android:text="@string/btn_vue2" />
58.
59. </RelativeLayout>
```

- lignes 22, 24 : le composant [*textView_nom*] est positionné dessous le composant [*textView_titre*] ;
- lignes 32, 34 : le composant [*editText_Nom*] est positionné à droite du composant [*textView_nom*] à une distance de 87 pixels. Il a un attribut *hint* ligne 36. Cet attribut est obtenu de la façon suivante :



- lignes 45-47 : le composant [*button_valider*] est positionné à droite du composant [*editText_Nom*] à une distance de 47 pixels ;
- ligne 54 : le côté gauche du composant [*button_vue2*] est aligné avec le côté gauche du composant [*textView_nom*] ;
- lignes 55-56 : le composant [*button_vue2*] est positionné à 53 pixels au-dessous du composant [*editText_Nom*] ;

Tous les textes proviennent du fichier [strings.xml] [6] suivant :

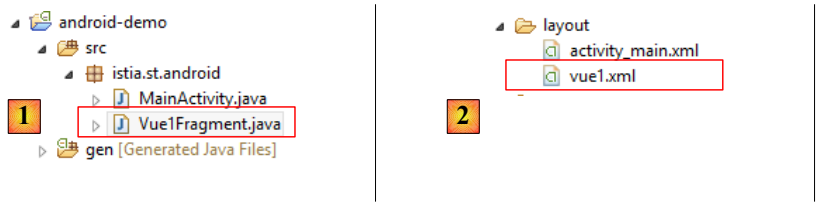
```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <resources>
3.
4.      <string name="app_name">Android-Demo</string>
5.      <string name="menu_settings">Settings</string>
6.      <string name="hello_world">Hello world!</string>
7.      <string name="title_section1">Section 1</string>
8.      <string name="title_section2">Section 2</string>
9.      <string name="title_section3">Section 3</string>
10.     <string name="vue1_txt_nom">Quel est votre nom :</string>
11.     <string name="vue1_titre">Vue n° 1</string>
12.     <string name="vue1_hint_nom">tapez votre nom</string>
13.     <string name="btn_valider">Validez</string>
14.     <string name="btn_vue2">Vue numéro 2</string>
15.
16. </resources>
```

Maintenant, modifions l'activité [MainActivity] pour que cette vue corresponde à l'onglet n° 1 :

```
1.  @Override
2.     public void onTabSelected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction)
    {
3.        // When the given tab is selected, show the tab contents in the
4.        // container view.
5.        int position = tab.getPosition();
6.        Fragment fragment=null;;
7.        switch (position) {
8.        case 0:
9.         // on crée un fragment de type [Vue1Fragment]
10.        fragment=new Vue1Fragment();
11.          break;
12.       default:
13.        // on crée un fragment de type [DummySectionFragment]
14.          fragment = new DummySectionFragment();
15.          Bundle args = new Bundle();
16.          args.putInt(DummySectionFragment.ARG_SECTION_NUMBER, tab.getPosition() + 1);
17.          fragment.setArguments(args);
18.       }
19.       getSupportFragmentManager().beginTransaction().replace(R.id.container,
    fragment).commit();
20.    }
```

- ligne 5 : on récupère la position de l'onglet cliqué ;
- lignes 8-11 : si c'est l'onglet n° 0, on crée un fragment de type [vue1Fragment] ;
- lignes 12-18 : pour les autres onglets ça ne change pas ;

Le fragment [Vue1Fragment] [1] est le suivant :



Le code de la classe [Vue1Fragment] est le suivant :

```
1.  package istia.st.android;
2.
3.  import android.os.Bundle;
4.  import android.support.v4.app.Fragment;
5.  import android.view.LayoutInflater;
6.  import android.view.View;
7.  import android.view.ViewGroup;
8.
9.  public class Vue1Fragment extends Fragment {
10.
11.    @Override
12.    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
    savedInstanceState) {
13.       // on crée la vue du fragment à partir de sa définition XML
14.       return inflater.inflate(R.layout.vue1, container, false);
15.    }
16.
17. }
```

- ligne 9 : le fragment [Vue1Fragment] étend la classe Android [Fragment] ;
- ligne 12 : la méthode [onCreateView] qui doit rendre une référence sur l'objet [View] à afficher ;
- ligne 14 : cet objet [View] est la vue définie par le fichier XML [vue1.xml] [2] ;

10.7 Gestion des événements

Gérons maintenant le clic sur le bouton [Validez] de la vue [Vue1] :

Le code de [MainActivity] évolue comme suit :

```
1.  @Override
2.     public void onActivityCreated(Bundle savedInstanceState) {
3.        // parent
4.        super.onActivityCreated(savedInstanceState);
5.        // activité sous-jacente aux vues
6.        Activity activity = getActivity();
7.        // on récupère les composants
8.        edtNom = (EditText) activity.findViewById(R.id.editText_Nom);
9.        btnValider = (Button) activity.findViewById(R.id.button_valider);
10.       btnVue2 = (Button) activity.findViewById(R.id.button_vue2);
11.       // gestionnaire d'évts
12.       btnValider.setOnClickListener(new OnClickListener() {
13.          @Override
14.          public void onClick(View arg0) {
15.             doValider();
16.          }
17.       });
18.
19.       // bouton Annuler
20.       btnVue2.setOnClickListener(new OnClickListener() {
21.          @Override
22.          public void onClick(View arg0) {
23.             // on passe à la vue n° 2
24.             navigateToView2();
25.          }
26.       });
27.    }
28.
29.    protected void navigateToView2() {
30.       // TODO Auto-generated method stub
31.
32.    }
33.
34.    protected void doValider() {
35.       // on affiche le nom saisi
```

```
36.       Toast.makeText(getActivity(), String.format("Bonjour %s",
    edtNom.getText().toString()),Toast.LENGTH_LONG).show();
37.    }
```

- ligne 2 : la méthode exécutée lorsque l'activité est créée. On l'utilise souvent pour récupérer des références sur les composants de la vue qui va être affichée ;
- ligne 4 : la méthode du parent est appelée. C'est obligatoire ;
- ligne 6 : on récupère l'activité qui affiche les fragments ;
- ligne 8 : on récupère la référence du composant d'id [*editText_Nom*] ;
- ligne 9 : on récupère la référence du composant d'id [*button_valider*] ;
- ligne 10 : on récupère la référence du composant d'id [*button_vue2*] ;
- lignes 12-17 : on définit un gestionnaire pour l'événement *clic* sur le bouton [Valider] ;
- ligne 34 : la méthode qui gère ce clic ;
- ligne 36 : affiche le nom saisi :
    - *Toast.makeText(...).show()* : affiche un texte à l'écran,
    - le 1er paramètre de *makeText* est l'activité du fragment,
    - le second paramètre est le texte à afficher dans la boîte qui va être affichée par *makeText,*
    - le troisième paramètre est la durée de vie de la boîte affichée : *Toast.LENGTH_LONG* ou *Toast.LENGTH_SHORT* ;

Le clic sur le bouton [Vue numéro 2] pourrait être géré comme suit :

```
1.  protected void navigateToView2() {
2.        // on affiche l'onglet n° 2
3.        // l'activité
4.        MainActivity activity=(MainActivity) getActivity();
5.        // la barre d'actions
6.        ActionBar actionBar=activity.getActionBar();
7.        // on sélectionne l'onglet n° 2
8.        actionBar.selectTab(actionBar.getTabAt(1));
9.     }
```

- ligne 2 : le but est de sélectionner l'onglet n° 2 ce qui déclenchera l'exécution de la méthode [onTabSelected] de l'activité [MainActivity] ;

10.8 Conclusion

Nous en savons assez sur la façon de construire un projet Android. Nous pouvons revenir au TP.

11 Le client natif Android du projet Domotique

Nous abordons maintenant l'écriture du client Android.

A lire

- le document [**ref2**] jusqu'à l'exemple 2 inclus. Nous allons suivre cet exemple pour écrire notre client Android. Le modèle AVAT doit être compris pour la suite du TP.

11.1 Les vues du client

Le client Android permet de gérer les Arduinos à distance. Il présente à l'utilisateur les écrans suivants :

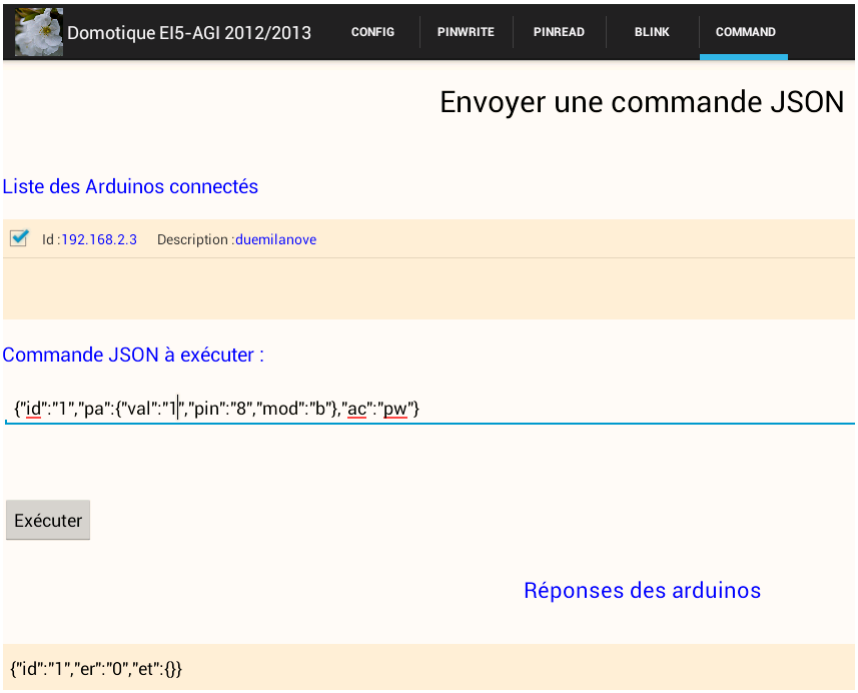
L'onglet [CONFIG] permet de se connecter au serveur et de récupérer la liste des Arduinos connectés :

L'onglet [PINWRITE] permet d'écrire une valeur sur une pin d'un Arduino :

L'onglet [PINREAD] permet de lire la valeur d'une pin d'un Arduino :

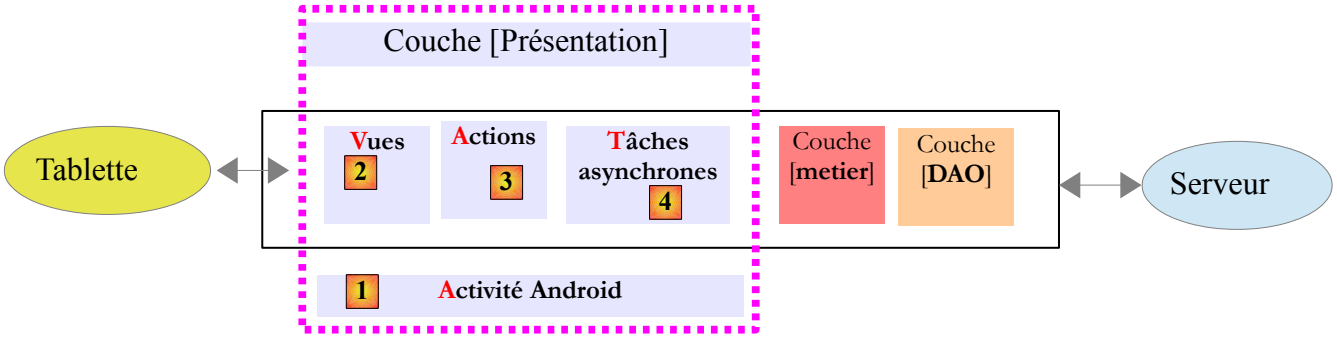
L'onglet [BLINK] permet de faire clignoter une led d'un Arduino :

L'onglet [COMMAND] permet d'envoyer une commande JSON à un Arduino :



11.2 L'architecture du client

L'architecture du client Android sera la suivante :



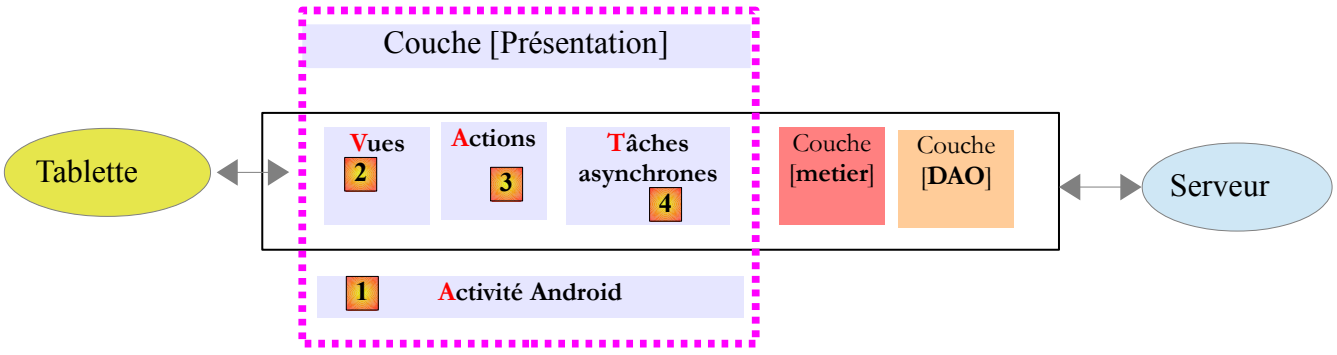
- la couche [DAO] communique avec le serveur REST. C'est un client REST implémenté par **Spring-Android** ;
- la couche [métier] reprend l'interface de la couche [métier] du serveur ;

Les versions actuelles d'Android exigent que les connexions réseau soient faites dans un autre thread que celui qui gère les interfaces visuelles. C'est ce qui explique la présence du bloc [4]. Les méthodes de la couche [métier] sont exécutées au sein d'un thread différent de celui de l'UI (**U**ser **I**nterface). Lorsqu'on fait exécuter une méthode de la couche[métier] dans un thread, on peut attendre ou non la réponse de la méthode. Dans le premier cas, synchrone, on bloque l'UI en attendant la réponse de la tâche. Dans le second cas, asynchrone, l'UI reste disponible pour l'utilisateur qui peut lancer d'autres actions. Ici, on a choisi la solution asynchrone pour deux raisons :

- le client Android doit pouvoir commander plusieurs Arduinos simultanément. Par exemple, on veut pouvoir faire clignoter une led placée sur deux Arduinos, en même temps et non pas l'une après l'autre. On ne peut donc pas attendre la fin de la première tâche pour commencer la seconde ;
- les connexions réseau peuvent être longues ou ne pas aboutir. Pour cette raison, on veut donner à l'utilisateur la possibilité d'interrompre une action qu'il a lancée. Pour cela, l'UI ne doit pas être bloquée.

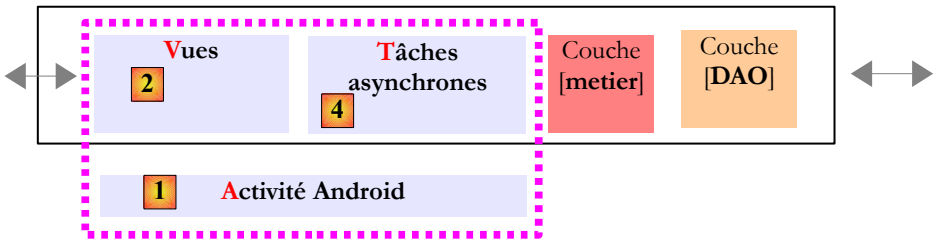
Les vues [2] sont les interfaces visuelles présentées à l'utilisateur. Sur une tablette, elles ressemblent à ceci :

A partir de cette vue, l'utilisateur lance une action avec le bouton [Exécuter].



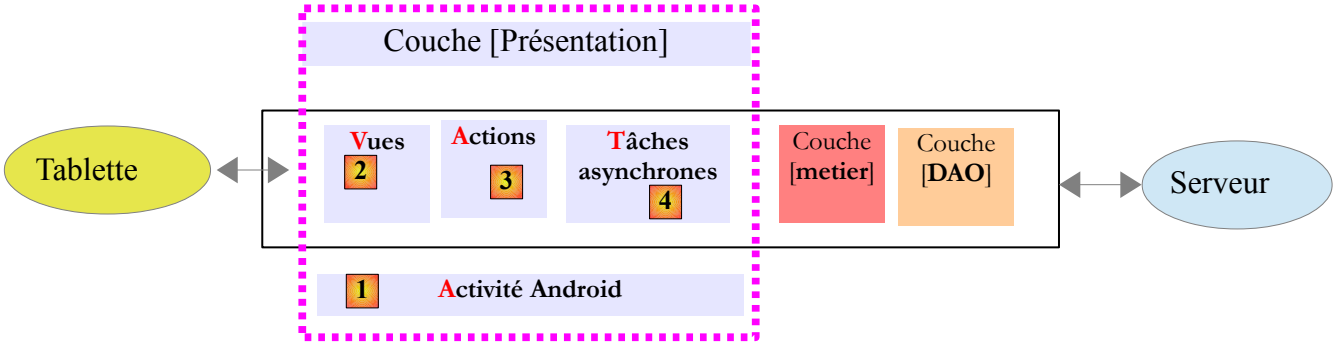
Le bloc [3] regroupe les actions exécutées par les vues. La vue ne fait que saisir des données et contrôler leur validité. On part du principe qu'elle ne sait pas à quoi elles vont servir. Elle sait simplement à quelle action elle doit transmettre les données saisies. L'action lancera les tâches asynchrones nécessaires, mettra en forme leurs résultats et rendra à la vue un modèle pour que celle-ci se mette à jour. La vue n'est pas bloquée par l'action, afin de donner à l'utilisateur la possibilité de l'interrompre.

Les activités [1] sont le coeur d'une application Android. Ici on n'en a qu'une et elle ne fait quasiment rien si ce n'est d'assurer les changements de vues. On appellera ce modèle, **AVAT** (**A**ctivité – **V**ues – **A**ctions – **T**âches). Il peut être allégé de la façon suivante :

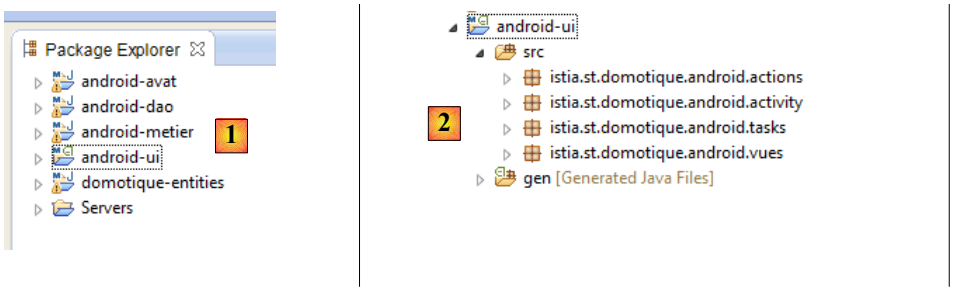


Dans le modèle AVAT, la vue est complètement ignorante de l'utilisation faite des données qu'elle a saisies. Dans le modèle **AVT** ci-dessus (**A**ctivité – **V**ues – **T**âches), la logique de l'action est transférée dans la vue. On trouve donc un peu de logique dans la vue. Celle-ci doit organiser les appels des tâches asynchrones elle-même. Cette logique est forcément faible. Si elle était importante, elle serait normalement transférée dans la couche [métier]. Ce modèle AVT est suffisant dans tous les cas.

11.3 Les projets Eclipse du client



Les projets Android suivent l'architecture ci-dessus :



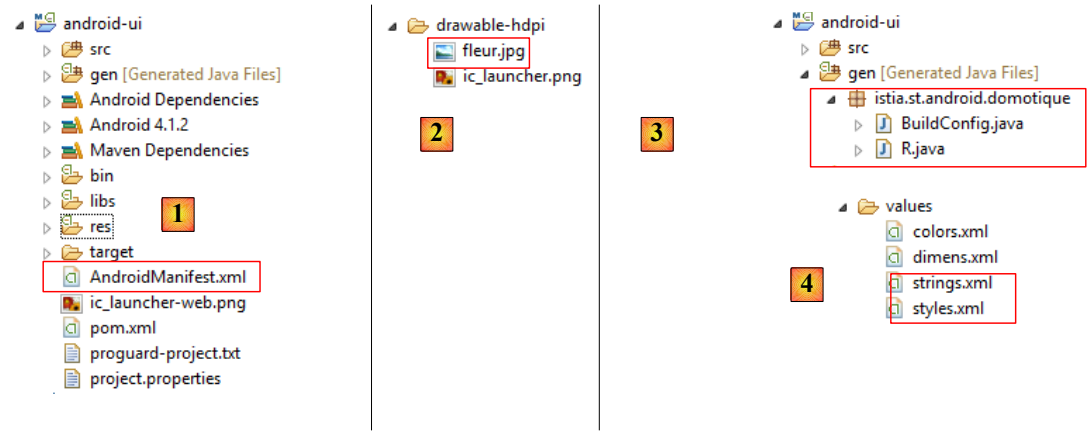
En [1] les cinq projets Eclipse du client Android :

- [domotique-entities] : projet déjà utilisé pour le serveur. Rassemble les entités échangées entre le client Android et le serveur REST ;
- [android-dao] : la couche [DAO] du client Android ;
- [android-metier] : la couche [métier] du client Android ;
- [android-ui], [android-avat] : la couche [présentation] du client Android qui s'appuie sur le modèle AVAT décrit dans [**ref2**].

En [2] les packages de la couche AVAT du client Android :

- [istia.st.domotique.android.**activity**] : rassemble ce qui a trait au 1er **A** de **A**VAT, l'**activité** ;
- [istia.st.domotique.android.**vues**] : rassemble ce qui a trait au **V** de A**V**AT, les **vues** ;
- [istia.st.domotique.android.**actions**] : rassemble ce qui a trait au 2ième **A** de AV**A**T, les **actions** ;
- [istia.st.domotique.android.**tasks**] : rassemble ce qui a trait au **T** de AVA**T**, les **tâches** ;

11.4 Le manifeste de l'application



Le fichier [AndroidManifest.xml] configure l'application. C'est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3.      package="istia.st.android.domotique"
4.      android:versionCode="1"
5.      android:versionName="1.0" >
6.
7.      <uses-sdk
8.          android:minSdkVersion="14"
9.          android:targetSdkVersion="16" />
10.
11.     <uses-permission android:name="android.permission.INTERNET" />
12.
13.     <application
14.         android:allowBackup="true"
15.         android:icon="@drawable/fleur"
16.         android:label="@string/app_name"
17.         android:theme="@style/AppTheme" >
18.         <activity
19.             android:name="istia.st.domotique.android.activity.MainActivity"
20.             android:label="@string/app_name"
21.             android:windowSoftInputMode="stateHidden" >
22.             <intent-filter>
23.                 <action android:name="android.intent.action.MAIN" />
24.
25.                 <category android:name="android.intent.category.LAUNCHER" />
26.             </intent-filter>
27.         </activity>
28.     </application>
29.
30. </manifest>
```

- ligne 3 : le paquetage du projet Android. Un certain nombre de classes seront automatiquement générées dans ce paquetage [3] ;
- ligne 8 : la version minimale d'Android pouvant exécuter l'application. Ici la version 14, une version récente ;
- ligne 9 : la version maximale d'Android. Mettre la dernière version de cet OS ;
- ligne 11 : ligne à mettre si l'application ouvre des connexions Internet. C'est le cas ici. Si on l'oublie, on a une exception parfois peu compréhensible lors des ouvertures de connexion réseau ;
- ligne 15 : l'icône [2] de l'application ;
- ligne 16 : le libellé de l'aplication. Il se trouve dans le fichier [strings.xml] [4] :

```
<string name="app_name">Domotique EI5-AGI 2012/2013</string>
```

- ligne 17 : le style de l'interface visuelle. Elle est définie dans le fichier [styles.xml] [4] :

```
1.      <style name="AppBaseTheme" parent="android:Theme.Light"/>
```

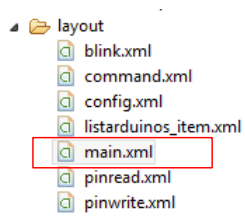
```
2.      <!-- Application theme. -->
3.  <style name="AppTheme" parent="AppBaseTheme"/>
```

- ligne 18 : une balise d'activité. Une application Android peut avoir plusieurs activités ;
- ligne 19 : le nom complet de la classe de l'activité ;
- ligne 20 : son libellé ;
- ligne 21 : une ligne qui cache le clavier logiciel. A l'affichage de la première vue, le curseur se positionne sur le premier champ de saisie. Le clavier logiciel est alors affiché et cache une partie de la vue. C'est disgracieux. Cette ligne élimine cet affichage ;
- ligne 23 : l'activité est désignée comme étant l'activité principale ;
- ligne 25 : et elle doit apparaître dans la liste des applications qu'il est possible de lancer sur l'appareil Android.

