Rapport de stage de deuxième année

Création d'application d'alerting Microsoft Teams

Du 15 novembre au 31 décembre 2021

Tuteur pendant le stage: M. Pierre Kauffmann
Tuteur académique: M. Michel Demède
Stagiaire: Carl Fremault

Formation BTS SIO option SLAM au CNED

Sommaire

1. Introduction

Pour mon stage de deuxième année du BTS Services Informatiques aux Organisations, option Solutions Logicielles et Applications Métiers, j'ai eu l'opportunité d'intégrer l'équipe de développeurs full-stack de la start-up fAIbrik, basée à Annecy en Haute-Savoie, pour une durée de sept semaines, du 15 novembre au 31 décembre 2021.

2. Contexte

2.1 Présentation de l'entreprise

L'entreprise fAIbrik est une jeune start-up créée par deux experts en Intelligence Artificielle. Sa mission est de proposer des solutions IA pour la gestion du service client auprès des PME, pour qui le coût d'investissement dans des solutions existantes (Salesforce, Zendesk, ...) est souvent hors de portée.

Dans notre société de plus en plus digitalisée, les clients sont plus facilement amenés à contacter leurs prestataires, et ce sur des canaux de plus en plus divers. Il n'est pas rare, même pour une PME, de recevoir mille mails par jour, et plusieurs centaines d'appels téléphoniques. À cela s'ajoutent les sollicitations par les différents réseaux sociaux, dont le nombre ne cesse de croître.

L'entreprise qui sait gérer et répondre à toutes ces demandes aura indéniablement une avantage par rapport à la concurrence.

2.2 L'application fAIbrik

fAIbrik propose une application web qui permet aux entreprises de regrouper l'ensemble des demandes clients dans un seul endroit. Dès qu'une communication entrante est signalée l'IA l'analyse et détecte entre autres la langue, le sujet, l'urgence et même le degré de satisfaction de l'émetteur. Ceci est un vrai exploit comme notamment les messages reçus sur les réseaux sociaux ne respectent pas forcément ni les formalismes ni les règles de grammaire ou de l'orthographe.

Une conversation peut être prise en charge par un agent (employée de l'entreprise utilisateur de la plateforme), elle peut être déléguée, ou l'agent peut la faire remonter

auprès d'un superviseur. Les différentes échanges (texte, audio, pièces jointes, ...) sont gardées puis archivées quand la demande est résolue.
La multitude de canaux proposés est une des forces de l'application. Les clients peuvent contacter les entreprises par mail, téléphone, SMS, Whatsapp, Facebook, ... Plutôt que d'attendre qu'un nouveau canal à la mode devient pertinent dans la relation client d'une entreprise, les dirigeants préfèrent anticiper et prévoir son intégration dans l'application, stratégie fructueuse pendant les négociations avec un nouveau client potentiel.

2.3 Objectif du stage

Dans ce cadre, la mission de stage qui m'a été confiée est de faire une application Microsoft Teams qui permet aux utilisateurs de cette plateforme de recevoir des alertes lors d'un évènement dans l'application fAIbrik, par exemple quand ils reçoivent une nouvelle réponse.

3. Le stage

3.1 La mission

L'objectif du stage était de développer une application Microsoft Teams pour faire un lien avec la plateforme fAIbrik. Ainsi, les clients de fAIbrik pourront décider d'installer cette application dans leur compte Teams pour recevoir des notifications lorsqu'un nouvel événement se produit sur la plateforme fAIbrik, par exemple quand ils recevront un nouveau message.

C'était un projet très complet avec tout d'abord une phase d'étude et d'analyse des possibilités de la plateforme Microsoft Teams, une phase de conception et finalement une phase de développement et de tests.

3.2 Phase 1 : Étude et analyse

3.2.1 Documentation

Comme évoqué ci-dessus, le cas d'utilisation de l'application est assez simple : après avoir installé l'application, un utilisateur doit pouvoir s'authentifier auprès de la plateforme fAIbrik puis souscrire aux alertes. Il doit pouvoir recevoir des alertes provenant de fAIbrik puis bien sûr, s'il le désire, se désabonner et déconnecter de son compte fAIbrik.

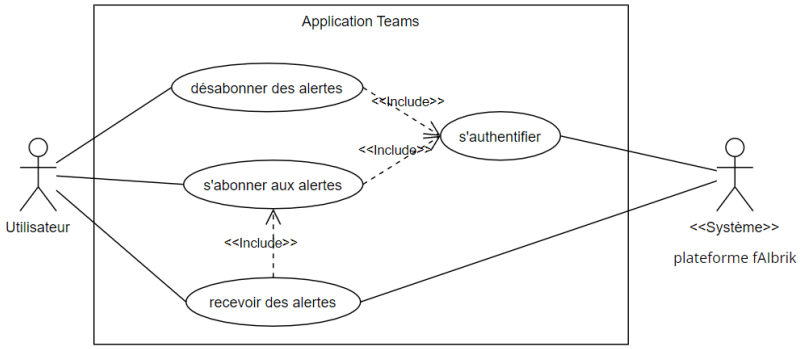

Figure 1: Diagramme de cas d'utilisation

En début de stage j'ai commencé par étudier la documentation de Microsoft Teams pour bien comprendre les différentes possibilités et leurs cas d'utilisation correspondantes. Effectivement, une application Teams peut prendre de différentes formes qui peuvent être regroupés sous quatre catégories principales (fig. 2) :

- Onglets ('Tabs') : un onglet Teams peut être intégré dans un 'channel', dans un 'chat' ou dans une 'meeting' pour présenter des données, des graphiques, voire une application web entière.
- Extension de messagerie : ce type d'application Teams propose des fonctionnalités par lesquelles un utilisateur peut enrichir les messages envoyés dans les chat.
- 'Webhooks & connectors' : permettent de recevoir des notifications dans une fenêtre de chat.
- 'Bots' : des applications de conversation qui peuvent effectuer un ensemble de tâches.

Au premier abord les 'connectors' semblent parfaitement répondre aux besoins. Par ailleurs, fAIbrik utilise déjà ce système en interne pour recevoir des alertes lorsque l'application fAIbrik génère des messages d'erreur. Toutefois leur fonctionnalité est limitée, et cette solution n'est donc pas retenue, d'autant plus que ultérieurement il est possible que l'application Teams sera développé davantage pour proposer plus de services aux utilisateurs.

La solution choisie est de développer un bot. Cela permettra d'interroger l'utilisateur par rapport à son choix d'abonnement (limité au début, mais qui pourra évoluer par la suite) et envoyer des messages 'proactives' (c.à.d. sans sollicitation de la part de l'utilisateur) lorsqu'un événement se produit au sein de l'application fAIbrik.

Figure 2: https://docs.microsoft.com/en-us/microsoftteams/platform/concepts/capabilities-overview

Par la suite je me suis documenté par rapport aux fonctionnalités et au développement d'un bot. La documentation Microsoft Teams renvoie vite vers la documentation de Microsoft Azure Bot Service, qui renvoie à son tour vers la documentation du Microsoft Bot Framework SDK, qui finit par renvoyer vers la documentation Teams ...

La préparation d'une petite présentation Powerpoint à l'issue de la deuxième semaine du stage m'a permis de mettre les choses au clair, que ce soit par rapport aux termes utilisés ou le fonctionnement interne d'un bot Microsoft Teams / Azure.

3.2.2 Essais

Par ailleurs les ressources proposées par Microsoft comprennent des dépôts Github avec des exemples concrets concernant les différents fonctionnalités. Ainsi, avant d'entamer le projet même, j'ai pu faire des tests et me familiariser avec les différents concepts. Notamment pour la partie authentification il m'a semblé judicieux de partir de l'exemple Microsoft (sous licence MIT qui donne le droit illimité d'utiliser, copier, modifier et distribuer le code tant que la source est indiquée). Ainsi j'évite d'introduire des failles de sécurité au niveau de la plateforme fAIbrik.

3.3 Phase 2 : Conception

3.3.1 Structure de l'application

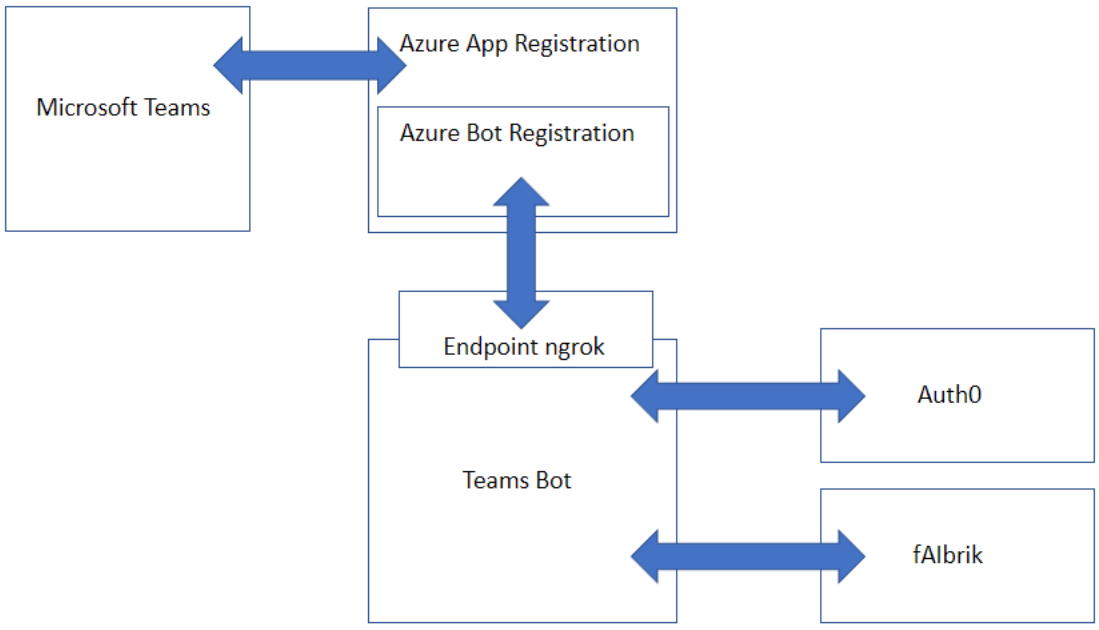

Figure 3: Structure de l'application

Le développement d'un bot pour Microsoft Teams implique l'utilisation de la plateforme cloud de Microsoft : Azure. Une 'installation' d'une application Teams ne consiste en rien

de plus que deux icônes ainsi qu'un fichier 'manifest.json' dans lequel se trouve notamment le 'Microsoft App Id' qui pointe vers l'enregistrement d'une application dans le cloud Azure. Dans notre cas, l'application Azure va contenir l'enregistrement d'un bot Azure.

