

# Rapport de stage de première année : Conception et développement d'un agent conversationnel

10 septembre 2017

Hugo Belhomme

Encadré par Jacky Casas, doctorant



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Cadre . . . . .	1
1.2	Sujet : Conception et développement d'un agent conversationnel . . . . .	1
<b>2</b>	<b>Développement</b>	<b>3</b>
2.1	Analyse . . . . .	3
2.1.1	Choix technologiques possibles . . . . .	3
2.1.2	Technologies utilisées . . . . .	3
2.1.3	Présentation des technologies choisies . . . . .	3
2.2	Conception . . . . .	6
2.2.1	Architecture du chatbot . . . . .	6
2.2.2	Conversation . . . . .	6
2.2.3	Fonctionnalités . . . . .	8
2.3	Implémentation . . . . .	10
2.3.1	Conversation . . . . .	10
2.3.2	Demande d'informations utilisateur . . . . .	11
2.3.3	Envoi de rappels . . . . .	12
2.3.4	Reconnaissance des repas . . . . .	12
2.3.5	Récapitulatif . . . . .	13
2.4	Tests et Validation . . . . .	16
<b>3</b>	<b>Bilan</b>	<b>22</b>
3.1	Respect du cahier des charges . . . . .	22
3.2	Problèmes rencontrés . . . . .	22
3.3	Idées d'amélioration et limites du <i>proof-of-concept</i> . . . . .	23
3.4	Conclusion . . . . .	24
<b>4</b>	<b>Bibliographie</b>	<b>25</b>
<b>5</b>	<b>Annexes</b>	<b>26</b>

### **Remerciements**

Je souhaite remercier Jacky Casas pour m'avoir accueilli au sein de HumanTech et pour ses remarques et conseils toujours pertinents. Je souhaite remercier l'institut HumanTech pour m'avoir trouvé un logement, chose peu aisée à l'étranger et qui a rendu ce stage possible.

# Chapitre 1

## Introduction

### 1.1 Cadre

Mon stage s'est déroulé au sein de l'institut de recherche HumanTech, un institut de recherche informatique hébergé par la Haute École d'Ingénierie et d'Architecture de Fribourg (HEIA-FR). L'institut est principalement composé de professeurs, doctorants et post-doctorants de l'école. Les missions de HumanTech sont principalement tournées vers l'accessibilisation du numérique et la santé. Les projets de recherche sont majoritairement financés par la Confédération (l'État suisse).

J'ai travaillé dans un bureau d'une dizaine d'autres postes, avec d'autres stagiaires. Le fonctionnement de l'institut fait que j'ai réalisé une présentation initiale et une présentation finale du projet devant les responsables de l'institut et les autres stagiaires.

### 1.2 Sujet : Conception et développement d'un agent conversationnel

Un chatbot est un programme informatique capable de converser avec un utilisateur. Les chatbots sont utilisés par des organisations pour gérer des conversations le plus souvent simples et limitées à un domaine précis. C'est un moyen pour les marques notamment de répondre automatiquement aux questions des utilisateurs (support, service après-vente). Ces bots peuvent œuvrer en tant qu'assistant pour différentes tâches (prendre des rendez vous, automatisations diverses, rappels). Les magasins en ligne s'en servent pour permettre aux gens de faire des commandes comme si on discutait avec une personne. Les sites de news s'en servent pour diffuser une synthèse des articles du jour.

Le but de ce stage était de concevoir puis développer un chatbot qui joue le rôle d'un compagnon nutritionnel. Le bot doit récolter des données sur le contenu des repas des utilisateurs (sous forme de texte dans un premier temps et d'image par la suite) et les stocker pour ensuite en retirer de la valeur pour l'utilisateur.

Le cahier des charges dans son intégralité est disponible en annexe.

Le but du stage était de rendre une *proof-of-concept*, en voici les points clés :

#### 1. Conversation avec l'utilisateur

Le chatbot devra être capable de converser simplement avec l'utilisateur dans des scénarios très simples et pour lui demander des informations précises. Il devra notamment toujours être capable de revenir au menu ou d'accéder à une aide lorsque nécessaire.

#### 2. Enregistrement et stockage des conversations

Le bot devra enregistrer les éléments clés de la conversation pour pouvoir proposer un suivi à l'utilisateur. Il peut aussi enregistrer les photos que l'utilisateur lui envoie.

#### 3. Envoi de rappels

Le bot devra pouvoir envoyer des rappels à l'utilisateur pour que celui-ci lui décrive son repas. Ces rappels doivent pouvoir être configurables par l'utilisateur.

4. **Reconnaissance des repas**

Le bot devra être capable de reconnaître ce que l'utilisateur a mangé, à partir de messages texte puis à partir d'images.

5. **Récapitulatif**

Le bot devra être capable de générer un récapitulatif de l'alimentation de l'utilisateur sous une forme visuellement agréable.

6. **Fluidité de l'expérience**

Le chatbot devra utiliser différents moyens de sorte à ce que la conversation soit intuitive et fluide : questions précises, boutons, cartes de contenu, etc.

## Chapitre 2

# Développement

### 2.1 Analyse

#### 2.1.1 Choix technologiques possibles

**Conversation :** Pour développer un chatbot performant, deux solutions sont possibles : *utiliser le framework de Microsoft* ou *développer sa propre solution*. Le framework permet à la fois de ne pas avoir à gérer les entrées et sorties et de pouvoir tester rapidement et simplement son bot en local (lien vers la procédure en annexe), là où tout développer soi-même a l'avantage d'avoir un contrôle total sur les données transmises et de pouvoir avoir une solution parfaitement adaptée.

**Compréhension du langage :** Le but d'un chatbot n'est pas de comprendre ce que l'utilisateur dit, mais de comprendre ce que l'utilisateur veut. Pour cela, il faut ou *limiter le vocabulaire de l'utilisateur* (c'est le cas de la majorité des logiciels actuels : l'utilisateur clique sur des boutons) ou *se rendre capable d'analyser les messages de l'utilisateur* et d'en ressortir son intention. Pour la dernière alternative, de nombreux services sont utilisables et permettent de ne pas avoir à développer un système d'apprentissage.

**Stockage des données :** Puisque le but du stage est de fournir un *proof-of-concept*, le stockage des données peut s'effectuer de différentes manières :

1. avec des fichiers textes.
2. avec une base de données relationnelle.
3. avec un autre type de bases de données.

#### 2.1.2 Technologies utilisées

**Conversation :** Le but du stage étant de concevoir et développer un chatbot et non un système de messagerie, j'ai choisi d'utiliser le framework Bot Builder de Microsoft.

**Compréhension du langage :** De même, pour pouvoir me concentrer sur le chatbot j'ai choisi Luis.ai pour gérer l'analyse des messages de l'utilisateur. Luis.ai étant détenu par Microsoft son utilisation avec le framework choisi est facilitée.

**Stockage des données :** Les objets utilisés par le framework et Luis.ai sont proches du JSON, mon maître de stage m'a donc suggéré d'utiliser MongoDB, un système de gestion de bases de données orienté documents qui manipule des objets en BSON (Binary JSON).

#### 2.1.3 Présentation des technologies choisies

##### Microsoft Bot Framework

Le SDK (Software Development Kit) BotBuilder est open source et disponible en .NET et en Node.js. Je me suis tourné vers sa version Node.js pour une question d'affinité avec les langages.

Le SDK permet de ne pas avoir à gérer les entrées / sorties du bot et de se concentrer sur la partie conversationnelle à proprement parler. BotBuilder fonctionne sur un système de pile de dialogues : le dialogue au sommet de la pile reçoit les messages et les traite jusqu'à qu'il se termine ou qu'il soit remplacé.

Pour gérer la pile de dialogue, les fonctions `beginDialog`, `replaceDialog`, `endDialog` et `endDialogWithResult` sont mises à disposition :

**`beginDialog`** permet de lancer un dialogue. Une fois celui-ci terminé, le dialogue "parent" sera repris à l'endroit où il avait été interrompu.

**`replaceDialog`** permet de quitter le dialogue courant et de ne pas y revenir à la fin du dialogue appelé.

**`endDialog`** permet de revenir au dialogue parent.

**`endDialogWithResult`** permet de revenir au dialogue parent en passant un résultat.

Dans l'exemple qui suit, sont présentés 3 dialogues : `('/')`, `'showReminders'` et `'createReminder'`. Les deux derniers sont déclenchés par le premier en fonction de la réponse de l'utilisateur à la question.

#### Extrait de code 2.1 – Exemples de dialogues

```
1 lib.dialog('/', [
2   function (session) {
3     builder.Prompts.choice(session, "What_do_you_want_to_do?",
4                           'See_my_reminders|Set_a_new_reminder'
5                           );
6   },
7   function (session, results) {
8     if (results.response) {
9       switch (results.response.index) {
10        case 0:
11          session.replaceDialog('showReminders');
12          break;
13        case 1:
14          session.replaceDialog('createReminder');
15          break;
16        default:
17          session.replaceDialog('default:/');
18          break;
19      }
20    }
21  }
22 ])
23
24 lib.dialog('showReminders', [
25   function (session) {
26     session.send("Here_are_all_your_reminders_");
27     session.send("...");
28     session.endDialog("Just_kidding,_it's_not_finished.");
29   }
30 ])
31
32 lib.dialog('createReminder', [
33   /* [...] */
34 ])
```

Il est possible de faire des boucles avec des dialogues, un exemple sera détaillé dans la partie **Implémentation**.

Chaque session (comprendre *discussion*) comporte des informations qui lui sont propres et sont accessibles en tout point par le bot. Notamment une partie `session.userData` et une partie `session.dialogData` qui permettent de stocker simplement des informations brèves.

## Luis.ai

Luis (Language Understanding Intelligent Service) est une application dite de NLU (Natural Language Understanding). Ces applications sont capables de s'entraîner à reconnaître les *intents* (ou intentions) d'un message ainsi que ses *entities* (ou paramètres). Pour la faire fonctionner, il faut définir des *intents* et des *entities* sur le site de Luis.ai puis rentrer des exemples de messages et indiquer à quoi ils correspondent.

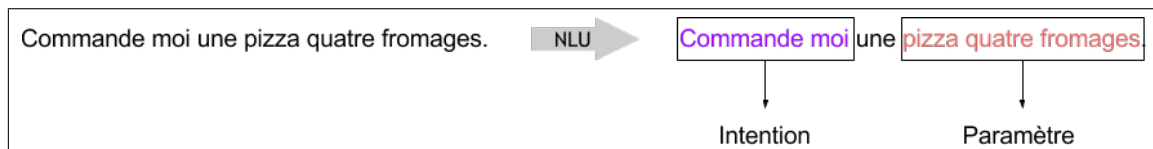


Schéma 2.1 – Utilisation d'une application de NLU.

## MongoDB

MongoDB est un système de gestion de base de données orienté document. Les documents sont stockés sous forme de JSON Binaire. Le BSON est un format binaire dans lequel zéro ou plusieurs couples de clé-valeur sont stockés dans une seule entité, appelée document. Il s'agit d'un système NoSQL, dans le sens où l'on est loin des requêtes SQL : les requêtes se font dans un langage plus proche de ce qui se fait dans l'informatique.