11.5 Le patron des vues

A lire :

- exemple 6 de [**ref2**] ;

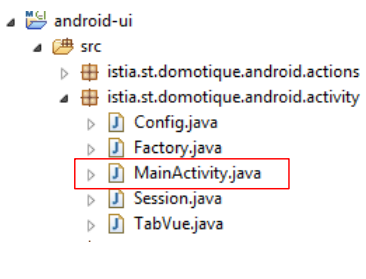


On lira complètement l'exemple 6 de [**ref2**] et on l'exécutera. Le patron [main.xml] des vues Android sera le suivant :

```
1.  <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.      xmlns:tools="http://schemas.android.com/tools"
3.      android:id="@+id/container"
4.      android:layout_width="match_parent"
5.      android:layout_height="match_parent"
6.      android:background="@color/floral_white"
7.      tools:context=".MainActivity"
8.  tools:ignore="MergeRootFrame" />
```

- ligne 1 : un unique conteneur de composants de type [FrameLayout] identifié par [*container*] (ligne 3). Le conteneur ne contient aucun composant ;

11.6 L'activité principale



L'activité a pour rôle principal de gérer cinq onglets. Lorsque l'un d'eux est cliqué, la vue correspondante doit être affichée. Il y en a cinq. Elles ont été présentées au paragraphe 11.1, page 82.

L'activité Android [MainActivity] est la suivante :

```
1.  package istia.st.domotique.android.activity;
```

```
2.
3.  ...
4.
5.  public class MainActivity extends FragmentActivity implements ActionBar.TabListener {
6.
7.    // on a 1 activité qui gère 5 vues (Fragments)
8.    // l'activité gère l'affichage de ces vues et les données qui leur sont
9.    // nécessaires - c'est tout
10.
11.   // les 5 vues
12.   private TabVue[] tabVues = new TabVue[5];
13.
14.   // exécuté à la création de l'activité
15.   @Override
16.   protected void onCreate(Bundle savedInstanceState) {
17. ...
18.   }
19.
20.   // affichage de certains onglets
21.   public void showTabs(Boolean[] show) {
22. ...
23.   }
24.
25.   // changement d'onglet
26.   @Override
27.   public void onTabSelected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction) {
28. ...
29.   }
30.
31.   @Override
32.   public void onTabReselected(Tab arg0, FragmentTransaction arg1) {
33.     // onglet resélectionné - on ne gère pas
34.   }
35.
36.   @Override
37.   public void onTabUnselected(Tab arg0, FragmentTransaction arg1) {
38.     // onglet déselectionné - on ne gère pas
39.   }
40.
41. }
```

- ligne 5 : la classe [MainActivity] étend la classe [FragmentActivity]. Ceci est imposé par le modèle AVAT ;
- ligne 5 : la classe [MainActivity] implémente l'interface [TabListener] car elle gère des onglets. Les méthodes de cette interface sont les méthodes des lignes 27, 32 et 37 ;
- ligne 12 : l'activité gère cinq vues qui sont mémorisées dans un tableau. Les éléments de celui-ci sont de type [TabVue] :

```
1.  package istia.st.domotique.android.activity;
2.
3.  import istia.st.avat.android.Vue;
4.  import android.app.ActionBar.Tab;
5.
6.  public class TabVue {
7.    // l'onglet
8.    private Tab tab;
9.    // la vue
10.   private Vue vue;
11.
12.   // constructeurs
13.   public TabVue() {
14.
15.   }
16.
17.   public TabVue(Tab tab, Vue vue) {
```

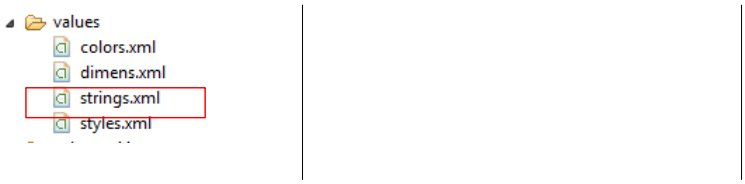
```
18.     this.tab = tab;
19.     this.vue = vue;
20.   }
21.
22.   // getters et setters
23. ...
24. }
```

- le type [TabVue] associe à un onglet [8] la vue que celui-ci affiche [10] ;

Retour au code de l'activité :

```
1.  // exécuté à la création de l'activité
2.    @Override
3.    protected void onCreate(Bundle savedInstanceState) {
4.      // parent
5.      super.onCreate(savedInstanceState);
6.      // le sablier
7.      requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
8.      setProgressBarIndeterminateVisibility(false);
9.      // template qui accueille les vues
10.     setContentView(R.layout.main);
11.     // la factory
12.     Factory factory = new Factory(this, new Config());
13.     // la session
14.     ISession session = (ISession) factory.getObject(Factory.SESSION, (Object[]) null);
15.     // on fixe l'URL par défaut du service REST
16.     session.add("urlServiceRest", getText(R.string.edt_UrlServiceRest).toString());
17.
18. ...
19.
20.   }
```

- ligne 3 : la méthode exécutée par toute activité lorsqu'elle est créée ;
- ligne 5 : appel de la méthode parent (obligatoire) ;
- ligne 7 : affiche une image animée en haut et à droite de la vue montrant qu'une opération longue est en cours ;
- ligne 8 : pour l'instant cette image n'est pas affichée ;
- ligne 10 : le modèle des vues [main.xml] devient la vue courante ;
- ligne 12 : la fabrique des objets AVAT est instanciée. On lui passe une classe de configuration sur laquelle on reviendra ;
- ligne 24 : on demande à la fabrique une référence sur la session AVAT. On s'en servira pour passer des informations d'une vue à l'autre ;
- ligne 16 : on met dans la session, une URL par défaut pour le service REST. Elle est prise dans le fichier [strings.xml] :




```
1.      <!-- Config -->
2.      <string name="txt_TitreConfig">Se connecter au serveur</string>
3.      <string name="txt_UrlServiceRest">Url du service REST</string>
4.      <string name="edt_UrlServiceRest">192.168.2.1:8080/rest</string>
5.      <string name="txt_MsgErreurUrlServiceRest">L\'Url du service doit être entrée sous
    la forme Ip1.Ip2.Ip3.IP4:Port/contexte</string>
6.      <string name="hint_UrlServiceRest">ex (192.168.1.120:8080/rest)</string>
7.  <string name="btn_annuler">Annuler</string>
```

La ligne 16 de l'activité utilise le message de la ligne 4 de [strings.xml].

Retour au code de l'activité :

```
1.  // exécuté à la création de l'activité
2.    @Override
3.    protected void onCreate(Bundle savedInstanceState) {
4.      ...
5.      // on fixe l'URL par défaut du service REST
6.      session.add("urlServiceRest", getText(R.string.edt_UrlServiceRest).toString());
7.
8.      // on utilise des onglets dans la barre d'actions
9.      ActionBar actionBar = getActionBar();
10.     actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
11.
12.     // on définit les 5 vues
13.     tabVues[0] = new
    TabVue(actionBar.newTab().setText(R.string.title_config).setTabListener(this),
14.         (Vue) factory.getObject(Factory.CONFIG_VUE, (Object[]) null));
15.     tabVues[1] = new
    TabVue(actionBar.newTab().setText(R.string.title_pinwrite).setTabListener(this),
16.         (Vue) factory.getObject(Factory.PINWRITE_VUE, (Object[]) null));
17.     tabVues[2] = new
    TabVue(actionBar.newTab().setText(R.string.title_pinread).setTabListener(this),
18.         (Vue) factory.getObject(Factory.PINREAD_VUE, (Object[]) null));
19.     tabVues[3] = new
    TabVue(actionBar.newTab().setText(R.string.title_blink).setTabListener(this),
20.         (Vue) factory.getObject(Factory.BLINK_VUE, (Object[]) null));
21.     tabVues[4] = new
    TabVue(actionBar.newTab().setText(R.string.title_command).setTabListener(this),
22.         (Vue) factory.getObject(Factory.COMMAND_VUE, (Object[]) null));
23.
24.     // au départ on a une liste d'arduinos vide dans la session
25.     session.add("arduinos", new Arduino[0]);
26.
27.     // on affiche le 1er onglet
28.     showTabs(new Boolean[] { true, false, false, false, false });
29.
30.   }
```

- ligne 9 : on récupère la barre d'action de l'activités. C'est elle qui va contenir les onglets ;
- ligne 10 : le mode de navigation est celui d'onglets qu'on clique ;
- lignes 13-21 : on définit les cinq vues de l'application ;
- lignes 13-14 : pour définir un élément de type [TabVue], il faut fournir deux éléments : un onglet [Tab] et une vue [Vue]. Lorsqu'on clique sur [Tab] alors la vue [Vue] est affichée. L'onglet [Tab] est construit comme suit :
    - *actionBar.newTab()* : construit un onglet [Tab] et rend sa référence,
    - *[Tab].setText(libellé)* : fixe un libellé à l'onglet et rend [Tab]. Celui-ci est défini dans le fichier [strings.xml] :

```
1.      <!-- les onglets -->
2.      <string name="title_config">Config</string>
3.      <string name="title_pinwrite">PinWrite</string>
4.      <string name="title_pinread">PinRead</string>
5.      <string name="title_blink">Blink</string>
6.  <string name="title_command">Command</string>
```

    - *[Tab].setTabListener* : fixe la classe qui gère les onglets et qui donc implémente l'interface [TabListener], ici *this*,
- ligne 14 : la référence de la vue à associer à l'onglet qui vient d'être construit est demandé à la fabrique :
    - 1er paramètre : le n° de l'objet à construire,
    - 2ème paramètre : un tableau d'arguments à passer à la vue. Ici, il n'y a pas d'arguments à passer ;
- ligne 28 : le 1er onglet est affiché ;
- ligne 25 : les différentes vues maintiennent en session la liste des Arduinos connectés sous forme d'un tableau. Au départ de l'activité, celui-ci est vide ;

La méthode [showTabs] qui affiche certains onglets est la suivante :

```
1.    // affichage de certains onglets
2.    public void showTabs(Boolean[] show) {
3.      // si show[i] est vrai, affiche la vue n° i
4.      ActionBar actionBar = getActionBar();
5.      // on passe ttes les vues en revue
6.      for (int i = 0; i < tabVues.length; i++) {
7.        Tab tab = tabVues[i].getTab();
8.        int position = tab.getPosition();
9.        if (show[i]) {
10.         // la vue doit être affichée si elle ne l'est pas déjà
11.         if (position == Tab.INVALID_POSITION) {
12.           actionBar.addTab(tab);
13.         }
14.       } else {
15.         // la vue doit être enlevée si elle ne l'est pas déjà
16.         if (position != Tab.INVALID_POSITION) {
17.           actionBar.removeTab(tab);
18.         }
19.       }
20.     }
21. }
```

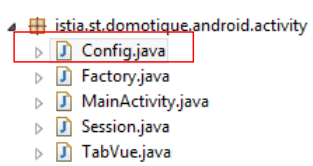
- ligne 7 : on récupère l'onglet n° i ;
- ligne 8 : on récupère sa position dans la liste des onglets affichés. Le résultat *INVALID_POSITION* signifie que l'onglet n'est pas affiché ;
- ligne 9 : si l'onglet n° i doit être affiché ;
- ligne 11 : s'il n'est pas déjà affiché, alors on l'affiche (ligne 12) ;
- ligne 14 : l'onglet ne doit pas être affiché ;
- ligne 16 : si l'onglet est déjà affiché, alors il faut l'enlever (ligne 17) ;

La gestion des onglets est faite avec les trois méthodes de l'interface [TabListener] :

```
1.  // changement d'onglet
2.    @Override
3.    public void onTabSelected(ActionBar.Tab tab, FragmentTransaction fragmentTransaction) {
4.      // on cherche l'onglet sélectionné
5.      Boolean trouvé = false;
6.      int i = 0;
7.      while (!trouvé) {
8.        trouvé = tab == tabVues[i].getTab();
9.        i++;
10.     }
11.     // la vue de l'onglet sélectionné est installée dans son conteneur
12.     fragmentTransaction.replace(R.id.container, tabVues[i - 1].getVue());
13.   }
14.
15.   @Override
16.   public void onTabReselected(Tab arg0, FragmentTransaction arg1) {
17.     // onglet resélectionné - on ne gère pas
18.   }
19.
20.   @Override
21.   public void onTabUnselected(Tab arg0, FragmentTransaction arg1) {
22.     // onglet déselectionné - on ne gère pas
23.   }
```

- ligne 3 : on ne gère que le clic qui sélectionne un onglet ;
    - *tab* : l'onglet cliqué,
    - *fragmentTransaction* : un objet qui permet de changer de fragment (Vue) dans le patron [main.xml] ;
- lignes 7-10 : l'onglet cliqué est cherché dans le tableau des onglets. L'élément trouvé a la position (i-1) dans le tableau ;
- ligne 12 ; dans le modèle [main.xml], on vient remplacer l'élément nommé [container] par la vue (Vue) associée à l'onglet cliqué ;

11.7 La configuration de l'application



L'application est configurée par la classe [Config] suivante :

```
1.  package istia.st.domotique.android.activity;
2.
3.  public class Config {
4.    // ici on met la config de l'application
5.
6.    // propriété verbose - à vrai on loguera les évts des tâches (WORK_STARTED,
7.    // WORK_TERMINATED, WORK_INFO, CANCELLED)
8.    private boolean verbose = true;
9.    // timeout des connexions au serveur en millisecondes
10.   private int timeout=1000;
11.
12.   // getters
13.   ...
14.
15. }
```

- ligne 8 : le booléen *verbose* contrôle le mode verbeux ou non du modèle AVAT. Le laisser à vrai le temps du débogage ;
- ligne 10 : fixe un délai maximal pour la réponse du serveur, ici 1 seconde. Au-delà, la couche [DAO] lance une exception ;

11.8 La fabrique de l'application

La fabrique est la classe qui fabrique les objets de l'application. Le modèle AVAT essaie de travailler le plus possible avec des interfaces plutôt qu'avec des objets concrets. Cela facilite l'évolution d'une application et ses tests. Les objets concrets sont délégués à une fabrique ayant pour seule méthode :

```
Object getObject(int num, Object... params) ;
```

La méthode [getObject] reçoit deux paramètres :

- le n° d'objet *num* de l'objet à construire ;
- une liste de paramètres pour initialiser l'objet à construire. Ces paramètres sont non typés parce qu'on ne peut prévoir a priori ce qu'ils seront ;

Le squelette de la fabrique est le suivant :

```
1.  package istia.st.domotique.android.activity;
2.
3.  ...
4.
5.  public class Factory implements IFactory {
```

```
6.
7.    // constantes
8.    public final static int CONFIG = 0;
9.    public final static int METIER = 1;
10.   public final static int SESSION = 2;
11.
12.   // les vues
13.   public final static int CONFIG_VUE = 102;
14. ...
15.   // les actions
16.   public final static int CONFIG_ACTION = 202;
17. ...
18.   // les tâches
19.   public final static int CONFIG_TASK = 302;
20. ...
21.
22.   // ------------------------------------ DEBUT SINGLETONS
23.   // la configuration
24.   private Config config;
25.   private boolean verbose;
26.   private int timeout;
27.
28.   // la couche métier
29.   private IMetier metier;
30.
31.   // la session
32.   private ISession session;
33.
34.   // les vues
35.   private Vue configVue;
36.   private Vue blinkVue;
37.   private Vue pinReadVue;
38.   private Vue pinWriteVue;
39.   private Vue commandVue;
40.
41.   // l'activité principale
42.   private Activity activity;
43.
44.   // ------------------------------------ FIN SINGLETONS
45.
46.   // constructeur
47.   public Factory(Activity activity, Config config) {
48.     // activité
49.     this.activity = activity;
50.     // la configuration
51.     this.config = config;
52.     verbose = config.getVerbose();
53.     timeout=config.getTimeout();
54.   }
55.
56.   // ------------------------------------ getObject
57.   public Object getObject(int id, Object... params) {
58.     switch (id) {
59.     case CONFIG:
60.       return config;
61.     case CONFIG_VUE:
62.       return getConfigVue();
63. ...
64.     case CONFIG_ACTION:
65.       return getConfigAction(params);
66. ...
67.     case CONFIG_TASK:
68.       return getConfigTask(params);
```

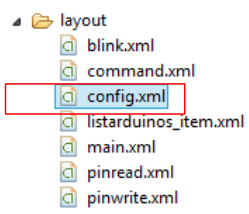
```
69. ...
70.     case METIER:
71.       return getMetier();
72.     case SESSION:
73.       return getSession();
74.     }
75.
76.     // pour le compilateur
77.     return null;
78.   }
79.
80.   // session
81.   private Object getSession() {
82.     if (session == null) {
83.       session = new Session();
84.     }
85.     return session;
86.   }
87.
88.   // couche métier
89.   private Object getMetier() {
90.     if (metier == null) {
91.       // couche dao
92.       IDao dao = new Dao();
93.       dao.setTimeout(timeout);
94.       // couche métier
95.       metier = new Metier();
96.       metier.setDao(dao);
97.     }
98.     return metier;
99.   }
100.
101.    // initialisation d'une vue
102.    private void init(Vue vue) {
103.      // la fabrique
104.      vue.setFactory(this);
105.      // l'équipe de travailleurs
106.      ITeam team = new Team();
107.      team.setMonitor(vue);
108.      vue.setTeam(team);
109.      // l'activité
110.      vue.setActivity(activity);
111.      // le mode verbeux ou non
112.      vue.setVerbose(verbose);
113.      // l'identifiant de la vue
114.      String longName = vue.getClass().getName();
115.      String[] parts = longName.split("\\.");
116.      vue.setBossId(parts[parts.length - 1]);
117.    }
118.
119.   // configVue
120.    private Object getConfigVue() {
121.      if (configVue == null) {
122.        configVue = new ConfigVue();
123.        init(configVue);
124.      }
125.      return configVue;
126.    }
127.
128.    // les tâches reçoivent deux paramètres : IBoss, id
129.    private Object getConfigTask(Object[] params) {
130.      ITask configTask = new ConfigTask();
131.      init(configTask, params);
```

```
132.      return configTask;
133.    }
134.
135.    // iitialisation d'une tâche terminale
136.    private void init(ITask task, Object[] params) {
137.      // le boss
138.      task.setBoss((IBoss) params[0]);
139.      // l'identifiant
140.      task.setWorkerId((String) params[1]);
141.      // la fabrique
142.      task.setFactory(this);
143.    }
144.
145.    // les actions reçoivent deux paramètres : IBoss, id
146.    private Object getConfigAction(Object[] params) {
147.      IAction configAction = new ConfigAction();
148.      init(configAction, params);
149.      return configAction;
150.    }
151.
152.    // initialisation d'une action
153.    private void init(IAction action, Object[] params) {
154.      // le boss
155.      action.setBoss((IBoss) params[0]);
156.      // l'équipe de travailleurs
157.      ITeam team = new Team();
158.      team.setMonitor(action);
159.      action.setTeam(team);
160.      // les identifiants
161.      action.setWorkerId((String) params[1]);
162.      action.setBossId((String) params[1]);
163.      // la fabrique
164.      action.setFactory(this);
165.      // le mode verbeux ou non
166.      action.setVerbose(verbose);
167.    }
168.  }
```

Nous ne présentons ici que certains éléments de la fabrique :

- lignes 81-86 : la session – instanciée en un seul exemplaire ;
- lignes 89-99 : les couches [métier] et [DAO] – instanciées en un seul exemplaire ;
- lignes 102-126 : la vue *CONFIG_VUE* qui est la 1ère vue présentée à l'utilisateur – instanciée en un seul exemplaire ;
- lignes 146-167 : l'action CONFIG_ACTION lancée par la vue CONFIG_VUE – instanciée à chaque demande ;
- lignes 129-143 : la tâche CONFIG_TASK lancée par l'action CONFIG_ACTION - instanciée à chaque demande ;

11.9 L'interface de la vue [Config]



La vue [config.xml] est la suivante :

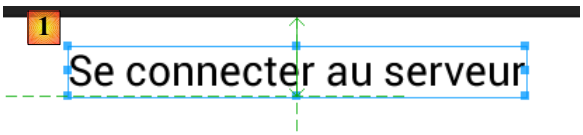
Le code XML de cette vue est le suivant :

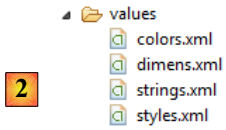
```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      xmlns:tools="http://schemas.android.com/tools"
4.      android:id="@+id/RelativeLayout1"
5.      android:layout_width="match_parent"
6.      android:layout_height="match_parent">
7.
8.      <TextView
9.          android:id="@+id/txt_TitreConfig"
10.         ../>
11.
12.     <TextView
13.         android:id="@+id/txt_UrlServiceRest"../>
14.
15.     <EditText
16.         android:id="@+id/edt_UrlServiceRest"..>
17.
18.         <requestFocus />
19.     </EditText>
20.
21.     <TextView
22.         android:id="@+id/txt_MsgErreurIpPort"../>
23.
24.     <TextView
25.         android:id="@+id/txt_arduinos"...>
26.
27.     <Button
28.         android:id="@+id/btn_Rafraichir"../>
29.
30.     <Button
31.         android:id="@+id/btn_Annuler"../>
32.
33.     <ListView
34.         android:id="@+id/ListViewArduinos"..>
35.     </ListView>
36.
37. </RelativeLayout>
```

- ligne 2 : les composants de la vue sont dans " *RelativeLayout* ". Dans ce conteneur de composants, ceux-ci sont placés les uns par rapport aux autres : " à droite de ", " à gauche de ", " au-dessous de ", " au-dessus de " ;
- ligne 5 : le conteneur va occuper tout l'espace horizontal de la tablette ;
- ligne 6 : idem pour l'espace vertical ;
- ligne 8 : le texte [1] ;
- ligne 12 : le texte [2] ;
- ligne 15 : le champ de saisie [3] ;

- ligne 21 : le message d'erreur [4] ;
- ligne 24 : le texte [5] ;
- ligne 27 : le bouton [6] ;
- ligne 30 : un bouton caché sous le bouton [6] ;
- ligne 33 : la liste [7] ;

11.9.1 Le texte [1]





Le texte [1] est obtenu par le code XML suivant :

```
1.  <TextView
2.          android:id="@+id/txt_TitreConfig"
3.          android:layout_width="wrap_content"
4.          android:layout_height="wrap_content"
5.          android:layout_alignParentTop="true"
6.          android:layout_centerHorizontal="true"
7.          android:layout_marginTop="20dp"
8.          android:text="@string/txt_TitreConfig"
9.          android:textSize="@dimen/titre" />
```

- ligne 2 : son identifiant [txt_TitreConfig]. Ce qui précède [@+id] est obligatoire ;
- ligne 3 : sa largeur – aussi large que le nécessite le texte affiché ;
- ligne 4 : sa hauteur – aussi haut que le nécessite le texte affiché ;
- ligne 5 : le haut du composant sera aligné sur le haut du conteneur ;
- ligne 7 : le composant aura une marge haute de 20 pixels. Il sera donc 20 pixels dessous le haut du conteneur ;
- ligne 6 : le composant sera centré horizontalement dans son conteneur ;
- ligne 8 : le texte à afficher est dans le fichier [Strings.xml] [2] identifié par [txt_TitreConfig] :

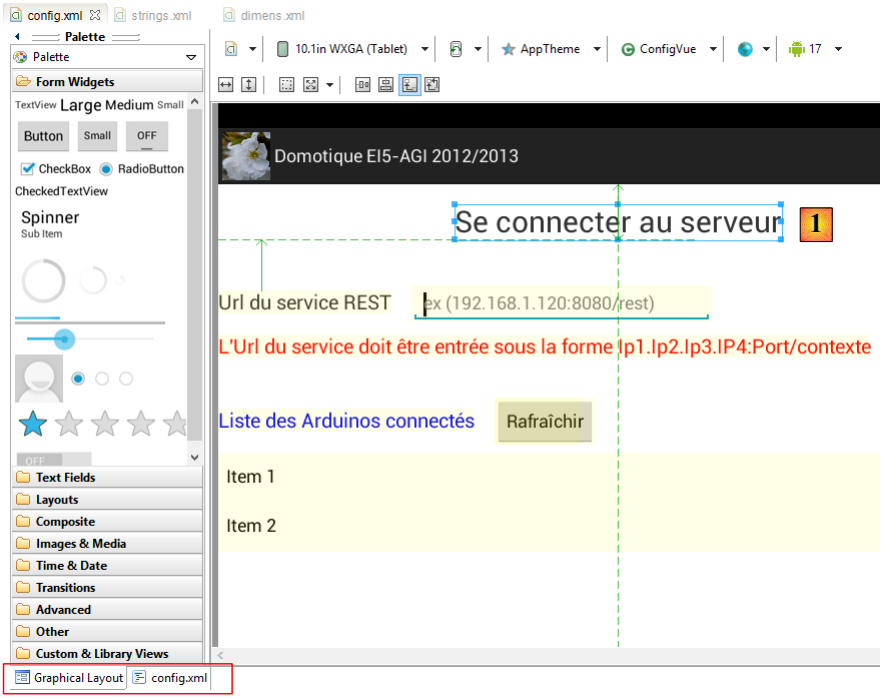
```
<string name="txt_TitreConfig">Se connecter au serveur</string>
```

- ligne 9 : la taille du texte est précisée dans le fichier [dimens.xml] [2] identifié par [titre] :

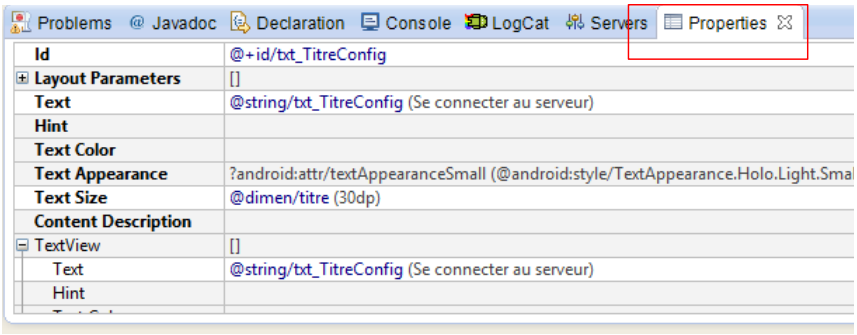
```
1. <?xml version="1.0" encoding="utf-8"?>
2. <resources>
3.     <dimen name="titre">30dp</dimen>
4. </resources>
```

Ligne 3, la taille [titre] est de 30 pixels.

