

Auteur : Julien Margarido

27/01/2017

Equipe : Anna Kostrikova, Félix Lima Gorito, Julien Margarido

Type : **Spécification Technique et Réalisations**

Chemin : github.com/suricats/lab-bot/docs

Statut : Final

Destinataire(s) : Jonathan Alimi, Cédric Besse, Mikael Gibert,
Lamia Laraqui, Bruno Lesueur

Page(s) : 23

STR du projet “Hello Suribot”

Version	Date	Modifications	Rédacteur(s)
1.0	27/01/2017		Julien Margarido



Table des matières

1. Objectifs	3
1.1. Objectifs de la STR	3
1.2. Objectifs du projet	3
2. Services	3
2.1. Nomenclature des services	3
2.2. Définition des services	4
3. Workflow	6
4. Architecture technique	7
4.1. Choix technologiques	7
4.2. Découpage en composants	7
5. Plan de déploiement, maintenance et exploitation	9
6. WBS de la réalisation	15
6.1. Diagramme du WBS	15
6.2. Explication des tâches	16
7. Exemples d'usages de cas métiers	17
8. Conclusion	21
9. Annexes	22
9.1. Ontologie	22
9.2. Documentation	23

1. Objectifs

1.1. Objectifs de la STR

Ce document a pour but de montrer les différents composants du projet.

Il explique également l'architecture du système, en montrant les différentes interactions et dépendances.

Enfin, il a également pour but de montrer les différentes évolutions du projet, sa maintenabilité et son extensibilité.

1.2. Objectifs du projet

L'objectif du projet "Hello Suribot" est de réaliser un bot conversationnel, aussi appelé chatbot, dont les interactions concernent le domaine des assurances, ceci dans le cadre d'un P.O.C*, pour montrer la faisabilité du projet, et éventuellement intéresser d'éventuels clients.

Ce bot pourra répondre à des demandes utilisateurs, en anglais ou en français, et fournir des données provenant d'API* externes.

La conception du bot doit être un maximum modulaire, maintenable, documentée et facilement déployable sur un serveur. Il devra également disposer d'un système automatique de tests.

2. Services

2.1. Nomenclature des services

Voici la liste des services réalisés dans le projet, à partir de l'analyse des cas d'utilisations de la STBE* (dans l'ordre d'utilisation):

1. Réception d'un message utilisateur transmis par un BotConnector*
2. Transmission à une AI* du message
3. Analyse des éléments de contextualisation fournis par une AI
4. Éventuelle interrogation d'une API* externe pour récupérer des données

5. Création d'une réponse dans un langage humain, adaptée à la demande initiale de l'utilisateur
6. Mise en page et transmission au BotConnector de la réponse du bot

2.2. Définition des services

2.2.1. Réception d'un message utilisateur transmis par un BotConnector

Ce service, par l'intermédiaire du système de "Controller" proposé par SpringBoot* (cf. Partie 4.1 Choix Technologiques), reçoit dans une requête une liste d'éléments (entête, identifiants de conversation, interlocuteurs, etc...), et doit y récupérer le message et l'identifiant de l'utilisateur, et transmettre ces éléments au composant suivant.

2.2.2. Transmission à une AI du message

Ce service a pour rôle de recevoir des messages (sous forme de chaînes de caractères), de les transmettre à une AI (dans le projet, ont été étudiés Recast.ai et API.ai).

Celui-ci écoute la réponse de l'analyse fournie par l'AI, la parse dans un format compréhensible par le bot, et le transmet au composant suivant.

2.2.3. Analyse des éléments de contextualisation fournis par une AI

Ce service analyse les intents* et entities* fournis par une AI, et décide éventuellement quelle API externe contacter pour obtenir des données (dans notre cas, l'API bouchonnée* créée pour récupérer de fausses données d'assurances) et quel type de réponse envoyer (question non comprise, manque d'arguments, données, ...).

Il coordonne donc le service d'interrogation d'une API et le service de création d'une réponse.

Une fois la réponse créée, il la transmet au service d'envoi au BotConnector.

2.2.4. Éventuelle interrogation d'une API externe pour récupérer des données

Ce service interroge une API externe (ici, dans le projet, l'API bouchonnée), récupère des données sous format JSON et les retourne au composant appelant.

2.2.5. Création d'une réponse dans un langage humain, adaptée à la demande initiale de l'utilisateur

Ce service a pour rôle de former une réponse dans un langage humain (anglais, français, ...). Par un système de ressource, il doit associer les données et clefs fournies par l'API et le bot à des données textuelles, en anglais/français.