### Extrait de code 2.2 – Exemple d'enregistrement de document avec MongoDB

```
1 processForm = function (resForm, id) {
2   MongoClient.connect(url, function(err, db) {
3     if (err) throw err;
4     resForm.userId = id;
5     db.collection("users").insertOne(resForm, function (err, res) {
6       if (err) throw err;
7       console.log("This_user_has_been_added_to_the_database_");
8       console.log(res.ops);
9     })
10  })
11  };
```

### Extrait de code 2.3 – Exemple de requête avec MongoDB

```
1 // Fowards the meals to the given dialog
2 lib.dialog('getMeals', [
3   function (session, nextDialog) {
4     var query = {userId: session.userData.userId};
5     db.collection('meals').find(query).toArray(function (err, res) {
6       if (err) throw err;
7       if (res.length === 0) {
8         session.send("...I couldn't find any meal at your name")
9         session.endDialog();
10      } else {
11        session.replaceDialog(nextDialog, res);
12      }
13    })
14  }
15  ])
```



## 2.2 Conception

### 2.2.1 Architecture du chatbot

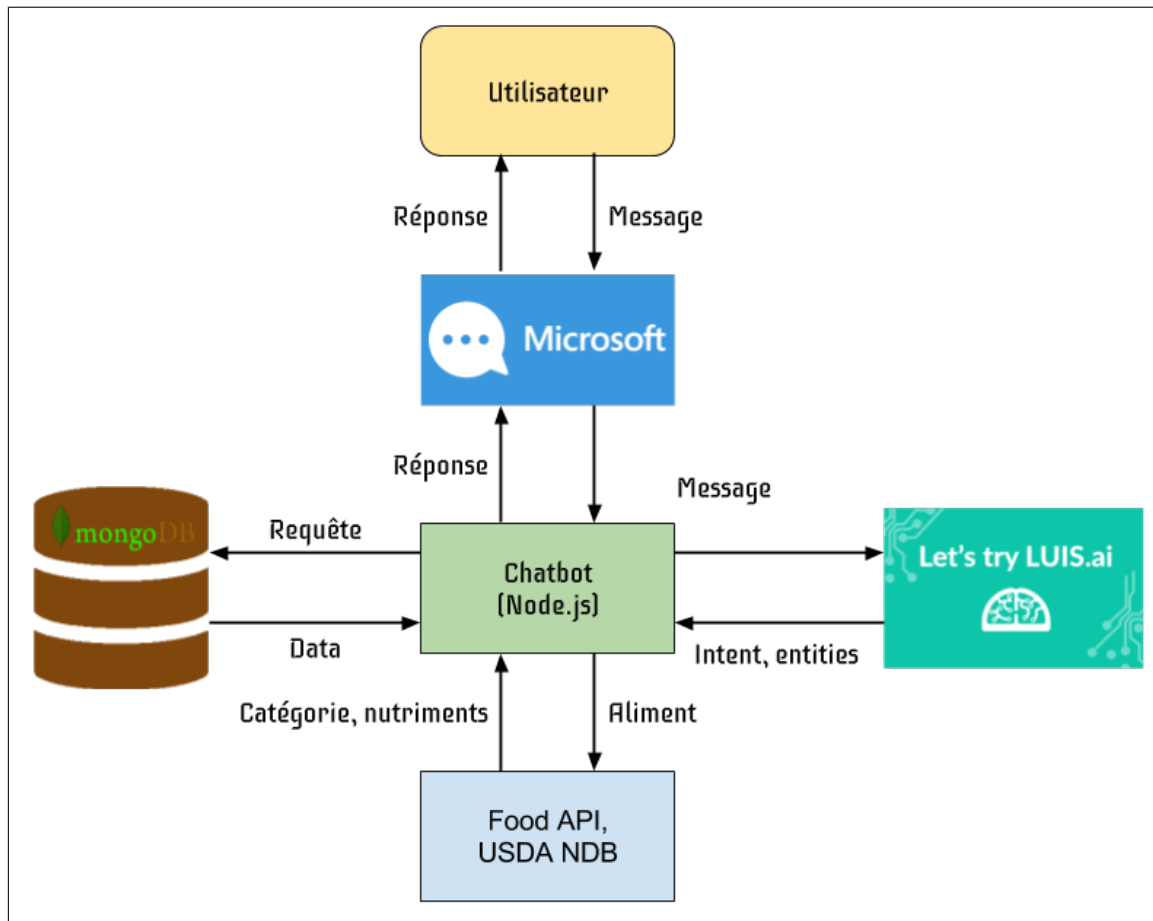


Schéma 2.2 – Architecture du chatbot.

Le connecteur de Microsoft se charge de transmettre le message de l'utilisateur à notre programme, qui le fait passer par Luis.ai pour en obtenir les *intents* et les *entities*. À partir de ces informations, le chatbot peut selon les cas poser une question à l'utilisateur, faire une recherche dans la base de données nutritionnelles ou faire une requête en base de données pour enregistrer ou aller chercher des informations.

### 2.2.2 Conversation

La conversation est le coeur du chatbot ; cela représente l'ensemble des dialogues que le bot sera en mesure d'avoir et les liens entre ces dialogues.

Le premier jet simpliste du schéma conversationnel du chatbot suit. Cette première approche contient :

- un dialogue par défaut vers lequel tous les messages sont dirigés et qui renvoie toujours au menu.
- un menu qui propose 3 choix vers des dialogues simples qui est déclenché dès que l'utilisateur écrit "menu" ou "help" dans son message.
- 3 dialogues basiques qui répondent à des besoins simples.

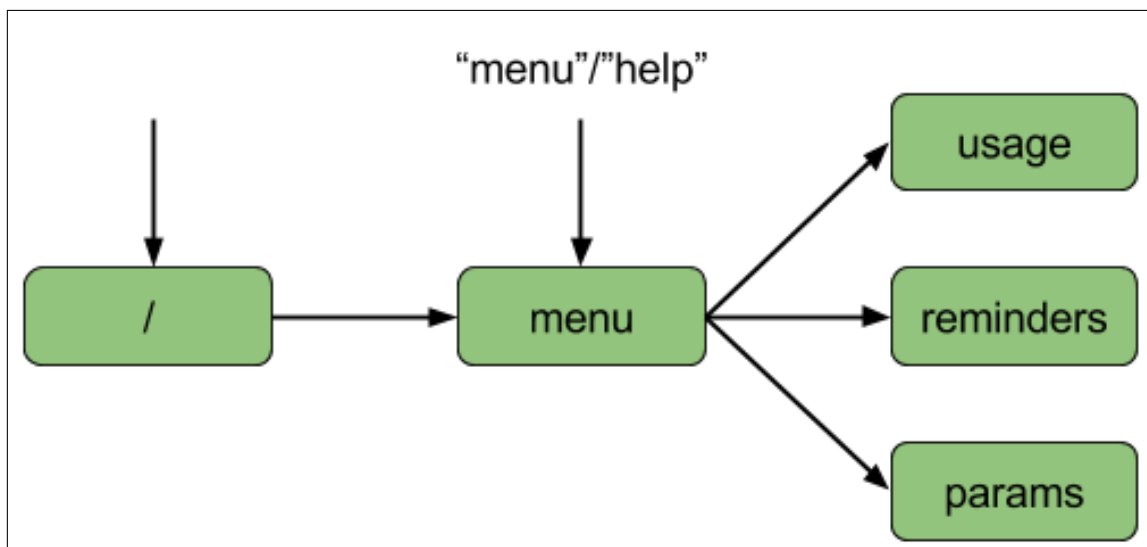


Schéma 2.3 – Premier jet simpliste du schéma conversationnel du chatbot.

Le schéma suivant représente le schéma conversationnel final du chatbot, il n'est pas exhaustif mais tous les dialogues majeurs sont représentés ainsi que leurs connections. Les dialogues en vert ne sont accessibles qu'à partir de leurs parents. Les dialogues en bleu sont accessibles en tout point si l'intention correspondante est repérée par notre application de NLU. Ce point est expliqué un peu plus en détails juste après ce schéma.

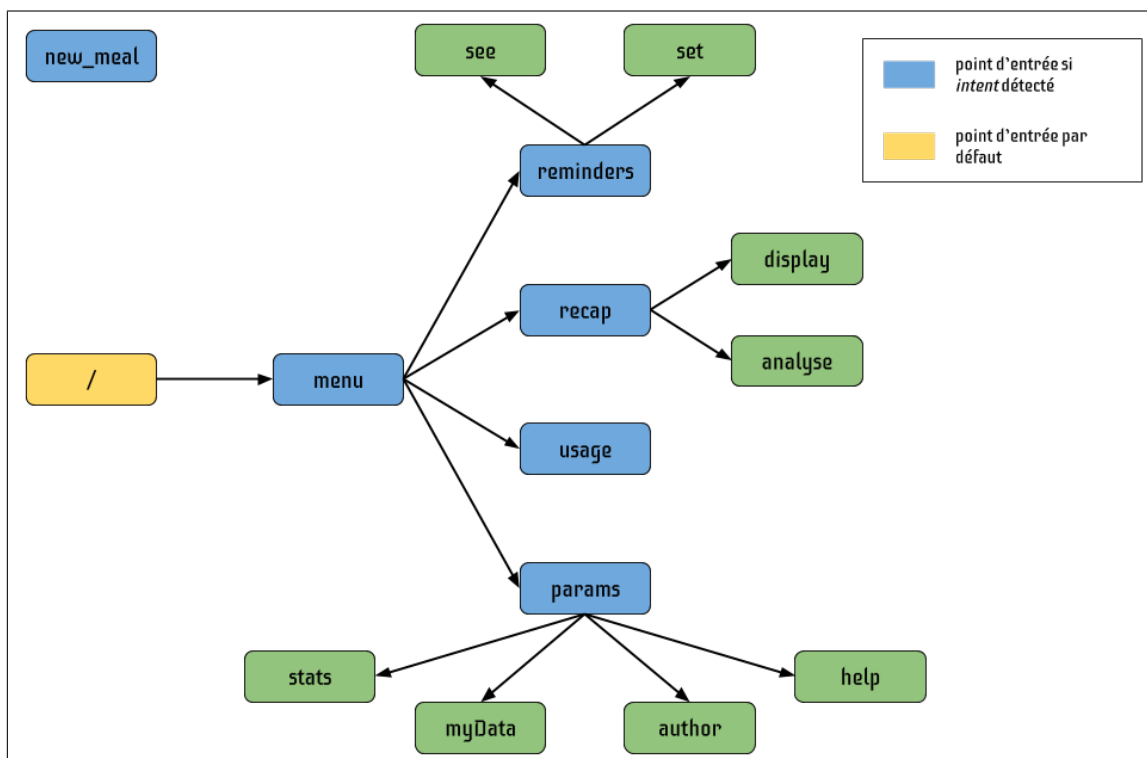


Schéma 2.4 – Schéma conversationnel final simplifié.

Tous les messages de l'utilisateur sont traités par Luis.ai qui retourne au bot les intentions et les éventuels paramètres qu'il a reconnu. (cf. exemple qui suit) Le chatbot contrôle alors la valeur associée à l'intention reconnue et juge si il interrompt le dialogue en cours pour lancer le nouveau dialogue ou non.

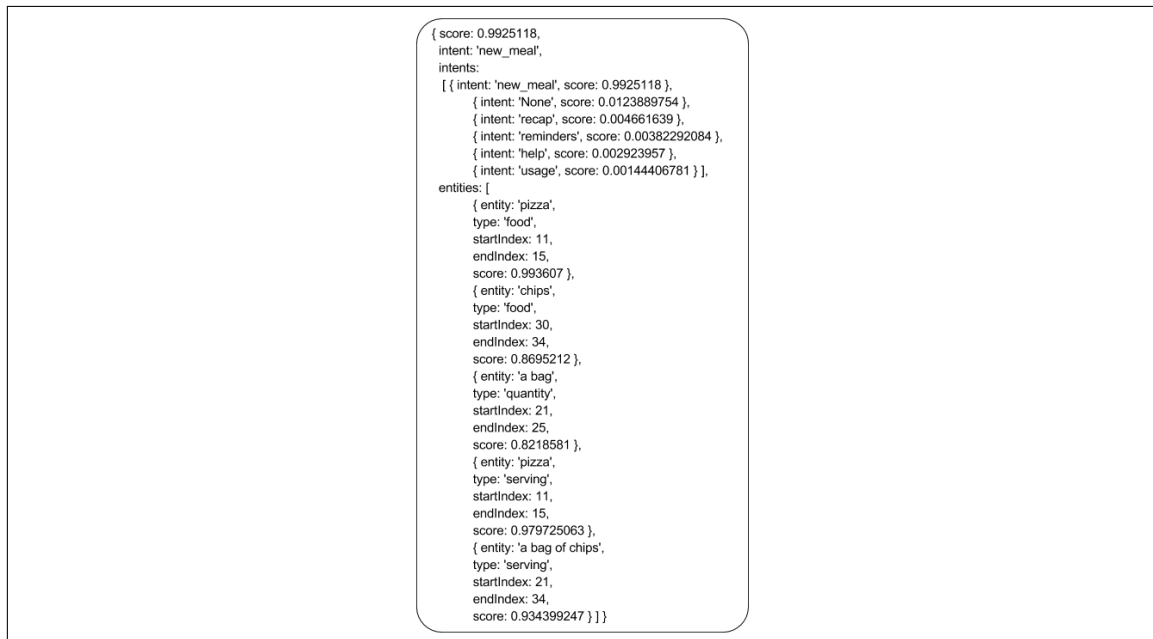


Schéma 2.5 – Retour de Luis.ai au message "I just ate pizza and a bag of chips"

### 2.2.3 Fonctionnalités

#### Demande d'informations utilisateur

Afin de pouvoir donner des conseils pertinents, un certain nombre de renseignements peuvent être utiles. Un dialogue propose donc à l'utilisateur de remplir un formulaire simple qui contient son prénom, son âge, sa taille et sa masse. Le remplissage de ce formulaire se fait au terme d'une série de questions/réponses.