Pendant la phase de développement la code du bot réside sur l'ordinateur du développeur, le bot tourne en local puis Teams y accède (par intermédiaire d'Azure) par un 'tunnel'. Pour générer ce tunnel j'ai utilisé le logiciel Ngrok qui permet à ses utilisateurs de proposer un service local sur internet en générant une URL (temporaire pour les comptes gratuits), lié à un port réseau de l'ordinateur. L'URL généré (le 'endpoint') est ensuite déclaré dans la configuration Azure du bot.

Cette architecture est intéressante comme fAIbrik préfère héberger le bot chez leur fournisseur habituel, plutôt que sur Azure comme proposé par Microsoft. Au lieu du endpoint qui dirige vers un tunnel Ngrok pendant le développement, une fois que le déploiement sera fait on peut donc très bien utiliser un endpoint qui pointe vers le lieu d'hébergement choisi. Le code du bot pourra donc facilement être intégré dans la solution fAIbrik existante à condition qu'il expose un endpoint accessible publiquement.

Un autre élément de l'architecture du bot est l'authentification. fAIbrik travaille avec Auth0, plateforme d'authentification qui utilise le protocole OAuth 2.0. La configuration de la connexion avec Auth0 est faite dans l'enregistrement du bot sur Azure. Ensuite, lorsque l'utilisateur demande au bot de se connecter, après authentification Auth0 retourne un 'opaque token', qui est une chaîne de 32 caractères. Ce token permet ensuite de demander un 'JSON Web Token' auprès de Auth0 qui contient plus d'informations concernant l'utilisateur, dont son identifiant duquel on va se servir pour pouvoir inscrire l'utilisateur aux alertes fAIbrik.

3.3.2 Fonctionnement d'un bot Azure

Le concept de base du fonctionnement d'un bot Azure est le 'turn', un 'tour de conversation'. En général un turn représente une activité entrante de l'utilisateur et la réponse renvoyée par le bot. Exception sur la règle, dont je vais me servir, sont les messages 'proactives' qui sont envoyés par le bot sans activité requise de la part de l'utilisateur.
Tout interaction entre l'utilisateur et le bot est appelé 'activity' et est géré par les 'activity handlers', par exemple 'onMessage' et 'onMembersAdded' qui sont déclenchés respectivement lors de la réception d'un message, ou quand un nouvel utilisateur est ajouté à une conversation.

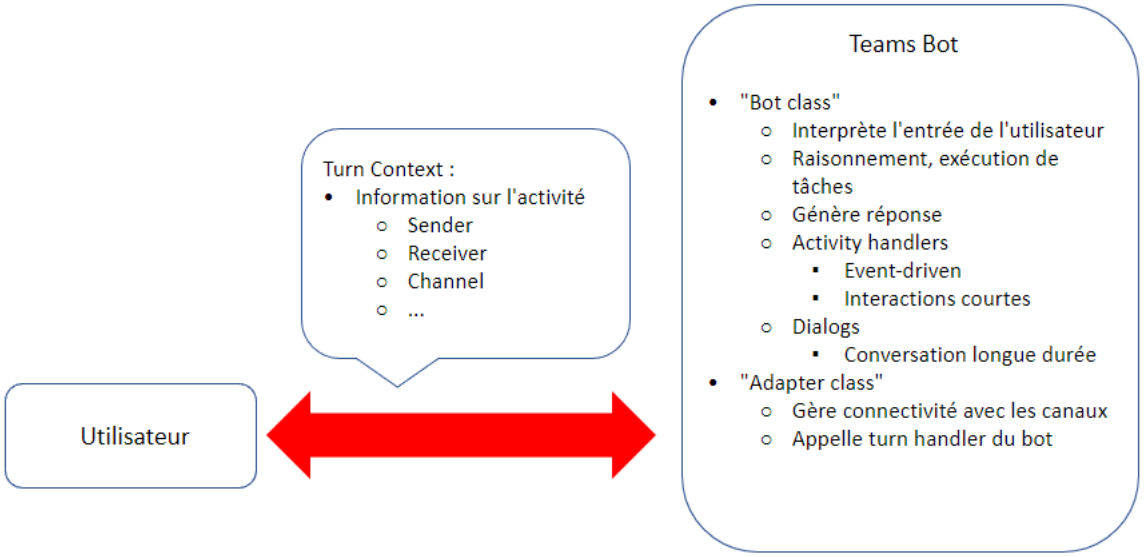

Figure 4: Fonctionnement basique d'un bot Azure

Chaque turn a son propre 'turn context' qui permet de mémoriser les propriétés du turn : émetteur, destinataire, l'id du canal, ... Chaque bot comporte au minimum deux classes: la classe 'bot' qui contient la logique pour gérer les conversations et la classe 'adapter' qui gère la connectivité avec les canaux, dans notre cas Microsoft Teams.

La gestion d'une conversation est faite par intermédiaire de dialogues ('dialogs'). Chaque dialogue représente une tâche de conversation. Un dialogue est similaire à une fonction, il peut recevoir des paramètres et il peut retourner une valeur. Un dialogue est l'unité de base de contrôle du 'workflow'. Il peut démarrer, arrêter, terminer, être mis en pause ou continuer, ou encore être annulé. Un dialogue s'exécute sur un ou plusieurs 'turns'.

Le Bot Framework propose plusieurs types de dialogues. Les types qui nous intéressent le plus, sont les 'component dialog', les 'waterfall dialog' et les 'prompt dialog'.

- Un 'component dialog' est un conteneur qui contient d'autres dialogues et qui les permet de interagir entre eux.
- Un 'waterfall dialog' (dialogue en cascade) contient plusieurs étapes ('steps'). Il se prête à une exécution linéaire de tâches. Chaque étape exécute ses tâches puis peut retourner une valeur à l'étape suivant.
- Un 'prompt dialog' demande une information de l'utilisateur, tel une valeur, ou une réponse à une question. Il peut être intégré dans un 'waterfall dialog'.

Les différents dialogues travaillent ensemble, et fonctionnent comme un 'stack'. Un dialogue démarre et est ainsi en haut du stack. Il peut lancer un autre dialogue, du coup le premier dialogue sera mis en suspension, le dialogue appelé se pose au dessus dans le stack et est exécuté. Lorsque le dialogue en haut du stack termine, le dialogue en dessous

continue son exécution. A chaque instant l'enchaînement des dialogues peut être interrompu.

Chaque dialogue possède un 'dialog context' permettant de stocker les variables pertinentes à la conversation.

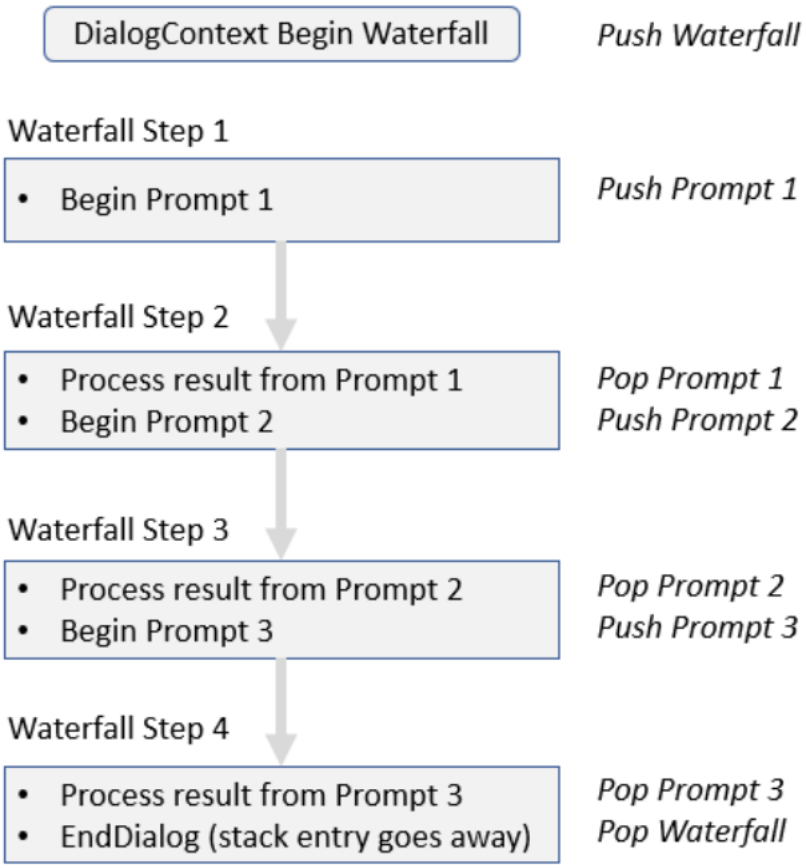

Figure 5: Séquence d'étapes dans un 'waterfall dialog' puis fonctionnement de stack (https://docs.microsoft.com/en-us/azure/bot-service/bot-builder-concept-waterfall-dialogs?view=azure-bot-service-4.0)

3.3.3 Fonctionnement du bot fAIbrik

Trois aspects principaux seront à intégrer dans l'application : l'authentification, la souscription et désinscription aux alertes, puis la réception des messages. On doit également s'occuper de la gestion des utilisateurs inscrits aux alertes.