Il peut former :

- Réponse de demande non comprise
- Réponse d'erreur interne
- Réponse de demande comprise mais il manque un argument
- Réponse de demande comprise mais il manque plusieurs arguments
- Réponse de demande comprise et contenant les données voulues

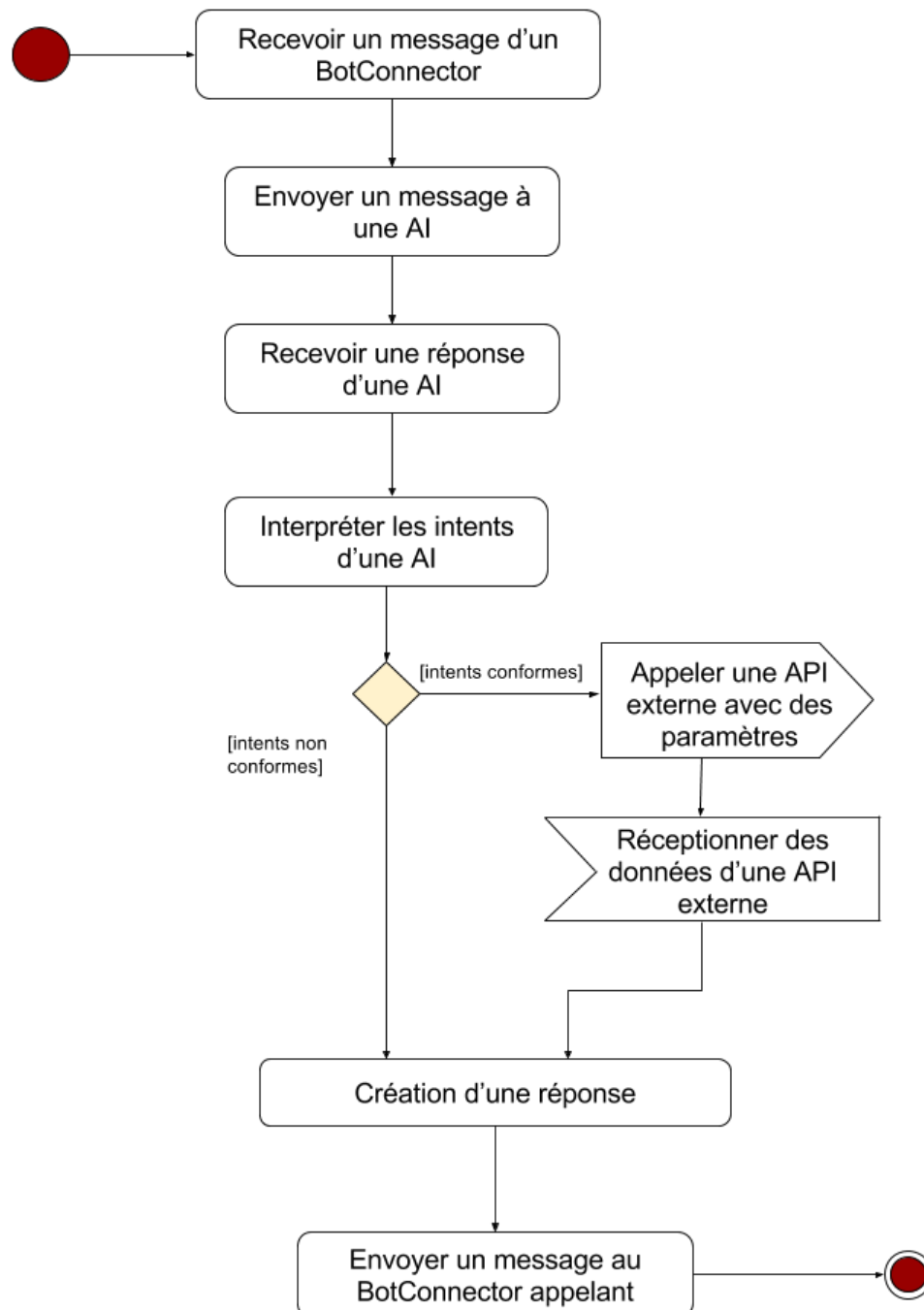
Il retourne ensuite la réponse formée au composant appelant.

2.2.6. Mise en page et transmission au BotConnector de la réponse du bot

Ce service, qui reçoit en argument la requête d'origine et la réponse du bot, a pour rôle de transmettre la réponse au BotConnector ayant envoyé la requête.

3. Workflow

Le diagramme d'activité ci-dessous montre l'utilisation et le déroulement des services dans le temps :



4. Architecture technique

4.1. Choix technologiques

Pour le projet, l'entreprise Suricats Consulting nous a mis à disposition un serveur, sur lequel doit être déployé le bot.

De par son API très riche, son nombre impressionnant de librairies tierces, son orientation POO* et sa stabilité, nous avons décidé d'utiliser le langage Java pour ce projet.