En outre, un identifiant unique est attribué à chaque utilisateur après réception de son premier message.

#### Enregistrement des messages

Le framework comporte une sorte de middleware par lequel chacun des messages passe avant d'être envoyé à un dialogue, il est donc aisé de stocker les messages à cet instant.

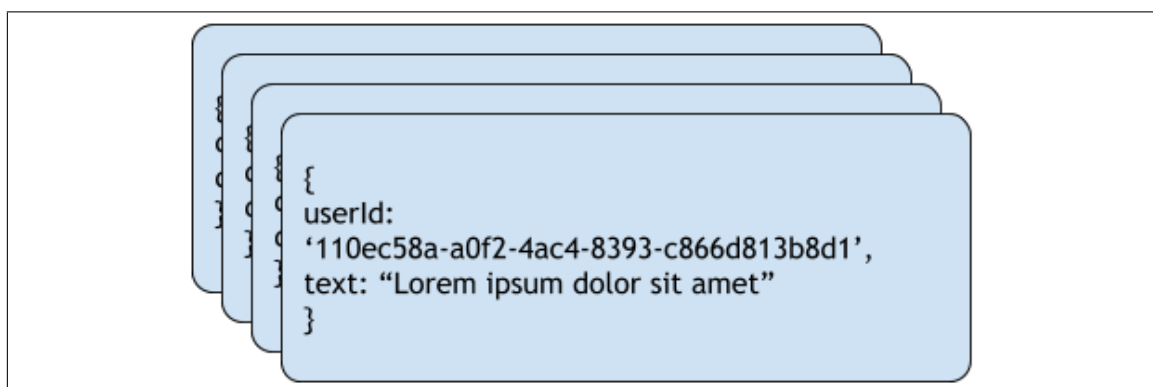


Schéma 2.6 – Représentation d'un message stocké en base de données.

#### Envoi de rappels

Un dialogue permet à l'utilisateur d'enregistrer un rappel. L'utilisateur configure un message, une heure/date et indique si il souhaite que le rappel soit quotidien. Le rappel est ensuite enregistré en base de données avec l'adresse utilisée par le connecteur de Microsoft pour envoyer des messages à l'utilisateur. La façon dont les rappels sont envoyés est expliquée dans la partie **Implémentation**.

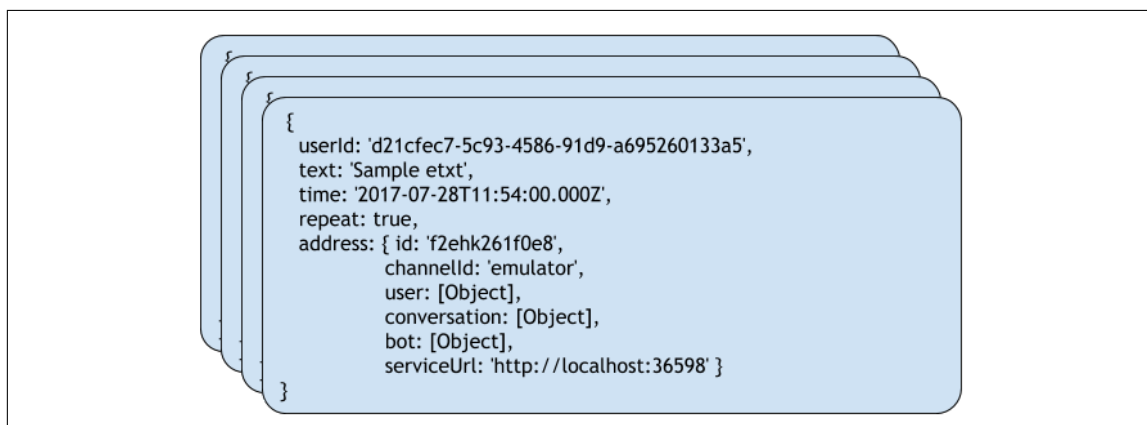


Schéma 2.7 – Représentation d'un rappel stocké en base de données.

## Reconnaissance des repas

Lorsque Luis.ai comprend que l'utilisateur souhaite enregistrer un nouveau repas, le dialogue 'new\_meal' est démarré avec le message et les *entities* repérées par Luis. Ces *entities* permettent d'amorcer le traitement illustré ci-dessous. Dans un premier temps, les repas du jour courant sont stockés dans la session de l'utilisateur. La journée de repas est ajoutée plus tard en base de données. Dans l'exemple qui suit l'utilisateur n'a indiqué qu'un diner pour sa journée.

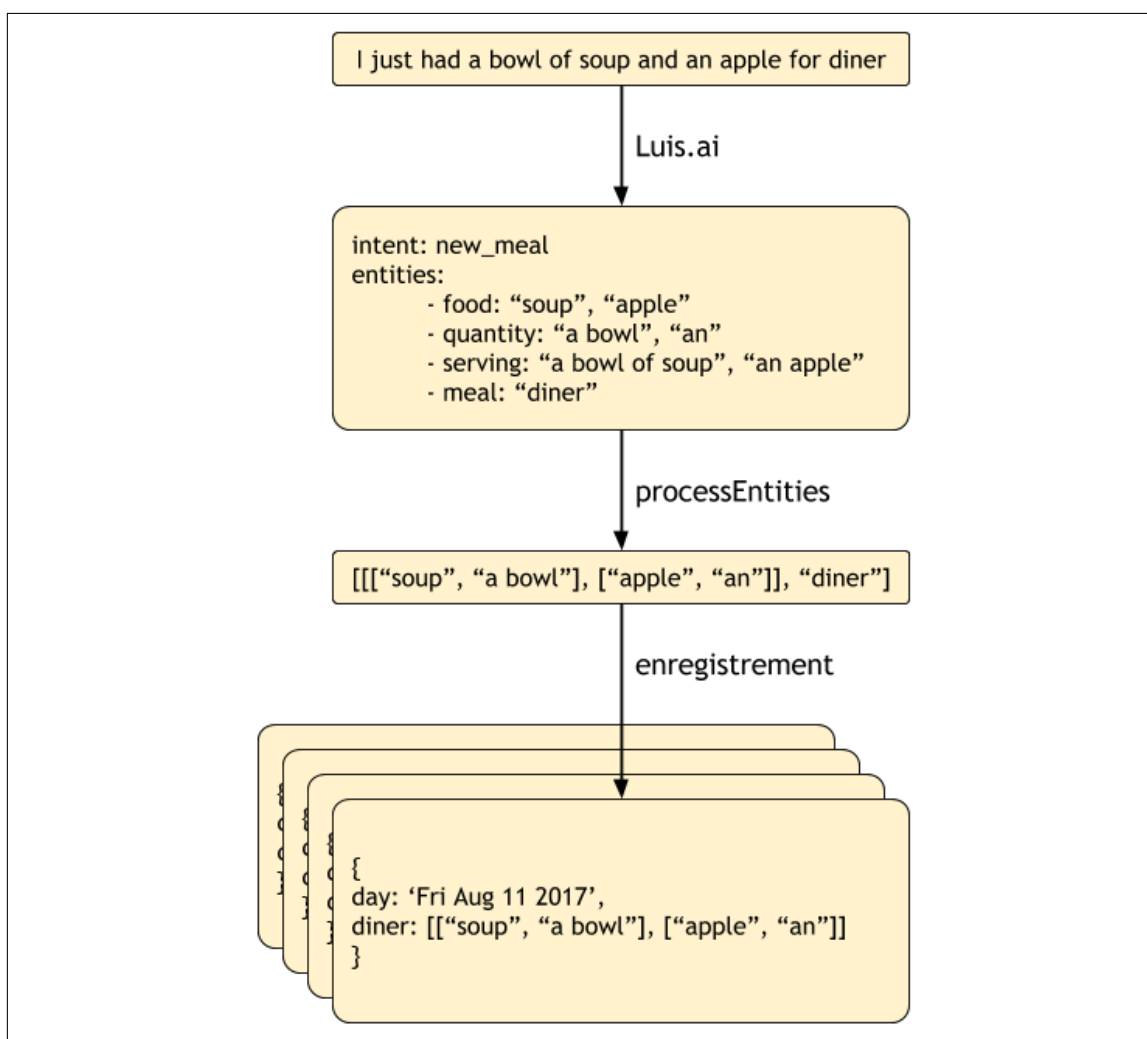


Schéma 2.8 – Procédé menant à l'enregistrement d'un repas.

## Récapitulatif

L'utilisateur peut déclencher le dialogue 'recap' pour que le bot lui fournisse un récapitulatif de ses repas, deux choix lui sont présentés : afficher ses repas ou calculer les apports nutritionnels d'une de ses journées.

L'affichage se réalise simplement à partir de la concaténation du résultat d'une requête en base de données et du jour en cours stocké dans la session.

J'ai choisi d'effectuer la recherche dans la base de données nutritionnelles au moment de la demande de récapitulatif et non au moment du stockage.

Les informations nutritionnelles sont fournies par une base de données gouvernementale Suisse qui a été téléchargée, convertie en fichier tsv (Tab Separated Values) puis retournée pour ne conserver qu'une partie des informations mises à disposition. Le tableau ainsi obtenu contient une ligne par aliment répertorié dans la banque de données

## 2.3 Implémentation

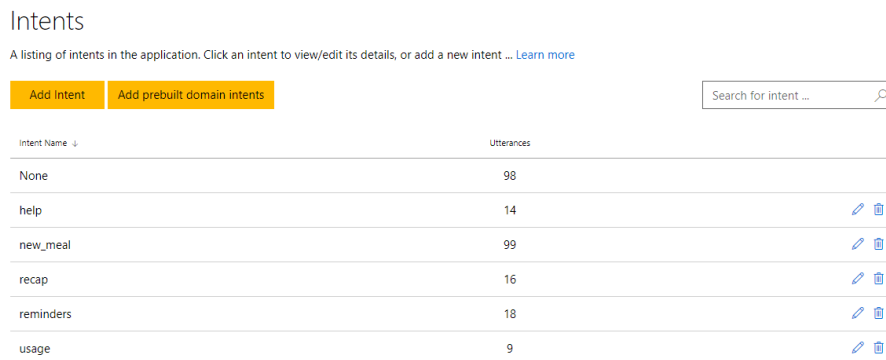
### 2.3.1 Conversation

#### Entraînement de l'application de NLU

Au fur et à mesure du développement et des tests effectués en parallèle, il a été fréquent que Luis.ai ne comprenne pas l'intention que j'avais voulu donner à mon message. Il a donc fallu ré-entraîner l'application en rajoutant des exemples de ce qui n'avait pas correctement fonctionné. Au moment où j'écris cette partie (18/08/17), 271 exemples ont été donnés et expliqués à Luis.ai principalement pour les intents 'new\_meal' et 'None'.

'new\_meal' parce que le but principal du chatbot est de repérer les repas dans les messages de l'utilisateur et 'None' pour s'assurer que des réponses à des questions directes ou des messages abérants ne déclenchent pas d'intents.

Capture d'écran 2.1 – Liste des *intents* reconnaissables par Luis.ai



Intents

A listing of intents in the application. Click an intent to view/edit its details, or add a new intent ... [Learn more](#)

[Add intent](#) [Add prebuilt domain intents](#)

Intent Name	Utterances	
None	98	
help	14	<a href="#">edit</a> <a href="#">delete</a>
new_meal	99	<a href="#">edit</a> <a href="#">delete</a>
recap	16	<a href="#">edit</a> <a href="#">delete</a>
reminders	18	<a href="#">edit</a> <a href="#">delete</a>
usage	9	<a href="#">edit</a> <a href="#">delete</a>

#### Gestion des *intents*

Les dialogues sont répartis entre plusieurs fichiers, pour assurer la redirection vers les dialogues lorsqu'un *intent* est détecté le framework propose une solution qui n'est pas satisfaisante. J'en ai donc mis une en place. Microsoft permet de se placer entre son connecteur et le bot et "d'intercepter" les messages. À cet endroit, j'envoie les messages à Luis.ai qui me donne une réponse telle que celle montrée en exemple dans la partie **Conception**. Le programme prend alors la décision de changer de dialogue ou non selon le degré de confiance annoncé par Luis.