Authentification

Bien entendu, afin de pouvoir s'inscrire aux alertes l'utilisateur doit être client chez fAIbrik. De plus, le bot aura besoin de son identifiant fAIbrik pour pouvoir lier la souscription aux alertes au bon utilisateur. Concrètement il s'agît d'un simple processus d'authentification, l'implémentation avec Auth0 a été décrit ci-dessus (cf. 3.3.1 Structure de l'application).

Souscription aux alertes

Quand un utilisateur souhaite s'inscrire aux alertes il convient de vérifier d'abord s'il s'est bien authentifié, puis s'il est déjà inscrit ou non. S'il est déjà inscrit le bot lui en informe, sinon il demande un confirmation de souscription. Après confirmation le bot enregistre les éléments dont il aura besoin pour pouvoir envoyer une alerte ultérieurement :

- l'identifiant d'utilisateur Teams
- l'identifiant 'tenant'. Le 'tenant Id' correspond à la souscription Microsoft Office de l'entreprise utilisateur.
- l'identifiant d'utilisateur fAIbrik, récupéré auprès de Auth0 en envoyant une demande http 'GET' avec le token opaque obtenu lors du processus d'authentification (ou de la vérification de l'authentification), suite à laquelle Auth0 retourne un 'JSON Web Token' (JWT) qui contient entre autres le fAIbrik user Id
- l'identifiant de conversation Teams. Cette information est optionnelle car si elle n'est pas disponible on peut créer une nouvelle conversation. Toutefois, afin d'optimiser l'utilisation des ressources il est utile de stocker cet identifiant.

Les identifiants utilisateur et tenant sont ensuite envoyés à la plateforme fAIbrik, ensemble avec l'identifiant utilisateur fAIbrik. Ceci permet à la plateforme de faire le lien entre la souscription aux alertes Teams et l'utilisateur de la plateforme. C'est la plateform efAIbrik qui s'occupe de stocker les éléments concernant la souscription dans sa base de données des utilisateurs. L'utilisateur reçoit une confirmation de son inscription.

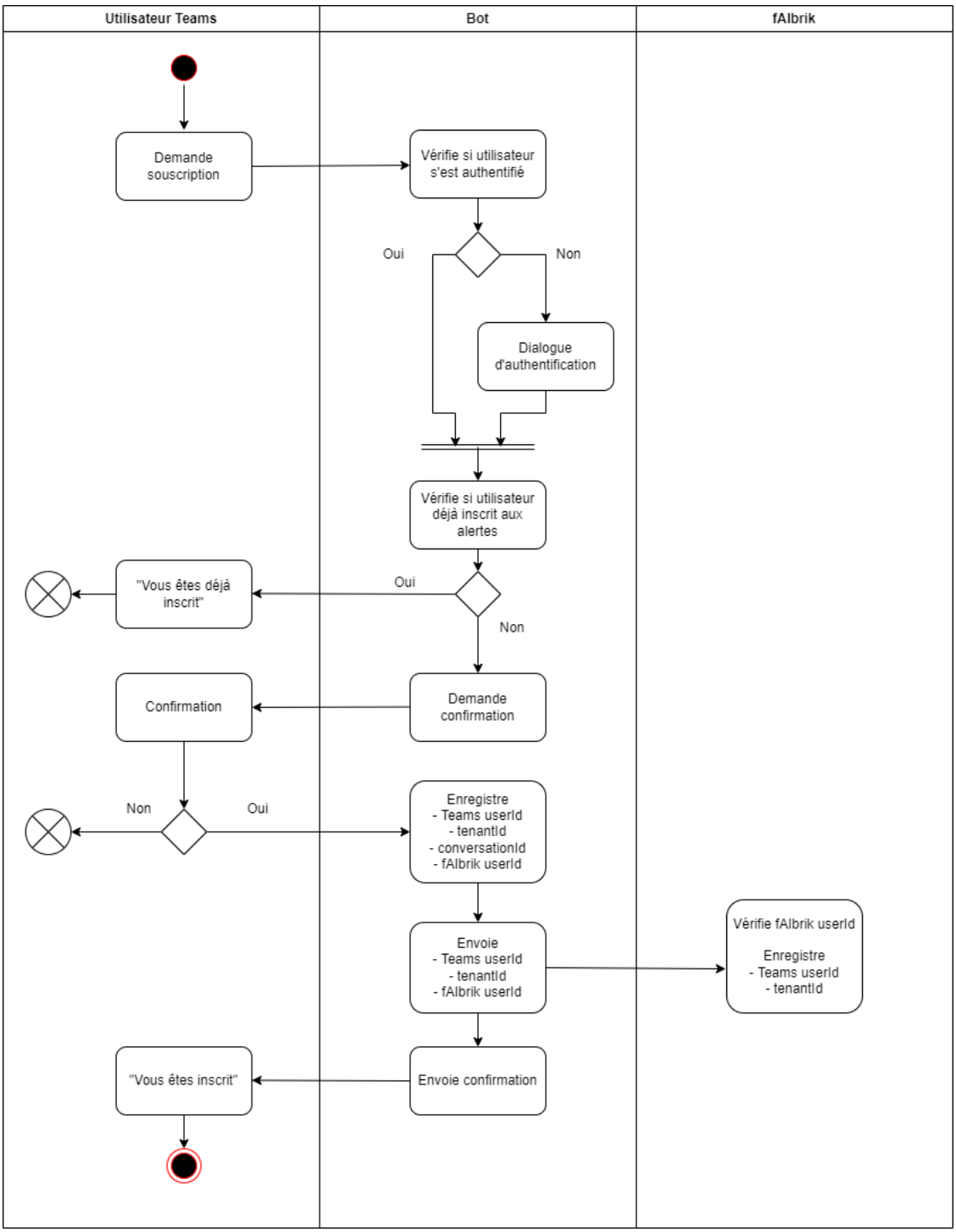

Figure 6: Diagramme d'activités : souscription

Désabonnement

Lorsque l'utilisateur demande d'être désinscrit des alertes, le bot vérifie d'abord si l'utilisateur est bien inscrit. Si c'est le cas, une confirmation de désabonnement est demandée. Si l'utilisateur confirme il est supprimé de la liste des abonnés et une demande de suppression est également envoyé à la plateforme fAIbrik.

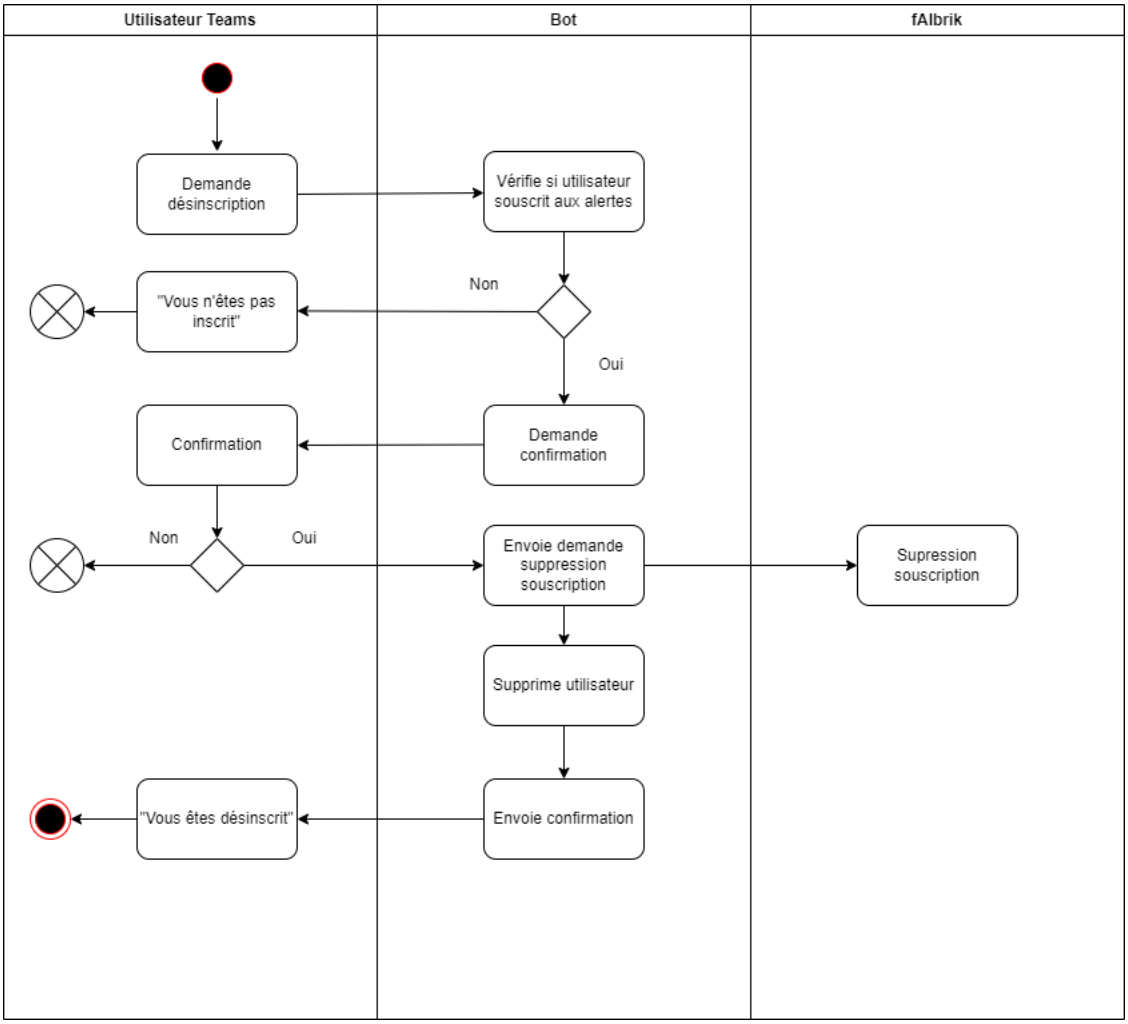

Figure 7: Diagramme d'activités : désabonnement

Gestion des utilisateurs inscrits aux alertes

Il a été décidé que le bot sera un composant 'stateless' : les utilisateurs inscrits seront gardés en mémoire mais pas stockés en base de données. Si le bot s'arrêterait, quelle qu'en soit la raison, il doit être possible de récréer la liste des utilisateurs inscrits à partir du message d'alerte envoyé depuis la plateforme fAIbrik (où les utilisateurs inscrits seront bien sauvegardés en base de données).