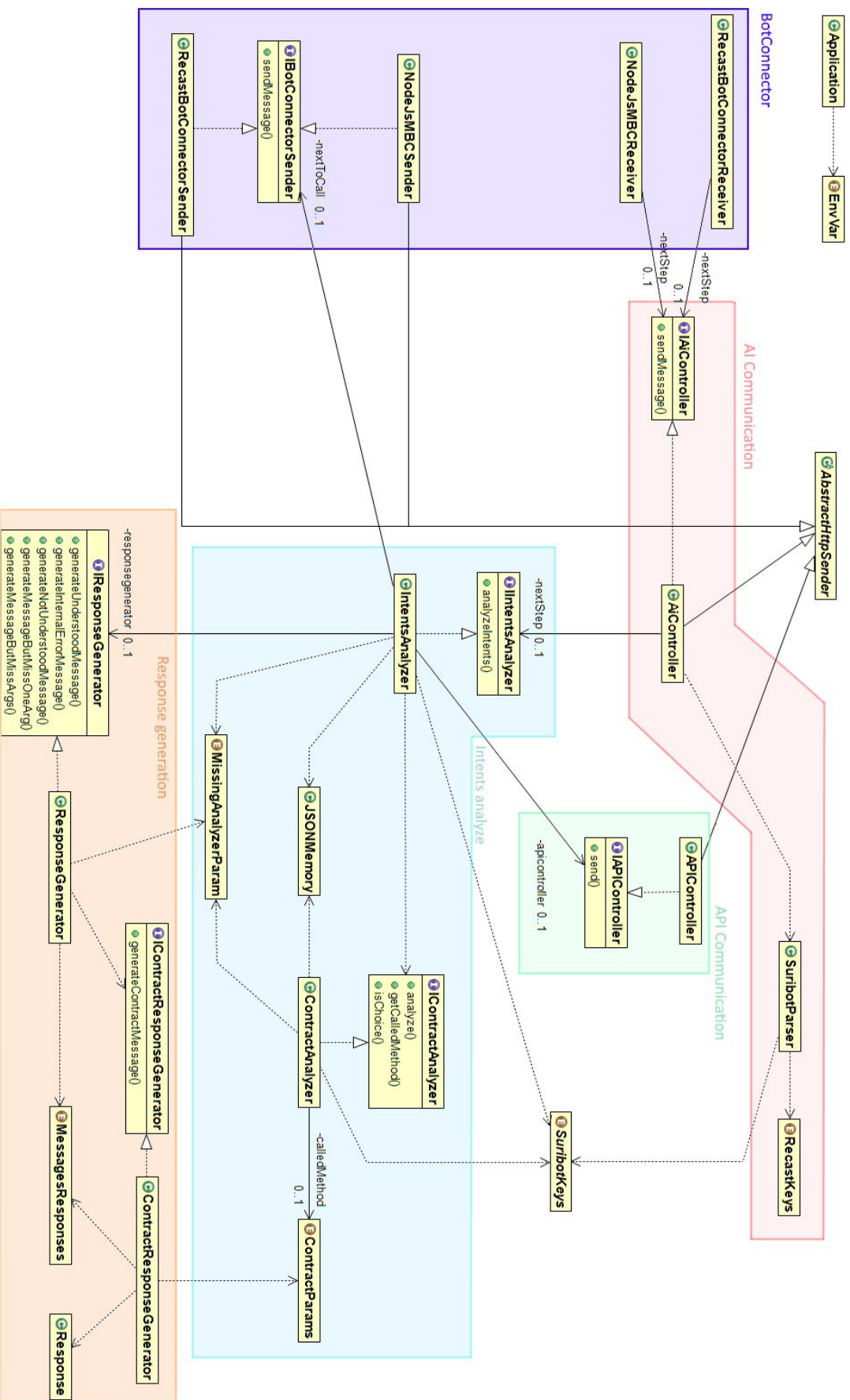
Devant créer 2 API (le bot, et une API bouchonnée), nous avons décidé d'utiliser le framework SpringBoot, qui permet rapidement de créer des backends REST, et possédant une documentation riche et claire.

Nous avons également utilisé 2 types de BotConnector (pour faire circuler les messages provenant des utilisateurs) : RecastBotConnector et MicrosoftBotConnector.

Nous avons également étudié 2 AI : Recast.ai et API.ai.

4.2. Découpage en composants

Voici un schéma d'architecture, reprenant des codes d'UML, mettant en avant les classes et composants du bot :



5. Plan de déploiement, maintenance et exploitation

Comment ajouter une intelligence artificielle

Avant de vouloir remplacer ou ajouter une nouvelle intelligence artificielle, il est important de bien configurer/entraîner celle-ci.

En effet, le remplacement de l'intelligence artificielle par défaut risque de provoquer un dysfonctionnement du chatbot si la nouvelle IA est mal configurée (intent¹ ou entities² différents sur l'intelligence artificielle et ceux définis dans le code).

L'entraînement de l'intelligence artificielle permet une meilleure compréhension des messages utilisateurs et par conséquent rendre le chatbot plus fiable.

L'entraînement de l'IA consiste à envoyer plusieurs phrases à celle-ci, et à vérifier que les données retournées soient cohérentes. Les phrases envoyées doivent être différentes de celles mises lors de la configuration. Par exemple, les phrases "Quelles sont les couvertures de l'objet 48885 ?" et "Est-ce que l'objet 48885 est couverts ?" doivent retourner le même résultat.