Toutes ces propriétés sont accessibles depuis l'onglet [Graphical Layout] du fichier [config.xml] :



Les propriétés du composant [1] s'obtiennent par un clic droit :



11.9.2 Les autres composants de la vue

Les autres composants de la vue sont les suivants :

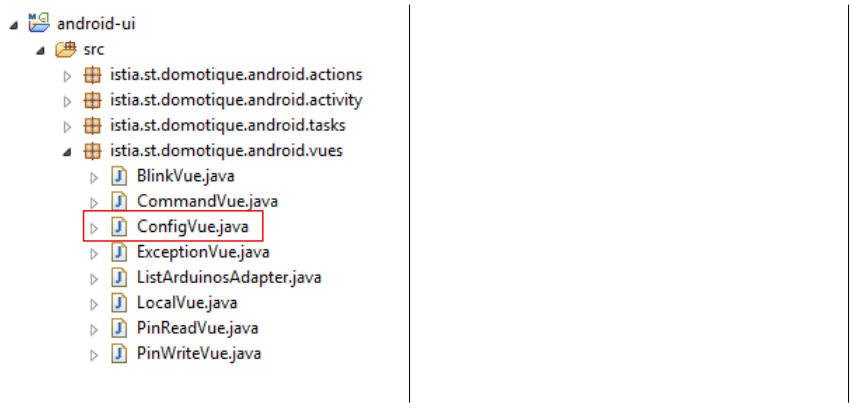
N°	Id	Type	Rôle
1	*txt_TitreConfig*	TextView	titre de la vue
2	*txt_UrlServiceRest*	TextView	libellé
3	*edt_UrlServiceRest*	EditText	saisie de l'URL du service REST
4	*txt_MsgErreurIpPort*	TextView	un message d'erreur affiché si l'URL saisie est incorrecte
5	*txt_arduinos*	TextView	libellé
6	*btn_Rafraichir*	Button	pour régénérer la liste des Arduinos connectés
6	*btn_Annuler*	Button	pour annuler la connexion au serveur si celle-ci est trop longue. Il est caché par le bouton [Rafraîchir].
7	*ListViewArduinos*	ListView	pour afficher la liste des Arduinos connectés

La description XML est la suivante :

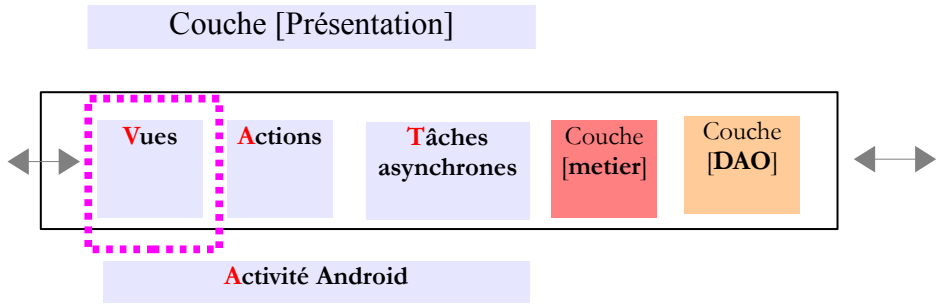
```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      xmlns:tools="http://schemas.android.com/tools"
4.      android:id="@+id/RelativeLayout1"
5.      android:layout_width="match_parent"
6.      android:layout_height="match_parent">
7.
8.      <TextView
9.          android:id="@+id/txt_TitreConfig"
10.         android:layout_width="wrap_content"
11.         android:layout_height="wrap_content"
12.         android:layout_alignParentTop="true"
13.         android:layout_centerHorizontal="true"
14.         android:layout_marginTop="20dp"
15.         android:text="@string/txt_TitreConfig"
16.         android:textSize="@dimen/titre" />
17.
18.     <TextView
19.         android:id="@+id/txt_UrlServiceRest"
20.         android:layout_width="wrap_content"
21.         android:layout_height="wrap_content"
22.         android:layout_alignParentLeft="true"
23.         android:layout_below="@+id/txt_TitreConfig"
24.         android:layout_marginTop="50dp"
25.         android:text="@string/txt_UrlServiceRest"
26.         android:textSize="20sp" />
27.
28.     <EditText
29.         android:id="@+id/edt_UrlServiceRest"
30.         android:layout_width="300dp"
31.         android:layout_height="wrap_content"
32.         android:layout_alignBaseline="@+id/txt_UrlServiceRest"
33.         android:layout_alignBottom="@+id/txt_UrlServiceRest"
34.         android:layout_marginLeft="20dp"
35.         android:layout_toRightOf="@+id/txt_UrlServiceRest"
36.         android:ems="10"
37.         android:hint="@string/hint_UrlServiceRest"
38.         android:inputType="textUri" >
39.
40.         <requestFocus />
41.     </EditText>
42.
43.     <TextView
44.         android:id="@+id/txt_MsgErreurIpPort"
45.         android:layout_width="wrap_content"
46.         android:layout_height="wrap_content"
47.         android:layout_alignParentLeft="true"
48.         android:layout_below="@+id/txt_UrlServiceRest"
```

```
49.         android:layout_marginTop="20dp"
50.         android:text="@string/txt_MsgErreurUrlServiceRest"
51.         android:textColor="@color/red"
52.         android:textSize="20sp" />
53.
54.     <TextView
55.         android:id="@+id/txt_arduinos"
56.         android:layout_width="wrap_content"
57.         android:layout_height="wrap_content"
58.         android:layout_alignParentLeft="true"
59.         android:layout_below="@+id/txt_MsgErreurIpPort"
60.         android:layout_marginTop="50dp"
61.         android:text="@string/titre_list_arduinos"
62.         android:textColor="@color/blue"
63.         android:textSize="20sp" />
64.
65.     <Button
66.         android:id="@+id/btn_Rafraichir"
67.         android:layout_width="wrap_content"
68.         android:layout_height="wrap_content"
69.         android:layout_alignBaseline="@+id/txt_arduinos"
70.         android:layout_alignBottom="@+id/txt_arduinos"
71.         android:layout_marginLeft="20dp"
72.         android:layout_toRightOf="@+id/txt_arduinos"
73.         android:text="@string/btn_rafraichir" />
74.
75.     <Button
76.         android:id="@+id/btn_Annuler"
77.         android:layout_width="wrap_content"
78.         android:layout_height="wrap_content"
79.         android:layout_alignBaseline="@+id/txt_arduinos"
80.         android:layout_alignBottom="@+id/txt_arduinos"
81.         android:layout_marginLeft="20dp"
82.         android:layout_toRightOf="@+id/txt_arduinos"
83.         android:text="@string/btn_annuler"
84.         android:visibility="invisible" />
85.
86.     <ListView
87.         android:id="@+id/ListViewArduinos"
88.         android:layout_width="match_parent"
89.         android:layout_height="100dp"
90.         android:layout_alignParentLeft="true"
91.         android:layout_below="@+id/txt_arduinos"
92.         android:layout_marginTop="20dp"
93.         android:clickable="true"
94.         tools:listitem="@android:layout/simple_list_item_multiple_choice" >
95.     </ListView>
96.
97. </RelativeLayout>
```

11.10 Le code de la vue [Config]



Nous décrivons maintenant la vue V du modèle AVAT :



Dans le modèle **AVAT**, la vue **V** saisit des données dans une interface Android. La validité de ces données est vérifiée. Si elles sont valides, la vue **V** lance une action **A** en lui passant les données saisies. L'action **A** a deux possibilités :

- si l'action **A** à faire doit exécuter une méthode réseau de la couche [métier], alors elle doit passer par une tâche T ;
- sinon l'action **A** fait exécuter une méthode de la couche [métier] ;
- la tâche asynchrone T exécute une méthode de la couche [métier] dans un thread à part ;
- la tâche T rend son résultat à l'action **A** qui le rend à la vue **V** ;

11.10.1 Squelette de la classe [ConfigVue]

Le code de la vue [ConfigVue] de configuration de l'application est le suivant :

```
1.  package istia.st.domotique.android.vues;
2.
3.  ...
4.
5.  public class ConfigVue extends LocalVue {
6.    // l'interface visuelle
7.    private Button btnRafraichir;
8.    private Button btnAnnuler;
9.    private EditText edtUrlServiceRest;
10.   private TextView txtMsgErreurUrlServiceRest;
11.   private ListView listArduinos;
12.
13.   // actions asynchrones
14.   private IAction configAction;
15.
16.   // les données saisies
17.   private String urlServiceRest;
18.
19.   // l'info reçue
20.   private Object info;
21.
22.   @Override
```

```
23.   public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
    savedInstanceState) {
24.     // on crée la vue du fragment à partir de sa définition XML
25.     return inflater.inflate(R.layout.config, container, false);
26.   }
27.
28.   @Override
29.   public void onActivityCreated(Bundle savedInstanceState) {
30.     // parent
31.     super.onActivityCreated(savedInstanceState);
32.     // on définit la liste des arduinos
33.     listArduinos = (ListView) activity.findViewById(R.id.ListViewArduinos);
34.     listArduinos.setItemsCanFocus(false);
35.
36.     // bouton Rafraîchir
37.     btnRafraichir = (Button) activity.findViewById(R.id.btn_Rafraichir);
38.     btnRafraichir.setOnClickListener(new OnClickListener() {
39.       @Override
40.       public void onClick(View arg0) {
41.         doRafraichir();
42.       }
43.     });
44.
45.     // bouton Annuler
46.     btnAnnuler = (Button) activity.findViewById(R.id.btn_Annuler);
47.     btnAnnuler.setOnClickListener(new OnClickListener() {
48.       @Override
49.       public void onClick(View arg0) {
50.         // on annule l'action
51.         cancelAll();
52.         // on annule l'atrente
53.         cancelWaiting();
54.         // on met à jour la vue principale
55.         mainActivity.showTabs(new Boolean[] { true, false, false, false, false });
56.       }
57.     });
58.
59.     // visibilité boutons
60.     btnRafraichir.setVisibility(View.VISIBLE);
61.     btnAnnuler.setVisibility(View.INVISIBLE);
62.
63.     // saisie de l'IP et du port du serveur
64.     edtUrlServiceRest = (EditText) activity.findViewById(R.id.edt_UrlServiceRest);
65.     edtUrlServiceRest.setText(R.string.edt_UrlServiceRest);
66.
67.     // le msg d'erreur éventuel
68.     txtMsgErreurUrlServiceRest = (TextView)
    activity.findViewById(R.id.txt_MsgErreurIpPort);
69.   }
70.
71.   // le fragment est désormais visible
72.   public void onStart() {
73. ...
74.   }
75.
76.   // rafraîchissement de la liste des arduinos
77.   public void doRafraichir() {
78. ...
79.   }
80.
81.   // affichage d'une erreur
82.   private void showErreur(String msg, Exception exception) {
83. ...
```

```
84.   }
85.
86.   // la vue gère les événements provenant de l'action lancée
87.   public void notifyEvent(IWorker worker, int eventType, Object event) {
88. ...
89.   }
90.
91.   @Override
92.   public void notifyEndOfTasks() {
93. ...
94.   }
95.
96.   // début de l'attente
97.   private void beginWaiting() {
98. ...
99.   }
100.
101.    // fin de l'attente
102.    protected void cancelWaiting() {
103.  ...
104.    }
105.
106.  }
```

- ligne 5 : la classe [ConfigVue] étend la classe [LocalVue] qui étend elle-même la classe [Vue] du modèle AVAT. Nous reviendrons un peu plus loin sur la classe [LocalVue] ;
- lignes 23-26 : la méthode [onCreateView] définit la vue associée au fragment. C'est la vue XML [config.xml] que nous venons de décrire ;
- lignes 29-69 : la méthode [onActivityCreated] récupère les références des composants de la vue [config.xml] ;
- lignes 41, 77 : la méthode [doRafraîchir] traite le clic sur le bouton [Rafraichir] ;

11.10.2 La méthode [doRafraichir]

Elle traite le clic sur le bouton [Rafraichir]. Son code est le suivant :

```
1.  // rafraîchissement de la liste des arduinos
2.    public void doRafraichir() {
3.      // on vérifie les saisies
4.      if (!pageValid()) {
5.        return;
6.      }
7.      // c'est bon -on mémorise l'URI
8.      session.add("urlServiceRest", urlServiceRest);
9.      // on efface la liste des arduinos précédemment mémorisée
10.     // on les met dans la session
11.     session.add("arduinos", new Arduino[0]);
12.     // on les affiche
13.     showArduinos(listArduinos, false, null);
14.     // action asynchrone
15.     configAction = (IAction) factory.getObject(Factory.CONFIG_ACTION, this,
    "configAction");
16.     // on signale le début de l'action
17.     super.notifyEvent(configAction, WORK_STARTED, null);
18.     // début de l'attente
19.     beginWaiting();
20.     // exécution action
21.     configAction.doWork(urlServiceRest);
22.     // début du monitoring
23.     beginMonitoring();
24.   }
```

- lignes 4-6 : les données saisies sont vérifiées. Si elle ne sont pas valides, on retourne à l'UI (ligne 5) ;
- ligne 8 : on mémorise en **session** l'URL du service REST. La session est un champ de la classe parent [LocalVue] ;

- ligne 11 : on met un tableau d'Arduinos vide en session ;
- ligne 13 : la liste vide est affichée avec la méthode [showArduinos] de la classe parent [LocalVue]. Cela permet d'afficher une liste vide, le temps que l'action de remplissage de la liste fasse son effet ;
- ligne 15 : l'action chargée de demander la liste des Arduinos connectés est instanciée par la fabrique. Les paramètres sont les suivants :
    - n° de l'action à instancier,
    - une référence **this** sur le boss de l'action (la vue),
    - l'identifiant de l'action ;
- ligne 17 : on signale à la classe parent [Vue] que l'action est lancée ;
- ligne 19 : on met en place l'attribut visuel qui signale qu'une opération potentiellement longue est en cours ;
- ligne 21 : l'action [configAction] est lancée. On lui passe l'URL du service REST ;
- ligne 23 : on demande à la classe parent de nous avertir de la fin des tâches.

11.10.3 Gestion des résultats de l'action

L'action [configAction] va rendre la liste des Arduinos connectés ou bien une exception si cette liste n'a pu être obtenue. Les méthodes qui récupèrent ces résultats sont les méthodes [notifyEvent] et [notifyEndOfTasks].

La méthode [notifyEvent] est la suivante :

```
1.  // la vue gère les événements provenant de l'action lancée
2.    public void notifyEvent(IWorker worker, int eventType, Object event) {
3.      // on passe l'événement à la Vue parent
4.      super.notifyEvent(worker, eventType, event);
5.      // on gère le type WORK_INFO
6.      if (eventType == IBoss.WORK_INFO) {
7.        // on note l'info
8.        info = event;
9.        // quel type d'info ?
10.       if (event instanceof Exception) {
11.         showException((Exception) event);
12.       } else if (event instanceof Collection<?>) {
13.         // on récupère le tableau d'arduinos
14.         @SuppressWarnings("unchecked")
15.         Arduino[] arduinos = ((Collection<Arduino>) event).toArray(new Arduino[0]);
16.         // on les met dans la session
17.         session.add("arduinos", arduinos);
18.         // on les affiche
19.         showArduinos(listArduinos, false, null);
20.       }
21.     }
22.   }
```

- ligne 2 : la méthode [notifyEvent] gère les événements WORK_INFO et WORK_TERMINATED que lui envoie l'action. Elle reçoit trois paramètres :
    - *worker* : l'action qui envoie la notification,
    - *eventType* : le type de notification. Il y en a trois WORK_STARTED, WORK_INFO et WORK_TERMINATED,
    - *info* : une information facultative ;
- ligne 4 : quelque soit l'événement, celui-ci est passé d'abord à la classe parent. L'événement [WORK_TERMINATED] va être ainsi traité ;
- ligne 6 : l'événement [WORK_INFO] est traité. Il transporte avec lui une information que la vue va afficher ;
- ligne 8 : l'information transmise par l'action est mémorisée ;
- lignes 10-11 : si l'information mémorisée est une exception, alors celle-ci est affichée par une méthode [showException] de la classe parent [LocalVue] ;
- ligne 12 : si l'information mémorisée est une collection alors il s'agit de la liste des Arduinos ;
- ligne 15 : la collection est transformée en tableau ;
- ligne 17 : ce tableau est mis en session. Il sera utilisé par toutes les autres vues ;
- ligne 19 : la liste des Arduinos est affichée et remplace donc la liste vide qui avait été affichée initialement ;

La méthode [notifyEndOfTasks] est la suivante :

```
1.    @Override
```

```
2.    public void notifyEndOfTasks() {
3.      // on arrête d'attendre
4.      cancelWaiting();
5.      // selon l'info, les onglets à afficher ne sont pas les mêmes
6.      if (info instanceof Exception) {
7.        mainActivity.showTabs(new Boolean[] { true, false, false, false, false });
8.      } else {
9.        mainActivity.showTabs(new Boolean[] { true, true, true, true, true });
10.     }
11. }
```

- ligne 2 : la méthode [notifyEndOfTasks] est appelée par la classe parent [Vue] lorsque toutes les tâches sont terminées. Ici, il s'agit de la fin de l'unique action qui a été lancée ;
- ligne 4 : on cache les éléments visuels qui indiquaient qu'une opération était en cours ;
- ligne 6 : si on a eu une exception, alors on n'affiche que l'onglet de configuration (ligne 7), sinon on affiche tous les onglets (ligne 9). En effet, à partir du moment où il existe une liste d'Arduinos, on peut travailler avec. Aussi les quatre onglets qui permettent d'envoyer des commandes aux Arduinos sont-ils affichés. Le champ [mainActivity] est un champ de la classe parent [LocalVue]. la méthode [showTabs] a été présentée au paragraphe ??, page 91.

11.10.4 La méthode [pageValid]

La méthode [pageValid] vérifie la valité des données saisies, ici une URL. Son code est le suivant :

```
1.  // vérification des saisies
2.    private boolean pageValid() {
3.      // on récupère l'Ip et le port du serveur
4.      urlServiceRest = edtUrlServiceRest.getText().toString().trim();
5.      // on vérifie sa validité
6.      try {
7.        new URI(String.format("http://%s", urlServiceRest));
8.      } catch (Exception ex) {
9.        // affichage erreur
10.       showErreur(ex.getMessage(), null);
11.       // retour à l'UI
12.       return false;
13.     }
14.     // c'est bon
15.     txtMsgErreurUrlServiceRest.setText("");
16.     return true;
17.   }
18.
19.   // affichage d'une erreur
20.   private void showErreur(String msg, Exception exception) {
21.     // on affiche l'erreur
22.     if (exception != null) {
23.       // affichage exception
24.       showException(exception);
25.     } else {
26.       // affichage msg d'erreur
27.       txtMsgErreurUrlServiceRest.setText(msg);
28.     }
29.     // pas d'arduinos
30.     session.add("arduinos", new Arduino[0]);
31.     showArduinos(listArduinos, false, null);
32.   }
```

- ligne 4 : l'URL saisie est récupérée ;
- ligne 7 : on essaie de créer une URI (Unique Resource Identifier) à partir de cette URL. Si l'URL n'a pas un format correct, une exception sera lancée ;
- ligne 10 : s'il y a exception, un message d'erreur est affiché ;
- ligne 20 : la méthode [showErreur] est utilisée pour afficher une erreur. Son premier paramètre est un message d'erreur à afficher et son second une exception à afficher ;
- ligne 22 : si on a une exception, on utilise la méthode [showException] de la classe parent [LocalVue] pour l'afficher ;

- ligne 27 : sinon, on affiche le message d'erreur ;
- lignes 30-31 : en cas d'erreur de saisie de l'URL, une liste d'Arduinos vide est mise en session et affichée ;
- ligne 15 : s'il n'y a pas d'erreur, on efface un éventuel message d'erreur qui aurait pu être affiché lors d'une saisie précédente.

11.10.5 Les autres méthodes de la vue

Les autres méthodes de la vue ont peu de logique.

```
1.    // début de l'attente
2.    private void beginWaiting() {
3.      // on met le sablier
4.      showHourGlass();
5.      // boutons
6.      btnRafraichir.setVisibility(View.INVISIBLE);
7.      btnAnnuler.setVisibility(View.VISIBLE);
8.    }
9.
10.   // fin de l'attente
11.   protected void cancelWaiting() {
12.     // on cache le sablier
13.     hideHourGlass();
14.     // visibilité boutons
15.     btnRafraichir.setVisibility(View.VISIBLE);
16.     btnAnnuler.setVisibility(View.INVISIBLE);
17. }
```

Les méthodes [showHourGlass] de la ligne 4 et [hideHourGlass] de la ligne 13 sont des méthodes de la classe parent [LocalVue].

- ligne 2 : la méthode [beginWaiting] est utilisée pour afficher une attente ;
- ligne 4 : on affiche l'image animée qui indique qu'une opération est en cours ;
- ligne 6 : le bouton [Rafraichir] va être caché ;
- ligne 7 : le bouton [Annuler] est affiché ;
- ligne 11 : la méthode [cancelWaiting] est utilisée pour arrêter l'attente ;
- ligne 13 : on cache l'image animée qui indique qu'une opération est en cours ;
- ligne 15 : le bouton [Rafraichir] est affiché ;
- ligne 16 : le bouton [Annuler] est caché ;

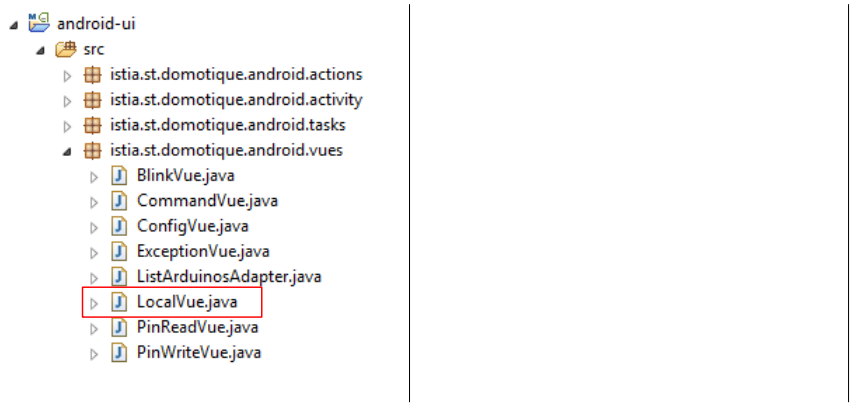
11.10.6 Annulation des tâches

Avec le bouton [Annuler], l'utilisateur peut annuler l'action lancée. La méthode qui traite le clic sur le bouton [Annuler] est la suivante :

```
1.  // bouton Annuler
2.      btnAnnuler = (Button) activity.findViewById(R.id.btn_Annuler);
3.      btnAnnuler.setOnClickListener(new OnClickListener() {
4.        @Override
5.        public void onClick(View arg0) {
6.          // on annule l'action
7.          cancelAll();
8.          // on annule l'attente
9.          cancelWaiting();
10.         // on met à jour la vue principale
11.         mainActivity.showTabs(new Boolean[] { true, false, false, false, false });
12.       }
13.     });
```

- ligne 5 : la méthode qui traite le clic sur le bouton [Annuler] ;
- ligne 7 : on demande à la classe parent d'annuler toutes les actions. Ici, il n'y en qu'une ;
- ligne 9 : on arrête l'attente ;
- ligne 11 : seul l'onglet [CONFIG] est affiché. Les autres disparaissent.