Alertes

Quand un événement se produit sur la plateforme fAIbrik, elle vérifiera si l'utilisateur fAIbrik concerné a souscrit aux alertes Teams. Si c'est le cas, la plateforme envoie une alerte au bot contenant:

- l'identifiant d'utilisateur Teams
- l'identifiant 'tenant'
- l'identifiant d'utilisateur fAIbrik
- le message

Le bot commence par vérifier s'il retrouve bien cet utilisateur, et si oui l'identifiant de la conversation en cours, dans sa mémoire. Si c'est le cas, le message est envoyé à l'utilisateur.

Si le bot trouve l'utilisateur mais pas l'identifiant de la conversation, quelle qu'en soit la raison, il crée une nouvelle conversation, envoie le message, puis sauvegarde la nouvelle conversation en mémoire (l'identifiant de la conversation n'est toutefois jamais envoyé dans la plateforme fAIbrik).

Si l'utilisateur n'est pas trouvé, par exemple suite à un arrêt et redémarrage du bot, un nouveau utilisateur est créé, une conversation est créée, le message est envoyé puis l'identifiant de conversation est sauvegardé.

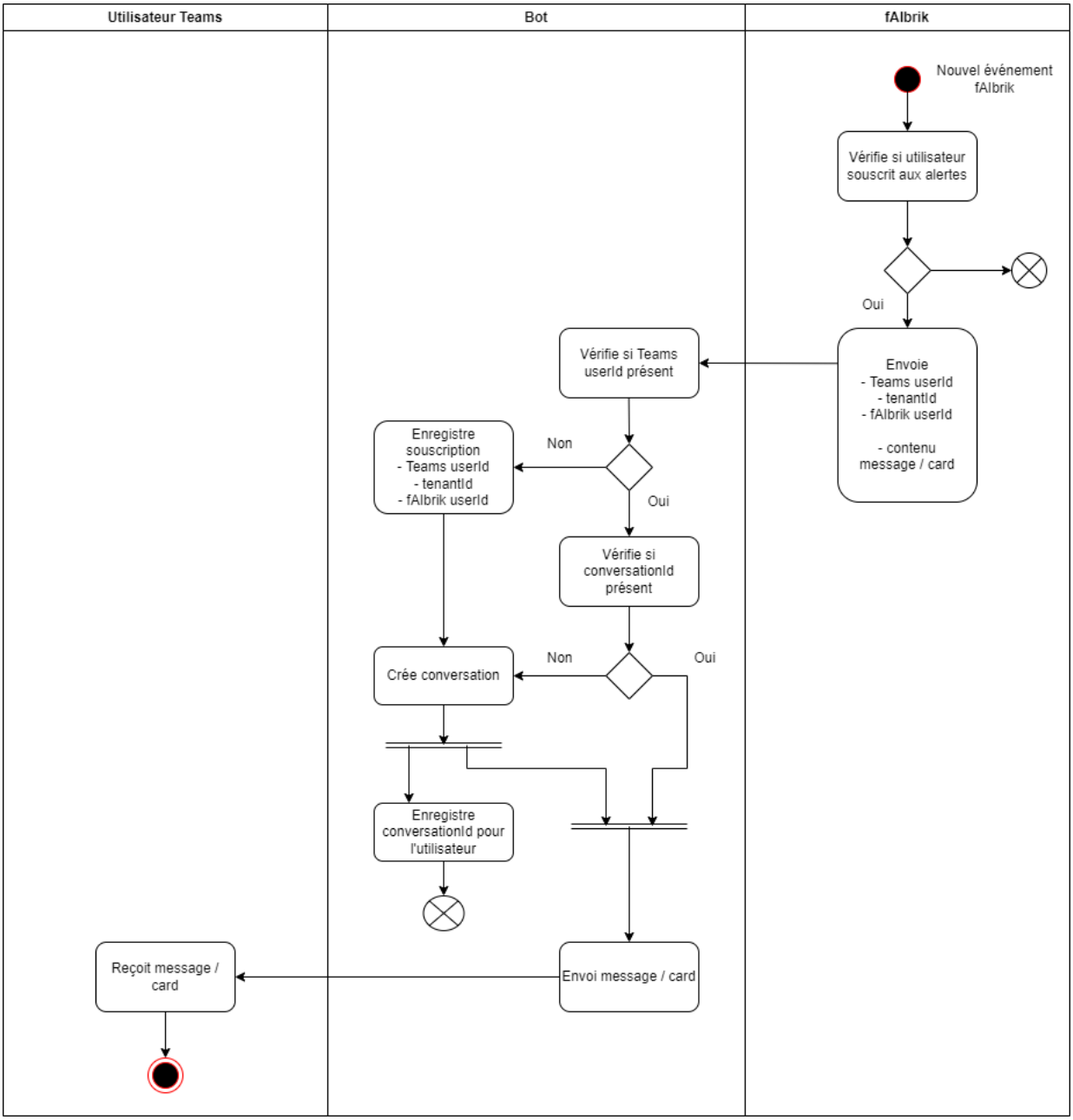

Figure 8: Diagramme d'activités : réception d'alertes

3.4 Phase 3 : Développement du bot 'standalone'

3.4.1 Première étape : Mise en place du projet & authentification

Dans un premier temps j'ai commencé à développer le bot en local sur ma machine, évidemment lié avec Teams par intermédiaire d'Azure (par un tunnel Ngrok), mais indépendamment de la plateforme fAIbrik.
Comme point de départ j'ai pris l'exemple d'un bot conversationnel avec authentification SSO tel proposé par Microsoft :

https://github.com/OfficeDev/Microsoft-Teams-Samples/tree/main/samples/bot-conversation-sso-quickstart/js

Effectivement, cette partie est sensible car elle pourrait introduire des failles de sécurité, d'autant plus que mes connaissances dans la matière sont limitées. Il me semblait donc logique de partir d'une base de code approuvée par Microsoft.
J'ai fait un clone du dépôt Github sur ma machine et j'ai pu suivre les instructions détaillées de Microsoft pour enregistrer un bot Azure et une application Azure :

https://github.com/OfficeDev/Microsoft-Teams-Samples/blob/main/samples/bot-conversation-sso-quickstart/BotSSOSetup.md

Dans cet exemple une connexion est faite avec un compte utilisateur Azure et affiche les détails du profil après l'authentification. Quand l'exemple était fonctionnel j'ai modifié la configuration de la connexion OAuth dans Azure pour qu'elle dirige vers le portal Auth0 de fAIbrik, et j'ai enlevé tout le code qui s'occupait du côté 'conversationnel' et de l'affichage du profil Azure, inutile pour mon cas d'usage.

```
class SsoOAuthPrompt extends OAuthPrompt {
    async continueDialog(dialogContext) {
        // If the token was successfully exchanged already, it should be cached in TurnState
        along with the TokenExchangeInvokeRequest
        const cachedTokenResponse = dialogContext.context.turnState.tokenResponse;

        if (cachedTokenResponse) {···
        }

        return await super.continueDialog(dialogContext);
    }
}
```

Figure 9: La classe SsoOAuthPrompt lance le processus d'authentification (dans la classe OAuthPrompt de la bibliothèque 'botbuilder-dialogs' de Microsoft) si cela n'a pas encore été fait.

Le 'prompt dialog' SsoOauthPrompt hérite de la classe OauthPrompt du package Microsoft 'botbuider-dialogs'. Cette classe s'occupe entièrement du processus d'authentification. Comme nous allons régulièrement passer par cette étape (fig. 6 Diagramme d'activités: souscription) SsoOauthPrompt commence par vérifier si un token est présent dans le contexte. Si ce n'est pas le cas elle appelle sa classe mère pour en obtenir un : une fenêtre 'pop-up' s'ouvre, où l'utilisateur saisit son identifiant et mot de passe, puis Auth0 retourne un 'opaque token' qui consiste d'une chaîne de 32 caractères qui sera stocké dans le contexte par la suite.

3.4.2 Deuxième étape : Enchaînement des dialogues

Le déroulement de la 'conversation' avec l'utilisateur est simple. Toute suite après l'installation de l'application il convient de lui poser la question s'il est déjà client fAIbrik. Si ce n'est pas le cas, par exemple si quelqu'un aurait installé l'application par curiosité, l'utilisateur est invité à visiter le site web de l'entreprise.

Cette démarche est initiée par le handler 'onMembersAdded' qui est déclenché à chaque fois qu'un utilisateur est 'ajouté à la conversation' (ou en autre mots: installe l'application').
S'il est bien client le processus d'authentification est entamé, suivi immédiatement par la proposition de souscription aux alertes, car au fin de compte c'est le but unique de l'application (pour l'instant).