Concrètement, j'ai écrit un dictionnaire avec en clé le nom de l'*intent* et en valeur le chemin vers le dialogue qui doit être appelé.

#### Extrait de code 2.4 – Dictionnaire liant les *intents* et les dialogues

```
1 var linker = {
```

```

2 new_meal: 'new_meal:/',
3 help: 'menu:showMenu',
4 usage: 'usage:showUsage',
5 reminders: 'reminders:/',
6 recap: 'recap:showRecap',
7 None: 'default:/'
8 }

```

Cette façon de faire permet de contrôler précisément les changements de dialogues dans toutes les situations.

### 2.3.2 Demande d'informations utilisateur

À chaque passage dans le dialogue de base ( '/') un dialogue pour remplir le formulaire est initié. Dans un premier temps ce dialogue vérifie que le formulaire n'a pas déjà été rempli (un flag est présent dans `session.userData`) et que la dernière demande était il y a suffisamment longtemps, de sorte à ne pas harceler l'utilisateur.

Si les conditions sont remplies, un dialogue pour demander l'accord de l'utilisateur est déclenché puis le dialogue clé du processus est démarré si l'utilisateur accepte de répondre au formulaire. Le fonctionnement du dialogue est détaillé en dessous.

#### Extrait de code 2.5 – Dialogue parcourant la liste des questions et stockant les réponses

```

1 lib.dialog('form', [
2   function (session, args) {
3     // Save previous state (create on first call)
4     session.dialogData.index = args ? args.index : 0;
5     session.dialogData.form = args ? args.form : {};
6
7     // Prompt user for next field
8     builder.Prompts.text(session,
9                           questions[session.dialogData.index].prompt
10                        );
11  },
12  function (session, results) {
13    var field = questions[session.dialogData.index++].field;
14    session.dialogData.form[field] = results.response;
15
16    // Check for end of form
17    if (session.dialogData.index >= questions.length) {
18      var msgThx = "Thank_you,I'll_keep_this_in_mind_and_"
19                  +"use_it_to_better_help_you";
20      session.send(msgThx);
21
22      // Return completed form
23      processForm(session.dialogData.form, session.userData.userId);
24      session.endDialogWithResult({response: session.dialogData.form});
25    } else {
26      // Next field
27      session.replaceDialog('form', session.dialogData);
28    }
29  }
30 ]);

```

Les questions sont stockées dans une liste sous forme d'un objet avec un champ 'field' qui correspond au nom de la question et un champ 'prompt' qui correspond à la question qui sera posée.

Lors du premier passage dans le dialogue, l'objet qui contiendra les résultats (`session.dialogData.form`) et l'index dans le tableau sont initialisés. Un prompt est créé par le framework et envoyé à l'utilisateur à partir de la question d'indice courant. La réponse à la question est passée à la deuxième

partie du dialogue, où elle est stockée, l'index est incrémenté à cet instant. Ensuite on contrôle si toutes les questions ont été posées : si oui on lève le flag pour signaler que le formulaire est complet, on envoie un message de fin et on stocke les informations ; si non on ré-appelle ce dialogue avec en arguments l'index courant et l'objet contenant les réponses.

### 2.3.3 Envoi de rappels

La création d'un rappel est simplement une série de prompts dont les résultats sont rassemblés dans un objet tel que celui présenté dans la partie **Conception**. Pour envoyer les rappels, j'ai choisi d'utiliser le package npm cron, qui permet d'avoir en parallèle du chatbot une fonction qui cherche toutes les minutes en base de données si des rappels doivent être envoyés. Ces rappels sont regroupés dans une liste et une fonction d'envoi de message est appelée pour chacun. Les heures des rappels enregistrés sont donc arrondies à 0 secondes, j'ai jugé qu'être précis à une minute près seulement n'était pas gênant pour l'utilisation du chatbot (les rappels ont pour but de rappeler à l'utilisateur d'indiquer ce qu'il a mangé et n'ont pas vocation à remplacer un réveil ou une montre).

Extrait de code 2.6 – CronJob utilisé pour gérer les rappels

```
1 var searchReminders = new CronJob('0_*_*_*_*_*',
2   function() {
3     var currentTime = new Date();
4     var query = {time: {$lte: currentTime}};
5     currentTime.setSeconds(0, 0);
6     console.log("Let's check for reminders at ", currentTime);
7     db.collection('reminders').find(query).toArray(function (err, res) {
8       if (err) throw err;
9       console.log("Reminders found: ", res);
10      res.forEach(remind.processReminder);
11    })
12  }, function () { // On complete
13    console.log("The CronJob searchReminders has ended (onComplete)");
14  },
15  true, // Autostarts
16  'Europe/Paris' // Time zone
17 );
```

### 2.3.4 Reconnaissance des repas

Cette fonctionnalité est centrale dans le chatbot : tout découle de la bonne reconnaissance des repas. Lorsque l'*intent* 'new\_meal' est détecté -c'est à dire quand Luis.ai devine que l'utilisateur souhaite enregistrer un repas- le dialogue déclenché va contrôler que toutes les informations sont présentes :

1. si aucun aliment n'est détecté par Luis.ai le bot demande explicitement à l'utilisateur ce qu'il a mangé.
2. si le nom du repas (breakfast, lunch, dinner, snack) n'est pas présent il est demandé à l'utilisateur.
3. si l'utilisateur a indiqué une quantité non répertoriée dans le bot, il lui est demandé de l'explicitier.

Notre application Luis.ai a été entraînée à reconnaître les aliments et les quantités liées à ces aliments cependant, l'ordre de ces derniers n'était pas cohérent et variait d'un message à l'autre. Une entité *serving* a donc été ajoutée : une entité *serving* est composée d'exactly une entité *food* et d'au plus une entité *quantity*. Cela permet d'être capable de réassembler les aliments et leurs quantités, peu importe l'ordre dans lequel ils sont fournis. Ce travail est effectué par la fonction qui suit, donc le fonctionnement est expliqué juste après.

Extrait de code 2.7 – Fonction regroupant les aliments avec leur quantités

```
1 processEntities = function (entities) {
2   if (entities) {
```

```

3   var foods = [];
4   var qtts = [];
5   var servs = [];
6   var meal = "";
7   var res = [];
8
9   for (var i = 0; i < entities.length; i++) {
10      if (entities[i].type == 'food'){
11         foods.push(entities[i].entity);
12      } else if (entities[i].type == 'quantity') {
13         qtts.push(entities[i].entity);
14      } else if (entities[i].type == 'serving') {
15         servs.push(entities[i].entity);
16      } else if (entities[i].type == 'meal') {
17         meal = entities[i].entity
18      }
19   }
20
21   for (var i = 0; i < servs.length; i++) {
22      var tmpRes = [];
23
24      for (var j = 0; j < foods.length; j++) {
25         if (servs[i].search(foods[j]) !== -1) {
26            tmpRes.push(foods[j]);
27         }
28      }
29      for (var j = 0; j < qtts.length; j++) {
30         if (servs[i].search(qtts[j]) !== -1) {
31            tmpRes.push(qtts[j]);
32         }
33      }
34
35      res.push(tmpRes);
36   }
37   return [res, meal];
38 } else {
39   return [[], ''];
40 }
41 }

```

La fonction reçoit en argument une liste d'entités telle que celle présentée sur le Schéma 2.5. Puisque le but est de pouvoir s'affranchir de l'ordre dans lequel les entités sont envoyées, dans un premier temps on regroupe chaque entité dans la liste de son type. On a alors la liste de tous les aliments détectés, la liste de toutes les quantités détectées, la liste de toutes les portions (aliment + quantité) détectées et éventuellement le nom du repas.

L'étape suivante consiste à parcourir toutes les portions pour retrouver l'aliment et la quantité qui correspond à chaque portion. Des tuples [aliment, quantité] sont formés et ajoutés au résultat. Le retour de cette fonction est le tuple formé par la liste des tuples et le nom du repas.

### 2.3.5 Récapitulatif

Dans la suite de dialogues récapitulatifs, la partie intéressante à présenter est la partie menant à l'affichage des apports nutritionnels d'une série de repas. Dans un premier temps, le bot demande à l'utilisateur pour quelle période il souhaite voir les apports nutritionnels. Une liste de tuples [aliment, quantité] est alors formée et le programme va chercher les aliments les plus justes possibles dans la base de données nutritionnelles.



La difficulté de cette partie est que la banque de données à disposition ne contient pas forcément exactement l'aliment indiqué par l'utilisateur et repéré par notre application de NLU ou alors la recherche de l'aliment va donner beaucoup trop de résultats, pas tous pertinents. C'est pourquoi le traitement est réalisé en plusieurs étapes successives.

Dans un premier temps, pour chaque aliment, une fonction parcourt toute la banque de données et retourne toutes les lignes qui comportent cet aliment.

La liste obtenue passe ensuite par deux fonctions de "preprocessing" :

Une fonction cherche si des références commencent exactement par l'aliment suivi d'une virgule, si elle n'en trouve pas elle cherche des références commençant par l'aliment suivi d'un espace. Si elle n'en trouve pas non plus, on conserve la liste originale non modifiée.

L'étape suivante est faite totalement arbitrairement : on retire les lignes concernant de la viande, du poisson ou des légumes crus, et tout les fruits non frais. En effet, pour un même aliment sont stockées plusieurs variations : types de cuisson, stockage ou assaisonnement. Pour réaliser cette étape il suffit de parcourir la liste et de retirer les lignes sus-mentionnées.

L'ensemble de ce procédé est illustré ci-dessous :