Après l'avoir bien configuré et entraîné, il faut ajouter l'appel à celle-ci dans la méthode "sendMessage" de la classe "AiController" du package "hello.suribot.communication.ai". Pour que le résultat retourné par l'API soit compréhensible par l' "IntentAnalyzer", il faudra ajouter une méthode dans la classe "SuribotParser" du sous-package "parser", qui transformera la réponse de l'API en un JSONObject compris par l'IntentAnalyzer (et donc par le bot).

Le JSONObject de sortie doit respecter une structure unique et contenir impérativement les mêmes clés que celles définies dans l'énumération RecastKeys, également contenue dans le sous package "parser".

Vous pouvez prendre exemple sur les autres méthodes de la classe Parser.

En résumé : l'ajout d'une nouvelle intelligence artificielle impacte la modification de 1 composant et plus particulièrement de 2 classes :

- "AiController" du package "hello.suribot.communication.ai"
- "SuribotParser" du package "hello.suribot.communication.ai.parser"

La durée estimée pour l'ajout ou le remplacement d'une intelligence artificielle prend 1,5 jour-homme.

Comment ajouter une api

L'ajout d'une api va entraîner une modification du code, mais également une modification du paramétrage de l'intelligence artificielle utilisée (Recast.ai, API.ai, ...).

La première étape (paramétrage) à effectuer pour l'ajout d'une nouvelle api est de définir correctement *l'intent* et l'ensemble des *entities* correspondant aux actions que l'on souhaite effectuer.

Une fois ceci fait, il faut les intégrer dans l'intelligence artificielle puis entraîner celle-ci pour qu'elle soit suffisamment performante.

La deuxième étape consiste à intégrer l'intent et les entities définies dans l'intelligence artificielle au chatbot. Pour ce faire, il faut créer un nouveau package, dans le package "hello.suribot.analyze" correspondant à l'intent créé. On y ajoutera la classe qui s'occupe d'analyser les entities reçus par l'intelligence artificielle. On peut prendre exemple sur la classe ContractAnalyzer.

L'analyseur créé devra retourner un JSONObject, qui signale si la demande a été comprise ou non. Dans le cas où elle n'a pas été comprise, le JSONObject indiquera si il manque des arguments dans la demande de l'utilisateur, et si il en manque, lesquels.

L'analyseur devra garder en mémoire la méthode appelée, et également si celle-ci retourne un résultat qui correspond à un choix ou non. Ces 2 arguments permettront par la suite de générer des réponses cohérentes à l'utilisateur.

La prochaine étape consiste à créer un nouveau générateur de réponse. Pour ce faire, il faut créer un nouveau package, qui correspond au nouvel intent, dans le package "hello.suribot.response". Dans ce package il faudra créer une nouvelle classe, qui retournera un objet Response qui sera compris par la classe qui enverra les réponses au bot connector.

Les messages à envoyer à l'utilisateur devront être ajoutés dans les fichiers "MessageBundle.properties" du package "main.ressources.message", et les clés de ces messages dans l'énumération "MessagesResponses" du package "hello.suribot.response". Ce dernier ajout permet de faire le lien entre le générateur de réponse, et les fichiers properties. Ceci permet d'éviter des fautes de frappe, ou autres erreurs orthographiques entre les clés entrées dans les fichiers properties et les générateurs de réponse.

Une fois l'analyseur et le générateur de réponses implémentés, il faut désormais les intégrer dans l' "IntentsAnalyzer" et dans le "ResponseGenerator".

Dans la méthode "analyzeIntents" de la classe "IntentsAnalyzer", il faut ajouter un cas à la condition qui vérifie le contexte (intent). Celui-ci devra vérifier que l'intent retourné par l'IA correspond à celui de la nouvelle API, et si oui appeler l'analyseur créé à l'étape 2, puis le générateur de réponse.

Dans la méthode "generateUnderstoodMessage" de la classe "ResponseGenerator" du package "hello.suribot.analyze", il faut également ajouter un cas, comme pour l'analyseur, qui va comprendre le contexte et appeler le nouveau générateur de réponse créé à l'étape précédente.

En résumé : L'ajout d'une nouvelle API va impacter 2 composants ("Analyse des intents" et "Génération de réponses") et également le paramétrage de(s) intelligence(s) artificielle(s).

Il entraîne également la création de 2 packages XXXX, un qui contiendra la classe XXXXAnalyzer et l'autre qui contiendra la classe XXXXResponseGenerator (XXXX étant le nom de l'intent).

La durée estimée pour l'ajout d'une nouvelle API est de 3,5 jour-homme.

Comment ajouter un bot connector

Si vous désirez ajouter un "bot connector" (comme Recast Bot Connector, ou Microsoft Bot Connector) pour faire le lien entre votre bot et plusieurs channels (Slack, Skype, Messenger, ...), cela se fait en 2 étapes.