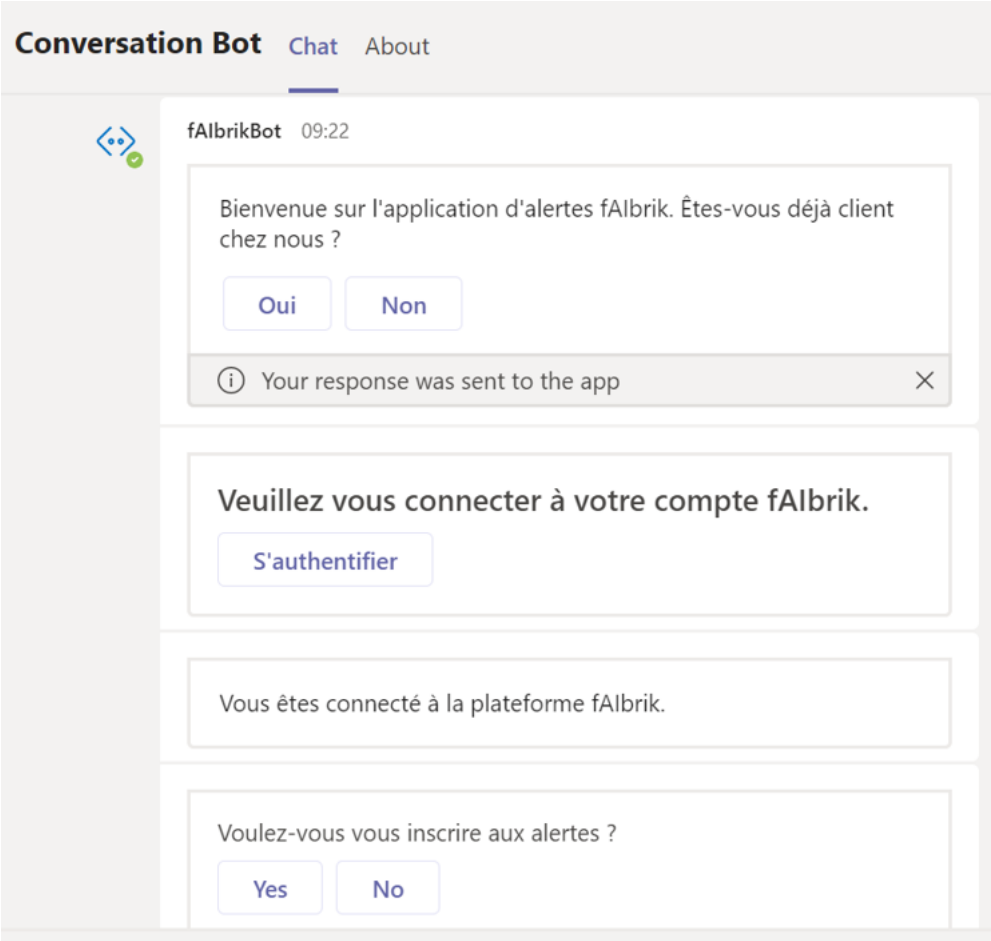

Figure 10: Le déroulement de la conversation initiale. A noter : les fonctionnalités incorporés par Microsoft s'adaptent à la langue du système d'exploitation. Le mien est en anglais, d'où le 'mélange' étrange.

L'application 'écoute' les entrées texte dans le chat Teams : 'connecter, déconnecter, désabonner'. Toutefois pour simplifier les interactions, notamment pour les utilisateurs 'mobile', le bot va répondre à n'importe quel message entré avec un menu qui propose les différentes commandes.

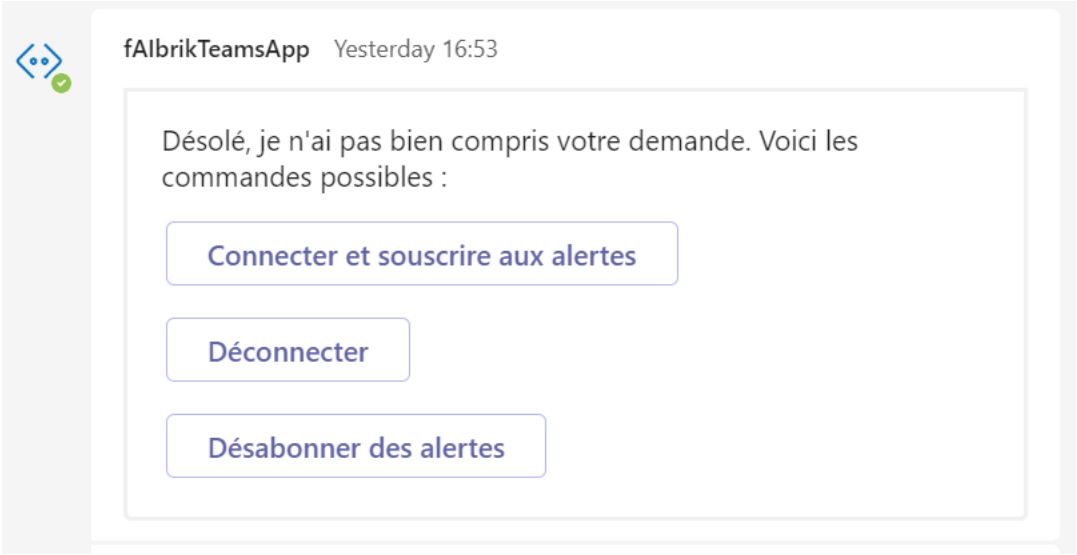

Figure 11: Le menu avec les différentes commandes

Par ailleurs le choix de ne pas proposer 'abonner' seul est conscient : de toute façon pour pouvoir s'abonner il faut d'abord se connecter, et de toute façon le seule but de se connecter est de s'abonner.

Pour incorporer cette conversation j'ai utilisé le système de dialogues en cascade ('waterfall dialogs'). Le dialogue principale ('main dialog') est emballé dans une couche 'logout dialog'. Cette architecture, étrange au premier abord, est hérité de l'exemple SSO de Microsoft. J'ai dû m'y adapter mais comme évoqué précédemment j'ai préféré ne pas trop modifier le code Microsoft qui s'occupe de l'authentification. D'autant plus qu'il y a une certaine logique, le 'logout dialog' fait un premier filtrage de toute message entrante et déconnecte dès qu'il en reçoit la demande.

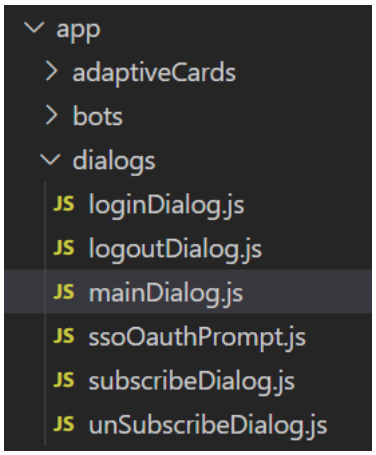

Figure 12: Les différents dialogues

Ensuite j'ai crée des 'waterfall dialogs' pour chaque fonctionnalité et je les ai attaché au dialogue principal qui va les lancer au besoin (fig. 11 et 12)

```
class MainDialog extends LogoutDialog {
    constructor(config) {
        super(MAIN_DIALOG, config.botConfig.connectionName);

        this.addDialog(new ConfirmPrompt(CONFIRM_PROMPT));
        this.addDialog(
            new WaterfallDialog(MAIN_WATERFALL_DIALOG, [
                this.faibrikClientStep.bind(this),
                this.subscribeStep.bind(this)
            ])
        );
        this.addDialog(new LoginDialog(config));
        this.addDialog(new SubscribeDialog());
        this.addDialog(new UnSubscribeDialog());

        this.initialDialogId = MAIN_WATERFALL_DIALOG;
    }
```

Figure 13: Les dialogues seront appelés depuis le 'main dialog'.

Chaque dialogue est composé de 'steps' (étapes). Ainsi le dialogue principal lance le 'faibrikClientStep' qui fait un deuxième filtrage des messages entrantes (après le 'logoutDialog', reçus ou bien par le chat, ou bien en cliquant sur les boutons des 'adaptive cards'. Un switch dirige l'utilisateur en mode 'cascade' vers la fonctionnalité voulue.

```
async faibrikClientStep(stepContext) {
    // Filter user input from adaptive cards or chat
    const cardInput = stepContext.context.activity.value?.messageText;
    const chatInput = stepContext.context.activity.text;
    const input = chatInput || cardInput;
    switch (input) {
        case "Oui":
            return await stepContext.beginDialog(LOGIN_DIALOG);
        case "Non":
            await stepContext.context.sendActivity(adaptiveCardProvider("visitSite"));
            return await stepContext.endDialog();
        case "connecter":
            return await stepContext.beginDialog(LOGIN_DIALOG);
        case "desabonner":
        case "désabonner":
            return await stepContext.beginDialog(UNSUBSCRIBE_DIALOG);
        default:
            await stepContext.context.sendActivity(adaptiveCardProvider("commands"));
            return await stepContext.endDialog();
    }
}
```

Figure 14: faibrikClientStep filtre les entrées

Si le client répond 'Oui' à la question s'il est client chez fAIbrik le 'login waterfall dialog' est lancé. Il contient le dialogue 'SsoOAuthPrompt' de Microsoft évoqué précedemment et deux steps.

```
class LoginDialog extends ComponentDialog {
    constructor(config) {
        super(LOGIN_DIALOG, config.botConfig.connectionName);

        this.addDialog(
            new SsoOAuthPrompt(OAUTH_PROMPT, {
                connectionName: config.botConfig.connectionName,
                text: ssoOAuthPromptText,
                title: ssoOAuthPromptButtonText,
                timeout: 300000
            })
        );

        this.addDialog(
            new WaterfallDialog(LOGIN_WATERFALL_DIALOG, [
                this.promptStep.bind(this),
                this.loginStep.bind(this)
            ])
        );

        this.initialDialogId = LOGIN_WATERFALL_DIALOG;
    }
```

Figure 15: Constructeur du LoginDialog

```
// This step verifies if the user is signed in already, and, if not, proposes a signin
dialog.
async promptStep(stepContext) {
    try {
        return await stepContext.beginDialog(OAUTH_PROMPT);
    } catch (err) {
        console.error(err);
    }
}

// This step verifies if login was successfull.
async loginStep(stepContext) {
    const tokenResponse = stepContext.result;
    if (!tokenResponse || !tokenResponse.token) {
        await stepContext.context.sendActivity(adaptiveCardProvider("simpleMessage",
        loginStepFail));
        return await stepContext.parent.cancelAllDialogs();
    } else {
        await stepContext.context.sendActivity(
            adaptiveCardProvider("simpleMessage", loginStepSuccess)
        );
    }
    return await stepContext.endDialog(tokenResponse.token);
}
```

Figure 16: prompStep et loginStep.