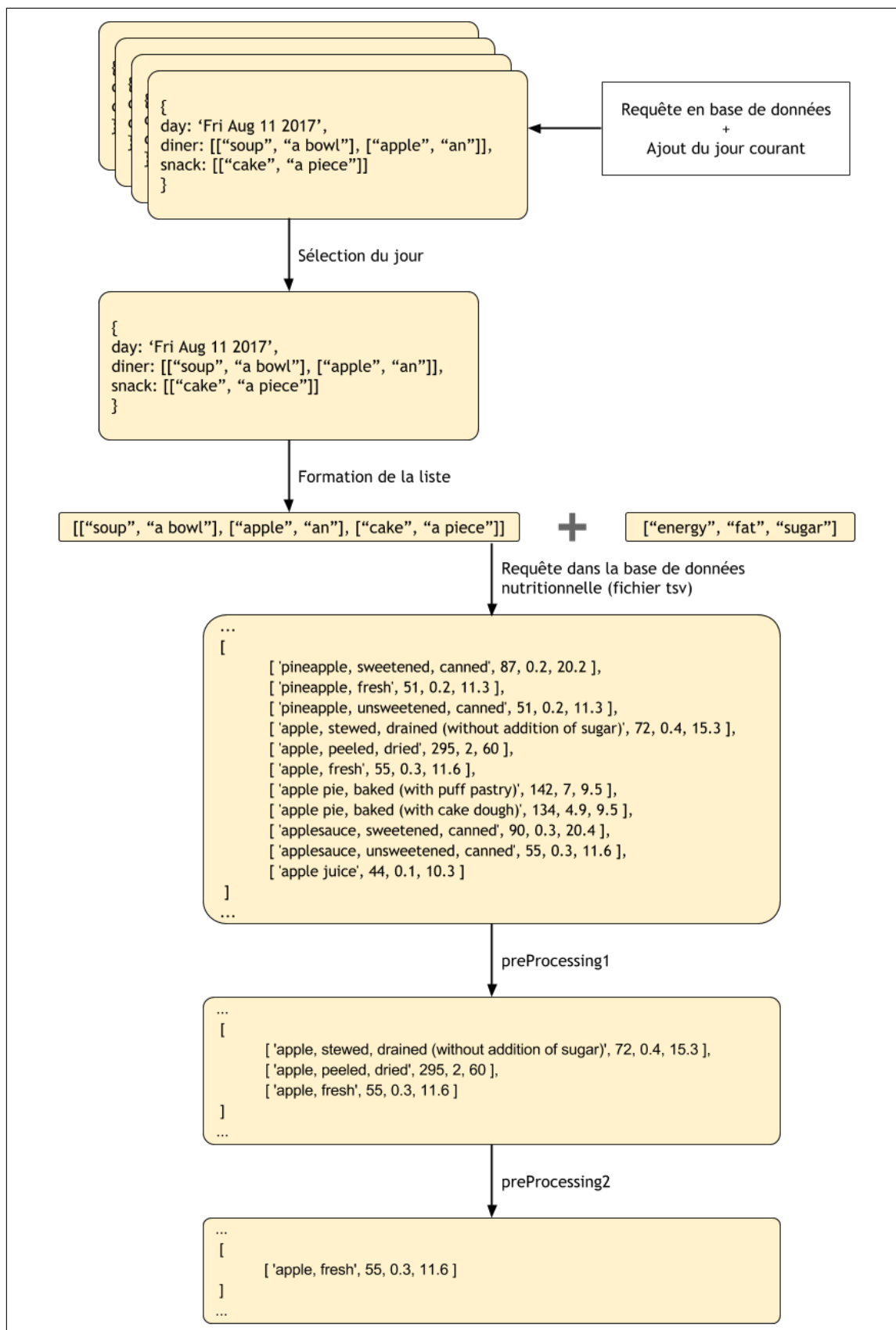


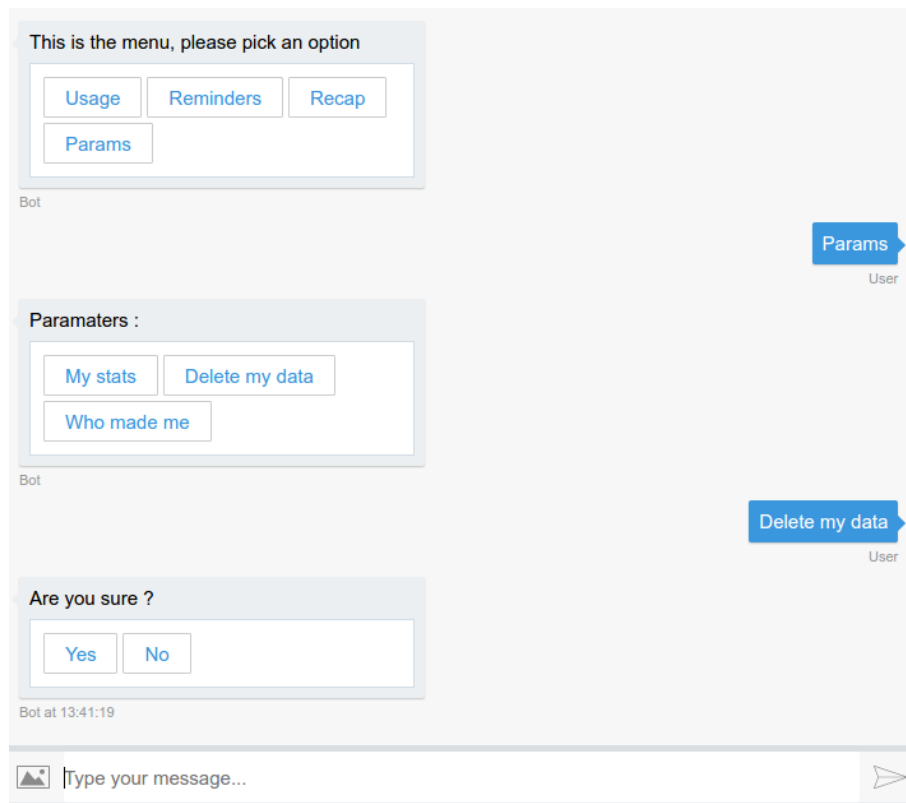
Schéma 2.9 – Procédé menant à l’affichage des apports nutritionnels.

## 2.4 Tests et Validation

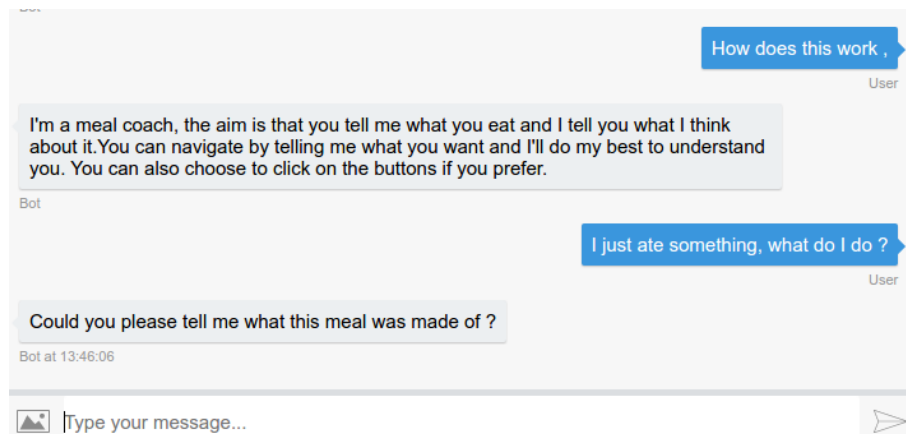
### Flot conversationnel

L'aspect conversationnel a été réalisé conformément au schéma disponible dans la partie **Conception** du rapport. Suivent deux captures d'écran pour illustrer les deux possibilités de navigation dans le schéma conversationnel du chatbot : La première représente une navigation en cliquant sur les boutons proposés par le bot. La seconde représente un passage par l'application de NLU : l'utilisateur exprime un souhait qui est compris par Luis.ai puis les dialogues correspondants sont enclenchés.

Capture d'écran 2.2 – Navigation par boutons dans le chatbot



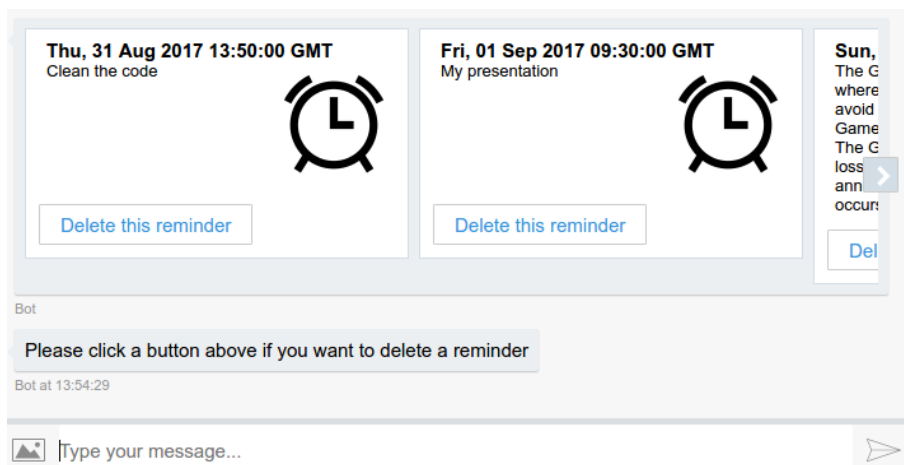
Capture d'écran 2.3 – Navigation par *intents* dans le chatbot



## Cartes, boutons

Il s'agissait avec cette partie de montrer comment il était possible d'utiliser les outils proposés par le SDK pour mettre en place des boutons et des carrousels (disposition horizontale de cartes de contenu divers). Sur la capture d'écran qui suit on peut voir que les rappels en cours de l'utilisateur sont représentés en carrousel, chaque rappel disposant d'un bouton pour le supprimer.

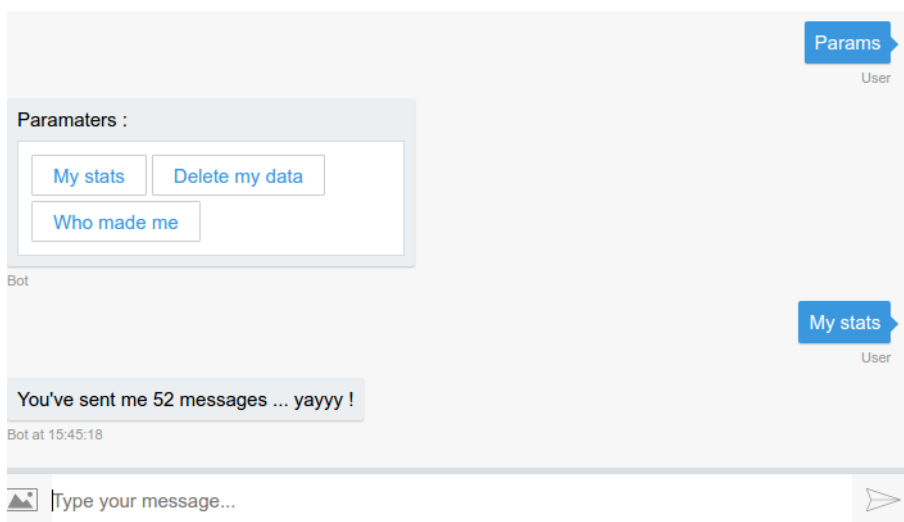
Capture d'écran 2.4 – Exemple de carrousel : affichage des rappels



## Enregistrement des conversations

La capture d'écran qui suit est un bref dialogue accessible depuis le dialogue des paramètres du chatbot, il indique à l'utilisateur le nombre de messages qu'il a enregistré.

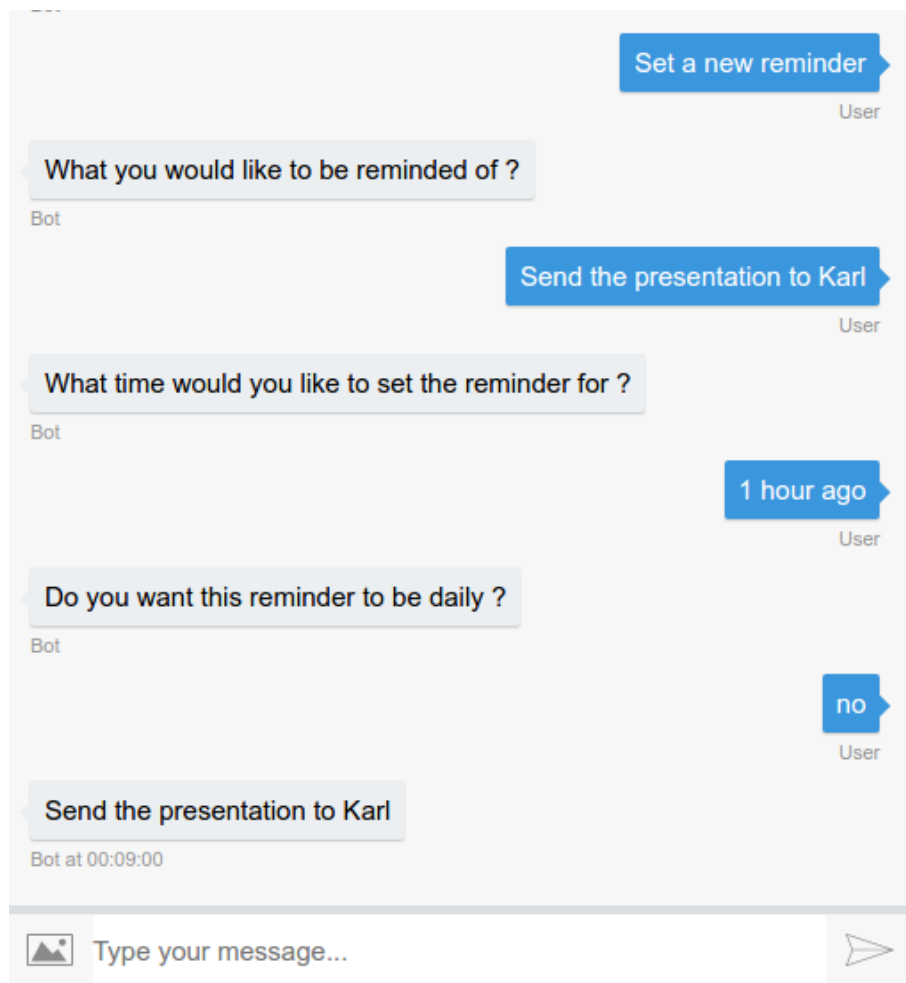
Capture d'écran 2.5 – Affichage du nombre de messages enregistrés par le chatbot.



## Rappels

La capture d'écran qui suit illustre l'enregistrement d'un rappel : l'utilisateur est guidé à travers les différents champs d'un rappel. On peut voir que le rappel est effectivement envoyé à l'instant choisi par l'utilisateur.

