Architecture de l'application
et
Manuel de réutilisation pour les programmeurs

Architecture technique

Présentation des concepts

Nous voulons faire une simulation de ifttt. Pour cela, nous utilisons deux niveaux d'abstraction en fonction desquels nous présentons les concepts utilisés par la suite.

Le niveau des modèles

Ce niveau est le plus abstrait et définit des archétypes dont on donnera en quelque sorte des réalisations dans le niveau inférieur.

Canal - Acteur

Un canal (models.Channel dans le code) est une catégorie d'éléments de l'internet des objets : ce peut être une classe d'objets physiques ou immatériels comme un capteur, une lampe, le compte d'un service en ligne, un site de news. Comme dans ifttt, des canaux ont des signaux (triggers en anglais et models.Trigger dans le code) et des actions.

Signal - Message

Les signaux sont levés par le canal pour signaler un changement : ce peut-être un changement d'environnement (l'ouverture d'une porte, la détection d'un mouvement) ou un changement interne (un délai expire, une date échoit). Sémantiquement, un signal mentionne généralement un objet et le nouvel état de ce dernier ou ce qui a provoqué ce changement d'état. Il peut mentionner des modalités (dans le cas d'une porte, on peut mentionner son degré d'ouverture).

Recette

Toujours de la même manière que ifttt, des recettes (models.Recipe) peuvent être créées pour matérialiser des réactions automatiques à un changement. Le slogan *if this then that* montre bien que si un événement arrive, alors un canal va agir d'une certaine manière : une recette matérialise une relation de causalité entre deux acteurs. Outre son nom, son propriétaire et d'autres attributs, elle contient principalement quatre membres : un canal émetteur, un signal émis, un canal récepteur et une action ; le canal émetteur lève un de ses signaux et à cause de cela, le canal récepteur accomplit l'action précisée dans la recette. Contrairement à certains de ses concurrents, une recette de ifttt ne peut lier plus d'un canal émetteur, un signal émis, un canal récepteur et une action : c'est atomique.

Le niveau des acteurs

Le niveau des acteurs est en quelque sorte une représentation concrète du monde. Il contient des acteurs qui évoluent indépendamment et communiquent par message. La communication entre ces acteurs suit le formalisme indiqué plus haut.

Relation avec un canal

Le mode d'action d'un acteur est définit par un canal. Un acteur peut donc être vu comme une instance d'un canal.

Message

Les messages échangés par des acteurs ont la sémantique soit d'un signal, soit d'une action. De la même manière qu'une lettre à la Poste, ils peuvent être envoyés anonymement mais ont forcément un destinataire. Les acteurs ne communiquent que par message. Les messages peuvent contenir des modalités qui précisent le changement décrit par le signal ou l'action à effectuer.

Relation de causalité – Recette Akka

A ce niveau concret, une recette est définie comme relation de causalité entre deux acteurs et « if this then that » devient : quand un acteur envoit un message qui contient un signal, alors un autre acteur reçoit un message qui lui dit d'accomplir telle action. Dans le monde physique, un capteur ne sait faire qu'une chose : lever des signaux. Pour respecter la simplicité des objets physiques, un acteur est définit (actors.ActorRouter dans le code) pour permettre aux acteurs de lever des signaux simplement en envoyant un message à icelui. C'est formellement un objet, pas un acteur. Les recettes sont diluées dans ce pseudo-acteur : à réception d'un message signal d'un acteur, le pseudo-acteur regarde si une relation de causalité existe. Le cas échéant, il envoit anonymement un message d'action à l'acteur définit.

ActorRouter

Cet acteur est utile pour envoyer des messages signaux et simuler des changements. Il joue le rôle de routeur, qui reçoit les messages signaux, trouve les messages d'actions correspondants, et envoie cette action à l'acteur désiré. Chaque groupe d'utilisateurs a un ActorRouter pour bien simuler le concept de maison (une maison a un système des gestion). Au lancement de l'application, un algorithme parcourt tous les groupes d'utilisateurs présents dans la base de données, et crée un ActorRouter pour chacun (SystemController.java), et lie ces acteurs au groupe d'utilisateurs par une HashMap.

Relation de génération entre les deux niveaux

Nous définissons plus haut les acteurs par rapport aux canaux. Cette démarche est cependant erronnée à cause de akka qui nous force en quelque sorte à définir programmatiquement les acteurs : leur définition n'est donc pas chargée en base mais codée « en dur » et iceux ne sont donc pas déduits des canaux comme le présentaient les explications précédentes.

Loin de poser problème, il suffit de considérer que les acteurs ont tout de même une intelligence qui échappe aux canaux et correspondent à des objets physiques qui ne sont pas facilement modifiables. Le paradigme d'ifttt est d'en abstraire des catégories en créant un canal qui regroupe des acteurs tandis que celui de l'utilisateur est de redescendre du canal vers son objet physique. La dichotomie de ces deux mouvements de pensée (induction d'ifttt et déduction de l'utilisateur)

renvoie à un phénomène déjà existant dans ifttt : il est donc naturel que nous le rencontrions également.
En conclusion, les éléments des deux niveaux doivent être définis en étroite accointance.

Manuel de réutilisation

Initialisation des données

Quand l'application est lancée, la première classe exécutée est la classe Global.java (attention : cette classe doit rester dans le default package pour être exécutée). Dans cette classe, il y a des appels directs à DatabaseEngine.java pour nettoyer la base de données et la peupler (c.f. paragraphe suivant pour savoir comment ajouter des utilisateurs, des channels…). Ensuite, il y a un appel à la méthode createActorRouterMap() de SystemController.java qui crée les acteurs qui joueront le rôle de routeur pour chaque groupe d'utilisateurs (un acteur routeur pour un groupe : bijection).
export() de Script.java permet de générer le graphe de relations entre les acteurs.

Création des données

Toute la gestion des données à l'initialisation est faite dans la classe DatabaseEngine.java. deleteDB() permet de supprimer toutes les données de la base et populateDB() permet de créer les données.
Donc la création d'utilisateurs ou de channels supplémentaires sera faite dans DatabaseEngine.populateDB() (c.f. code pour savoir comment créer les utilisateurs).
Pour la création d'un nouveau channel, il faut créer une nouvelle instance de la classe Channel. Ensuite il faut créer les triggers et les actions de ce canal (s'il y en a), et de lier les triggers ou actions à l'objet channel créé.
Important : Chaque channel doit avoir son acteur correspondant **ET** chaque trigger ou action doit avoir son message correspondant.
Donc quand on crée un channel (ex : Lamp) il faut créer une classe qui extend le type UntypedActor dans la class AllActors.java : le nom de l'acteur créé doit avoir le même nom du channel + Actor (ex : LampActor). S'il y a des espaces (ex : temperature detector), l'acteur a comme nom TemperatureDetectorActor. Il est très important de bien faire attention au noms.
La méthode onReceive(Object message) est la méthode la plus importante de la classe, puisque c'est elle qui reçoit le message envoyer à l'acteur
Idem pour les triggers (actions par analogie). Une fois le trigger est créé, il faut créer un message qui extend la classe MessageEnvelope et qui implémente Serializable.

Utilisation

Toutes les pages html se trouvent dans le package views. Les annotations « bizarres » dans html et javascript (ex : @for(log <- logs){…}) sont du scala.
La class Scheduler.java permet de lancer les triggers à….