Si l'authentification s'est bien passé le LoginDialog est terminé avec la méthode endDialog et disparaît ainsi du haut du stack. Le dialogue précédent (mainDialog dans ce cas ci) peut continuer avec le subscribeStep (cf. fig 12) qui lance immédiatement le SubscribeDialog.

Le SubscribeDialog vérifie d'abord si l'utilisateur est déjà souscrit aux alertes puis, si ce n'est pas le cas, lui demande s'il souhaite souscrire. Si la réponse est positive les différents éléments nécessaires sont collectés et envoyés en paramètres de la fonction 'addMember' du 'memberStore'

Avant de pouvoir enregistrer un membre toutefois l'identifiant utilisateur fAIbrik doit être récupéré depuis Auth0. Cela se fait en envoyant le token obtenu lors de l'authentification à un endpoint chez Auth0, qui retourne un JSON Web Token duquel on peut extraire l'identifiant.

```
// Get faibrik userID
async function getFaibrikID(token) {
    // Get userinfoUrl from platformConfig
    const myConfig = getConfig();
    const userinfoUrl = myConfig.botConfig.userinfoUrl;

    const config = {
        method: "get",
        url: userinfoUrl,
        headers: {
            Authorization: `Bearer ${token}`
        }
    };

    try {
        const response = await axios(config);
        return response.data.name;
    } catch (error) {
        CSlogger.error(error);
    }
}
```

Figure 17: Récupération de l'identifiant Auth0

Afin de mieux visualiser l'enchaînement des différents 'steps' et dialogues j'ai fait un diagramme séquentiel :

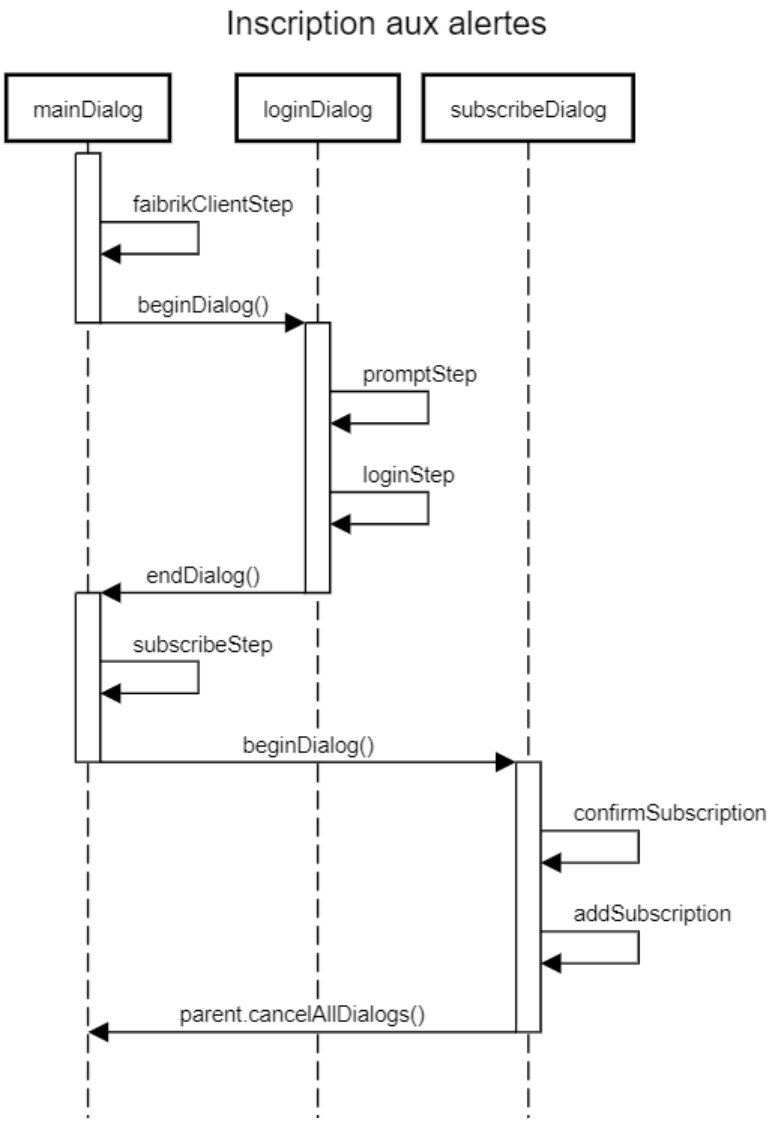

Figure 18: Diagramme séquentiel du processus de souscription

```
// If the user answered yes (s)he is added to the memberStore
async addSubscription(stepContext) {
    const result = stepContext.result;

    if (result) {
        const userToken = stepContext._info.options;
        const conversationID = stepContext.context.activity.conversation.id;
        const faibrikID = await memberStore.getFaibrikID(userToken);
        const tenantID = stepContext.context.activity.channelData.tenant.id;
        const userID = stepContext.context.activity.from.id;

        // Member is added only if not already present (this is handled within addMember
        function)
        const newMember = memberStore.addMember(conversationID, faibrikID, tenantID,
        userID);
        const messageText = newMember ? newMemberAddedText : memberExistsText;

        // Member is sent to queue
        memberStore.sendMemberToQueue(faibrikID, tenantID, userID);

        await stepContext.context.sendActivity(adaptiveCardProvider("simpleMessage",
        messageText));
    } else {
        await stepContext.context.sendActivity(
            adaptiveCardProvider("simpleMessage", subscriptionDecline)
        );
    }
    return await stepContext.parent.cancelAllDialogs();
}
```

Figure 19: addSubscription step

3.4.3 Troisième étape : Le 'memberStore'

Le fichier 'memberStore' s'occupe du traitement des utilisateurs qui souscrivent aux alertes : ajouter ou supprimer des membres, obtenir l'identification fAIbrik, retourner un membre, ... L'ensemble des membres est stockée dans un tableau.

Un 'membre' est représenté par un objet qui contient toutes les informations nécessaires. La structure de cet objet est déterminé par l'objet 'conversationReference' dont Teams a besoin pour pouvoir envoyer un message à un utilisateur. En effet, l'objet 'member' est une 'conversationReference' simplifiée. Par la suite, pour envoyer une alerte, l'application va récupérer le membre destinataire du tableau puis rajouter quelques informations supplémentaires dans l'objet.

```
// Array containing all subscribed users
const members = [];

const member = (conversationID, faibrikID, tenantID, userID) => {
    return {
        conversation: {
            id: conversationID,
            tenantId: tenantID
        },
        faibrikID: faibrikID,
        user: {
            id: userID
        }
    };
};
```

Figure 20: Le tableau des membres et la fonction qui retourne un objet 'member'

Après l'enregistrement du membre dans la mémoire du bot il sera envoyé sur la plateforme fAIbrik. Ceci est fait non pas avec des requêtes HTTP classiques mais par la publication dans une 'queue'. Pendant cette première phase du développement je me contente d'afficher l'objet qui sera envoyé dans la console.

Bien entendu un membre dois pouvoir se désabonner aussi. Ainsi j'ai défini un format JSON pour les messages sortantes pour chacun de ces activités.

```
{
    "action": "addTeamsConnection",
    "data": {
        "teams": {
            "teamsTenantID": "ec8570ae-d78a-4465-9cd9-979d44127af1",
            "teamsUserID":
            "29:1GVQDZGpuJcOL-qCgVcFTXxNz6-MkSOQzVdTx4VN8ohSdeVrg9ERrucSZDqeBq9yeyAt4x6kOUedD
            yUeQJqkDNg"
        },
        "userID": "carl@faibrik.com"
    }
}
```

Figure 21: Message sortant pour abonner un utilisateur

```
{
    "action": "removeTeamsConnection",
    "data": {
        "userID": "carl@faibrik.com"
    }
}
```

Figure 22: Message sortant pour désabonner un utilisateur

3.4.4 Quatrième étape : Mise en forme des messages du bot

Afin de mieux présenter les communications provenant du bot j'ai utilisé le format 'Adaptive Cards' de Microsoft. C'est un format qui est indépendant du plateforme utilisé et permet ainsi aux données fournis de s'incorporer dans le style de l'application choisie (Teams, Outlook, ...). Les 'Adaptive Cards' sont créées en utilisant JSON.

J'ai créé plusieurs formats ainsi qu'un 'adaptiveCardProvider' qui permet de facilement intégrer les cartes dans les fonctionnalités du bot. Le 'adaptiveCardProvider' prend deux paramètres en entrée : le premier, de type chaîne de caractères, définit le format de carte à utiliser. Le deuxième représente le contenu à afficher.

```
const commandsCard = CardFactory.adaptiveCard(adaptiveCardCommands());
const richMessageCard = content => CardFactory.adaptiveCard(adaptiveCardRichMessage(content))
;
const simpleMessageCard = content => CardFactory.adaptiveCard(adaptiveCardSimpleMessage
(content));
const visitSiteCard = CardFactory.adaptiveCard(adaptiveCardVisitSite());
const welcomeCard = CardFactory.adaptiveCard(adaptiveCardWelcome());

const adaptiveCardProvider = (cardType, cardContent) => {
    switch (cardType) {
        case "commands":
            return {
                attachments: [commandsCard]
            };
        case "richMessage":
            return { attachments: [richMessageCard(cardContent)] };
        case "simpleMessage":
            return { attachments: [simpleMessageCard(cardContent)] };
        case "visitSite":
            return {
                attachments: [visitSiteCard]
            };
        case "welcomeCard":
            return { attachments: [welcomeCard] };
    }
};
```

Figure 23: La fonction 'adaptiveCardProvider'

```
const adaptiveCardSimpleMessage = content => {
    return {
        $schema: "http://adaptivecards.io/schemas/adaptive-card.json",
        body: [
            {
                type: "TextBlock",
                size: "default",
                text: content
            }
        ],
        type: "AdaptiveCard",
        version: "1.2"
    };
};
```