Capture d'écran 2.6 – Exemple d'enregistrement d'un rappel, puis reception de ce rappel

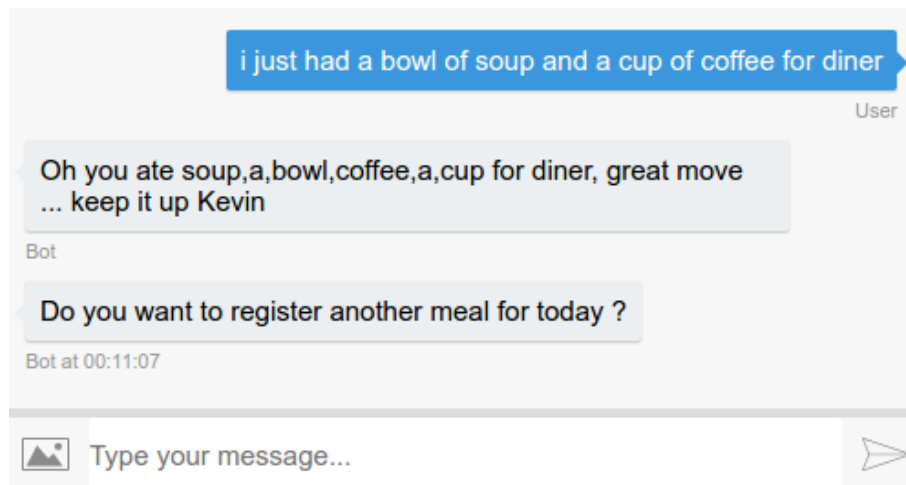


Cet exemple est particulier puisque à un instant  $t$  le chatbot envoie tous les rappels dont l'heure est inférieure ou égale à  $t$ . J'ai donc enregistré le rappel dans le passé pour qu'il soit envoyé au plus vite.

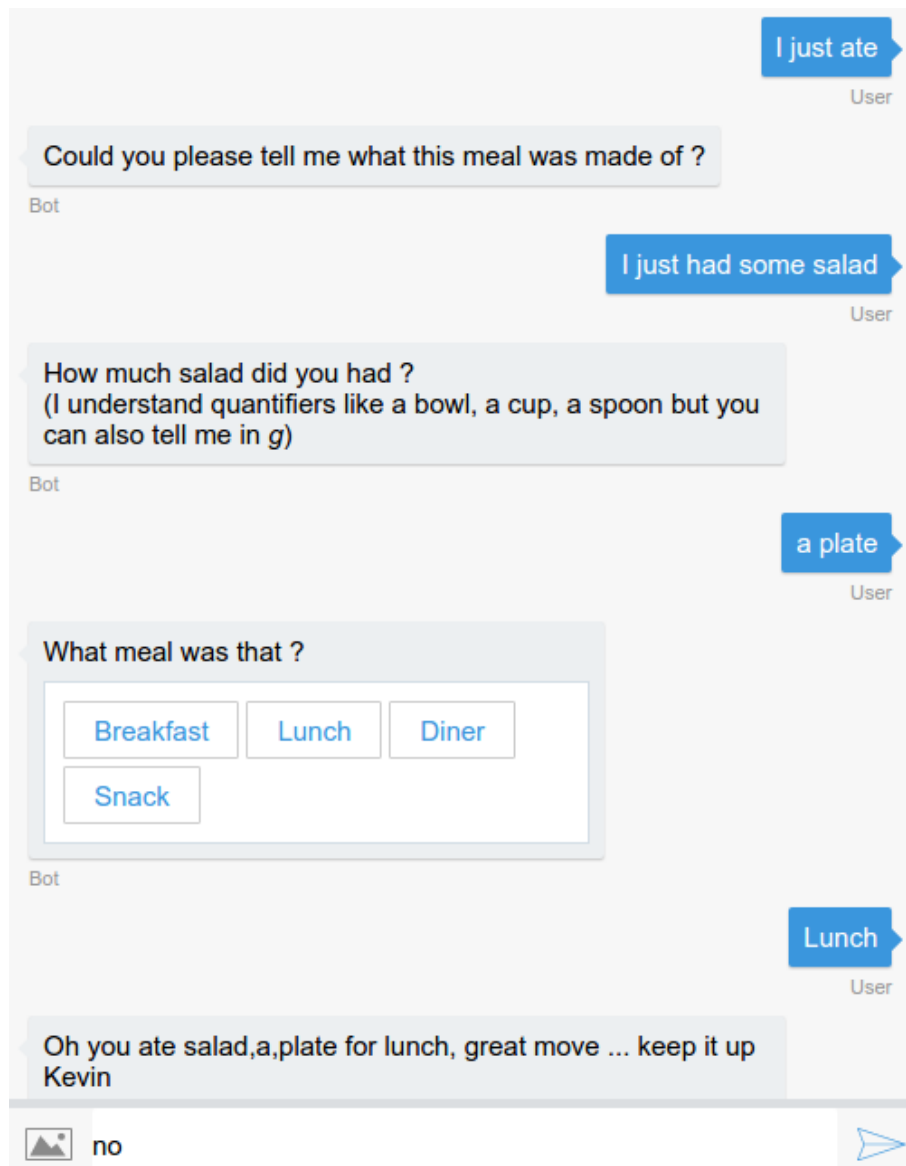
### Reconnaissance et stockage des repas

Deux captures d'écran montrent les deux exemples d'enregistrement de repas les plus extrêmes : sur la première on peut voir que l'utilisateur a inclu dans son message toutes les informations nécessaires et que le programme a été capable de les reconnaître, sur la seconde on voit que l'utilisateur se fait guider par le chatbot pour fournir toutes les informations requises.

Capture d'écran 2.7 – Exemple d'enregistrement d'un repas : message complet



Capture d'écran 2.8 – Exemple d'enregistrement d'un repas : guidage de l'utilisateur



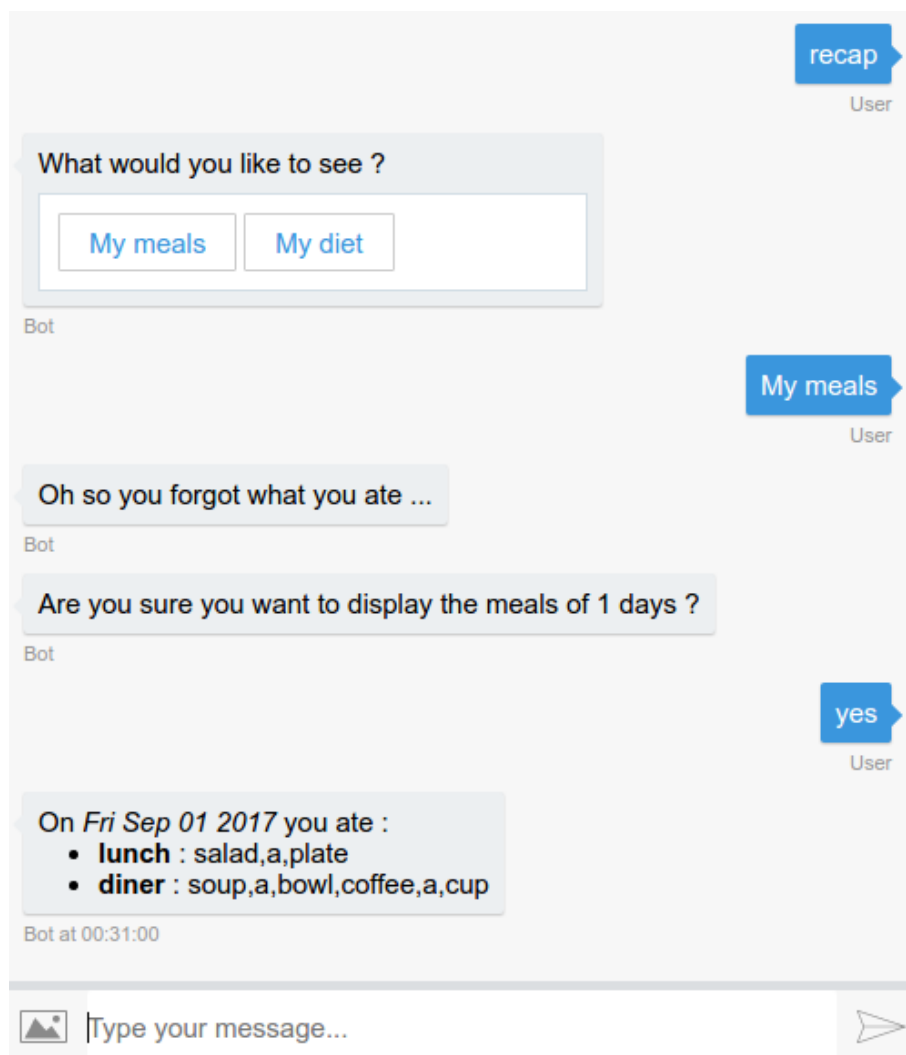
Bien sur, si l'utilisateur n'indique qu'une partie des informations requises, le chatbot va l'interroger sur les champs manquants mais dans un soucis de relative concision de ce rapport ces différents cas ne sont pas illustrés.

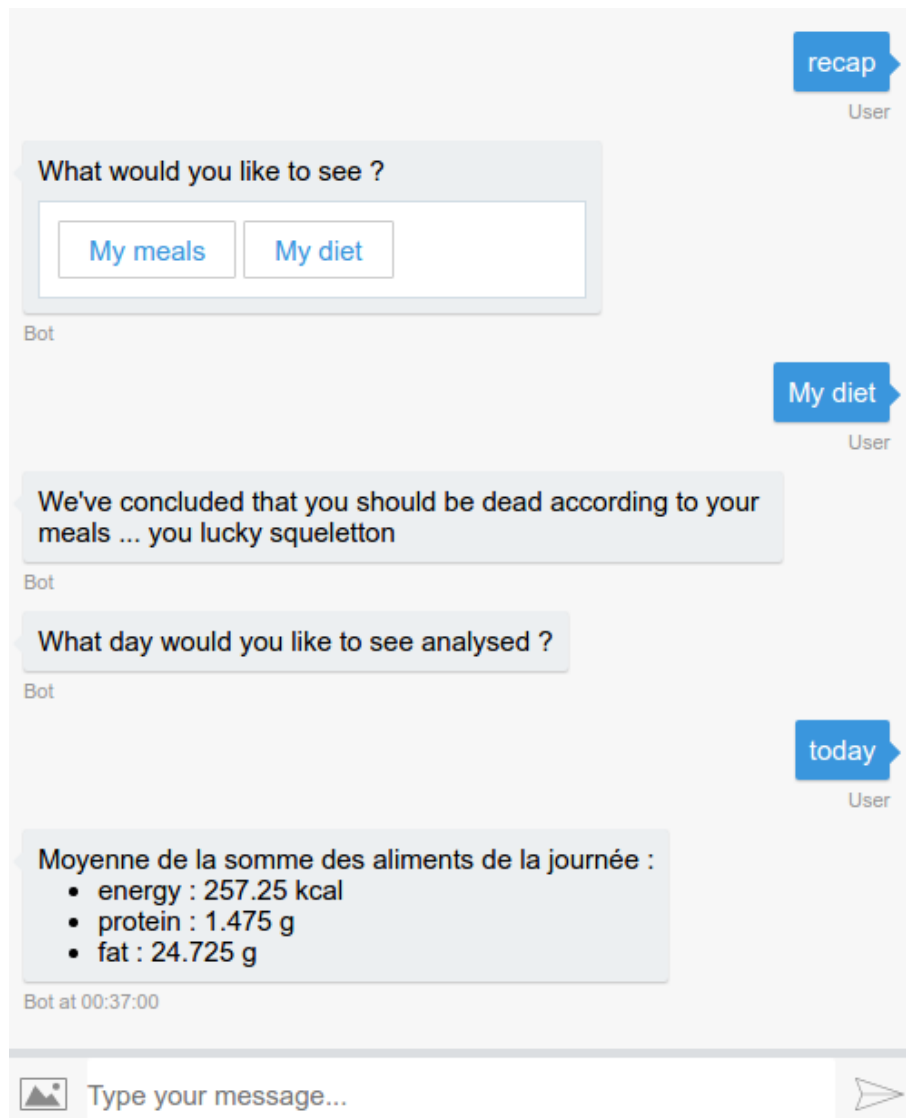
Il est important de noter que la reconnaissance des repas -c'est à dire des aliments et de leurs éventuelles quantités- est totalement dépendante de l'application de NLU. (Ce point est développé dans la partie **Idées d'amélioration** du rapport.)