11.11 La classe parent [LocalVue]



La classe [LocalVue] est la classe parent de toutes les vues. Elle factorise les comportements communs aux différentes vues. Son code est le suivant :

```
1.  package istia.st.domotique.android.vues;
2.
3.  ...
4.
5.  // encapsule des méthodes et données communes aux classes filles
6.  public abstract  class LocalVue extends Vue {
7.    // la session
8.    protected ISession session;
9.    // la main activity
10.   protected MainActivity mainActivity;
11.
12.   @Override
13.   public void onActivityCreated(Bundle savedInstanceState) {
14.     // parent
15.     super.onActivityCreated(savedInstanceState);
16.     // la session
17.     session = (ISession) factory.getObject(Factory.SESSION, (Object[]) null);
18.     // la main activity
19.     mainActivity= (MainActivity) activity;
20.   }
21.
22.   // affichage de la liste des arduinos
23.   protected void showArduinos(ListView listArduinos, Boolean selectable, Arduino[]
    selectedArduinos) {
24. ...
25.   }
26.
27.   // affichage exception
28.   protected void showException(Exception ex) {
29. ...
30.   }
31.
32. ...
33.   // affichage du sablier
34.   protected void showHourGlass() {
35.     activity.setProgressBarIndeterminateVisibility(true);
36.   }
37.
38.   // effacement du sablier
39.   protected void hideHourGlass() {
40.     activity.setProgressBarIndeterminateVisibility(false);
41.   }
```

```
42.
43.   @Override
44.   abstract public void notifyEndOfTasks();
45. }
```

- ligne 6 : la classe [LocalVue] étend la classe [Vue] du modèle AVAT. C'est obligatoire ;
- ligne 13 : la méthode exécutée lorsque la vue est créée ;
- ligne 15 : l'événement est passé à la classe parent ;
- ligne 17 : on demande une référence de la session à la fabrique ;
- ligne 19 : l'activité [MainActivity] est injectée dans toutes les vues dans un champ *activity* de type *Activity*. Afin d'éviter des transtypages dans les différentes vues, cette activité est mémorisée avec le bon type dans le champ *mainActivity* de la ligne 10 ;
- les lignes 35 et 40 agissent sur le composant qui a été défini ainsi dans [MainActivity] :

```
1.      // le sablier
2.      requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
3.  setProgressBarIndeterminateVisibility(false);
```

La méthode de la ligne 23 affiche les Arduinos de la façon suivante :

```
1.    // affichage de la liste des arduinos
2.    protected void showArduinos(ListView listArduinos, Boolean selectable, Arduino[]
    selectedArduinos) {
3.
4.      // on définit la source de données de la liste
5.      Arduino[] arduinos = (Arduino[]) session.get("arduinos");
6.      ArrayAdapter<Arduino> adapter = new ListArduinosAdapter(getActivity(),
    R.layout.listarduinos_item, arduinos,
7.          selectable);
8.      listArduinos.setAdapter(adapter);
9.      // TODO sélectionner certains arduinos
10.     return;
11. }
```

- ligne 2 : la méthode admet les paramètres suivants :
    - le composant [ListView] chargé d'afficher les Arduinos,
    - un booléen *selectable* qui indique si l'affichage d'un Arduino dans la liste doit se faire avec une case à cocher ou non. Elle se fera sans case à cocher dans la vue [ConfigVue] avec une case à cocher dans les autres vues,
    - le tableau des Arduinos déjà sélectionnés. Il faut cocher leur case. Ce paramètre n'a pas été utilisé dans l'exemple. Ce travail reste à faire (ligne 9) ;
- ligne 5 : la liste des Arduinos à afficher est récupérée dans la session ;
- ligne 6 : cette liste est utilisée pour alimenter le [ListView] (ligne 8) au moyen d'un *ArrayAdapter* (ligne 6).

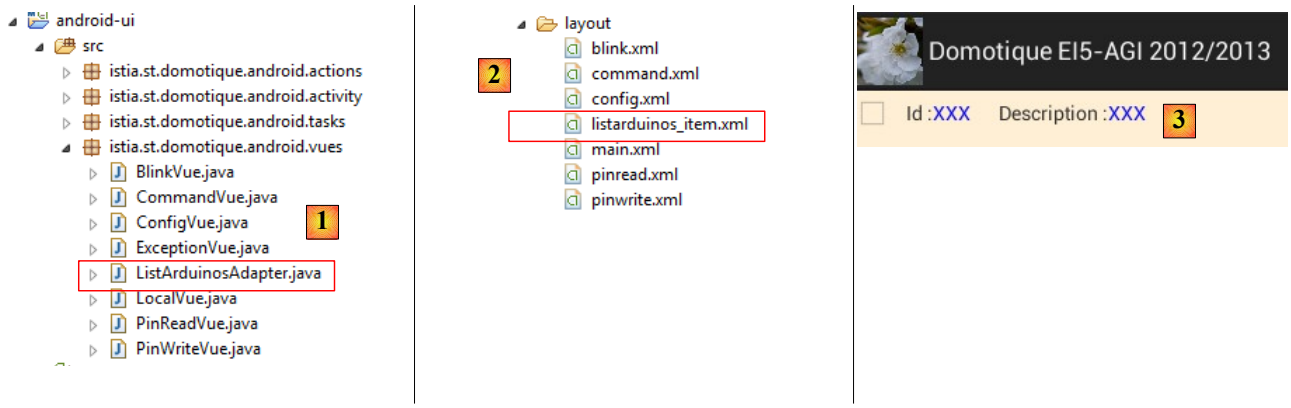
11.12 L'adaptateur [ListArduinosAdapter]

Revenons sur le code précédent :

```
1.      // on définit la source de données de la liste
2.      Arduino[] arduinos = (Arduino[]) session.get("arduinos");
3.      ArrayAdapter<Arduino> adapter = new ListArduinosAdapter(getActivity(),
    R.layout.listarduinos_item, arduinos,
4.          selectable);
5.  listArduinos.setAdapter(adapter);
```

- ligne 5 : le [ListView] listArduinos est alimenté par un adaptateur. C'est lui qui fournit au [ListView] les données à afficher ;
- ligne 3 : la définition de l'adaptateur. Il reçoit quatre paramètres :
    1. l'activité principale,
    2. le fichier XML qui définit une ligne du [ListView],
    3. le tableau des Arduinos qui vont être affichés dans le [ListView],
    4. un booléen indiquant si les lignes du [ListView] peuvent être ou non sélectionnées par une case à cocher ;

L'adaptateur [ListArduinosAdapter] est le suivant :

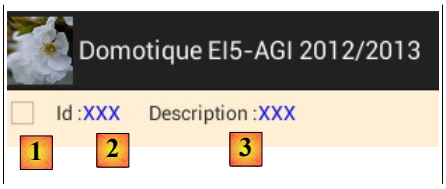


- en [1], la classe de l'adaptateur du [ListView] ;
- en [2], le fichier [listarduinos_item.xml] est la définition XML des lignes affichées par le [ListView] ;
- en [3], la forme d'une ligne du[ListView].

Le code du fichier [listarduinos_item.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="utf-8"?>
2.  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
3.      android:id="@+id/RelativeLayout1"
4.      android:layout_width="match_parent"
5.      android:layout_height="match_parent"
6.      android:background="@color/wheat"
7.      android:orientation="vertical" >
8.
9.      <CheckBox
10.         android:id="@+id/checkBoxArduino"
11.         android:layout_width="wrap_content"
12.         android:layout_height="wrap_content"
13.         android:layout_alignParentLeft="true"
14.         android:layout_alignParentTop="true"
15.         android:layout_toRightOf="@+id/txt_arduino_description" />
16.
17.     <TextView
18.         android:id="@+id/TextView1"
19.         android:layout_width="wrap_content"
20.         android:layout_height="wrap_content"
21.         android:layout_alignBaseline="@+id/checkBoxArduino"
22.         android:layout_marginLeft="40dp"
23.         android:text="@string/txt_arduino_id" />
24.
25.     <TextView
26.         android:id="@+id/txt_arduino_id"
27.         android:layout_width="wrap_content"
28.         android:layout_height="wrap_content"
29.         android:layout_alignBaseline="@+id/checkBoxArduino"
30.         android:layout_alignParentTop="true"
31.         android:layout_toRightOf="@+id/TextView1"
32.         android:text="@string/dummy"
33.         android:textColor="@color/blue" />
34.
35.     <TextView
36.         android:id="@+id/TextView2"
37.         android:layout_width="wrap_content"
38.         android:layout_height="wrap_content"
39.         android:layout_alignBaseline="@+id/checkBoxArduino"
```

```
40.         android:layout_alignParentTop="true"
41.         android:layout_marginLeft="20dp"
42.         android:layout_toRightOf="@+id/txt_arduino_id"
43.         android:text="@string/txt_arduino_description" />
44.
45.     <TextView
46.         android:id="@+id/txt_arduino_description"
47.         android:layout_width="wrap_content"
48.         android:layout_height="wrap_content"
49.         android:layout_alignBaseline="@+id/checkBoxArduino"
50.         android:layout_alignTop="@+id/TextView2"
51.         android:layout_toRightOf="@+id/TextView2"
52.         android:text="@string/dummy"
53.         android:textColor="@color/blue" />
54.
55. </RelativeLayout>
```



- lignes 9-15 : la case à cocher ;
- lignes 17-33 : l'identifiant de l'Arduino ;
- lignes 35-53 : la description de l'Arduino.

La classe [ListArduinosAdapter] exploite cette définition XML :

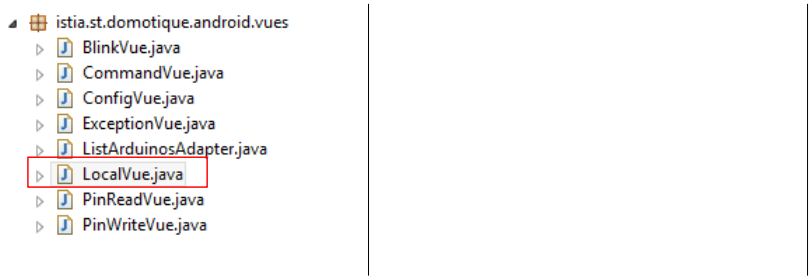
```
1.  package istia.st.domotique.android.vues;
2.
3.  ...
4.
5.  public class ListArduinosAdapter extends ArrayAdapter<Arduino> {
6.
7.    // le tableau des arduinos
8.    private Arduino[] arduinos;
9.    // le contexte d'exécution
10.   private Context context;
11.   // l'id du layout d'affichage d'une ligne de la liste des arduinos
12.   private int layoutResourceId;
13.   // la ligne comporte ou non un checkbox
14.   private Boolean selectable;
15.
16.   // constructeur
17.   public ListArduinosAdapter(Context context, int layoutResourceId, Arduino[] arduinos,
    Boolean selectable) {
18.     super(context, layoutResourceId, arduinos);
19.     // on mémorise les infos
20.     this.arduinos = arduinos;
21.     this.context = context;
22.     this.layoutResourceId = layoutResourceId;
23.     this.selectable = selectable;
24.   }
25.
26.   @Override
27.   public View getView(int position, View convertView, ViewGroup parent) {
28.     // la ligne à peupler
29.     View row = convertView;
30.     // déjà vue ?
31.     if (row == null) {
```

```
32.       // on crée la ligne
33.       row = ((Activity) context).getLayoutInflater().inflate(layoutResourceId, parent,
    false);
34.       TextView txtArduinoId = (TextView) row.findViewById(R.id.txt_arduino_id);
35.       TextView txtArduinoDesc = (TextView) row.findViewById(R.id.txt_arduino_description);
36.       // on remplit la ligne
37.       txtArduinoId.setText(arduinos[position].getId());
38.       txtArduinoDesc.setText(arduinos[position].getDescription());
39.       // checkbox ?
40.       CheckBox ck = (CheckBox) row.findViewById(R.id.checkBoxArduino);
41.       ck.setVisibility(selectable ? View.VISIBLE : View.INVISIBLE);
42.     }
43.     // on rend la ligne
44.     return row;
45.   }
46. }
```

- ligne 27 : pour générer les lignes du [ListView], la méthode [getView] est appelée de façon répétée. Elle rend la ligne à afficher. Le 1er paramètre est le n° de la ligne dans le [ListView]. Le second paramètre est la ligne actuellement à cette position. Ce paramètre est *null* lors de la première génération de la ligne ;
- ligne 29 : on récupère l' "ancienne " ligne ;
- ligne 31 : si cette ligne existe déjà (*row!=null*) elle est renvoyée telle quelle (ligne 44). Si elle n'existe pas (*row==null*), alors on la crée (lignes 32-41) ;
- ligne 33 : la ligne est créée à partir de sa définition XML [listarduinos_item.xml] ;
- lignes 34-35 : on récupère les références des composants *TextView* qui doivent afficher respectivement, l'identifiant et la description de l'Arduino à afficher ;
- lignes 37-38 : les textes de ces deux composants sont intialisés ;
- ligne 40 : on récupère la référence de la case à cocher ;
- ligne 41 : selon que le constructeur a été appelé avec la propriété *selectable* à vrai ou faux, on montre ou cache la case à cocher.

11.13 La méthode [getSelectedArduinos] de la vue [LocalVue]

Dans les vues [PINWRITE], [PINREAD], [BLINK] et [COMMANDS], l'utilisateur sélectionne dans une liste, les Arduinos concernés par la commande qu'il va exécuter. On a donc besoin de savoir quels sont les arduinos qui ont été sélectionnés par l'utilisateur. Cela est obtenu par la méthode [getSelectedArduinos] de la vue [LocalVue] :

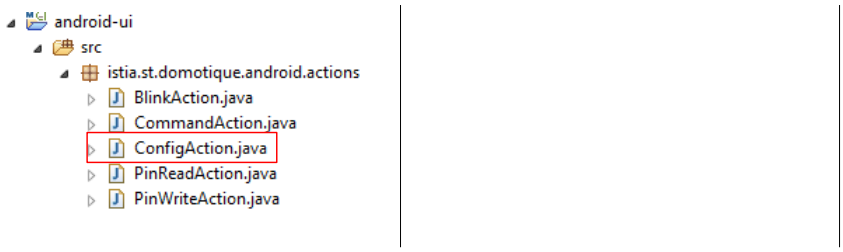



```
1.  // arduinos sélectionnés dans la liste
2.    protected Arduino[] getSelectedArduinos(ListView listArduinos) {
3.      // la liste des arduinos
4.      Arduino[] arduinos = (Arduino[]) session.get("arduinos");
5.      List<Arduino> selectedArduinos = new ArrayList<Arduino>();
6.      // on parcourt la liste des arduinos
7.      for (int i = 0; i < listArduinos.getChildCount(); i++) {
8.        CheckBox ck = (CheckBox)
    listArduinos.getChildAt(i).findViewById(R.id.checkBoxArduino);
9.        if (ck.isChecked()) {
10.         selectedArduinos.add(arduinos[i]);
11.       }
12.     }
13.     // on rend le résultat
```

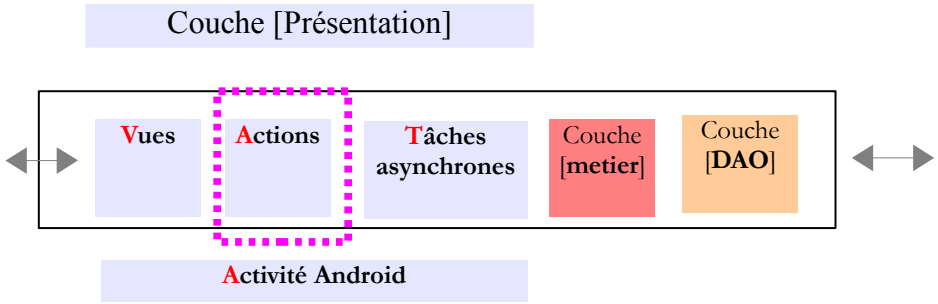
```
14.     return selectedArduinos.toArray(new Arduino[0]);
15.   }
```

- ligne 4 : on récupère en session le tableau des Arduinos connectés ;
- ligne 5 : la liste des Arduinos sélectionnés par l'utilisateur ;
- ligne 7 : on parcourt les éléments du [ListView] ;
- ligne 8 : on récupère une référence sur la case à cocher de la ligne n° i ;
- lignes 9-10 : si la case est cochée, alors on ajoute l'Arduino n° i à la liste des Arduinos sélectionnés ;
- ligne 14 : on rend un tableau plutôt qu'une liste.

11.14 L'action [ConfigAction]



La classe [ConfigAction] est chargée d'obtenir la liste des Arduinos connectés auprès du serveur REST.



Cette action va déboucher sur l'ouverture d'une connexion réseau vers le serveur REST. Pour cela, on a besoin d'une tâche asynchrone.

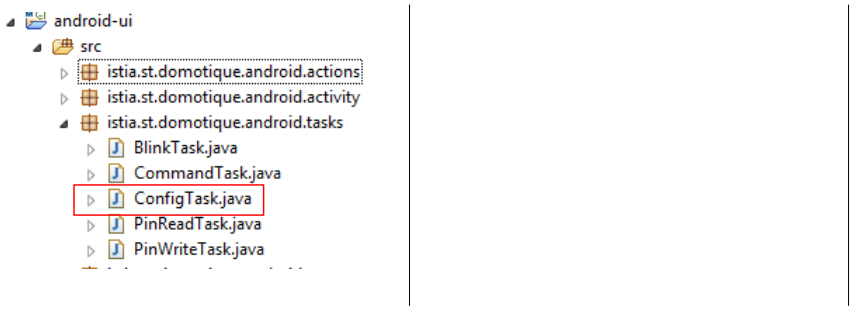
L'action [ConfigAction] est la suivante :

```
1.  package istia.st.domotique.android.actions;
2.
3.  ...
4.
5.  public class ConfigAction extends Action {
6.
7.    @Override
8.    // urlServiceRest
9.    public void doWork(Object... params) {
10.
11.     // on crée la tâche asynchrone
12.     Task configTask = (Task) factory.getObject(Factory.CONFIG_TASK,this, "configTask");
13.     // on lance la tâche (urlServiceRest)
14.     configTask.doWork(params);
15.     // on commence le monitoring de la tâche
16.     beginMonitoring();
17.   }
```

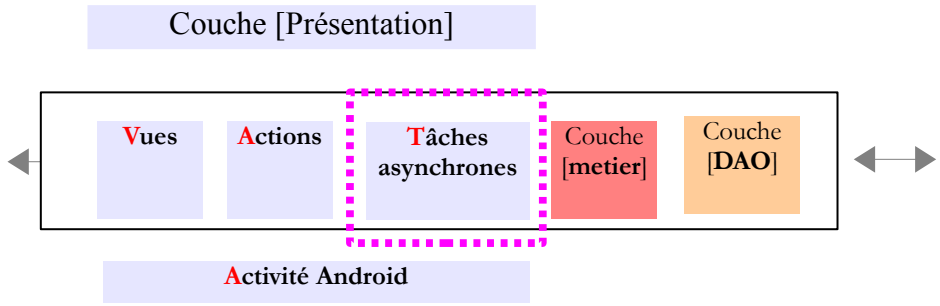
```
18.
19.   @Override
20.   public void notifyEndOfTasks() {
21.     // fin de l'action
22.     boss.notifyEvent(this, IBoss.WORK_TERMINATED, null);
23.   }
24.
25.   // on gère certains événements
26.   public void notifyEvent(IWorker worker, int eventType, Object event) {
27.     // on passe l'événement au parent
28.     super.notifyEvent(worker, eventType, event);
29.     // on gère le type WORK_INFO
30.     if (eventType == IBoss.WORK_INFO) {
31.       // on passe l'info brute au boss
32.       boss.notifyEvent(this, eventType, event);
33.     }
34.   }
35.
36. }
```

- ligne 5 : la classe [ConfigAction] étend la classe [Action] du modèle AVAT. C'est obligatoire ;
- ligne 9 : la méthode exécutée par l'action lorsque la vue la lance. Ici, elle reçoit un paramètre : l'URL du service REST ;
- ligne 12 : on demande à la fabrique la tâche asynchrone de n° [Factory.ConfigTask]. On passe à la fabrique deux paramètres :
    - *this* : une référence sur le boss de la tâche, l'action elle-même ,
    - *configTask* : l'identifiant de la tâche créée ;
- ligne 14 : on lance la tâche avec les paramètres qu'a reçus l'action ;
- ligne 16 : l'action attend la fin des tâches asynchrones qu'elle a lancées, ici une seule.
- ligne 26 : en tant que boss, l'action va voir passer dans la méthode [notifyEvent] les notifications des tâches qu'elle a lancées. Cette méthode a trois paramètres :
    - *worker* : la tâche qui envoie la notification,
    - *eventType* : le type de notification,
    - *info* : une éventuelle information ;
- ligne 28 : la notification est passée à la classe parent pour que celle-ci puisse envoyer à son tour la notification [endOfTasks] ;
- ligne 30 : si la notification est de type WORK_INFO, la notification est passée au boss de l'action et celle-ci se substitue au *worker* qui a envoyé cette notification (paramètre *this* de la ligne 32) ;
- ligne 20 : la classe parent active cette méthode de la classe fille lorsque toutes les tâches lancées par celle-ci sont terminées ;
- ligne 22 : l'action envoie alors à son boss (la vue) la notification indiquant qu'elle a terminé son travail.

11.15 La tâche [ConfigTask]



La tâche [ConfigTask] est chargée d'obtenir la liste des Arduinos connectés auprès du serveur REST.



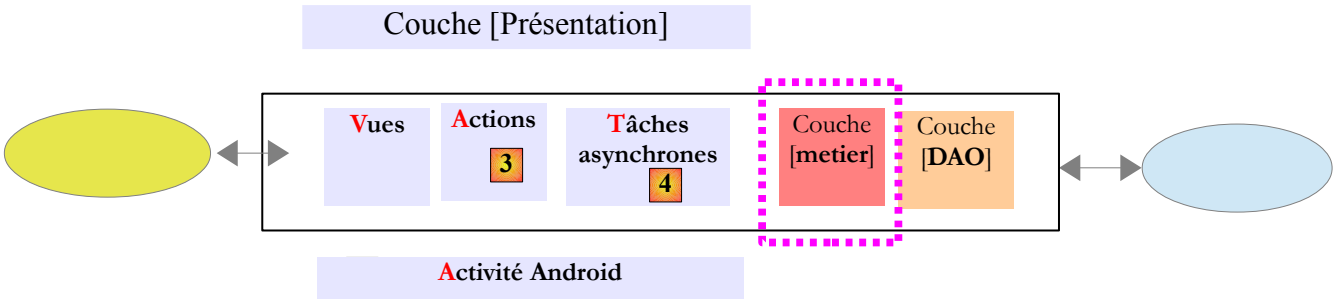
Son code est le suivant :