Figure 24: Exemple d'une fonction qui crée une adaptive card très simple : 'adaptiveCardSimpleMessage'

3.4.5 Cinquième étape : Réception des alertes

Lorsque le bot sera incorporé dans l'application fAIbrik, l'envoi et la réception de messages se fera avec des "queues" (à l'exception de l'interaction avec Teams qui passera toujours, par intermédiaire d'Azure, par HTTP POST). Toutefois, pendant cette première phase du développement en 'standalone' j'ai créé une deuxième route '/api/notify' sur laquelle j'envoie les alertes, à savoir : des objets JSON, avec Postman.

```
{
    "header": {
        "type": "simpleMessage",
        "sendTo": [
            {
                "teamsUserID":
                "29:1DWQE3LfwhYFPIz5lv2Dti6lBobY95TpIGmRhk21oD_MTZ6qSq0SW2Z3XSfRo98CwqxfZNHml
                rrrRh-NVBoQvoQ",
                "teamsTenantID": "ec8570ae-d78a-4465-9cd9-979d44127af1",
                "userID": "carl@faibrik.com"
            }
        ]
    },
    "payload": {
        "messageText": "Vous avez un nouveau message sur fAIbrik. "
    }
}
```

Figure 25: Format pour un message entrant de type 'message simple'

Les récipients du message sont stockés dans un tableau. Effectivement, il doit être possible d'envoyer le même message à plusieurs abonnés d'un coup.

Dès réception d'un message le bot va examiner le contenu et récupérer :

- le type de message pour déterminer le format de 'adaptiveCard' à utiliser
- le contenu du message ('payload') pour l'insérer dans la 'adaptiveCard'
- le tableau des destinataires

Ensuite il va vérifier, pour chaque destinataire, s'il est présent en mémoire et s'il a bien une conversation en cours. Si c'est le cas il récupère ce membre (qui est en fait, comme expliqué précédemment, un 'conversationReference') et lui envoie le message.

Si le membre est trouvé, mais, quelque soit la raison il n'a pas de 'conversation id' (élément qui n'est pas stocké en base de données fAIbrik) le bot créé une nouvelle conversation et envoie le message. Ensuite il va stocker l'identifiant de la nouvelle conversation en mémoire dans l'objet du membre concerné.

Si finalement, par exemple après un redémarrage du bot, l'utilisateur abonné n'est pas en mémoire, le bot va pouvoir récréer la souscription comme il a toutes les données nécessaires dans l'objet reçu. Il créé une nouvelle conversation, envoie le message, puis enregistre l'identifiant de la conversation dans l'objet représentant le membre.

```
const recipientArray = message.header.sendTo;
recipientArray.forEach(recipient => {
    // Get conversation reference from memberstore
    const conversationReference = memberStore.getMember(recipient.teamsUserID);
    // Get serviceUrl from platformConfig
    const serviceUrl = getConfig().botConfig.serviceUrl;

    if (conversationReference && conversationReference.conversation?.id) {
        /*
         * Send message in existing conversation if member found and conversation exists
         */

        // Add serviceURL
        conversationReference.serviceUrl = serviceUrl;

        adapter.continueConversation(conversationReference, async context => {
            await context.sendActivity(teamsActivity);
        });
    } else if (conversationReference) {
        /*
        * Send message in new conversation if member found and conversation does not
        exist
        */

        // Add serviceURL
        conversationReference.serviceUrl = serviceUrl;

        adapter.createConversation(conversationReference, async context => {
            await context.sendActivity(teamsActivity);
            memberStore.addConversationID(conversationReference, context.activity.
            conversation.id);
        });
```

Figure 26: Le traitement d'un message reçu ...

```
} else {
    /*
     * adds member in member store, sends message in new conversation
     */
    const conversationID = "";
    const faibrikID = recipient.userID;
    const tenantID = recipient.teamsTenantID;
    const userID = recipient.teamsUserID;
    const conversationReference = memberStore.addMember(
        conversationID,
        faibrikID,
        tenantID,
        userID
    );

    // Add serviceURL
    conversationReference.serviceUrl = serviceUrl;

    adapter.createConversation(conversationReference, async context => {
        await context.sendActivity(teamsActivity);
        memberStore.addConversationID(conversationReference, context.activity.
        conversation.id);
    });
}
```

Figure 27: ... (suite)

3.5 Phase 4 : Incorporation du bot 'standalone' dans un 'channel manager' fAIbrik

Maintenant que le bot fonctionne en version 'standalone' il va falloir l'incorporer dans un 'channel manager' fAIbrik. Effectivement, pour toute communication avec les canaux extérieures (mail, Whatsapp, ...) fAIbrik utilise des 'channel mannager'. Ils s'occupent à écouter les messages entrantes, éventuellement de reformater le message vers un format 'pivot' compris par l'application fAIbrik puis de rediriger le message vers le composant concerné. Lors de l'envoi d'un message depuis fAIbrik vers un des canaux, à nouveau c'est le 'channel manager' qui s'occupe de récupérer le message, de le formater depuis le format pivot vers le format compris par le canal extérieur, puis de l'envoi du message.

Comme le code du 'channel manager' est interne à l'entreprise, à partir de cette phase je ne pourrai plus systématiquement faire de captures d'écran.

Pour commencer j'ai fait un clone d'un dépôt Gitlab interne à l'entreprise, qui est un 'blueprint' pour un 'channel manager'. Il met en place les fonctionnalités pour écouter des

endpoint HTTP mais surtout pour écouter et envoyer vers des 'queues' RabbitMQ. C'est la façon dont les composants internes de l'application fAIbrik communiquent entre eux.

J'ai donc intégré les différents fichiers du bot dans ce projet, puis le travail principal était de 'merger' le fichier index.js, point de départ du bot 'standalone' avec le fichier server.js, point de départ du 'channel manager', de sorte que les fonctionnalités de chaque se mettent en route lors du démarrage.

La principale difficulté est que la totalité des fonctionnalités du 'channel manager' est mis en route à l'intérieur d'une fonction 'callback' d'une fonction asynchrone. Lors du démarrage cette fonction récupère les différents éléments de configuration depuis la base de données et met en route ses différents composants.

Comme le bot aussi à besoin de certains éléments de configuration (qui du coup ont été enregistrés dans la base de données) il fallait trouver comment s'introduire dans ce processus. Comme le JavaScript est un langage 'single thread' il effectue toutes les opérations du code, met de côté les fonctions 'asynchrones' dans un 'event loop' puis n'exécute ces fonctions qu'après avoir vidé son 'call stack'.

La récupération de la configuration depuis la base de données étant asynchrone, le bot ne peut pas démarrer avant que toutes ces actions soient faites.

La solution était d'incorporer le 'callback' de démarrage du blueprint dans une fonction 'init' qui, en plus, retourne un objet avec la configuration reçue. Cette fonction est mise dans une IIFE (Immediately Invoked Function Expression) et suivie par deux autres fonctions qui prennent la configuration en entrée et initialisent respectivement l'adaptateur et les différents composants du bot.

Ainsi dès le démarrage du channel manager la configuration est récupérée depuis la base de données, dispatchée parmi les fonctions d'initialisation puis le bot est prêt pour écouter aussi bien les entrées utilisateur depuis Teams puis d'envoyer et recevoir des messages dans les queues.

3.6 Phase 5 : Gestion des inscriptions d'alertes

Dorénavant les nouvelles inscriptions pour les alertes seront non seulement sauvegardés en mémoire du bot, mais aussi envoyés dans une queue pour être enregistrés dans la base de données (MongoDB) des utilisateurs.

Du côté du bot, j'ai rajouté une fonction qui crée l'objet JSON à envoyer dans la queue. Ensuite j'ai ajouté un appel a cette fonction lors de l'inscription aux alertes (en même temps que l'appel de fonction pour sauvegarder l'utilisateur en mémoire du bot).

```
const queueMessage = (action, faibrikID, tenantID, userID, active = true) => {
    return {
        action: action,
        data: {
            teams: {
                active: active,
                teamsTenantID: tenantID,
                teamsUserID: userID
            },
            userID: faibrikID
        }
    };
};
```

Figure 28: La fonction 'queueMessage()' crée l'objet JSON à envoyer dans la queue

```
// Send member to configAD queue
function sendMemberToQueue(faibrikID, tenantID, userID) {
    const message = queueMessage("addTeamsConnection", faibrikID, tenantID, userID, true);
    sendToQueue(message);
}

function sendToQueue(message) {
    try {
        const myConfig = getConfig();
        const queueOut = myConfig.botConfig.queueOut;
        publish("ORC", queueOut, JSON.stringify(message));
    } catch (error) {
        CSlogger.error(
            "Failed to send message to queue. Queue : " + queueOut + " - Message : " + message
        );
    }
}
```

Figure 29: L'envoi de l'objet

Pourquoi avoir fait un deuxième format d'objet ? Parce que ce deuxième objet permet d'extraire l'objet imbriqué "teams" pour le rajouter tel quel au document de l'utilisateur concerné. L'objet 'member' de son côté correspond parfaitement au format dont Teams a besoin pour pouvoir envoyer un message.

Par ailleurs j'ai ajouté une paire clé-valeur 'active' à l'intérieur de cet objet 'teams'. Suite aux tests j'avais constaté que, quand un utilisateur désinstalle l'application sans se désabonner des alertes avant, il continue de les recevoir. Comme la plateforme fAIbrik utilise déjà une fonctionnalité pour activer/désactiver des utilisateurs (par exemple, pour ne pas les déranger quand un employé est en vacances) il a été décidé d'utiliser cette logique pour l'application Teams aussi.