La première consiste à configurer le bot connector avec les différents channels de communication, mais surtout le "endpoint", c'est-à-dire l'url de destination des messages (le serveur/machine hébergeant le bot).

Une fois ceci fait, les messages des utilisateurs transiteront du channel, au Bot Connector, pour arriver finalement à la machine hébergeant votre bot.

La seconde et dernière partie consiste en l'ajout des composants de communication avec ce nouveau bot connector. Vous pouvez prendre exemple sur les classes du package rbc. Ainsi, il est nécessaire de créer un sous-package dans le composant "Communication avec Bot Connector" du bot (package communication.botConnector). Ce sous-package aura pour nom le diminutif du nouveau bot connector que vous souhaitez ajouter.

Vous devrez y créer 2 nouvelles classes.

La première sera le "Receiver", qui recevra les requêtes. Il sera sensiblement le même que les 2 Receivers existants (RecastBotConnectorReceiver et NodeJsMBCReceiver). Il devra simplement changer la BotConnectorIdentity.

La seconde est le "Sender", qui retournera au Bot Connector les réponses du bot. Il devra adapter la mise en page (saut de lignes, boutons, etc...) au Bot Connector en question, ainsi que le moyen d'envoyer la requête sur le réseau.

D'autre part, il sera nécessaire de :

- modifier l'énumération BotConnectorIdentity pour y ajouter une identité au nouveau Bot Connector.
- modifier le *switch case* de la méthode getSender dans la classe IntentsAnalyzer (package analyze), pour y ajouter la nouvelle instantiation du nouveau "Sender".

En résumé : l'ajout d'un nouveau "Bot Connector" impacte 2 composants (communication avec les BotConnector et Analyseur d'intents).

Il entraîne également la création d'un package, et la création de 2 classes :

- un "XXXReceiver", où XXX est le nom du Bot Connector.
- un "XXXSender", où XXX est le nom du Bot Connector.

Il entraîne également la modification de 2 classes :

- BotConnectorIdentity
- IntentsAnalyzer

La durée estimée pour l'ajout ou le remplacement d'un Bot Connector est de 1/2 jour-homme.

Manuel d'installation

Avant de vouloir lancer le chatbot, il faut vérifier que la machine sur laquelle vous souhaitez le lancer possède les variables d'environnement nécessaires. Elles sont aux nombres de 7 : RBCSLUG, RCBOTID, RBCTOKEN, BOTPORT, CONTRACTPORT, TOKENRECAST, TOKENAPIAI et NODEJSPORT.

La machine d'exécution doit posséder au minimum java 1.8. Pour installer java 8, utilisez la commande "sudo apt-get install oracle-java8-set-default".

Une fois cette vérification effectuée, vous avez deux possibilités pour lancer le bot et l'API bouchonnée que nous avons créée. La première consiste à les lancer manuellement en utilisant les commandes suivantes :

- java -jar -Dserver.port=\$BOTPORT lab-bot-1.0.jar & (pour lancer le bot)
- java -jar -Dserver.port=\$CONTRACTPORT mock-api-1.0.jar & (pour lancer l'API)

La deuxième possibilité consiste à utiliser l'intégration continue, et plus particulièrement l'outil Travis, que nous avons intégré à notre projet. Pour pouvoir l'utiliser, vous devez forker le projet depuis github (création d'un nouveau "repository" sur votre github) et l'importer sur votre machine, puis crypter les valeurs contenues dans le fichier ".travis.yml".

Pour ce faire, vous devez disposer de "Travis" sur votre machine, et par conséquent de Ruby. Pour crypter les variables, vous devez lancer l'invite de commandes Ruby, vous déplacez jusqu'à la racine du projet que vous avez importé, puis utiliser la commande : "travis encrypt NOMVARIABLE=VALEURVARIABLE --add".

Une fois les variables cryptées pour votre dossier github précédemment téléchargé, vous devez ajouter votre projet à l'outil "Travis" en allant sur le site <https://travis-ci.org/profile>: ainsi chaque commit provoquera le déploiement automatique par Travis de la nouvelle version du projet vers la machine cible, suivant les instructions renseignées dans le fichier travis.yml.

NB: Si vous souhaitez passer de Recast Bot Connector à Microsoft Bot Connector (les 2 ne pouvant pas tourner simultanément sur le même port), votre machine doit avoir node.js d'installé.

Pour ce faire il faut utiliser les 2 commandes suivantes : "sudo apt-get install nodejs npm" puis "sudo apt-get install node".

Il faut modifier également les variables d'environnement APPID, APPSECRET, BOTPORT et CONTRACTPORT (variables d'environnement sur la machine, et dans l'enum EnvVar.java).