```
1.  package istia.st.domotique.android.tasks;
2.
3.  ...
4.
5.  public class ConfigTask extends Task {
6.
7.     // résultat de la tâche
8.     private Object info;
9.
10.    // constructeurs
11.    public ConfigTask() {
12.
13.    }
14.
15.    @Override
16.    protected void onPreExecute() {
17.       // on est dans le thread de l'UI
18.       // la tâche démarre
19.       boss.notifyEvent(this, IBoss.WORK_STARTED, null);
20.    }
21.
22.    @Override
23.    // on exécute la tâche (metier, urlServiceRest)
24.    protected Void doInBackground(Object... params) {
25.
26.       // paramètres
27.       String urlServiceRest = (String) params[0];
28.
29.       // on exécute la méthode métier
30.     IMetier metier = (IMetier) factory.getObject(Factory.METIER, (Object[]) null);
31.       try {
32.          info = metier.getArduinos(urlServiceRest);
33.       } catch (Exception ex) {
34.          // on mémorise l'exception
35.          info = ex;
36.       }
37.       // pour le compilateur
38.       return null;
39.    }
40.
41.    @Override
42.    protected void onPostExecute(Void result) {
43.       // on est dans le thread de l'UI
44.       // on passe l'info au boss
45.       boss.notifyEvent(this, IBoss.WORK_INFO, info);
46.       // la tâche est terminée
47.       boss.notifyEvent(this, IBoss.WORK_TERMINATED, null);
48.    }
49.
50. }
```

- ligne 5 : la classe [ConfigTask] étend la classe [Task] du modèle AVAT. C'est obligatoire ;
- ligne 8 : l'information produite par la tâche ;
- ligne 16 : la méthode [onPreExecute] est exécutée avant la méthode asynchrone [doInBackGround]. Elle est exécutée dans le thread de l'interface comme le sont également l'activité, l'action et la vue ;
- ligne 19 : la tâche envoie à son boss la notification WORK_STARTED pour indiquer qu'elle a démarré ;
- ligne 24 : la méthode [doInBackGround] est exécutée dans un thread à part de celui de l'UI. Elle reçoit les paramètres que lui a donnés l'action qui l'a lancée ;
- ligne 27 : on récupère l'unique paramètre qu'a transmis l'action ;
- ligne 30 : on demande à la fabrique une référence sur la couche [métier] ;
- ligne 32 : la méthode [getArduinos] de la couche [métier] est exécutée. On mémorise le résultat de cette méthode ;
- ligne 35 : s'il se produit une exception, elle est mémorisée ;
- ligne 42 : la méthode [onPostExecute] est exécutée après la méthode asynchrone [doInBackGround]. Elle est exécutée dans le thread de l'interface comme le sont également l'activité, l'action et la vue ;
- ligne 45 : l'information produite par la méthode [doInBackGround] est transmise au boss (l'action) ;
- ligne 47 : la tâche signale a son boss (l'action) qu'elle a terminé son travail.

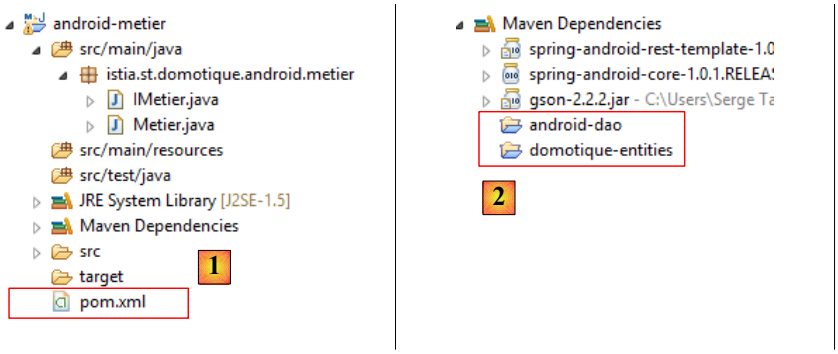
11.16 La couche [métier]

La couche [métier] est chargée d'obtenir auprès du serveur REST les informations demandées par les vues.



11.16.1 Le projet Eclipse

Le projet Eclipse de la couche [métier] est le suivant :



- en [1], le projet Eclipse est un projet Maven ;
- en [2], les dépendances Maven. Il y en a une : celle sur le projet [android-dao] de la couche [DAO] du client Android. Les autres en découlent.

11.16.2 Les dépendances Maven

Le fichier [pom.xml] du projet est le suivant :

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
2.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
   4.0.0.xsd">
3.     <modelVersion>4.0.0</modelVersion>
4.
5.     <groupId>istia.st.domotique.android</groupId>
6.     <artifactId>android-metier</artifactId>
7.     <version>1.0-SNAPSHOT</version>
8.     <packaging>jar</packaging>
9.
10.    <name>android-metier</name>
11.    <url>http://maven.apache.org</url>
12.
13.    <properties>
14.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.    </properties>
16.
17.    <dependencies>
18.       <dependency>
19.          <groupId>istia.st.domotique.android</groupId>
20.          <artifactId>android-dao</artifactId>
21.          <version>1.0-SNAPSHOT</version>
22.       </dependency>
23.
24.    </dependencies>
25. </project>
```

- lignes 5-7 : l'identité Maven de la couche [métier] ;
- lignes 19-21 : la couche [métier] a une dépendance sur la couche [DAO] du client Android.

11.16.3 L'interface de la couche [métier]

L'interface [IMetier] de la couche [métier] est la suivante :

```
1.  package istia.st.domotique.android.metier;
2.
3.  ...
4.
5.
6.  public interface IMetier {
7.    // liste des arduinos
8.    public Collection<Arduino> getArduinos(String urlServiceRest);
9.    // lecture d'une pin
10.   public Reponse pinRead(String urlServiceRest,String idCommande, String idArduino, int
    pin, String mode);
11.   // écriture d'une pin
12.   public Reponse pinWrite(String urlServiceRest,String idCommande, String idArduino, int
    pin, String mode,int val);
13.   // faire clignoter une led
14.   public Reponse faireClignoterLed(String urlServiceRest,String idCommande, String
    idArduino, int pin, int millis, int nbIter);
15.   // envoyer une suite de commandes Json à un Arduino
16.   public List<String> sendCommandesJson(String urlServiceRest,String idArduino,
    List<String> commandes);
17.   // envoyer une suite de commandes à un Arduino
18.   public List<Reponse> sendCommandes(String urlServiceRest,String idArduino, List<Commande>
    commandes);
19.   // la couche [dao]
20.   public void setDao(IDao dao);
21. }
```

Cette interface est celle de l'interface [IMetier] du serveur décrite au paragraphe ??, page 53. Les méthodes correspondent une à une avec une unique différence : l'interface [IMetier] du client Android a un paramètre supplémentaire pour chaque méthode : l'URL du service REST.

11.16.4 La méthode [getArduinos]

Nous avons vu que la tâche [ConfigTask] a appelé la couche [métier] de la façon suivante :

```
        info = metier.getArduinos(urlServiceRest);
```

La méthode [getArduinos] est la suivante :

```
1.  public Collection<Arduino> getArduinos(String urlServiceRest) {
2.      // adresse du service REST
3.      String urlService = String.format("http://%s/arduinos/", urlServiceRest);
4.      // on demande la liste des Arduinos au service REST
5.      String réponse = dao.executeRestService("get", urlService, null, new HashMap<String,
    String>());
6.      // on exploite la réponse
7.      try {
8.        // résultat
9.        return gson.fromJson(réponse, new TypeToken<Collection<Arduino>>() {
10.       }.getType());
11.     } catch (Exception ex) {
12.       throw new DomotiqueException(String.format("Réponse incorrecte du serveur: %s",
    réponse), ex, 167);
13.     }
14.   }
```

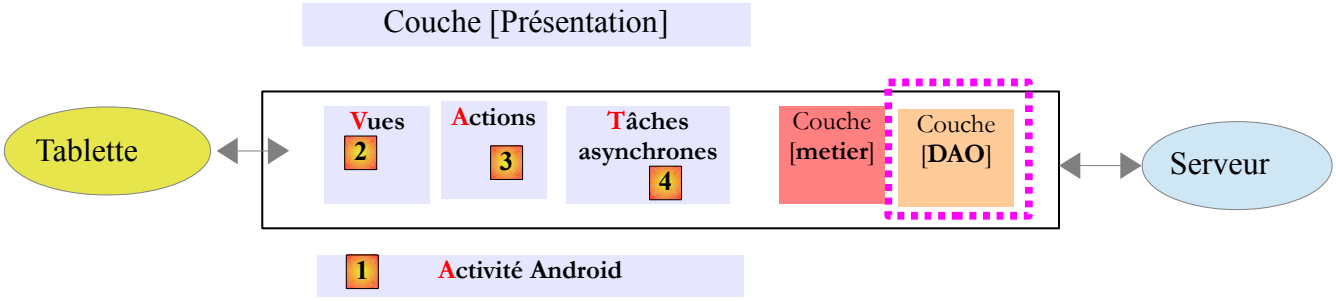
- la méthode [getArduinos] du client doit appeler la méthode [getArduinos] du serveur REST. L'URL de cette méthode a été définie de la façon suivante (cf paragraphe 32, page 63) :

```
1.  // liste des arduinos
2.     @RequestMapping(value = "/arduinos/", method = RequestMethod.GET)
3.     public ModelAndView getArduinos() {
4.  ...
5.     }
```

  On a donc une URL sans paramètres.
- l'URL complète de la méthode est construite ;
- ligne 5 : on demande à la couche [DAO] de se connecter à cette URL et d'en récupérer la réponse JSON. Les paramètres sont les suivants :
  - " *get* " : la méthode HTTP à utiliser pour se connecter à l'URL,
  - *urlService* : l'URL à laquelle se connecter ;
  - *null* : un dictionnaire pour les paramètres d'une commande HTTP POST. Ici, il s'agit d'un GET. Donc on passe un pointeur *null*,
- ligne 5 : la méthode [dao.executeRestService] va rendre une collection d'Arduinos sous la forme d'une chaîne JSON ou bien va lancer une exception de type [DomotiqueException]. Cette dernière n'est pas gérée. Elle va remonter jusqu'à la vue où elle sera affichée ;
- ligne 9 : la chaîne JSON récupérée est transformée en type *Collection<Arduino>* et rendue à la tâche [ConfigTask] ;
- ligne 12 : cas où la chaîne JSON récupérée est invalide.

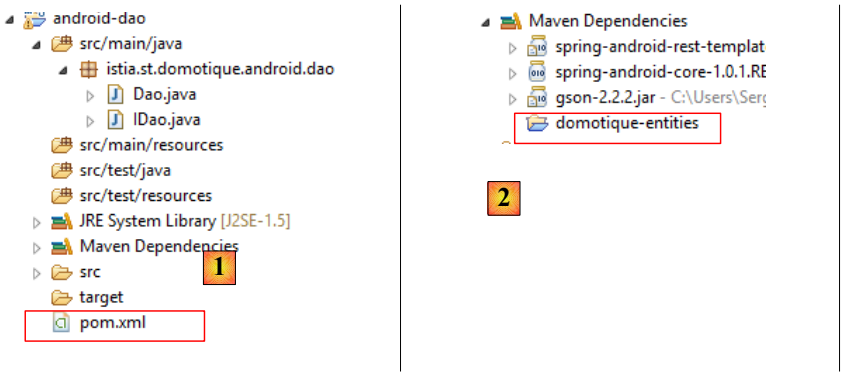
11.17 La couche [DAO]

La couche [DAO] exécute les demandes de la couche [métier]. C'est elle qui véritablement se connecte au serveur REST :



11.17.1 Le projet Eclipse

Le projet Eclipse de la couche [DAO] est le suivant :



- en [1], le projet Eclipse est un projet Maven ;
- en [2], les dépendances Maven. Il y en a deux :
    - celle sur le projet [domotique-entities] déjà utilisé pour le serveur. Ce projet rassemble les objets que s'échangent le client Android et le serveur REST ,
    - celle sur la bibliothèque [spring-android-rest-template] qui est un client REST fourni par le framework Spring ;

11.17.2 Les dépendances Maven

Le fichier [pom.xml] est le suivant :

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
    4.0.0.xsd">
3.     <modelVersion>4.0.0</modelVersion>
4.     <groupId>istia.st.domotique.android</groupId>
5.     <artifactId>android-dao</artifactId>
6.     <version>1.0-SNAPSHOT</version>
7.
8.     <properties>
9.        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
10.       <spring-android-version>1.0.1.RELEASE</spring-android-version>
11.    </properties>
12.
13.    <repositories>
14.       <repository>
15.          <id>springsource-repo</id>
16.          <name>SpringSpring Repository</name>
17.          <url>http://repo.springsource.org/release</url>
```

```
18.       </repository>
19.    </repositories>
20.
21.    <dependencies>
22.       <dependency>
23.          <groupId>org.springframework.android</groupId>
24.          <artifactId>spring-android-rest-template</artifactId>
25.          <version>${spring-android-version}</version>
26.       </dependency>
27.       <dependency>
28.          <groupId>istia.st.domotique.common</groupId>
29.          <artifactId>domotique-entities</artifactId>
30.          <version>1.0-SNAPSHOT</version>
31.       </dependency>
32.    </dependencies>
33. </project>
```

- lignes 4-6 : l'identité Maven de la couche [DAO] du client Android ;
- lignes 22-26 : la dépendance sur la bibliothèque [spring-android-rest-template] ;
- lignes 27-31 : la dépendance sur le projet [domotique-entities] ;
- lignes 14-18 : le site où trouver la dépendance [spring-android-rest-template].

11.17.3 Le code de la couche [DAO]

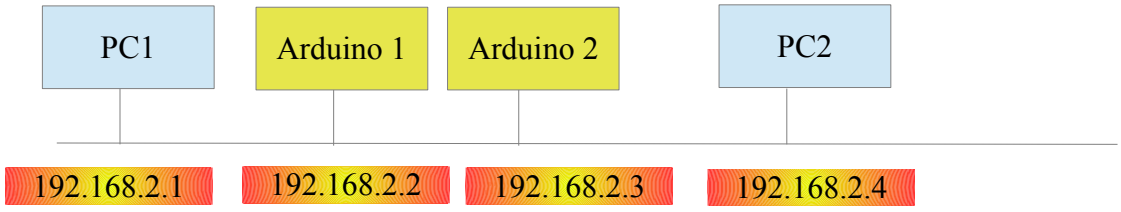
Le code de la couche [DAO] a été décrit dans [ref2] au paragraphe 4.4.2, page 69. Vous êtes invités à lire ce paragraphe. La couche [DAO] est à utiliser telle quelle. Elle vous sera donnée.

11.18 Les tests

Testez la vue [ConfigVue] :

Pour les tests :

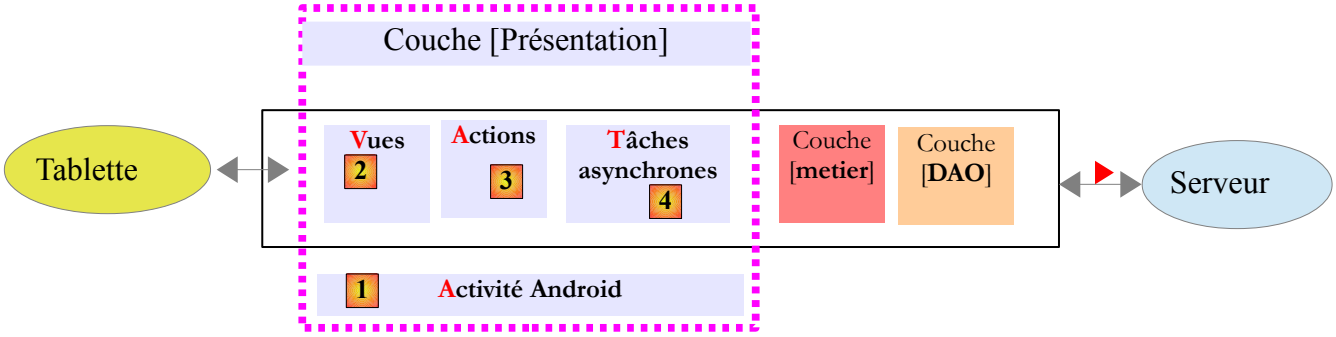
- vous devez configurer le réseau du PC et des Arduinos :



- lancez votre serveur REST. Vérifiez-le ;
- testez votre client Android.

11.19 Travail à faire

Nous venons d'expliquer toute la chaîne d'exécution pour la vue [ConfigVue] :



Question 5 : En suivant cet exemple, réalisez successivement les différents onglets de l'application présentés au paragraphe 11.1, page 82.

12 Le client web mobile du projet Domotique

Nous abordons maintenant l'écriture du client web mobile.

A lire

- [ref1] : " Introduction aux frameworks JSF2, Primefaces et Primefaces Mobile " disponible à l'URL [http://tahe.developpez.com/java/primefaces/] ;
- [ref3] : " Introduction à Java EE avec Netbeans 6.8 " disponible à l'URL [http://tahe.developpez.com/java/javaee].

12.1 Les vues du client web mobile

Le client web mobile permet de gérer les Arduinos à distance avec le navigateur d'un smartphone ou d'une tablette. Il présente à l'utilisateur les écrans suivants :

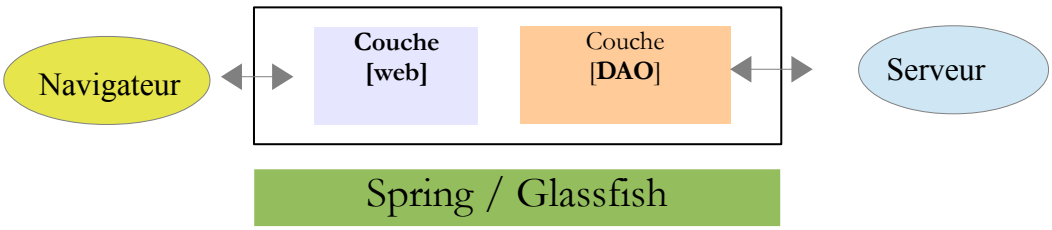
La vue d'accueil [1] propose les opérations possibles. La vue [pinWrite] [2] permet d'écrire une valeur sur une pin d'un Arduino.

La vue [pinRead] [1] permet de lire la valeur d'une pin d'un Arduino. La vue [blink] [2] permet de faire clignoter une led d'un Arduino :

La vue [commands] [1] permet d'envoyer une commande JSON à un Arduino. De chaque vue, on peut revenir à la vue d'accueil avec l'icône [2].

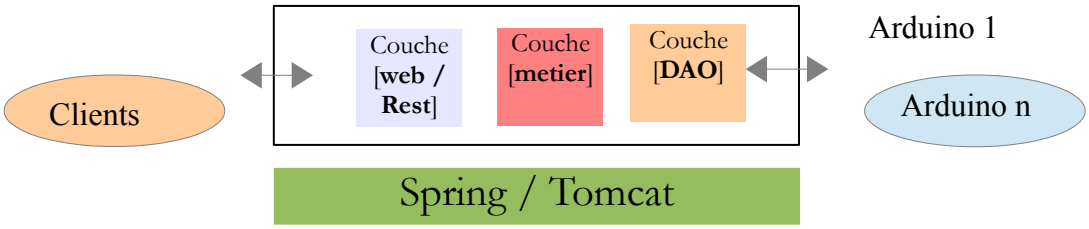
12.2 L'architecture du client web mobile

L'architecture du client web mobile est la suivante :



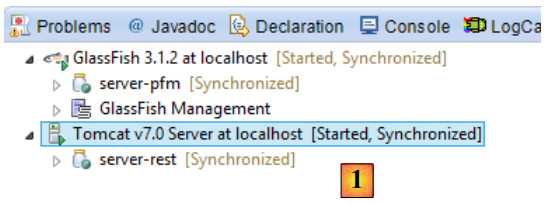
- la couche [DAO] communique avec le serveur REST. C'est un client REST implémenté par **Spring-Android** ;
- la couche [web] est une couche JSF2 (Java Server Faces) avec la bibliothèque Ajax PFM (Primefaces Mobile).

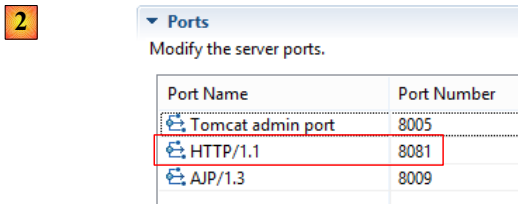
La couche [DAO] dialogue avec le serveur REST des Arduinos :



Le serveur REST des Arduinos est exécuté sur un serveur Tomcat. Pour l'application web mobile, nous utiliserons le serveur Glassfish. En mode développement on est amené à redéployer souvent les applications sur les serveurs. Or redéployer une application sur le serveur Tomcat relance Tomcat lui-même donc l'application REST des Arduinos. En mettant le client web mobile sur Glassfish on évite de relancer le serveur REST.

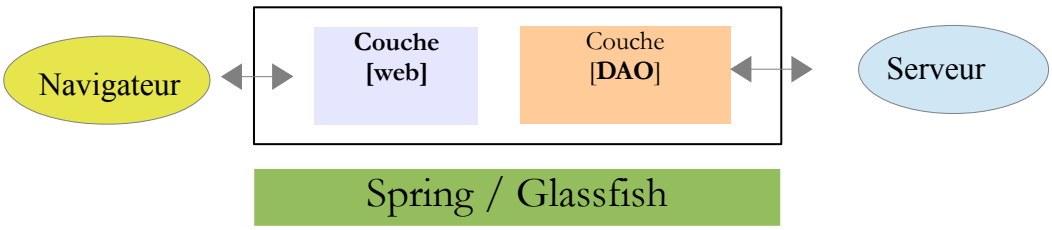
Pour que les serveurs Glassfish et Tomcat puissent fonctionner ensemble, il faut qu'ils travaillent sur des ports différents. Or par défaut, ils travaillent tous les deux sur le port 8080. On pourra changer le port du serveur Tomcat de la façon suivante :



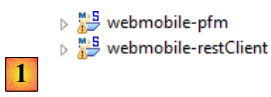


- en [1], dans la fenêtre *Servers* double-cliquer sur le serveur Tomcat ;
- en [2], modifier son port HTTP.

12.3 Les projets Eclipse du client web mobile



Les projets Eclipse suivent l'architecture ci-dessus :

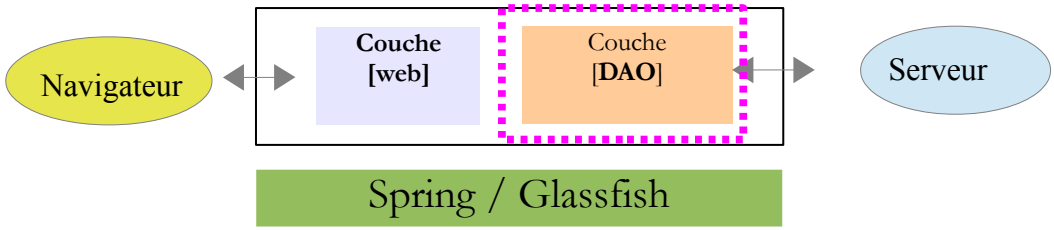


En [1] les deux projets Eclipse du client web mobile :

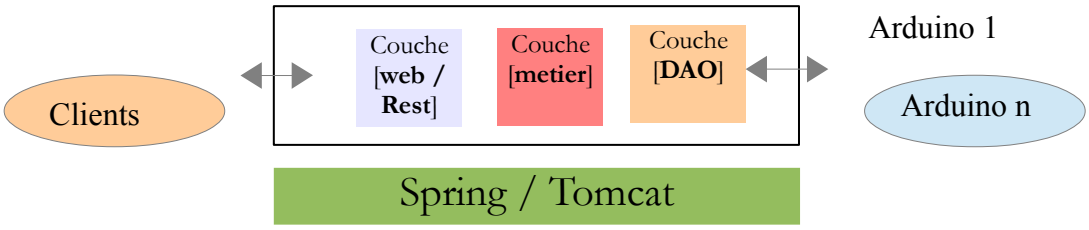
- [webmobile-restClient] : projet de la couche [DAO] ;
- [webmobile-pfm] : le projet web Primefaces Mobile

12.4 La couche [DAO] du client web mobile

Dans l'architecture du client web mobile



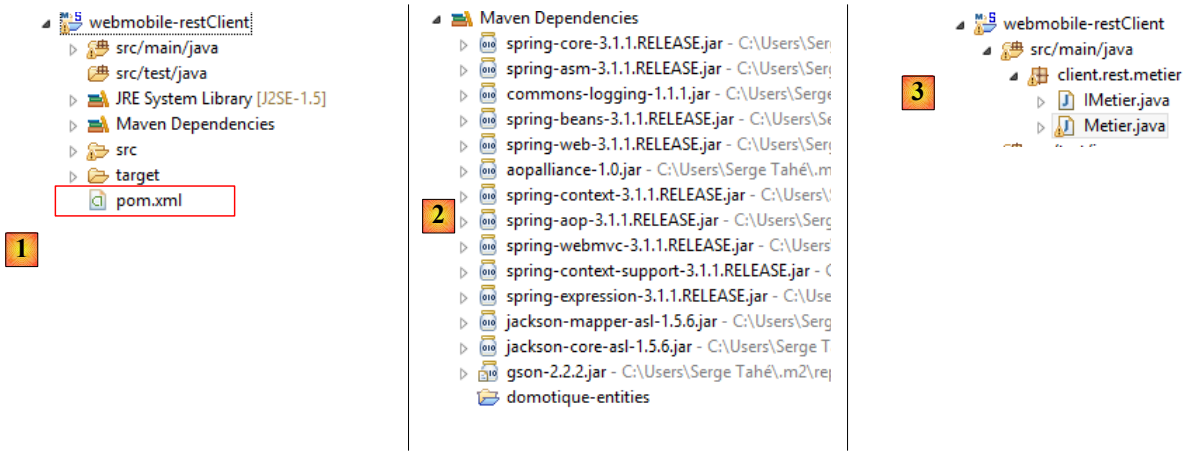
La couche [DAO] dialogue avec le serveur REST des Arduinos :



La couche [DAO] est un client REST comme l'étaient les couches [métier] et [DAO] du projet Android (paragraphe 11.16, page 116 et paragraphe 11.17, page 118). On y trouve du code analogue.

12.4.1 Le projet Eclipse

Le projet Eclipse de la couche [DAO] est le suivant :



- en [1], le projet [webmobile-restClient] est un projet Maven ;
- en [2], les dépendances Maven ;
- en [3], l'implémentation du client REST ;

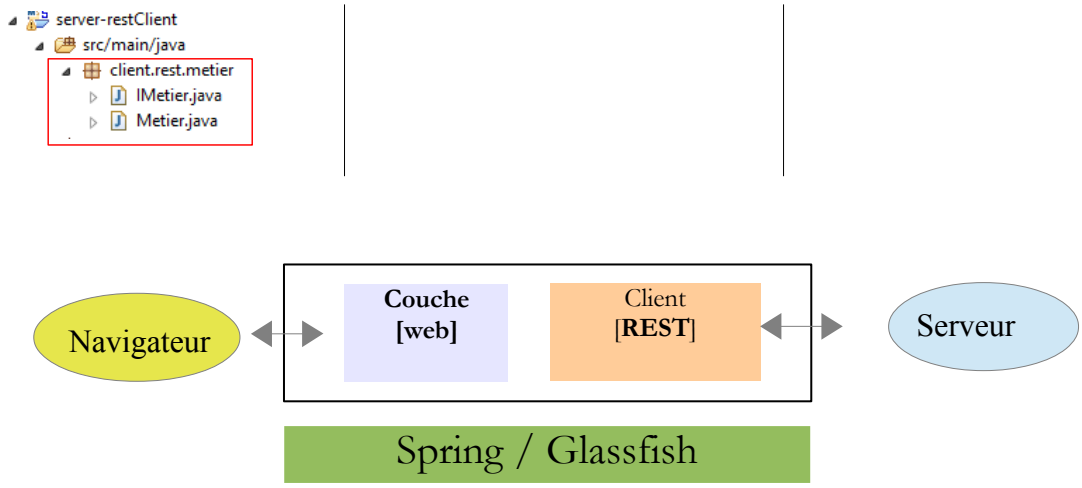
12.4.2 Les dépendances Maven

Le fichier [pom.xml] du projet est le suivant :