### Génération de récapitulatifs

Les deux captures d'écran qui suivent illustrent les deux types de récapitulatifs disponibles : l'affichage des repas enregistrés et le calcul des apports nutritionnels d'une journée de repas.

Capture d'écran 2.9 – Affichage des repas enregistrés







# Chapitre 3

## Bilan

### 3.1 Respect du cahier des charges

### 3.2 Problèmes rencontrés

#### Node.js

C'était la première fois que je faisais réellement du Javascript et cela a entraîné quelques difficultés pendant la première partie du stage. Node.js fonctionnant de manière asynchrone avec des fonctions de callback j'ai eu du mal à me détacher de la programmation "*linéaire*" que je connaissais. Notamment je suis resté quelques heures bloqué parce que je ne comprenais pas pourquoi mes fonctions n'étaient pas exécutées.

Je me suis également plusieurs fois retrouvé à écrire des fonctions qui existaient déjà parce que je ne connaissais pas le langage.

#### BotBuilder

Le SDK est relativement récent (environ 1 an), ce qui fait qu'il n'y a pas autant de ressources disponibles pour débiter que pour d'autres technologies et que dès qu'on souhaite aller plus loin que les exemples proposés par Microsoft il faut passer par la documentation officielle pour voir ce qui existe comme fonctionnalité puis s'employer à la faire fonctionner pour réaliser ce qu'on désire.

#### Dialogues en parallèle

Toute action de l'utilisateur, qu'il clique sur un bouton, qu'il envoie un message ou qu'il envoie une photo est comprise comme un message par le framework. Tous les messages sont analysés par l'application Luis.ai. Ceci a eu pour effet que dans les débuts du chatbot des conversations se déclenchaient à des moments non voulus.

Imaginez par exemple que le nom du rappel enregistré par l'utilisateur soit pour une quelconque raison "Je viens de manger". L'application de NLU reconnaîtrait alors que ce message indique un nouveau repas et lancerait le dialogue "*new\_meal*" ce qui rendra impossible la finalisation du rappel.

J'ai donc dans un premier temps implémenté un système de flag : à chaque début de dialogue qui ne devait pas être interrompu un drapeau était levé (puis baissé en sortie du dialogue) et la valeur du drapeau était testée au moment de changer de dialogue. Le problème de cette façon de faire est que si le dialogue est interrompu et que le drapeau n'est pas baissé alors plus aucune redirection n'est possible et le bot devient inutilisable.

La solution finale que j'ai adoptée pour ce problème consiste à regarder la pile de dialogues et déterminer le dialogue courant avant le traitement de chaque message (c'est à dire avant chaque potentielle redirection). Grâce à un dictionnaire, chaque dialogue dispose d'un seuil différent des autres. Pour que le dialogue courant soit quitté directement il faut que la note (comprise entre 0

et 1) attribuée par Luis.ai au nouveau dialogue soit supérieure à ce seuil. Les seuils de certains dialogues sensibles ont été fixés à 2 pour rendre impossible toute redirection dans ces dialogues.

## Vision d'ensemble

Mon manque de vision sur le projet dans son intégralité a fait que j'ai parfois implémenté des fonctionnalités sans avoir en tête l'impact qu'elle allait avoir sur le projet. Cela a entraîné plusieurs réusinages de code qui auraient pu être évités si j'avais pris plus de temps pour préparer les structures de données pour tout le projet et non pas fonctionnalité par fonctionnalité.

## Luis.ai

J'ai commencé par entraîner l'application à reconnaître les aliments, puis les quantités. Le problème est que Luis.ai ne renvoie pas les entités dans un ordre fixe. J'ai donc dû créer une entité *serving* pour regrouper les aliments et leurs quantités et à chaque fois reprendre tous les exemples sur lesquels l'application de NLU pour les marquer à nouveau avec les nouvelles entités.

## Gestion de la quantité

En plus de reconnaître les aliments, le bot doit reconnaître les quantités associées à ces aliments pour être capable de réellement produire des informations utiles à l'utilisateur.

Les données nutritionnelles dont le bot dispose (disponibles en annexes) sont disponibles en unité/100g pour la majorité des aliments solides. La solution la plus simple serait donc de demander à l'utilisateur combien son steak pesait mais cela va à l'encontre du rôle d'un chatbot : être plus simple et agréable d'utilisation qu'un formulaire. Il faut donc faire en sorte que notre bot soit capable de comprendre des quantifieurs simples et basiques tels qu'un bol, une cuillère ou une assiette. La nouvelle difficulté est alors de traduire ces volumes en masses : on ne peut pas raisonnablement considérer qu'une assiette de salade et une assiette d'huîtres aient une masse d'aliment identique. L'approche que j'ai considérée a été d'affecter une "densité" à chaque sous-catégorie d'aliment et proposer à l'utilisateur d'utiliser des quantifieurs classiques.

Une autre difficulté liée à la quantité qui n'a pas été traitée est la gestion de l'unité. Comment faire en sorte que le programme puisse savoir qu'une cerise ne pèse pas autant qu'une entrecôte ? À première vue il me semble que la solution la plus simple serait de trouver une base de données qui recense les masses moyennes des aliments les plus courants.

## 3.3 Idées d'amélioration et limites du *proof-of-concept*

### Diversifier l'entraînement de l'application de NLU

L'application de NLU a été entraînée de façon extrêmement biaisée : je suis le seul à lui avoir fourni des exemples et par conséquent ses capacités de compréhension sont limitées. C'est à dire que si un utilisateur converse avec le chatbot avec des tournures de phrase ou une grammaire différente de la mienne il n'est absolument pas certain que la compréhension soit bonne. Pour remédier à cela, il faudrait que l'application de NLU soit entraînée par quelqu'un d'autre. Luis.ai propose également de classifier et d'ajouter simplement les messages que les utilisateurs lui ont envoyé et pour lesquels l'application n'a pas été entraînée. Cette possibilité n'est utilisable que si le chatbot est rendu public, ce qui n'est pas encore le cas.

### Ajouter de nouvelles langues d'utilisation

À terme, le chatbot sera utilisé pour la population suisse. Il devra donc être capable de converser en allemand, en italien et en français. La façon de faire qui me vient en tête consiste à remplacer tous les textes par des objets avec un champ par langue, et d'ajouter un paramètre langue à la session. Il faudrait certainement créer une application Luis.ai par langue ce qui est assez laborieux puisqu'il faut traduire le JSON contenant les exemples sur lesquels l'application s'est entraînée (le JSON que j'ai utilisé et construit pendant mon stage est disponible dans le dossier assets du projet.)

### **Ajouter un correcteur orthographique**

Il est inconscient de partir du principe que les utilisateurs écriront tous toujours sans faire de fautes. Il y a deux solutions à ce problème : entraîner l'application de NLU en faisant volontairement des fautes -ce qui me semble incroyablement laborieux- ou faire passer tous les messages de l'utilisateur par un correcteur orthographique avant de les traiter.

### **Permettre à l'utilisateur d'apprendre des mots de vocabulaire au chatbot**

La capacité du chatbot à générer des informations utiles est limitée par sa connaissance des unités de mesure. Une possibilité de palier à ce problème est de permettre à l'utilisateur d'enseigner au chatbot de nouvelles unités.

### **Améliorer et diversifier les synthèses nutritionnelles**

Pour le projet je me suis limité à montrer qu'il était possible de générer un compte-rendu nutritionnel. À terme il faudrait que le chatbot puisse générer des graphiques à partir des informations nutritionnelles récoltée, qu'il puisse donner des conseils en fonction des tendances qu'il observe dans l'alimentation de l'utilisateur et, avec l'accord express de l'utilisateur, qu'il collecte des données pour générer des statistiques plus globales sur l'alimentation des suisses.

## **3.4 Conclusion**

Sur le plan professionnel ce stage en a été riche en enseignements. J'ai travaillé pour la première fois dans un environnement professionnel ce qui m'a permis de travailler avec de réelles contraintes et des impératifs indépendants de ma volonté. C'était également la première fois que je travaillais en Node.js ce qui me permet d'en avoir maintenant une bonne vision du fonctionnement.

J'ai également progressé dans ma façon de présenter mon travail car j'ai eu deux présentations à faire en interne à l'institut HumanTech au cours de mon stage.

La nature internationale du stage était ici limitée puisque la partie de la Suisse dans laquelle j'étais est francophone. J'ai pu cependant acquérir un assez bon point de vue de la vie professionnelle en Suisse et nouer de bons contacts.

Toutefois je regrette ne pas avoir eu le temps d'aller au bout de l'objectif final : la reconnaissance des repas à partir d'images.

## Chapitre 4

# Bibliographie

Voici les ressources principales que j'ai utilisé pendant mon stage :

- Documentation officielle du SDK
- Le github de Microsoft
- Documentation MongoDB
- Tutoriel MongoDB

## Chapitre 5

## Annexes

ENSIIE  
Stage de première année

# État de l'art

## Conception et développement d'un agent conversationnel

Hugo BELHOMME

Jacky CASAS  
Omar ABOU KHALED

4 juillet 2017

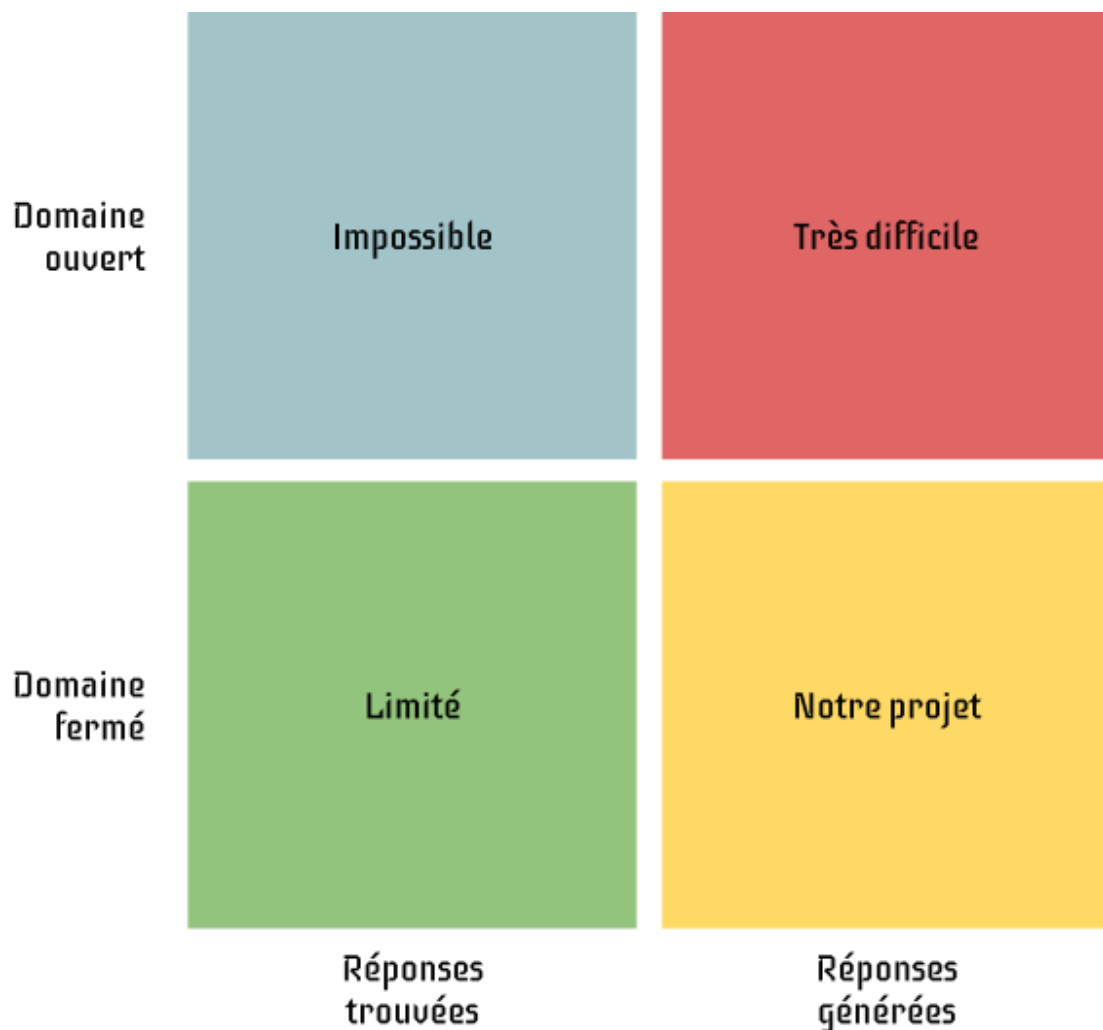


# Qu'est-ce qu'un chatbot ?

Un chatbot -ou agent conversationnel- est un programme informatique capable de répondre lorsqu'un utilisateur lui transmet un message. Ils sont utilisés dans le domaine de la communication, de la relation client, de l'aide à la personne ou des loisirs. Ainsi il existe des chatbots qui permettent de répondre à des questions simples de clients, qui peuvent prendre des rendez-vous ou passer des commandes pour l'utilisateur, qui peuvent répondre aux questions de l'utilisateur sur la santé ou qui peuvent faire jouer l'utilisateur à des quizz.