De plus, il faut compiler le fichier node.js/app.js, en lançant “npm install” et “node app.js” et en ajoutant ces deux commandes au fichier travis.yml.
Enfin, il faut modifier la variable “PORT” du fichier “.travis.yml” en lui affectant une nouvelle valeur, correspondant à celle définie dans les variables d’environnement (BOTPORT).





Quelques mots sur la maintenance :

Nous avons implémentés un système de logs* journaliers, reportant toutes les traces et erreurs apparues sur le Bot et sur l’API.

Chaque log indique la classe concernée, la date, un message, et éventuellement une “stack trace*” s’il y a une erreur.

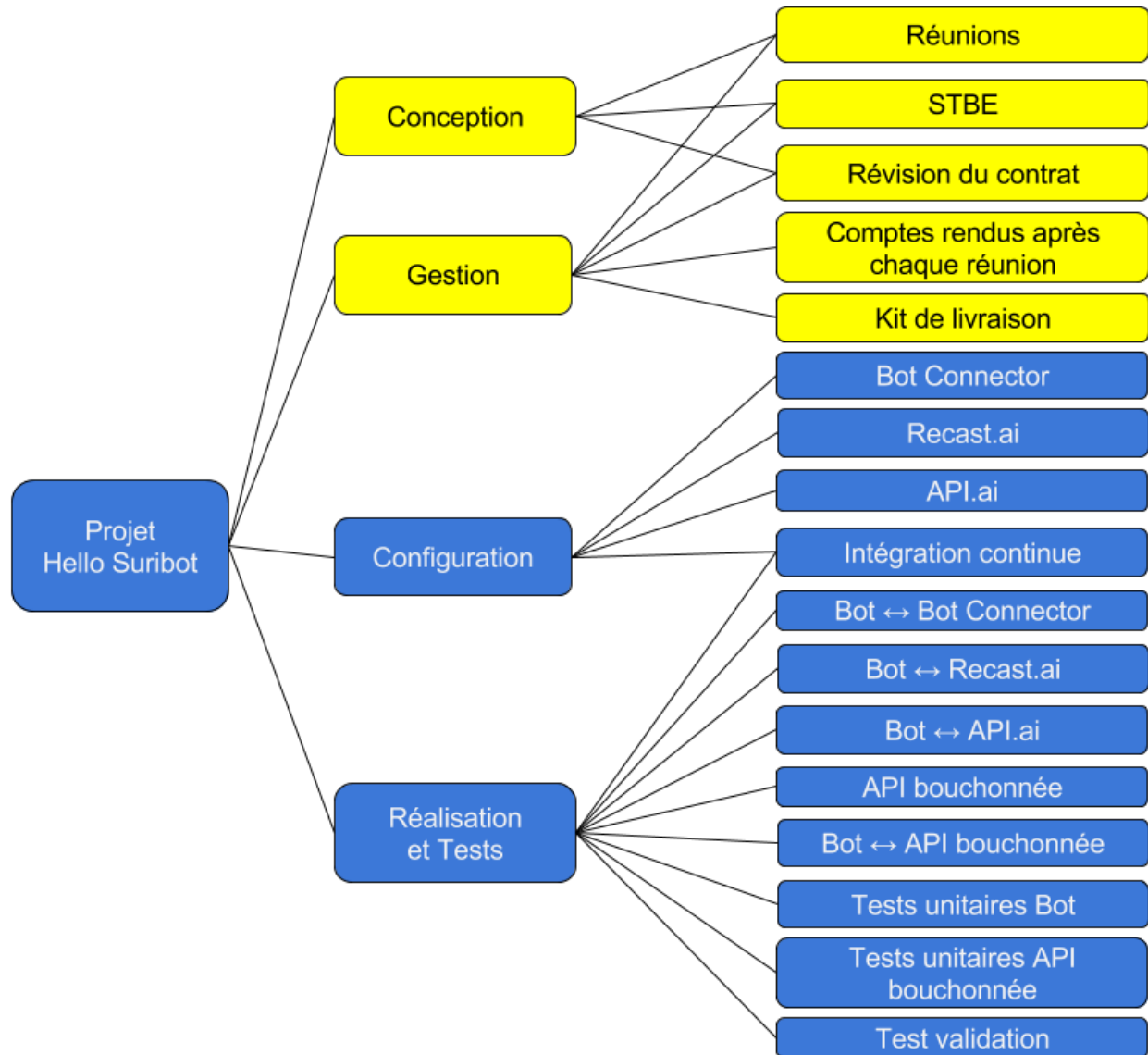
Ces logs sont enregistrés sur le serveur, dans le dossier logs, au même endroit où se trouvent les jars*.