```
1. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
   4.0.0.xsd">
3.    <modelVersion>4.0.0</modelVersion>
4.
5.    <groupId>istia.st.domotique.webmobile</groupId>
6.    <artifactId>webmobile-restClient</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>jar</packaging>
9.
10.   <name>webmobile-restClient</name>
11.   <url>http://maven.apache.org</url>
12.
13.   <properties>
14.      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.      <spring.version>3.1.1.RELEASE</spring.version>
16.      <jackson.mapper.version>1.5.6</jackson.mapper.version>
17.   </properties>
18.
19.   <repositories>
20.      <repository>
21.         <id>com.springsource.repository.bundles.release</id>
22.         <name>SpringSource Enterprise Bundle Repository - Release</name>
23.         <url>http://repository.springsource.com/maven/bundles/release</url>
24.      </repository>
25.   </repositories>
26.
27.
28.   <dependencies>
29.      <dependency>
30.         <groupId>org.springframework</groupId>
31.         <artifactId>spring-webmvc</artifactId>
32.         <version>${spring.version}</version>
33.      </dependency>
34.      <dependency>
35.         <groupId>org.codehaus.jackson</groupId>
36.         <artifactId>jackson-mapper-asl</artifactId>
37.         <version>${jackson.mapper.version}</version>
38.      </dependency>
39.      <dependency>
40.         <groupId>istia.st.domotique.common</groupId>
41.         <artifactId>domotique-entities</artifactId>
42.         <version>1.0-SNAPSHOT</version>
43.      </dependency>
44.   </dependencies>
45. </project>
```

- lignes 5-7 : l'identité Maven du projet ;
- lignes 29-33 : Pour le client REST, on a besoin du framework *SpringMVC* ;
- lignes 34-38 : la dépendance sur la bibliothèque JSON *Jackson* ;
- lignes 39-43 : le client et le serveur REST échangent des entités définies dans le projet Maven [domotique-entities].

12.4.3 L'implémentation du client REST



Le client REST offre à la couche [web], l'interface [IMetier] suivante :

```
1.  package client.rest.metier;
2.
3.  ...
4.
5.  public interface IMetier {
6.    // liste des arduinos
7.    public Collection<Arduino> getArduinos();
8.    // lecture d'une pin
9.    public Reponse pinRead(String idCommande, String idArduino, int pin, String mode);
10.   // écriture d'une pin
11.   public Reponse pinWrite(String idCommande, String idArduino, int pin, String mode,int
    val);
12.   // faire clignoter une led
13.   public Reponse faireClignoterLed(String idCommande, String idArduino, int pin, int
    millis, int nbIter);
14.   // envoyer une suite de commandes Json à un Arduino
15.   public List<String> sendCommandesJson(String idArduino, List<String> commandes);
16.   // envoyer une suite de commandes à un Arduino
17.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes);
18. }
```

C'est l'interface de la couche [métier] du serveur. C'est normal. Lorsqu'on met bout à bout le client et le serveur, on voit que la couche [web] du client dialogue avec la couche [métier] du serveur.

L'implémentation [Metier] de cette interface pourrait être la suivante :

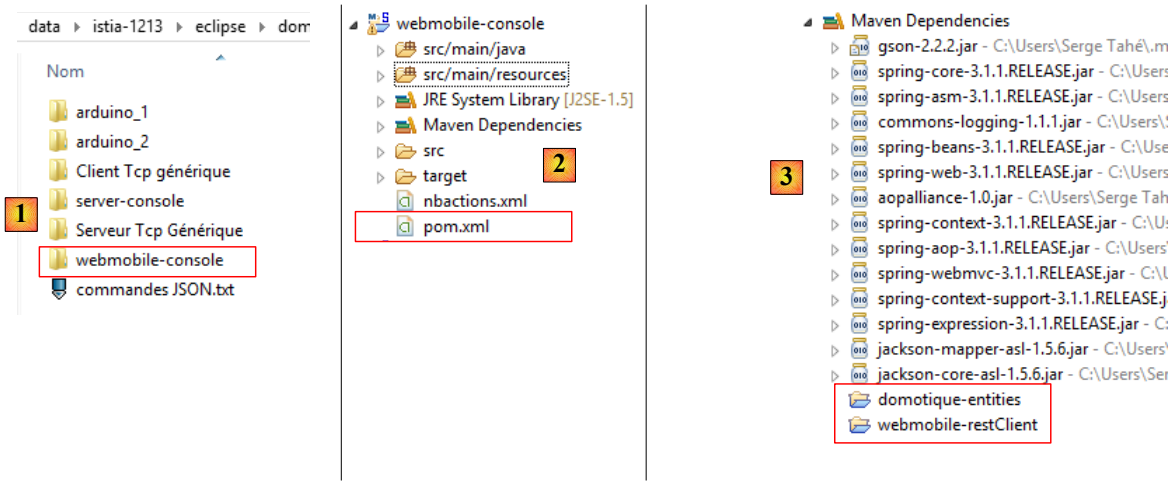
```
1.  package client.rest.metier;
2.
3.  ...
4.
5.  @Service
6.  public class Metier implements IMetier {
7.
8.    // client REST
9.    @Autowired
10.   private RestTemplate restTemplate;
11.   // mapper JSON
12.   private Gson gson = new Gson();
13.   // ip service REST
14.   private String urlServiceRest;
```

```
15.
16.   // constructeur
17.   public Metier() {
18.   }
19.
20.   public Collection<Arduino> getArduinos() {
21. ...
22.   }
23.
24.   public Reponse pinRead(String idCommande, String idArduino, int pin, String mode) {
25. ...
26.   }
27.
28.   public Reponse pinWrite(String idCommande, String idArduino, int pin, String mode, int
    val) {
29. ...
30.   }
31.
32.   public Reponse faireClignoterLed(String idCommande, String idArduino, int pin, int
    millis, int nbIter) {
33. ...
34.   }
35.
36.   @SuppressWarnings("unchecked")
37.   public List<String> sendCommandesJson(String idArduino, List<String> commandes) {
38. ...
39.   }
40.
41.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes) {
42. ...
43.   }
44.
45.   // exécution de l'appel au service REST
46.   public String executeRestService(String method, String urlService, Object request,
    Map<String, String> paramètres) {
47. ....
48.   }
49.
50.   // setters Spring
51. ...
52. }
```

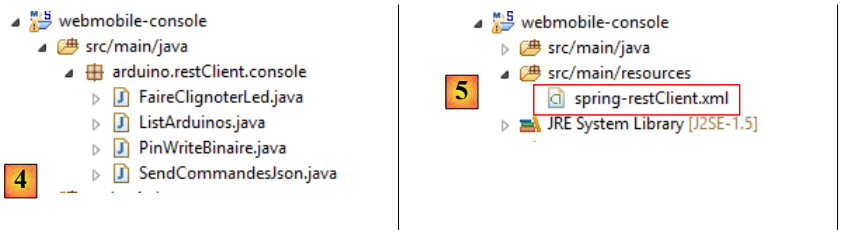
Vous connaissez cette implémentation. C'est celle que vous avez écrite pour le client Android. Revoyez les couches [métier] au paragraphe 11.16, page 116 et [DAO] au paragraphe 11.17, page 118 du projet Android. Il y a une différence : la bibliothèque JSON utilisée est *Jackson* et non *Gson*.

Question 6 : écrire la classe [RestMetier].

12.4.4 Les tests du client REST



- dans le dossier [support] du TP, vous trouverez un projet Eclipse [webmobile-console] [1] ;
- importez-le dans Eclipse. C'est un projet Maven [2] ;
- en [3], les dépendances Maven du projet. Celui-ci dépend du client REST [webmobile-restClient]. Les autres dépendances en découlent ;



- en [4], vous trouverez quatre applications Java testant certaines fonctionnalités du client REST ;
- les quatre applications utilisent Spring pour instancier le client REST [5].

Examinons le code de la classe [ListArduinos] qui affiche la liste des Arduinos connectés :

```
1.  package arduino.restClient.console;
2.
3.  ...
4.  public class ListArduinos {
5.
6.     public static void main(String[] args) {
7.        IMetier metier = (IMetier) new ClassPathXmlApplicationContext("spring-
    restClient.xml").getBean("metier");
8.        System.out.println("Liste des Arduinos connectés");
9.        Collection<Arduino> arduinos = metier.getArduinos();
10.       if (arduinos.size() == 0) {
11.          System.out.println("Aucun Arduino n'est connecté actuellement...");
12.       } else {
13.          for (Arduino arduino : arduinos) {
14.             System.out.println(arduino);
15.          }
16.       }
17.    }
18. }
```

- ligne 7 : le client REST est instancié ;
- ligne 9 : on lui demande la liste des Arduinos connectés ;

Le fichier de configuration de Spring [spring-restClient.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
```

```
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.         xmlns:p="http://www.springframework.org/schema/p"
5.         xmlns:context="http://www.springframework.org/schema/context"
6.         xmlns:oxm="http://www.springframework.org/schema/oxm"
7.         xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
8.          http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd
9.          http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm-3.0.xsd"
    xmlns:util="http://www.springframework.org/schema/util">
10.
11.   <!-- couche applicative -->
12.   <bean id="metier" class="client.rest.metier.Metier">
13.     <property name="urlServiceRest" value="localhost:8081/server-rest"/>
14.     <property name="restTemplate" ref="restTemplate"/>
15.   </bean>
16.   <!-- rest template -->
17.   <bean id="restTemplate" class="org.springframework.web.client.RestTemplate">
18.     <property name="messageConverters">
19.       <list>
20.         <ref bean="jacksonConverter"/>
21.         <ref bean="stringConverter"/>
22.       </list>
23.     </property>
24.   </bean>
25.   <bean id="jacksonConverter"
    class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"/>
26.   <bean id="stringConverter"
    class="org.springframework.http.converter.StringHttpMessageConverter"/>
27. </beans>
```

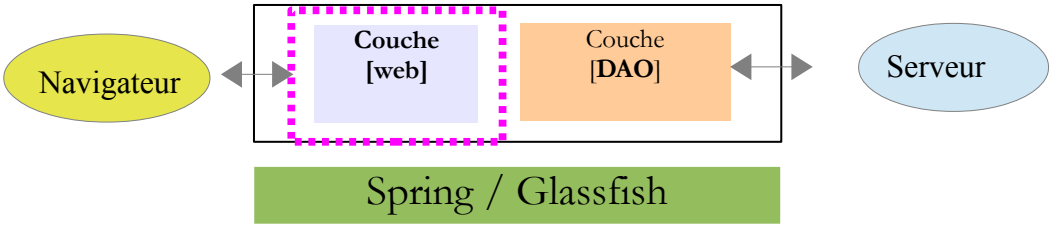
- ligne 12 : définition du client REST ;
- ligne 13 : injection de l'URL du service REST distant. Vous serez peut-être amenés à modifier cette valeur ;
- ligne 14 : injection du client REST de Spring ;
- ligne 17 : le client REST de Spring. La classe appartient au paquetage [org.springframework] ;
- ligne 18 : on injecte dans cette classe une liste de convertisseurs de messages. Cette notion a été expliquée au paragraphe ??, page 60.
- ligne 25 : le convertisseur des messages JSON. On utilise ici la bibliothèque Jackson ;

Travail à faire : Passer les quatre tests du client REST.

12.5 La couche [web] de l'application web mobile

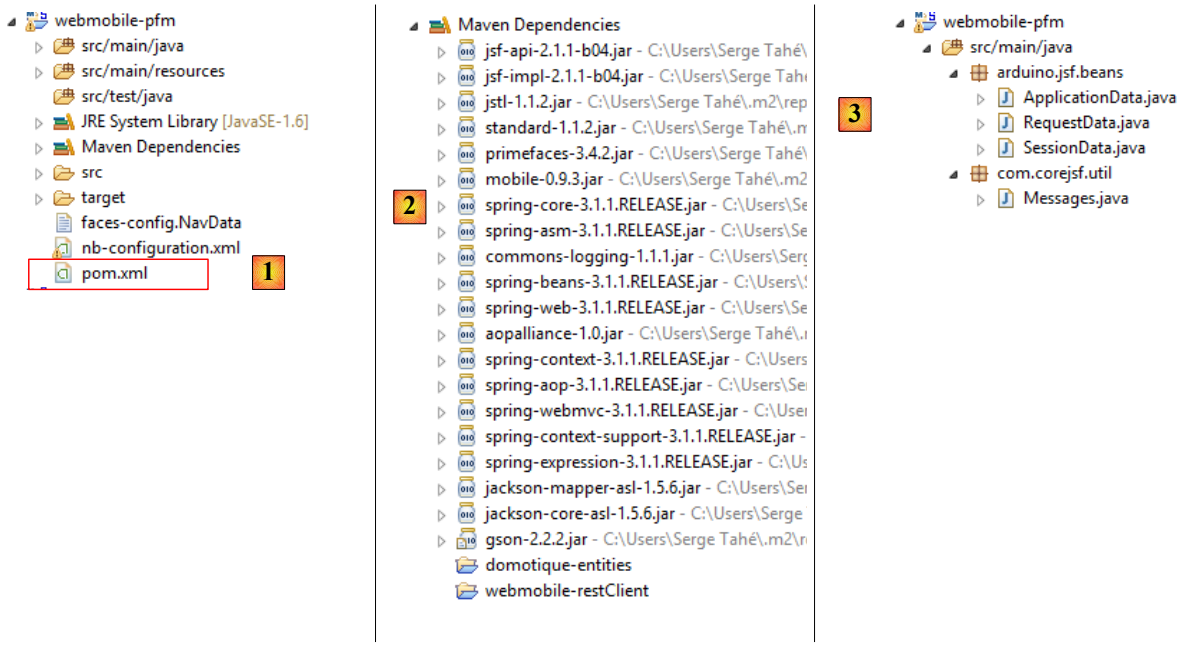
A lire :

- le cours JSF (Java Server Faces) de [**ref1**] ;



12.5.1 Le projet Eclipse

Le projet Eclipse de la couche [web] est le suivant :



- en [1], le projet est un projet Maven ;
- en [2], les dépendances Maven du projet. Celui-ci dépend uniquement du client REST [webmobile-restClient]. Les autres dépendances en découlent ;
- en [3], les classes du projet :
    - le paquetage [arduino.jsf.beans] contient les trois beans JSF de l'application de portées respectives *Application*, *Session* et *Request,*
    - le paquetage [com.corejsf.util] est un paquetage utilitaire pour gérer l'internationalisation des messages. Nous ne l'expliciterons pas ici. C'est fait dans le cours JSF de [**ref1**] ;

12.5.2 Le projet Maven

Le fichier [pom.xml] est le suivant :

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2.            xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
3.    <modelVersion>4.0.0</modelVersion>
4.
5.    <groupId>istia.st.domotique.webmobile</groupId>
6.    <artifactId>webmobile-pfm</artifactId>
7.    <version>1.0-SNAPSHOT</version>
8.    <packaging>war</packaging>
9.
10.   <name>server-pfm</name>
11.
12.   <properties>
13.     <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
14.     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
15.   </properties>
16.
17.   <dependencies>
18.     <dependency>
19.       <groupId>com.sun.faces</groupId>
20.       <artifactId>jsf-api</artifactId>
21.       <version>2.1.1-b04</version>
```

```
22.     </dependency>
23.     <dependency>
24.       <groupId>com.sun.faces</groupId>
25.       <artifactId>jsf-impl</artifactId>
26.       <version>2.1.1-b04</version>
27.     </dependency>
28.     <dependency>
29.       <groupId>org.primefaces</groupId>
30.       <artifactId>mobile</artifactId>
31.       <version>0.9.3</version>
32.       <type>jar</type>
33.     </dependency>
34.     <dependency>
35.       <groupId>istia.st.domotique.webmobile</groupId>
36.       <artifactId>webmobile-restClient</artifactId>
37.       <version>1.0-SNAPSHOT</version>
38.     </dependency>
39.   </dependencies>
40.
41.   <build>
42.     ...
43.   </build>
44.
45.   <repositories>
46.     <repository>
47.       <id>prime-repo</id>
48.       <name>PrimeFaces Maven Repository</name>
49.       <url>http://repository.primefaces.org</url>
50.       <layout>default</layout>
51.     </repository>
52.   </repositories>
53. </project>
```

- lignes 5-7 : l'identité Maven du projet ;
- lignes 18-27 : les dépendances vis à vis des bibliothèques JSF ;
- lignes 28-33 : les dépendances vis à vis de la bibliothèque Primefaces Mobile ;
- lignes 34-38 : la dépendance sur le client REST ;
- lignes 46-51 : le dépôt MAven où peut être trouvée la bibliothèque Primefaces Mobile.

12.5.3 La configuration de la couche [web]

- la configuration et les pages XHTML de la couche [web] sont dans le dossier [webapp] [1] ;
- le projet est configuré par les fichiers [web.xml] et [faces-config.xml] [2].

12.5.4 Le fichier [web.xml]

Le fichier [web.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-
    app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
3.    <context-param>
4.      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
5.      <param-value>client</param-value>
6.    </context-param>
7.    <context-param>
8.      <param-name>javax.faces.FACELETS_SKIP_COMMENTS</param-name>
9.      <param-value>true</param-value>
10.   </context-param>
11.   <context-param>
12.     <param-name>javax.faces.PROJECT_STAGE</param-name>
13.     <param-value>Development</param-value>
14.   </context-param>
15.   <servlet>
16.     <servlet-name>Faces Servlet</servlet-name>
17.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
18.     <load-on-startup>1</load-on-startup>
19.   </servlet>
20.   <servlet-mapping>
21.     <servlet-name>Faces Servlet</servlet-name>
22.     <url-pattern>/faces/*</url-pattern>
23.   </servlet-mapping>
24.   <session-config>
25.     <session-timeout>30</session-timeout>
26.   </session-config>
27.   <welcome-file-list>
28.     <welcome-file>faces/index.xhtml</welcome-file>
29.   </welcome-file-list>
30. </web-app>
```

C'est le fichier standard de configuration d'une application web JSF. Il n'y a rien ici de spécifique à Primefaces Mobile.