L'intérêt d'un chatbot est qu'il peut répondre correctement et instantanément à tout type de question pour laquelle il a été préparé. Ceci avec très peu de coûts d'entretien par rapport à des employés puisque la majeure partie du travail est effectuée en amont.

Il existe de nombreux types de chatbots et une façon simple de les différencier est de prendre en compte leur capacité de conversation. L'image suivante est la plus utilisée pour illustrer cette classification :



On distingue deux catégories de conversations : domaine ouvert ou domaine fermé; ainsi que deux catégories de réponses : préparées à l'avance ou générées intelligemment.

Un **chatbot** donnant des réponses prédéfinies dans un domaine libre est impossible puisqu'il faudrait pour cela avoir anticipé chaque scénario possible.

Un **chatbot** donnant des réponses prédéfinies dans un domaine limité représente la base des chatbots et est relativement aisé à mettre en place. Attention : il est néanmoins possible d'approcher fortement l'intelligence, mais tout doit être anticipé et hardcodé.

Les **chatbots** donnant des réponses générées dans un domaine restreint ont été rendu possibles par l'évolution des technologies de Machine Learning. Selon le domaine et la qualité voulue la difficulté peut varier grandement.

Les **chatbots** donnant des réponses générées dans un domaine quelconque sont très difficile à mettre en place et sont sujets à des recherches au sein des plus grandes entreprises du numérique. Le but de ces chatbots est d'arriver à suivre n'importe quelle conversation avec un humain, et aussi bien (voir mieux) qu'un être humain.



# Principe général de fonctionnement

Un des enjeux des chatbots est de parvenir à comprendre les messages de l'utilisateur de façon fiable. On se rendra compte aisément que chercher si un mot est dans le message pour déclencher une réponse prédéfinie atteint rapidement ses limites. La compréhension du langage est un large champ de recherche à lui tout seul et il existe des services dits de NLU (Natural Language Understanding) que l'on peut rendre capables de repérer l'intention (*intent*) de l'utilisateur et les paramètres (*entities*) de cette intention.

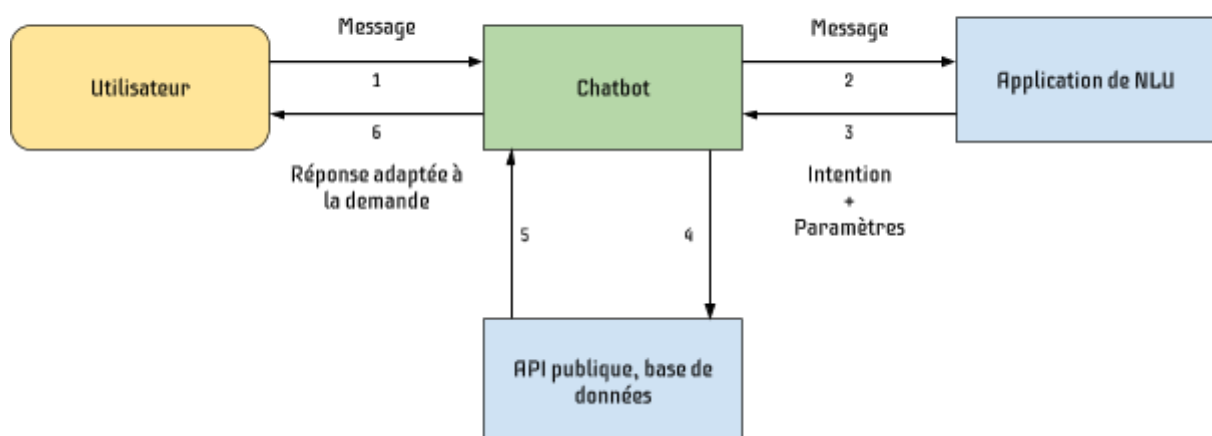
## Utilisation d'un service de NLU



En effet le but n'est pas de comprendre ce que l'utilisateur a dit, mais de comprendre ce qu'il désire. Pour être efficace, il faut entraîner l'app de NLU en lui fournissant des messages et en lui indiquant quels sont les intentions et les paramètres dans les messages.

Une fois la compréhension de l'utilisateur possible, il suffit par exemple d'aller chercher une pizzeria dans une base de données et de passer une commande.

## Fonctionnement simplifié d'un chatbot



# Blocages classiques et solutions

## Mécanisme de retry :

Il peut arriver que le chatbot ait mal compris le message de l'utilisateur, il faut alors que l'utilisateur ait la possibilité de reformuler son message. Cependant, on ne peut pas se permettre de demander à l'utilisateur si il est satisfait à chaque fois que le chatbot envoie une réponse sans rendre l'expérience utilisateur mauvaise. Personne n'a envie de doubler tous ses messages d'un *"La réponse était satisfaisante"* même si il suffit de cliquer sur un bouton.

Il faudrait donc disposer d'un moyen d'évaluer la fiabilité de la compréhension du message puis définir un seuil à partir duquel on considère que le message n'a pas été compris pour proposer à l'utilisateur de reformuler son message.

## Messages consécutifs :

Un utilisateur peut très bien envoyer plusieurs messages consécutifs avant que le chatbot n'ait répondu au premier. Je pense qu'il faut prendre parti ici : **ou (1)** l'on considère que plusieurs messages consécutifs correspondent à autant *"d'instructions"* et il faut donc porter attention à leur ordre de traitement **ou (2)** l'on considère que plusieurs messages consécutifs forment une seule et unique *"instruction"*.

Problème avec **(2)** : peut être que la réponse n'a pas été envoyée, mais qu'elle est en cours de traitement, ce qui rendrait impossible la *"fusion"*. On pourrait envisager de mettre un délai avant de commencer à traiter la requête mais cela risque de rendre l'expérience désagréable.

Problème avec **(1)** : il faut bien maîtriser tous les éléments utilisés (la plateforme, la structure du code) pour être certain que tous les messages seront traités dans l'ordre et que les réponses seront dans l'ordre également. S'ajoute à cela la possibilité que des accès simultanés en base de données peuvent générer des conflits.

## Gérer les rappels :

Souvent dans les fonctionnalités d'un chatbot figure un système de rappels mais un tel système peut être compliqué à gérer en raison des différents fuseaux horaires si ils ne sont pas gérés. Si un utilisateur demande à ce qu'un rappel lui soit envoyé à 20h, il veut le recevoir à 20h heure de là où il vit et non pas à 20h heure de chez le développeur ou 20h heure du serveur du chatbot. Il faudrait donc d'une manière ou d'une autre déterminer puis stocker le fuseau horaire de l'utilisateur.







Plutôt que de demander à l'utilisateur son fuseau horaire, il pourrait être envisageable de demander à l'utilisateur dans combien de temps il veut recevoir un rappel. Il serait alors possible de décrémenter un compteur toutes les X minutes jusqu'à atteindre 0 et envoyer le rappel mais cela demanderait dans la majorité des cas un travail supplémentaire de la part de l'utilisateur, ce qui est le contraire de ce que l'on veut accomplir avec un chatbot.

Je pense qu'il est préférable de demander à l'utilisateur à sa première utilisation.

## Utilisateur *"malveillant"* :

Il arrivera qu'un utilisateur ne réponde pas au chatbot de façon appropriée. Un chatbot devrait être capable de s'en rendre compte et d'agir en conséquence plutôt que de répondre d'un façon qui sera nécessairement erratique.

## Les plateformes de messagerie




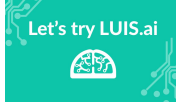

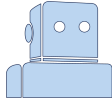
Plateforme	Utilisateurs actifs	Type de bot	Usage	Restrictions particulières
Discord 	45 M / mois (mid 2017)	Intégré à un serveur	Facilitateur, récréatif	Doit être ajouté au serveur par un utilisateur ayant les droits
Messenger 	1,2 B / mois (mid 2017)	Page Facebook	Tous	
Skype 	300 M / mois (fin 2016)	Skype app	Tous	Ne peut pas lire de fichiers
Slack 	5,8 M / semaine (fin 2016)	Intégré à un serveur	Facilitateur, récréatif	Doit être ajouté au serveur par un utilisateur ayant les droits
Telegram 	100 M / mois (deb 2016)	Bot user	Tous	
Twitter 	328 M / mois (deb 2017)	Twitter app	Tous	

Messenger semble être la plateforme la plus profitable pour héberger un chatbot, de son important nombre d'utilisateurs réguliers et de sa facilité d'accès.

Les chatbots sur Discord et Slack semblent avoir une portée moindre puisque les bots sont intégrés à des serveurs : pour communiquer avec un chatbot il faut aller sur un serveur où il est installé ou alors l'installer sur un de ses serveurs.

## Tableau récapitulatif des services de NLU

NLU	Maison mère	Coût pour 10 000 requêtes	Plateformes	Interface	Langages	Autre
-----	-------------	---------------------------	-------------	-----------	----------	-------

 Amazon Lex	Amazon	7,5 USD	Messenger et Slack (au moins)	Console	?, peut se porter sur mobile	<a href="#">Pas disponible en Europe</a>
 api.ai	Google	Gratuit	Messenger, Skype, Slack, Telegram, Twitter	Web	SDK pour : JS, Node.js; Ruby, C#, C++, Python, PHP, Java, Android, IOS	Beaucoup de doc et de samples
 IBM Watson	IBM	Gratuit jusqu'à 10000 et limité en intent et entities	Messenger et Slack (au moins)	Console	SDK pour : Node.js, Java, .NET, Python	
 Let's try LUIS.ai	Microsoft	Gratuit jusqu'à 10000, 7.5 USD / 10000 supplémentaires	Toutes	Web	Node.js, .NET, REST	Microsoft Bot Framework (voir plus bas)
 RASA	RASA	Gratuit	Toutes	Console	Python, Tous	Tourne en local ⇒ pas de HTTPS à envoyer Bonne doc
 Wit.AI	Facebook	Gratuit	Surtout Messenger	Web	SDK : Node.js, Python, Ruby	API HTTP Bonne doc

Amazon Lex et IBM Watson se distinguent par leur manque de doc.

Luis implique d'utiliser le Microsoft bot framework.

Rasa a le mérite d'être en local et n'est pas détenu par une grosse entreprise.

Api et Wit sont proches : gratuité totale, beaucoup de doc, interface web.

## Microsoft bot framework

Microsoft met à disposition des développeurs divers services pour leur faciliter la création d'un chatbot, ceci en est un bref tour d'horizon.

Un SDK -Software Development Kit- open source est à disposition en .NET ou en Node.js. Il a pour but de faciliter le développement et de généraliser le code des chatbots. Avec ce SDK sont fournis beaucoup d'exemples de bots, plus ou moins un par feature du SDK.

Microsoft propose de déployer tout bot écrit avec ce SDK vers diverses plateformes de messagerie -Courriel, SMS, Messenger, Skype, Slack et Telegram notamment- à partir de leur plateforme.

Il est possible de télécharger le Bot Framework Emulator pour tester et déboguer son bot en local.

Microsoft dispose également de nombreux services qui peuvent être utiles à un chatbot : Azure Search (Search as a Service), QnA Maker permet de transformer une FAQ en un chatbot simple, Computer Vision API permet de faire de la reconnaissance d'images.