Les logs apparaissent également sur la console.

 suribot.log	8 KB	25/01/2017 09:41:47
 suribot_api.log	1 KB	25/01/2017 09:41:34
 suribot_api-2017-01-24.log	1 KB	24/01/2017 16:44:18
 suribot-2017-01-24.log	3 KB	24/01/2017 16:44:17
 suribot_api-2017-01-23.log	1 KB	23/01/2017 17:06:08
 suribot-2017-01-23.log	4 KB	23/01/2017 17:04:05
 suribot-2017-01-22.log	11 KB	22/01/2017 19:57:25
 suribot_api-2017-01-22.log	1 KB	22/01/2017 19:37:36
 suribot-2017-01-21.log	10 KB	21/01/2017 22:47:46
 suribot_api-2017-01-21.log	2 KB	21/01/2017 22:45:46

6. WBS de la réalisation

6.1. Diagramme du WBS:



6.2. Explication des tâches

Configuration Bot Connector :

La configuration des différents BotConnector (Microsoft BotConnector et Recast BotConnector) consiste principalement à relier l'«endpoint*» (adresse du serveur sur lequel fonctionne le bot) aux BotConnector, et à relier les app* de différentes messageries aux Bot Connector.

Configuration Recast.ai :

La configuration de Recast.ai consiste à intégrer l'ensemble des entités et des intents à l'intelligence artificielle et à entraîner celle-ci.

Pour plus de détails sur la configuration, consultez la partie « Comment ajouter une intelligence artificielle » du manuel « Manuels HelloSuribot »

Configuration API.ai :

La configuration de API.ai consiste à intégrer l'ensemble des entités et des intents à l'intelligence artificielle et à entraîner celle-ci.

Configuration/Réalisation intégration continue :

Pour l'intégration continue, nous avons créé un fichier .travis.yml à la racine de notre projet, dans lequel nous définissons l'ensemble des tâches à effectuer à chaque commit*. Une fois ce fichier créé, nous avons lié notre projet à l'outil Travis* pour que chaque commit provoque un déploiement automatique suivant le fichier .travis.yml.

Connexion entre le bot et Bot Connector :

Pour réaliser cette tâche nous avons créé un composant qui réceptionnera les données (format JSON) reçues par les Bot Connector. Une fois ces données reçus, il les transmettra au composant suivant « AI communication » avec le message de l'utilisateur extrait ainsi que son identifiant. Le JSON sera récupéré à la fin du traitement effectué par le bot, afin d'envoyer la réponse générée au bot connector et ainsi à l'utilisateur.

Connexion entre le bot et les intelligences artificielles (Recast.ai et API.ai) :

L'appel à ces 2 API nécessitant des identifiants uniques, nous avons stockés ceux-ci en variable d'environnement pour qu'elles restent privées et n'apparaissent pas en clair sur les « repository » github.

Une fois ces variables stockées, nous avons créé un composant qui appellera l'intelligence artificielle correspondante, puis transformera la réponse en un JSON compréhensible par le composant d'analyse d'intents.

Création d'une API bouchonnée puis connexion avec le bot :

Pour la création de l'API bouchonnée, nous avons décidé d'utiliser SpringBoot. Celui-ci nous a permis de créer une API REST répondant à un usage de cas métiers en lien avec les assurances.

Pour la connexion entre notre API bouchonnée et notre bot, nous utilisons le composant "API Controller", qui va s'occuper de communiquer avec les différentes API en fonction des paramètres envoyés par l'analyseur d'intent.

Pour tester l'ensemble des fonctionnalités que nous avons créées, nous avons mis en place un processus automatique et un protocole de test.

Chaque commit* effectué va provoquer le lancement automatique des tests unitaires que nous avons implémentés, et si l'un d'eux ne passe pas, le déploiement est interrompu.

Ensuite, nous testons "à la main" une suite de phrase envoyées au bot pour vérifier que celui-ci nous retourne toujours des réponses en adéquation avec le besoin du client.

7. Exemples d'usages de cas métiers

Voici un tableau montrant quelques exemples de questions/demandes accompagnées de leurs réponses (toutes les données étant de fausses données provenant de l'API bouchonnée) :

Demande	Réponse
"Puis-je obtenir des informations sur mon contrat ?" "Contrat" "Donne-moi des infos sur mon contrat s'il te plait !"	"Il manque un argument à votre demande : votre identifiant de contrat."
"Contrat 699522" "Contrat n°5985680" "Puis-je obtenir des informations sur mon contrat d'identifiant 2748482 ?"	"Veuillez préciser si votre demande concerne : vos prélèvements vos couvertures une personne"