12.5.5 Le fichier de configuration [faces-config.xml]

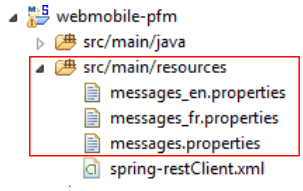
Le fichier [faces-config.xml] configure les différents éléments d'une application JSF. Il est ici le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8'?>
2.
3.  <!-- ========== FULL CONFIGURATION FILE ================================== -->
4.
5.  <faces-config version="2.0"
6.                xmlns="http://java.sun.com/xml/ns/javaee"
7.                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8.                xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">
9.
10.   <application>
11.     <!-- le fichier des messages -->
12.     <resource-bundle>
13.       <base-name>
14.         messages
15.       </base-name>
16.       <var>msg</var>
17.     </resource-bundle>
18.     <message-bundle>messages</message-bundle>
19.     <default-render-kit-id>PRIMEFACES_MOBILE</default-render-kit-id>
```

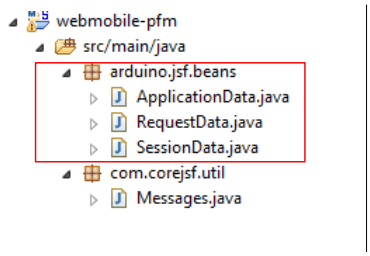
```
20.   </application>
21.   <!-- définition des différents beans -->
22.   <!-- de portée application -->
23.   <managed-bean>
24.     <managed-bean-name>applicationData</managed-bean-name>
25.     <managed-bean-class>arduino.jsf.beans.ApplicationData</managed-bean-class>
26.     <managed-bean-scope>application</managed-bean-scope>
27.   </managed-bean>
28.   <!-- de portée session -->
29.   <managed-bean>
30.     <managed-bean-name>sessionData</managed-bean-name>
31.     <managed-bean-class>arduino.jsf.beans.SessionData</managed-bean-class>
32.     <managed-bean-scope>session</managed-bean-scope>
33.   </managed-bean>
34.   <!-- de portée request -->
35.   <managed-bean>
36.     <managed-bean-name>requestData</managed-bean-name>
37.     <managed-bean-class>arduino.jsf.beans.RequestData</managed-bean-class>
38.     <managed-bean-scope>request</managed-bean-scope>
39.     <managed-property>
40.       <property-name>applicationData</property-name>
41.       <value>#{applicationData}</value>
42.     </managed-property>
43.   </managed-bean>
44.
45. </faces-config>
```

- lignes 12-18 : définissent le fichier des messages internationalisés. Il s'appelle [messages.properties] et se trouve dans le *Classpath* du projet [1] :

1



2



- ligne 19 : un kit de rendu des pages XHTML. Ici, c'est le kit de Primefaces Mobile qui est utilisé ;
- lignes 23-27 : le bean de portée [Application][2] ;
- lignes 29-33 : le bean de portée [Session] [2] ;
- lignes 35-43 : le bean de portée [Request] [2]. On notera qu'on y injecte la référence du bean de portée [Application] (lignes 39-42).

12.5.6 Le bean de portée [Application]

Le bean de portée [Application] est instancié au démarrage de l'application JSF. Il reste en mémoire pendant la durée de vie de l'application d'où son nom. Il est accessible à toutes les requêtes de tous les utilisateurs. A cause de cette concurrence d'accès, il est généralement utilisé uniquement en lecture.

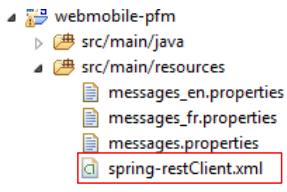
Le code de la classe [ApplicationData] est le suivant :

```
1.  package arduino.jsf.beans;
2.
3.  ...
4.
5.  public class ApplicationData implements IMetier, Serializable {
6.    // couche [métier]
7.
8.     private static final long serialVersionUID = 1L;
```

```
9.    private IMetier metier;
10.
11.   // initialisation
12.   @PostConstruct
13.   public void init() {
14.     // couche métier
15.     metier = (IMetier) new ClassPathXmlApplicationContext("spring-
    restClient.xml").getBean("metier");
16.   }
17.
18.   @Override
19.   public Collection<Arduino> getArduinos() {
20.     return metier.getArduinos();
21.   }
22.
23.   @Override
24.   public Reponse pinRead(String idCommande, String idArduino, int pin, String mode) {
25.     return metier.pinRead(idCommande, idArduino, pin, mode);
26.   }
27.
28.   @Override
29.   public Reponse pinWrite(String idCommande, String idArduino, int pin, String mode, int
    val) {
30.     return metier.pinWrite(idCommande, idArduino, pin, mode, val);
31.   }
32.
33.   @Override
34.   public Reponse faireClignoterLed(String idCommande, String idArduino, int pin, int
    millis, int nbIter) {
35.     return metier.faireClignoterLed(idCommande, idArduino, pin, millis, nbIter);
36.   }
37.
38.   @Override
39.   public List<String> sendCommandesJson(String idArduino, List<String> commandes) {
40.     return metier.sendCommandesJson(idArduino, commandes);
41.   }
42.
43.   @Override
44.   public List<Reponse> sendCommandes(String idArduino, List<Commande> commandes) {
45.     return metier.sendCommandes(idArduino, commandes);
46.   }
47. }
```

- ligne 5 : la classe [ApplicationData] implémente l'interface [IMetier] du client REST ;
- ligne 9 : une référence sur le client REST ;
- lignes 13-16 : la référence sur le client REST est obtenue avec Spring ;
- lignes 18-46 : implémentation de l'interface [IMetier]. Tout est délégué au client REST. Nous allons ainsi pouvoir cacher le client REST aux autres beans de l'application JSF. Ceux-ci s'adresseront au bean [ApplicationData] plutôt qu'au client REST.

12.5.7 Le fichier de configuration de Spring



Le fichier [spring-restClient.xml] est le même que celui décrit au paragraphe ??, page 129.

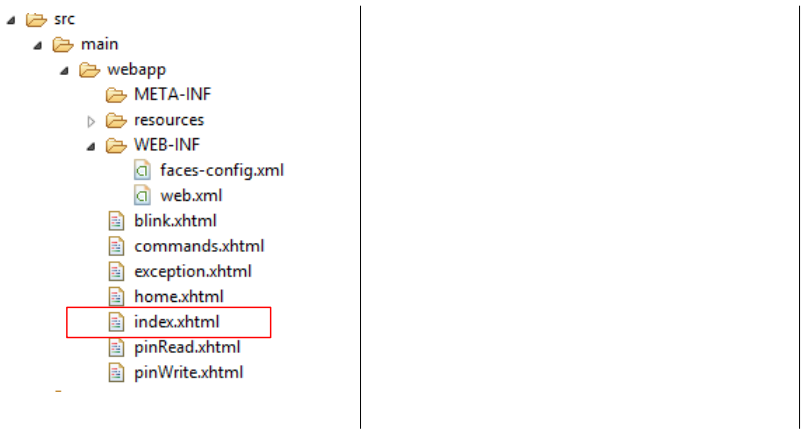
12.5.8 Le bean de portée Session

Le bean de portée [Session] est la mémoire d'un utilisateur. Toutes les requêtes de celui-ci peuvent écrire et lire de l'information dans cette mémoire. Ici, nous ne mémorisons qu'une information :

```
1. package arduino.jsf.beans;
2.
3.
4. public class SessionData {
5.   // données de session
6.   String locale="fr";
7.
8.   // getters / setters
9. ...
10. }
```

- ligne 6 : l'unique information de la session : la langue utilisée pour les pages affichées par l'application web mobile, ici le français.

12.5.9 La page principale du projet PFM (PrimeFaces Mobile)



La page [index.html] ci-dessus est l'unique page du projet PFM. Son code est le suivant :

```
1. <f:view xmlns="http://www.w3.org/1999/xhtml"
2.         xmlns:f="http://java.sun.com/jsf/core"
3.         xmlns:h="http://java.sun.com/jsf/html"
4.         xmlns:ui="http://java.sun.com/jsf/facelets"
5.         xmlns:p="http://primefaces.org/ui"
6.         xmlns:pm="http://primefaces.org/mobile"
7.         contentType="text/html"
8.         locale="#{sessionData.locale}">
9.
10.  <pm:page title="#{msg['appli.titre']}">
11.    <pm:view id="home">
12.      <ui:include src="home.xhtml"/>
13.    </pm:view>
14.    <pm:view id="commands">
15.      <ui:include src="commands.xhtml"/>
16.    </pm:view>
17.    <pm:view id="blink">
18.      <ui:include src="blink.xhtml"/>
19.    </pm:view>
20.    <pm:view id="pinWrite">
21.      <ui:include src="pinWrite.xhtml"/>
22.    </pm:view>
23.    <pm:view id="pinRead">
```

```
24.       <ui:include src="pinRead.xhtml"/>
25.     </pm:view>
26.   </pm:page>
27. </f:view>
```

- la page [index.html] est l'unique page affichée par l'application. Elle définit cinq **vues**. A un moment donné, une seule de ces cinq vues est visible. Au démarrage c'est la première qui est visible. On passe ensuite aux suivantes par navigation ;
- lignes 11-13 : la vue [home]. Elle présente quatre liens pour passer à l'une des quatre autres vues. Ces dernières ont toutes une icône permettant de revenir à la vue [home] ;
- lignes 14-16 : la vue [commands] permet d'envoyer une commande JSON aux Arduinos connectés ;
- lignes 17-19 : la vue [blink] permet de faire clignoter une led sur les Arduinos connectés ;
- lignes 20-22 : la vue [pinWrite] permet d'écrire une valeur binaire ou analogique sur l'une des pins des Arduinos connectés ;
- lignes 23-25 : la vue [pinRead] permet de lire la valeur binaire ou analogique de l'une des pins des Arduinos connectés.

12.5.10 La vue [home]

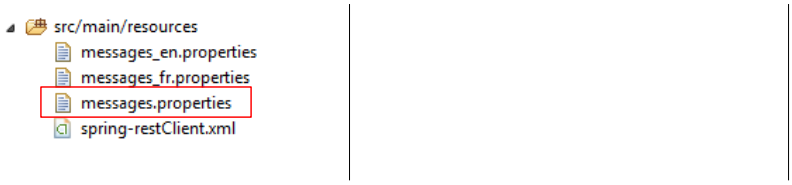
Le code [home.xhtml] de cette vue est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.
10.   <!-- menu -->
11.   <pm:header title="#{msg['appli.titre']}" swatch="b">
12.     <f:facet name="left">
13.       <p:button icon="gear" value=" "  href="#config" />
14.     </f:facet>
15.   </pm:header>
16.   <pm:content>
17.     <h:form id="frmHome">
18.       <p:dataList id="list1" type="inset">
19.         <f:facet name="header">#{msg['menu.options']}</f:facet>
20.         <h:outputLink value="#pinRead">#{msg['menu.pinRead']}</h:outputLink>
21.         <h:outputLink value="#pinWrite">#{msg['menu.pinWrite']}</h:outputLink>
22.         <h:outputLink value="#blink">#{msg['menu.clignoter']}</h:outputLink>
23.         <h:outputLink value="#commands">#{msg['menu.commande']}</h:outputLink>
24.       </p:dataList>
25.     </h:form>
26.   </pm:content>
27.   <pm:footer>
28.     <div align="center">
29.       &copy; Votre nom<br/>ISTIA, université d'Angers
30.     </div>
```

```
31.   </pm:footer>
32.
33. </html>
```

- lignes 11-15 : définissent l'entête de la vue ;
- ligne 11 : le libellé [1] de l'entête ;
- ligne 13 : le bouton [2]. Un clic sur ce bouton fait apparaître la vue [config.xhtml] (attribut *href*). Ici, elle n'a pas été définie. Donc le bouton sera inactif. Vous pourrez le rendre actif en créant une vue [config.xhtml] permettant de choisir une autre langue que le français ;
- ligne 16 : le contenu de la vue. La balise est obligatoire ;
- ligne 17 : le formulaire. Il y en normalement un dans chaque vue ;
- ligne 18 : l'unique composant de ce formulaire, un [DataList] ;
- ligne 19 : le libellé [3] du *DataList* ;
- ligne 20 : un lien vers la vue [pinRead] (attribut *value*) ;
- ligne 21 : un lien vers la vue [pinWrite] (attribut *value*) ;
- ligne 22 : un lien vers la vue [blink] (attribut *value*) ;
- ligne 23 : un lien vers la vue [commands] (attribut *value*) ;
- ligne 29 : le bas de page de la vue affichée. Mettez votre nom par exemple.

12.5.11 Le fichier des messages



La vue [home.xhtml] détaillée précédemment utilise des messages provenant du fichier [messages_fr.properties] :

```
1.  # appli
2.  appli.titre=Mobilit\u00e9 & Domotique
3.  appli.footer=
4.  # ent\u00eate
5.  menu.options=Options
6.  menu.pinWrite=| Ecrire une valeur sur une pin
7.  menu.pinRead=| Lire la valeur d'une pin
8.  menu.clignoter=| Faire clignoter une led
9.  menu.commande=| Envoyer une commande Json
10. menu.rafraichir=Rafra\u00eechir la liste
11. # page pinWrite
12. cmd.executer=Ex\u00e9cuter
13. cmd.rafraichir=Rafra\u00eechir
14. ...
```

Si vous internationalisez l'application en y ajoutant l'anglais, vous aurez à remplir le fichier [messages_en.properties] avec les mêmes clés que dans [messages_fr.properties] et des textes en anglais.

12.5.12 La vue [blink.xhtml]

Nous allons examiner l'une des vues comme exemple à suivre. La vue [blink.xhtml] est la suivante :

Le code XHTML est le suivant :

```
1.  <?xml version='1.0' encoding='UTF-8' ?>
2.  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3.  <html xmlns="http://www.w3.org/1999/xhtml"
4.        xmlns:h="http://java.sun.com/jsf/html"
5.        xmlns:p="http://primefaces.org/ui"
6.        xmlns:f="http://java.sun.com/jsf/core"
7.        xmlns:pm="http://primefaces.org/mobile"
8.        xmlns:ui="http://java.sun.com/jsf/facelets">
9.    <!-- blink -->
10.   <pm:header title="#{msg['appli.titre']}" swatch="b">
11.     <f:facet name="left">
12.       <p:button icon="home" value=" "  href="#home" />
13.     </f:facet>
14.   </pm:header>
15.   <pm:content>
16.     <h:form id="frmBlink">
17.       <!-- liste des arduinos -->
18.       <h2>#{msg['listeArduinos']}</h2>
19.       <h:selectManyCheckbox id="arduinos" value="#{requestData.selectedArduinosIds}">
20.         <f:selectItems var="arduino" value="#{applicationData.arduinos}"
    itemLabel="#{arduino.ip}" itemValue="#{arduino.id}"/>
21.       </h:selectManyCheckbox>
22.       <br/>
23.       <!-- exécution -->
24.       <p:commandButton id="cmdRafraichir" value="#{msg['menu.rafraichir']}"
    immediate="true" update=":frmBlink:arduinos"/>
25.       <h2>#{msg['commande']}</h2>
26.       <!-- n° de pin -->
27.       <pm:field>
28.         <h:outputLabel for="pin" value="#{msg['pinNumber']}"/>
29.         <h:selectOneMenu id="pin" value="#{requestData.pin}">
30.           <f:selectItems var="pin" value="#{requestData.pins}" itemLabel="#{pin}"
    itemValue="#{pin}"/>
31.         </h:selectOneMenu>
32.       </pm:field>
```

```
33.       <h:outputText id="msgErreur1" value="#{msg[requestData.pinMessageKey]}" style="color:
   red"/>
34.       <!-- durée d'un clignotement -->
35.       <pm:field>
36.         <h:outputLabel for="millis" value="#{msg['clignoter.dureeClignotement']}"/>
37.         <h:inputText id="millis" value="#{requestData.dureeClignotement}"
38.                      required="true"
   requiredMessage="#{msg['clignoter.dureeClignotement.required']}"
39.                      validatorMessage="#{msg['clignoter.dureeClignotement.error']}">
40.           <f:validateLongRange minimum="100" maximum="10000"/>
41.         </h:inputText>
42.       </pm:field>
43.       <h:message id="msgErreur2" for="millis" style="color: red"/>
44.       <!-- nombre de clignotements -->
45.       <pm:field>
46.         <h:outputLabel for="nbClignotements" value="#{msg['clignoter.nbClignotements']}"/>
47.         <h:inputText id="nbClignotements" value="#{requestData.nbClignotements}"
48.                      required="true"
   requiredMessage="#{msg['clignoter.nbClignotements.required']}"
49.                      validatorMessage="#{msg['clignoter.nbClignotements.error']}">
50.           <f:validateLongRange minimum="1" maximum="100"/>
51.         </h:inputText>
52.       </pm:field>
53.       <h:message id="msgErreur3" for="nbClignotements" style="color: red"/>
54.       <p:commandButton id="clignoter" value="#{msg['cmd.executer']}"
   action="#{requestData.clignoter}" update=":frmBlink:msgErreur1 :frmBlink:msgErreur2
   :frmBlink:msgErreur3 :frmBlink:reponses :frmBlink:exceptions"/>
55.       <!-- réponses -->
56.       <p:outputPanel id="reponses">
57.         <ui:fragment rendered="#{! empty requestData.réponses}">
58.           <!-- réponses de l'arduino -->
59.           <hr/>
60.           <h2>#{msg['réponsesArduinos']}</h2>
61.           <ul>
62.             <ui:repeat var="réponse" value="#{requestData.réponses}">
63.               <li>#{réponse}</li>
64.             </ui:repeat>
65.           </ul>
66.         </ui:fragment>
67.       </p:outputPanel>
68.       <!-- exceptions -->
69.       <p:outputPanel id="exceptions">
70.         <ui:fragment rendered="#{! empty requestData.exceptions}">
71.           <!-- exceptions -->
72.           <hr/>
73.           <h2>#{msg['exceptions']}</h2>
74.           <ul>
75.             <ui:repeat var="exception" value="#{requestData.exceptions}">
76.               <li>#{exception}</li>
77.             </ui:repeat>
78.           </ul>
79.         </ui:fragment>
80.       </p:outputPanel>
81.     </h:form>
82.   </pm:content>
83.   <pm:footer>
84.     <div align="center">
85.       &copy; Votre nom<br/>ISTIA, université d'Angers
86.     </div>
87.   </pm:footer>
88.
89. </html>
```

- lignes 10-14 : l'entête [1] de la vue ;
- ligne 12 : l'icône [A] qui permet de revenir sur la vue [home] (attribut *href*) ;
- ligne 15 : le contenu de la vue ;
- ligne 16 : le formulaire de la vue ;
- ligne 18 : le libellé [2] ;
- ligne 19-21 : la liste de cases à cocher [3] ;
- ligne 19 : la liste des éléments cochés sera mémorisée dans le champ [selectedArduinosIds] du bean de portée [request] *RequestData* :

```
private String[] selectedArduinosIds;
```

- ligne 20 : la liste est alimentée par la liste des Arduinos connectés. Celle-ci est trouvée dans le bean [ApplicationData] :

```
@Override
public Collection<Arduino> getArduinos() {
  return metier.getArduinos();
}
```

Le libellé de chaque case à cocher sera l'adresse IP de l'Arduino (attribut *itemLabel*). La valeur postée sera l'identifiant de l'Arduino (attribut *itemValue*) ;

- ligne 24 : le bouton [4] qui demande au client REST, la liste des Arduinos connectés. Elle le fait avec un appel Ajax, donc sans rechargement de page. L'attribut *immediate=true* permet de s'affranchir des tests de validité du formulaire. Ils ne seront pas exécutés. La liste des Arduinos de la ligne 19 sera régénérée (attribut *update*) ;
- ligne 25 : le libellé [5] ;
- ligne 27 : un champ PFM ;
- ligne 28 : le libellé [6]. Notez l'attribut *for* qui référence le composant d'id *pin*. PFM utilise cette information pour sa mise en page ;
- lignes 29-31 : la liste déroulante [7]. La valeur postée sera affectée (attribut *value*) au champ suivant de [RequestData] :

```
private int pin;
```

- ligne 30 : la liste déroulante est alimentée (attribut *value*) par le champ suivant de [RequestData] :

```
private String[] pins = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13" };
```

Le libellé affiché (*itemLabel*) ainsi que la valeur postée (*itemValue*) seront le n° de pin.

- ligne 33 : un message d'erreur non représenté sur la copie d'écran. Ce message apparaît si le n° de pin posté est invalide ;
- lignes 35-42 : le champ PFM des composants 8 et 9 ;
- ligne 36 : le libellé [8] ;
- ligne 37 : le champ de saisie [9]. La valeur postée sera mémorisée (attribut *value*) dans le champ suivant de [RequestData] :

```
private String dureeClignotement;
```

On peut se demander si un type *int* n'aurait pas été préférable. A l'affichage, la valeur initiale de [dureeClignotement] est affichée dans le champ de saisie. Si le type est *int*, la valeur 0 est affichée. Avec le type *String*, rien n'est affiché. D'où le choix de ce dernier ;

- ligne 38 : le champ est obligatoire (attribut *required*) ;
- ligne 40 : la durée du clignotement doit être entre 100 ms et 10 s ;
- ligne 43 : affiche un message d'erreur si la saisie est incorrecte ;
- lignes 45-53 : on a un code analogue pour la saisie du nombre de clignotements ;
- lignes 54 : le bouton qui va faire clignoter les leds des Arduinos sélectionnés ;
  La méthode [RequestData].*clignoter* va être exécutée (attribut *action*) par un appel Ajax. Les zones identifiées par *msgErreur1* (ligne 33), *msgErreur2* (ligne 43), *msgErreur3* (ligne 53), *reponses* (ligne 56), *exceptions* (ligne 69) seront mises à jour avec le résultat de l'appel Ajax (attribut *update*) ;
- lignes 56-67 : un bloc de la vue qui contient le fragment de la ligne 57. Celui-ci n'est affiché que si après l'appel Ajax, la liste des réponses des Arduinos est non vide ;

- ligne 60 : le libellé [1] ;
- lignes 62-64 : ce code produit la liste [2] alimentée par le champ suivant de [RequestData] :

```
private List<String> réponses;
```

- ligne 63 : la réponse d'un Arduino est affichée sous la forme [2] ;

- lignes 69-80 : un bloc de la vue qui contient le fragment de la ligne 70. Celui-ci n'est affiché que si après l'appel Ajax, la liste des exceptions produite par l'appel Ajax est non vide ;

- ligne 73 : le libellé [1] ;
- lignes 74-78 : ce code produit la liste [2] alimentée par le champ suivant de [RequestData] :

```
private List<String> exceptions;
```

- ligne 76 : chaque exception est affichée sous la forme [2]. Ici le câble réseau des Arduinos avait été déconnecté de l'ordinateur hôte ;

La vue [blink.xhtml] utilise les messages suivants du fichier [messages_fr.properties] :

```
1. # appli
2. appli.titre=Mobilit\u00e9 & Domotique
3. appli.footer=
4. ..............
5. # vue clignoter
6. cmd.clignoter=Faire clignoter
7. clignoter.nbClignotements=Nombre de clignotements
8. clignoter.dureeClignotement=Dur\u00e9e du clignotement en millisecondes
9. clignoter.dureeClignotement.required=Donn\u00e9e requise
10. clignoter.nbClignotements.required=Donn\u00e9e requise
11. clignoter.dureeClignotement.error=La dur\u00e9e doit \u00eatre comprise entre 100 et 10000
    millisecondes
12. clignoter.nbClignotements.error=Le nombre de clignotements doit \u00eatre compris entre 1
    et 100
13. .............
14. exceptions=Il y a eu des exceptions
```

Pour terminer cet exemple, il ne nous reste plus qu'à décrire la méthode [RequestData].*clignoter* qui fait clignoter les leds.

12.5.13 La méthode [RequestData].clignoter

Le code de la méthode est le suivant :

```
1.  // on fait clignoter les arduinos sélectionnés
2.     public String clignoter() throws IOException {
3.        // raz page
4.        réponses = new ArrayList<String>();
5.        exceptions = new ArrayList<String>();
6.        // exécution de la commande sur les arduinos sélectionnés
7.        for (String arduinoId : selectedArduinosIds) {
8.           // exécution de la commande
9.           try {
10.             réponses.add(applicationData.faireClignoterLed(arduinoId, arduinoId, pin,
    Integer.parseInt(dureeClignotement),
11.                   Integer.parseInt(nbClignotements)).toString());
12.          } catch (Exception ex) {
13.             // on mémorise l'exception
14.             recordException(ex);
15.          }
16.       }
17.       // même page
18.       return null;
19.    }
```

- pour comprendre ce code, il faut se souvenir que les champs [selectedArduinosIds, dureeClignotement, nbClignotements] ont reçu les valeurs saisies par l'utilisateur ;
- ligne 7 : on parcourt la liste des Arduinos sélectionnés par l'utilisateur ;
- ligne 10 : pour chacun d'eux, on demande au bean [ApplicationData] de faire clignoter la led. Les réponses JSON des Arduinos sont cumulés dans la liste de la ligne 4 ;
- ligne 14 : les exceptions sont cumulées dans la liste de la ligne 5 par la méthode privée [recordException] suivante :

```
1.     // mémoriser la pile des exceptions
2.     private void recordException(Exception ex) {
3.        // on enregistre la pile des exceptions
4.        Throwable th = ex;
5.        while (th != null) {
6.           exceptions.add(th.getMessage());
7.           th = th.getCause();
```

```
8.       }
9.  }
```

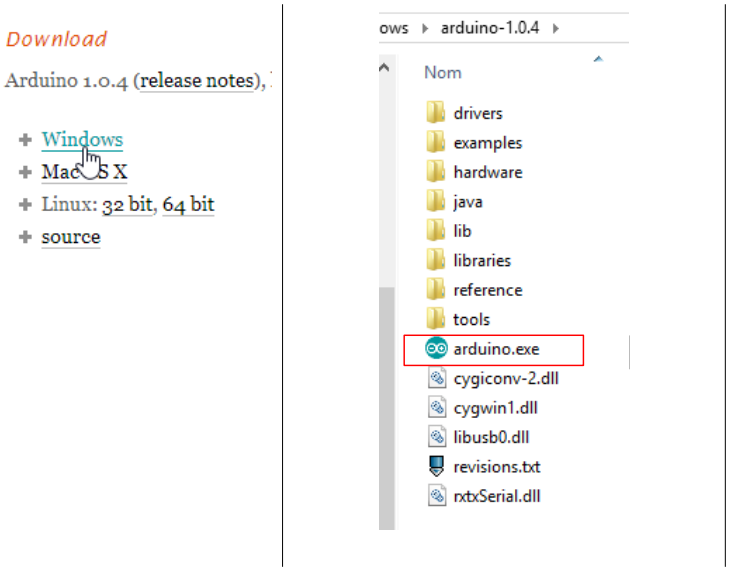
12.6 Travail à faire

Question 7 : en suivant le modèle qui vient d'être décrit pour la vue [**blink**], réalisez les vues [**pinWrite**], [**pinRead**] et [**commands**].

13 Annexes

13.1 Intallation de l'IDE Arduino

Le site officiel de l'Arduino est [http://www.arduino.cc/]. C'est là qu'on trouvera l'IDE de développement pour les Arduinos [http://arduino.cc/en/Main/Software] :



Le fichier téléchargé [1] est un zip qui une fois décompressé donne l'arborescence [2]. On pourra lancer l'IDE en double-cliquant sur l'exécutable [3].