Ainsi j'ai ajouté le handler 'onInstallationUpdateRemove' (de la classe mère ActivityHandler, de la librairie 'botbuilder') dans le bot qui écoute pour la désinstallation de l'application) et appelle la fonction 'deactivateMember' qui à son tour va envoyer un autre objet pour actionner ce booléen 'active'.

```
// Deactivation of member alerts when app uninstalled
this.onInstallationUpdateRemove(async (context, next) => {
    const memberToRemove = context.activity.from.id;
    const tenantID = context.activity.channelData.tenant.id;
    deactivateMember(memberToRemove, tenantID);
    await next();
});
```

Figure 30: Le handler 'onInstallationUpdateRemove'

```
// Deactivate member after app uninstallation
function deactivateMember(userID, tenantID) {
    // Deactivate in fAIbrik platform user database
    const message = queueMessage("deactivateTeamsConnection", undefined, tenantID, userID, false);
    sendToQueue(message);

    // Remove from member array if present
    const member = getMember(userID);
    if (member) {
        removeMemberFromArray(member);
    }
}
```

Figure 31: ... et la fonction 'deactivateMember' appelée.

Dans un des composants du back-end de la plateforme fAIbrik, dont je ne peux donc pas publier des captures d'écran, j'ai modifié la fonction mise en place par un des développeurs de l'entreprise, qui réceptionnait déjà les messages d'inscription pour sauvegarder les données dans la base de données d'utilisateurs. J'y ai ajouté la fonctionnalité pour pouvoir actionner la propriété booléenne 'active'.

3.7 Phase 6 : Réception d'alertes

Il ne restait pas suffisamment de temps à la fin du stage pour s'occuper de la génération des alertes sur la plateforme fAIbrik. D'un côté ce sera sans doute la partie la plus longue et compliquée, d'autre part il y a beaucoup de possibilités et des choix à faire sur lesquelles les dirigeants devaient encore se concerter.

Des possibilités, entres autres, seraient de proposer aux utilisateurs le choix entre les canaux (par exemple recevoir des alertes pour des nouveaux messages sur Whatsapp mais pas pour les mails), proposer des alertes selon les fonctions des utilisateurs (par exemple d'autres alertes pour les administrateurs que les utilisateurs en contact avec les clients), ...

Aussi, dans un premier temps il sera utile de faire tourner le bot en interne, et de l'utiliser pour la réception des alertes de l'application par les développeurs. À ce fin le CTO et dirigeant de l'entreprise à mis en place une petite application me permettant de générer des messages d'alerte et de les envoyer dans la 'queue' de l'application.

Figure 32: Un message d'alerte pour publication dans la 'queue'

Ensuite, j'ai ajouté la fonctionnalité nécessaire dans un composant 'queueManager' qui écoute toutes les messages qui passent sur la 'queue' de l'application fAIbrik.

Dans ce cas-ci, tout objet qui passe et qui contient le mot 'alert' déclenche une fonction 'publishToTeamsQueue'. Cette fonction appelle d'abord une autre fonction 'createTeamsAlert' qui à son tour lance la fonction 'getTeamsAlertSubscriptions'. Cette dernière fonction va chercher tous les utilisateurs dans la base de données dont le 'assistantID' correspond à celui envoyé dans l'alerte, et dont le booléen 'active' de l'objet imbriqué 'teams' est 'true'. Si des utilisateurs concernés sont trouvés, la fonction retourne un tableau de ceux-ci à la fonction 'createTeamsAlert' qui va construire l'objet JSON (qui, souvenez-vous, contient entre autre un tableau de récipients) à envoyer dans Teams, selon les caractéristiques définis dans l'alerte. Cet objet est retourné dans la fonction initiale 'publishToTeamsQueue' qui le publie dans la queue pour être affiché dans Teams.

Figure 33: L'alerte reçu dans Microsoft Teams

3.8 Phase 7 : Documentation technique

Afin de clôturer la période de stage il était évidemment nécessaire de bien documenter l'application que j'ai créée. Ainsi j'ai fait une documentation qui traite de la structure et du fonctionnement du bot, qui détaille le processus d'enregistrement d'application et de bot dans Microsoft Azure, et qui explique les différents formats utilisés pour les objets JSON, pour pouvoir envoyer et recevoir des messages. Cette documentation sera également disponible sur mon portfolio, où vous avez trouvé ce rapport de stage.

3.9 Phase 8 : Préparation du déploiement sur le Teams Store

Pendant le dernier jour du stage je me suis occupé de vérifier au maximum les conditions spécifiés par Microsoft pour pouvoir publier une application sur le Teams Store. La liste est assez intimidante, ainsi j'ai essayé de faire le maximum possible pour mettre l'application en état de conformité (pour autant que ce n'était pas encore le cas). Bien entendu certaines des conditions nécessitent l'intervention voire des décisions de la part des dirigeants. Pour compléter la documentation technique j'ai fait un document détaillant les points qui n'étaient pas encore traités.

Parmi les conditions Microsoft certaines traitent de la déontologie (nommage, contenu, ...), de la sécurité et de l'authentification et ne nécessitaient pas de travail supplémentaire. J'ai complété les différentes descriptions dans le fichier 'manifest.json' (d'où seront tirés les descriptions que l'utilisateur verra quand il consulte l'offre dans le Teams Store). J'ai implémenté les icônes selon les prérequis Microsoft (taille, fond, couleurs) ainsi qu'un menu 'pop-up' qui s'affiche dès que l'utilisateur se positionne dans le champ de texte du chat, pour l'informer sur les actions possibles.

Ils resteront quelques tâches à faire, notamment la mise à jour des Conditions d'Utilisation disponibles sur le site web de fAIbrik et référencés dans l'application, qui doivent la

mentionner. Il serait également nécessaire de fournir des comptes utilisateur et des scénarios à Microsoft pour qu'ils puissent tester l'application.

Finalement, il sera nécessaire que fAIbrik crée un compte 'Partenaire Microsoft' afin de pouvoir entamer le processus de publication.

4. Technologies

4.1 JavaScript, Node.js, MongoDB

J'avais déjà entamé l'apprentissage du JavaScript lors de la première année du BTS, principalement en dehors des cours, et notamment pour préparer mon stage de première année. Pendant ce stage de deuxième année j'ai pu approfondir mes connaissances, notamment en Node.js (qui permet de tourner du JavaScript pour le back-end d'une application) et plus spécifiquement en ce qui concerne les opérations asynchrones.

J'ai également eu une première expérience de travailler sur une 'vraie' application web avec une interaction poussée entre divers composants. Même si le bot Teams était en quelque sorte une application à part, pendant les dernières semaines du stage j'ai pu travailler sur l'intégration de cette application dans la plateforme fAIbrik existante, notamment sur le Channel Manager et le Queue Manager de fAIbrik.

Aussi, j'ai pu mettre en pratique les connaissances MongoDB acquises lors des cours du CNED, pour sauvegarder et récupérer les souscriptions vers et depuis la base de données des utilisateurs.

4.2 VSCode, débogueur

En tandem avec mon progrès au niveau JavaScript / Node.js, j'ai eu une belle expérience en ce qui concerne l'utilisation de VSCode et particulièrement le débogueur intégré Brièvement abordé dans les cours de première année, les applications restaient simples donc l'utilisation du débogueur limité.

Chez fAIbrik j'ai appris à m'en servir d'une façon bien plus approfondie. Au début, lors de la phase d'étude du projet, pour suivre et comprendre le fonctionnement des applications 'exemple' fournis par Microsoft. Après pour la conception des dialogues, leurs enchaînements, l'envoi des objets, ... J'ai donc bien pu constater à quel point c'est un outil

indispensable dont on se sert sans cesse à chaque instant du développement d'une application.

4.3 Outils

Tout au long du stage je me suis servi sans arrêt de deux outils très utiles :

- **Ngrok** m'a permis de créer des 'tunnel'. Un 'endpoint' temporaire mais sécurisé est généré, ce qui permet de donner accès à une application locale à des utilisateurs ou des services qui tournent sur internet. Ainsi, pendant la plupart du développement initiale, le bot a résidé sur mon ordinateur mais en créant un tunnel, puis en le spécifiant dans la configuration Azure, il était possible de communiquer avec le bot local sur Teams.

- **Postman API** est une application web qui permet d'envoyer des requêtes HTTP. Cet outil a été indispensable pour tester la fonctionnalité du bot pendant la phase initiale, avant qu'il a été connecté aux 'queues' de la plateforme fAIbrik. Non seulement peut-on créer des requêtes et recevoir les réponses, le site permet de sauvegarder les requêtes faites dans des espaces de travail, et on peut sauvegarder des variables (par exemple des valeurs de tokens) dans la plateforme afin de mieux gérer la sécurité.

5. Vie d'entreprise

Start-up de petite taille en pleine croissance, la direction de fAIbrik vise la transparence. Tous les jours à 9h a eu lieu une réunion de synchronisation ou les développeurs sont informés des avances au niveau de la commercialisation du produit. À leur tour, chaque développeur détaille les points sur lesquels il a pu avancer la veille, ce qui reste à faire et sur ses occupations prévus pour la journée.

J'ai également eu de la chance par rapport au timing du stage. L'entreprise était en train de boucler quelques projets de longue date afin de remplacer certains composants clés de l'application avec des nouvelles versions. Même si les implications pour ma mission étaient moindres, j'ai été invité pour participer à réunions des développeurs expérimentés sur l'architecture de l'application et l'implémentation de ces nouveaux composants ainsi que sur le format 'pivot' utilisé par ces composants pour échanger des données.

6. Conclusion & remerciements

J'estime avoir eu beaucoup de chance d'avoir trouvé ce stage au sein de fAIbrik. La mission proposée était très complète et passait par toutes les phases du développement d'une nouvelle application, de la phase d'étude jusqu'au produit (presque) fini.

Aspirant développeur web et très fan de JavaScript, j'ai été heureux de pouvoir travailler avec des technologies que j'aime et qui seront très utile pour la suite de mon parcours de professionnalisation.

Je suis très reconnaissant envers toute l'équipe de fAIbrik pour leur aide et surtout pour l'accueil qui m'a été réservé. Ainsi j'aimerais les remercier vivement pour cette belle opportunité !