<p>“Quels sont les propriétaires de mon contrat n°856584 ?”</p> <p>“Donne-moi les proprios de mon contrat d’identifiant 896986”</p>	<p>“De quelle personne parlez-vous ?</p> <p>Thomas Dupont - 758598</p> <p>Eric Elcor - 970079”</p> <p><i>(chaque personne étant un bouton cliquable)</i></p>
<p><i>Si l'utilisateur clique sur le bouton d'une personne ou si il écrit directement une phrase type :</i></p> <p>“Donne-moi des informations la personne n°585 de mon contrat n°4848”</p>	<p>“Voici les informations sur la personne :</p> <p>date de fin de contrat : 2016-12-25</p> <p>numéro client : 5639358</p> <p>date de naissance : 1994-12-05</p> <p>nom : dupont</p> <p>code postale : 93700</p> <p>prénom : thomas</p> <p>identifiant : 78595”</p>
<p>“Quels sont les prélèvements de mon contrat n°856584 ?”</p>	<p>“De quel prélèvement parlez-vous ?</p> <p>prélèvement n°970079</p> <p>prélèvement n°985959”</p> <p><i>(chaque prélèvement étant un bouton cliquable)</i></p>
<p><i>Si l'utilisateur clique sur le bouton d'un prélèvement ou si il écrit directement une phrase type :</i></p> <p>“Donne-moi des informations le prélèvement n°585 de mon contrat n°4848”</p>	<p>“Voici les informations sur le prélèvement :</p> <p>moyen de paiement : chèque</p> <p>montant : 542.97</p> <p>identifiant : 123987456</p> <p>fréquence : hebdomadaire</p> <p>prochain prélèvement : 2017-11-10”</p>
<p>“Quels sont les couvertures assurées par mon contrat n°336 ?”</p>	<p>“De quelle couverture d'objet parlez-vous ?</p> <p>objet n°970079</p> <p>objet n°985959”</p> <p><i>(chaque objet étant un bouton cliquable)</i></p>
<p><i>Si l'utilisateur clique sur le bouton d'un objet ou si il écrit directement une phrase type :</i></p> <p>“Donne-moi des informations la couverture n°985959 de mon contrat n°4848”</p>	<p>“Voici les informations sur la couverture de l'objet :</p> <p>incendie : non assuré</p> <p>vandalisme : assuré</p> <p>inondation : assuré”</p>
<p>“Donne-moi le rib associé à mon contrat”</p>	<p>“Voici votre RIB :”</p> <p><i>suivi d'une photo du RIB</i></p>
<p><i>En cas de non-compréhension du message</i></p>	<p>“Veuillez reformuler votre question”</p>

En cas de problème/maintenance interne au bot

“Problème interne au bot.
Nous essayons actuellement de le régler.
Veuillez nous excuser pour la gêne occasionnée.”



felix.gorito 10:07 PM

Bonjour quelles informations puis-je avoir pour mon contrat ?



lab_bot_suricats 10:07 PM

Il manque un argument à votre demande : votre identifiant de contrat



felix.gorito 10:07 PM

J'ai le contrat 1289648



lab_bot_suricats 10:07 PM

Veuillez préciser si votre demande concerne :

vos prélèvements
vos couvertures
une personne



lab_bot_suricats 9:07 PM

De quel prélèvement parlez-vous ?

prélèvement n°96268885

prélèvement n°2355596664

prélèvement n°123987456

Voici les informations sur le prélèvement :

moyen de paiement : prelevement
montant : 123.97
identifiant : 96268885
fréquence : annuel
prochain prélèvement : 2016-11-10

8. Conclusion

Ce projet est disponible sur Github, sur le compte de Suricats : github.com/suricats/lab-bot-api et github.com/suricats/lab-bot.

En quelques chiffres, le projet comporte :

- plus de 180 tests unitaires
- 265 commits
- environ 3300 lignes de codes produites pour le bot
- environ 1300 lignes de codes produites pour l'API

L'équipe de développement remercie l'entreprise Suricats Consulting, et particulièrement Jonathan Alimi et Mikael Gibert, pour leur encadrement et leurs conseils qui ont permis à ce projet de se dérouler dans d'excellentes conditions.

Nous remercions également nos encadrants de l'UE PiSTL, de l'Université Pierre et Marie Curie, qui sont Cédric Besse, Lamia Laraqui et Bruno Lesueur.

9. Annexes

9.1. Ontologie

STBE = Spécification Technique des Besoins et Exigences

BotConnector = Outil permettant de faire le lien entre différentes plateformes de messages et des bots

AI = Intelligence Artificielle

API = Interface de programmation applicative (Application Programming Interface)

SpringBoot = Framework de développement permettant la création d'application Spring

Intents = Paramètre donnant le contexte d'une phrase (ex: contrat, banque, transport, ...)

Entities = mot-clé extrait d'une expression obtenu lors du décryptage d'une phrase (ex: rib, couverture, prélèvement, numéro de contrat, ...)

API bouchonnée = API

POO = Programmation Orienté Objet

Endpoint = Url de destination

App = Nom de la configuration d'un bot sur Slack et Messenger

Commit = Action propre à github permettant de mettre à jour le code d'un répertoire en ligne.

Travis = Logiciel libre permettant le déploiement automatique

9.2. Documentation

- <https://botconnector.recast.ai/>
- <https://travis-ci.org/>
- <https://recast.ai/>
- <https://blog.recast.ai/nodejs-bot-tutorial/>
- <https://chatbotsmagazine.com/the-complete-beginner-s-guide-to-chatbots-8280b7b906ca>
- <https://docs.botframework.com/en-us/>