13.2 Intallation du pilote (driver) de l'Arduino

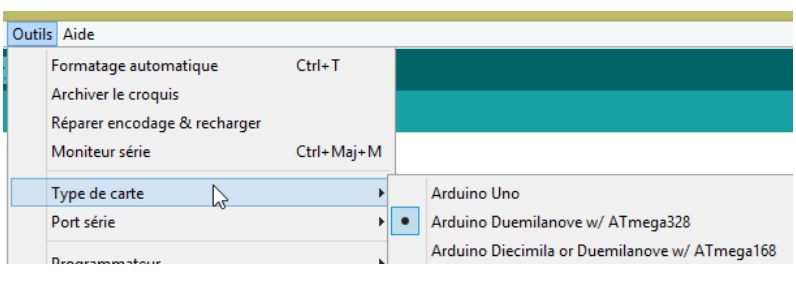
Afin que l'IDE puisse communiquer avec un Arduino, il faut que ce dernier soit reconnu par le PC hôte. Pour cela, on pourra procéder de la façon suivante :

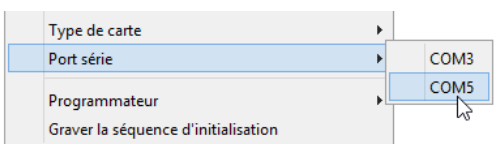
- brancher l'Arduino sur un port USB de l'ordinateur hôte ;
- le système va alors tenter de trouver le pilote du nouveau périphérique USB. Il ne va pas le trouver. Indiquez alors que le pilote est sur le disque à l'endroit **<arduino>/drivers** où <arduino> est le dossier d'installation

13.3 Tests de l'IDE

Pour tester l'IDE, on pourra procéder de la façon suivante :

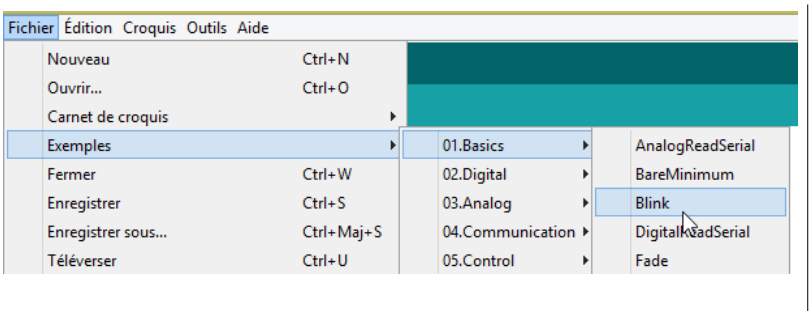
- lancer l'IDE ;
- connecter l'Arduino au PC via son câble USB. Pour l'instant, ne pas inclure la carte réseau ;





- en [1], choisissez le type de la carte Arduino connectée au port USB ;
- en [2], indiquez le port USB sur lequel l'Arduino est connectée. Pour le savoir, débranchez l'Arduino et notez les ports. Rebranchez-le et vérifiez de nouveau les ports : celui qui a été rajouté est celui de l'Arduino ;

Exécutez certains des exemples inclus dans l'IDE :



L'exemple est chargé et affiché :

Les exemples qui accompagnent l'IDE sont très didactiques et très bien commentés. Un code Arduino est écrit en langage C et se compose de deux parties bien distinctes :

- une fonction [**setup**] qui s'exécute une seule fois au démarrage de l'application, soit lorsque celle-ci est " téléversée " du PC hôte sur l'Arduino, soit lorsque l'application est déjà présente sur l'Arduino et qu'on appuie sur le bouton [Reset]. C'est là qu'on met le code d'initialisation de l'application ;
- une fonction [**loop**] qui s'exécute continuellement (boucle infinie). C'est là qu'on met le coeur de l'application.

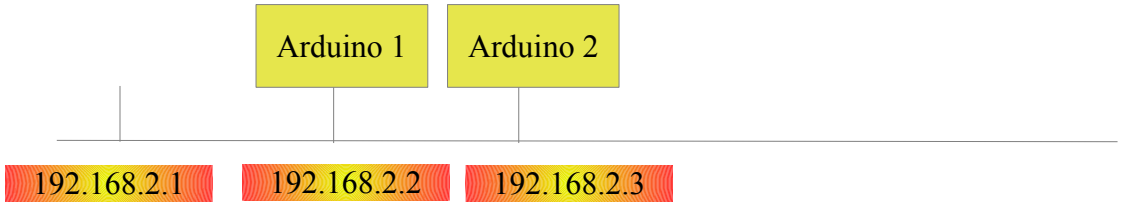
Ici,

- la fonction [setup] configure la pin n° 13 en sortie ;
- la fonction [loop] l'allume et l'éteint de façon répétée : la led s'allume et s'éteint toutes les secondes.

Le programme affiché est transféré (téléversé) sur l'Arduino avec le bouton [1]. Une fois transféré, il s'exécute et la led n° 13 se met à clignoter indéfiniment.

13.4 Connexion réseau de l'Arduino

L'Arduino ou les Arduinos et le PC hôte doivent être sur un même réseau privé :

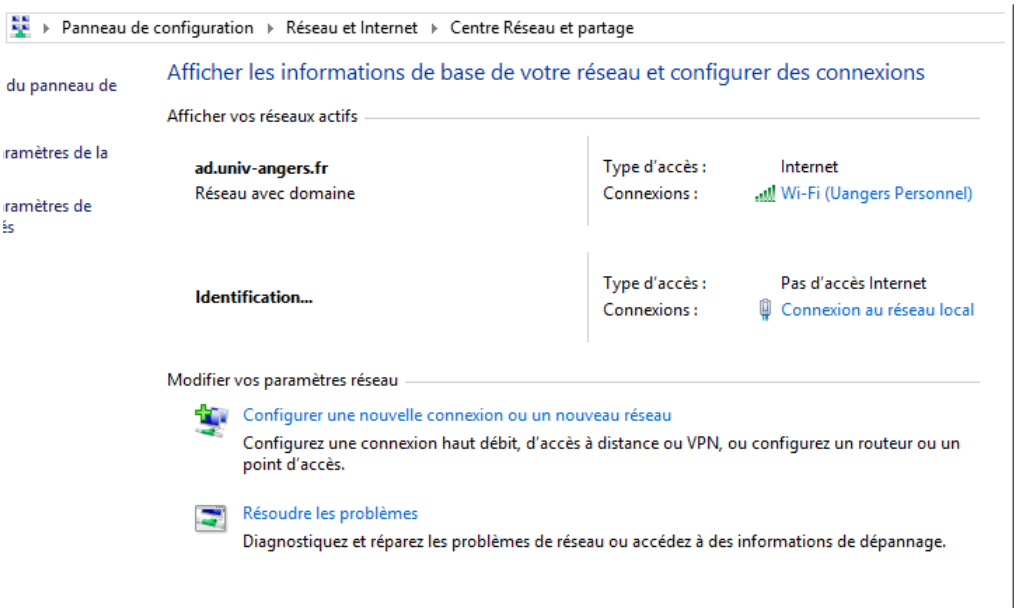


S'il n'y a qu'un Arduino, on pourra le relier au PC hôte par un simple câble RJ 45. S'il y en a plus d'un, le PC hôte et les Arduinos seront mis sur le même réseau par un mini-hub.

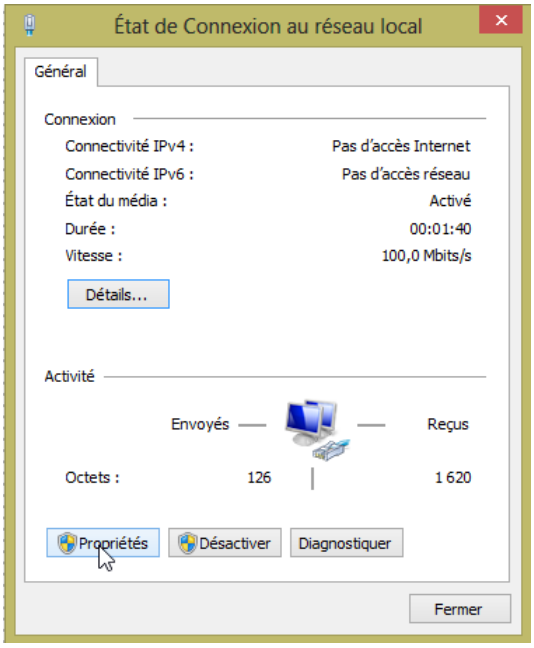
On mettra le PC hôte et les Arduinos sur le réseau privé **192.168.2.x**.

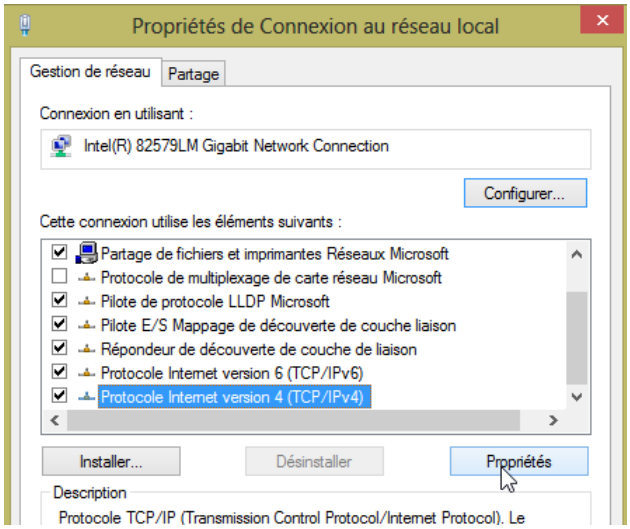
- l'adresse IP des Arduinos est fixée par le code source. Nous verrons comment ;
- l'adresse IP de l'ordinateur hôte pourra être fixée comme suit :

- prendre l'option [Panneau de configuration\Réseau et Internet\Centre Réseau et partage] :

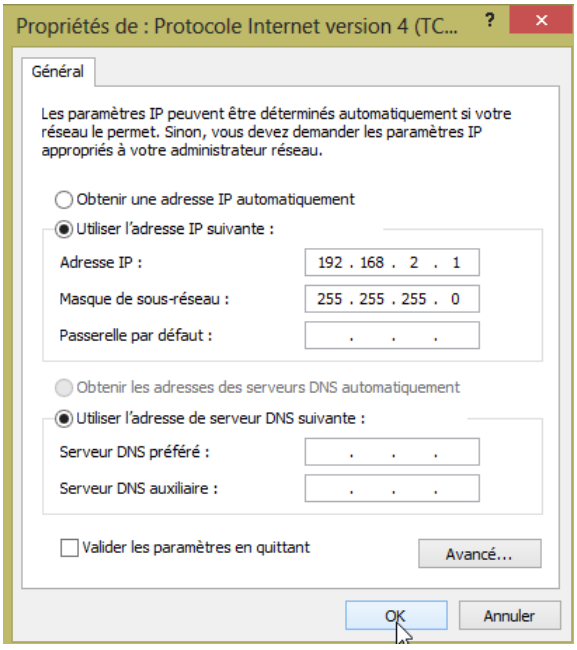


- en [1], suivre le lien [Connexion au réseau local]





- en [2], visualiser les propriétés de la connexion ;
- en [4], visualiser les propriétés IP v4 [3] de la connexion ;



- en [5], donner l'adresse IP [192.168.2.1] à l'ordinateur hôte ;
- en [6], donner le masque [255.255.255.0] au réseau ;
- valider le tout en [7].

13.5 Test d'une application réseau

Avec l'IDE, chargez l'exemple [Exemples / Ethernet / WebServer] :

```
1. /*
```

```
2.    Web Server
3.
4.   A simple web server that shows the value of the analog input pins.
5.   using an Arduino Wiznet Ethernet shield.
6.
7.   Circuit:
8.   * Ethernet shield attached to pins 10, 11, 12, 13
9.   * Analog inputs attached to pins A0 through A5 (optional)
10.
11.  created 18 Dec 2009
12.  by David A. Mellis
13.  modified 9 Apr 2012
14.  by Tom Igoe
15.
16.  */
17.
18. #include <SPI.h>
19. #include <Ethernet.h>
20.
21. // Enter a MAC address and IP address for your controller below.
22. // The IP address will be dependent on your local network:
23. byte mac[] = {
24.   0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
25. IPAddress ip(192,168,1, 177);
26.
27. // Initialize the Ethernet server library
28. // with the IP address and port you want to use
29. // (port 80 is default for HTTP):
30. EthernetServer server(80);
31.
32. void setup() {
33.  // Open serial communications and wait for port to open:
34.   Serial.begin(9600);
35.    while (!Serial) {
36.     ; // wait for serial port to connect. Needed for Leonardo only
37.   }
38.
39.
40.   // start the Ethernet connection and the server:
41.   Ethernet.begin(mac, ip);
42.   server.begin();
43.   Serial.print("server is at ");
44.   Serial.println(Ethernet.localIP());
45. }
46.
47.
48. void loop() {
49.   // listen for incoming clients
50.   EthernetClient client = server.available();
51.   if (client) {
52.     Serial.println("new client");
53.     // an http request ends with a blank line
54.     boolean currentLineIsBlank = true;
55.     while (client.connected()) {
56.       if (client.available()) {
57.         char c = client.read();
58.         Serial.write(c);
59.         // if you've gotten to the end of the line (received a newline
60.         // character) and the line is blank, the http request has ended,
61.         // so you can send a reply
62.         if (c == '\n' && currentLineIsBlank) {
63.           // send a standard http response header
64.           client.println("HTTP/1.1 200 OK");
```

```
65.           client.println("Content-Type: text/html");
66.           client.println("Connection: close");
67.           client.println();
68.           client.println("<!DOCTYPE HTML>");
69.           client.println("<html>");
70.                     // add a meta refresh tag, so the browser pulls again every 5 seconds:
71.           client.println("<meta http-equiv=\"refresh\" content=\"5\">");
72.           // output the value of each analog input pin
73.           for (int analogChannel = 0; analogChannel < 6; analogChannel++) {
74.             int sensorReading = analogRead(analogChannel);
75.             client.print("analog input ");
76.             client.print(analogChannel);
77.             client.print(" is ");
78.             client.print(sensorReading);
79.             client.println("<br />");
80.           }
81.           client.println("</html>");
82.           break;
83.         }
84.         if (c == '\n') {
85.           // you're starting a new line
86.           currentLineIsBlank = true;
87.         }
88.         else if (c != '\r') {
89.           // you've gotten a character on the current line
90.           currentLineIsBlank = false;
91.         }
92.       }
93.     }
94.     // give the web browser time to receive the data
95.     delay(1);
96.     // close the connection:
97.     client.stop();
98.     Serial.println("client disconnected");
99.   }
100.  }
```

Cette application crée un serveur web sur le port 80 (ligne 30) à l'adresse IP de la ligne 25. L'adresse MAC de la ligne 23 est l'adresse MAC indiquée sur la carte réseau de l'Arduino.

La fonction [setup] initialise le serveur web :

- ligne 34 : initialise le port série sur lequel l'application va faire des logs. Nous allons suivre ceux-ci ;
- ligne 41 : le noeud TCP-IP (IP, port) est initialisé ;
- ligne 42 : le serveur de la ligne 30 est lancé sur ce noeud réseau ;
- lignes 43-44 : on logue l'adresse IP du serveur web ;

La fonction [loop] implémente le serveur web :

- ligne 50 : si un client se connecte au serveur web *[server].available* rend ce client, sinon rend *null* ;
- ligne 51 : si le client n'est pas *null* ;
- ligne 55 : tant que le client est connecté ;
- ligne 56 : *[client].available* est vrai si le client a envoyé des caractères. Ceux-ci sont stockés dans un buffer. *[client].available* rend vrai tant que ce buffer n'est pas vide ;
- ligne 57 : on lit un caractère envoyé par le client ;
- ligne 58 : ce caractère est affiché en écho sur la console de logs ;
- ligne 62 : dans le protocole HTTP, le client et le serveur échangent des lignes de texte.
    - le client envoie une requête HTTP au serveur web en lui envoyant une série de lignes de texte terminées par une ligne vide,
    - le serveur répond alors au client en lui envoyant une réponse et en fermant la connexion ;

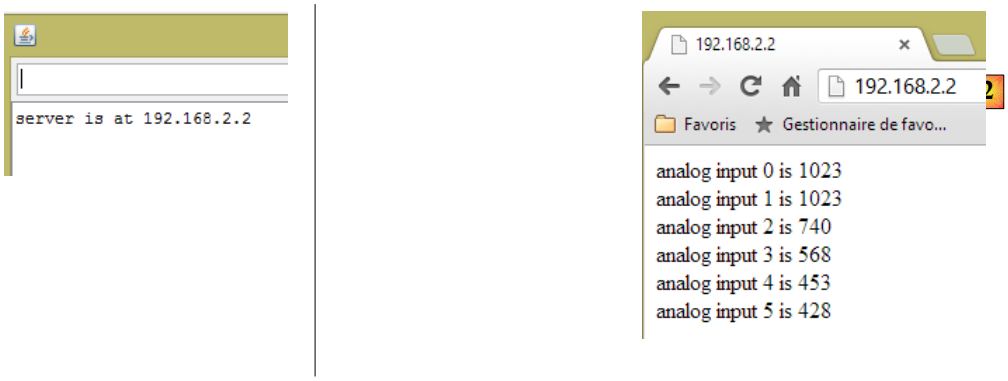
  Ligne 62, le serveur ne fait rien avec les entêtes HTTP qu'il reçoit du client. Il attend simplement la ligne vide : une ligne qui contient le seul caractère **\n** ;

- lignes 64-67 : le serveur envoie au client les lignes de texte standard du protocole HTTP. Elles se terminent par une ligne vide (ligne 67) ;
- à partir de la ligne 68, le serveur envoie un document à son client. Ce document est généré dynamiquement par le serveur et est au format HTML (lignes 68-69) ;
- ligne 71 : une ligne HTML particulière qui demande au navigateur client de rafraîchir la page toutes les 5 secondes. Ainsi le navigateur va demander la même page toutes les 5 secondes ;
- lignes 73-80 : le serveur envoie au navigateur client, les valeurs des 6 entrées analogiques de l'Arduino ;
- ligne 81 : le document HTML est fermé ;
- ligne 82 : on sort de la boucle *while* de la ligne 55 ;
- ligne 97 : la connexion avec le client est fermée ;
- ligne 98 : on logue l'événement dans la console de logs ;
- ligne 100 : on reboucle sur le début de la fonction **loop** : le serveur va se remettre à l'écoute des clients. Nous avons dit que le navigateur client allait redemander la même page toutes les 5 secondes. Le serveur sera là pour lui répondre de nouveau.

Modifiez le code en ligne 25 pour y mettre l'adresse IP de votre Arduino, par exemple :

```
IPAddress ip(192,168,2,2);
```

Téléversez le programme sur l'Arduino. Lancez la console de logs (Ctrl-M) (M majuscule) :



- en [1], la console de logs. Le serveur a été lancé ;
- en [2], avec un navigateur, on demande l'adresse IP de l'Arduino. Ici [192.168.2.2] est traduit par défaut comme [http://198.162.2.2:80];
- en [3], les informations envoyées par le serveur web. Si on visualise le code source de la page du navigateur, on obtient :

```
1.  <!DOCTYPE HTML>
2.  <html>
3.  <meta http-equiv="refresh" content="5">
4.  analog input 0 is 1023<br />
5.  analog input 1 is 1023<br />
6.  analog input 2 is 727<br />
7.  analog input 3 is 543<br />
8.  analog input 4 is 395<br />
9.  analog input 5 is 310<br />
10. </html>
```

On reconnaît là, les lignes de texte envoyées par le serveur. Du côté Arduino, la console de logs affiche ce que le client lui envoie :

```
1.  new client
2.  GET /favicon.ico HTTP/1.1
3.  Host: 192.168.2.2
4.  Connection: keep-alive
5.  Accept: */*
6.  User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.22 (KHTML, like Gecko)
    Chrome/25.0.1364.172 Safari/537.22
7.  Accept-Encoding: gzip,deflate,sdch
8.  Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.6,en;q=0.4
9.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

```
10.
11. client disconnected
```

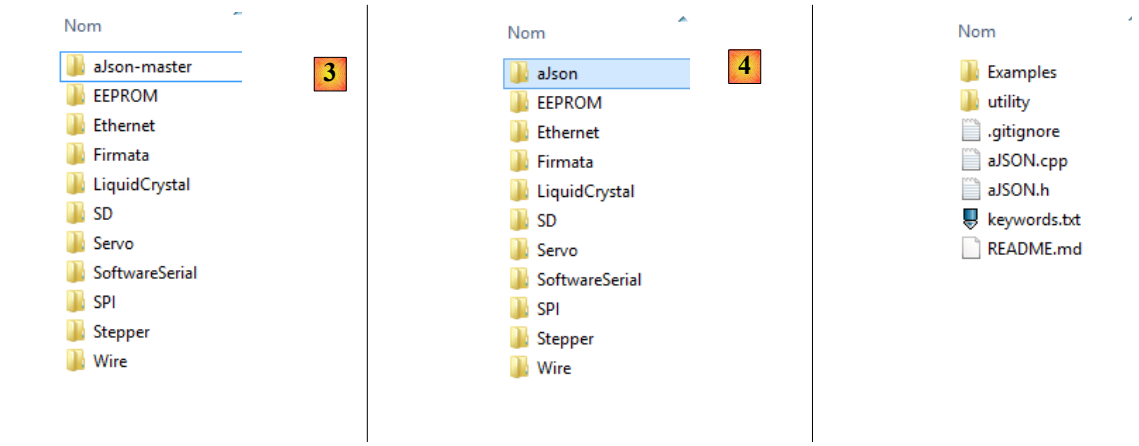
- lignes 2-9 : des entêtes HTTP standard ;
- ligne 10 : la ligne vide qui les termine.

Etudiez bien cet exemple. Il donne des indications qui vous seront utiles pour construire un serveur Arduino pour le TP.

13.6 La bibliothèque aJson

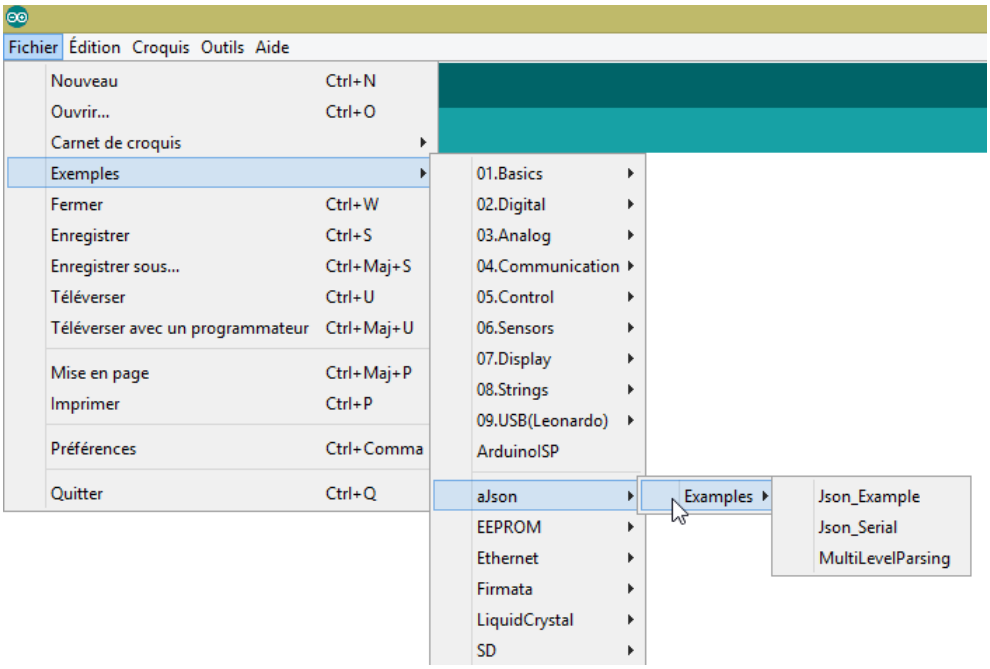
Dans le TP à écrire, les Arduinos échangent des lignes de texte au format JSON avec leurs clients. Par défaut, l'IDE Arduino n'inclut pas de bibliothèque pour gérer le JSON. Nous allons installer la bibliothèque **aJson** disponible à l'URL [https://github.com/interactive-matter/aJson].

- en [1], téléchargez la version zippée du dépôt Github ;
- décompressez le dossier et copiez le dossier [aJson-master] [3] dans le dossier [<arduino>/libraries] [3] où <arduino> est le dossier d'installation de l'IDE Arduino ;



- en [4], renommez ce dossier [aJson] ;
- en [5], son contenu.

Maintenant, lancez l'IDE Arduino :



Vérifiez que dans les exemples, vous avez désormais des exemples pour la bibliothèque **aJson**. Exécutez et étudiez ces exemples. Vous en aurez besoin.

Table des matières