Chatbot – Recrutement – Telegram

Sylvain METAYER – Romain DURADE

EPSI – I4 – Groupe 2

Photo : Hunters Race - Unsplash

Sommaire

Solutions techniques

Notre chatbot de recrutement a été développé sous la plateforme de communication « Telegram »

Pour réaliser ce chatbot, nous avons utilisé NodeJS ainsi que (pour les modules principaux) la librairie Node Telegram Bot API pour la partie développement et Telegram Test API pour la partie tests unitaire et d'intégration.

Motivations et avantages

Nous avons fait le choix d'utiliser cette plateforme car il s'agit d'une messagerie sécurisée utilisée par de nombreuses personnes dans le monde.[1]

Le potentiel de recrutement y est donc important, ce qui en fait une plateforme de choix.

De plus, et contrairement à certains concurrents tel que Messenger de Facebook ou Hangout de Google, les données des utilisateurs ne sont, **à priori**[2], pas analysées ni utilisées pour faire de la publicité ciblée.

Concernant le choix du langage, nous avons choisi NodeJS car nous connaissions tous les deux ce langage et souhaitions progresser techniquement sur ce dernier. De plus, une rapide étude des librairies disponibles en NodeJS nous a montré que de nombreuses librairies étaient disponibles afin d'interagir avec l'API Telegram, ce qui nous a évité d'avoir à réinventer la roue en ne partant pas de zéro.

Nous avons également fait le choix de ne pas utiliser d'outil externe tel que DialogFlow par exemple, car nous souhaitions avoir un contrôle total sur notre solution. Cela a entrainé un démarrage du projet plus lent que d'autres groupes, mais cela nous a garanti une maitrise totale de notre application, tant sur les tests que sur le code de notre chatbot. De plus, nous ne dépendons pas d'un intermédiaire pour le fonctionnement de notre chatbot, hormis le bon fonctionnement de l'API Telegram et de notre serveur privé, sur lequel est hébergé le chatbot.

De plus, cela nous a permis d'écrire des tests unitaires et des tests d'intégrations, puisque nous étions en mesure de tester unitairement chaque fonctionnalités de notre application, là où des outils tel que DialogFlow ne donne accès qu'à des tests d'intégrations / scénarios.

1 https://expandedramblings.com/index.php/telegram-stats/

2 On ne peut jamais en être sur lorsque qu'un projet garde une partie de son fonctionnement fermé !

Organisation du projet

Le projet a été organisé de façon a pouvoir être déployé de façon automatique, tout en assurant une intégration continue de notre application, en lançant automatiquement nos tests. Nous allons maintenant détailler ci-dessous l'organisation du projet, ainsi que les différents services que nous avons utilisé.

Voici divers liens utiles pour consulter les éléments cités ci-dessous :

- Dépôt Github
- Issues
- Github Project
- Milestones
- Releases
- CircleCI
- CodeCov
- StandardJS

Git (Github)

Nous avons évidemment versionné notre code, afin de faciliter le travail en groupe, et d'avoir un historique de notre travail.

Mais nous avons également utilisé les fonctionnalités offertes par Github afin de nous organiser au mieux et d'ajouter une touche d'agilité au projet. Ainsi, nous avons utilisé l'onglet projet de Github, qui permet de définir simplement un tableau « Kanban » afin de savoir les tâches en cours, à faire ou terminées.

Nous avons aussi utilisé les « issues » de Github afin de répartir les différentes tâches. Ce sont ces issues qui sont représentées dans le tableau Kanban dont nous venons de parler.

Nos issues tendaient à être nos Users Stories, même si certaines issues représentaient des tâches annexes, et nous ont permis de savoir les différentes fonctionnalités que notre application allait offrir. Ces issues ont ensuite été classées par « Milestone », afin de prioriser l'ordre des tâches à effectuer.

Dès qu'un milestone était complet, nous pouvions ajouter un tag à notre dépôt git, et ainsi proposer une « release » (version) de l'application.

De plus, nous avons utilisés l'intégration de Github avec d'autres services externe afin d'enrichir le déploiement et la qualité de notre application. Ces services permettent un retour visuel directement sur Github (voir illustration 1) afin de savoir l'état de l'application.

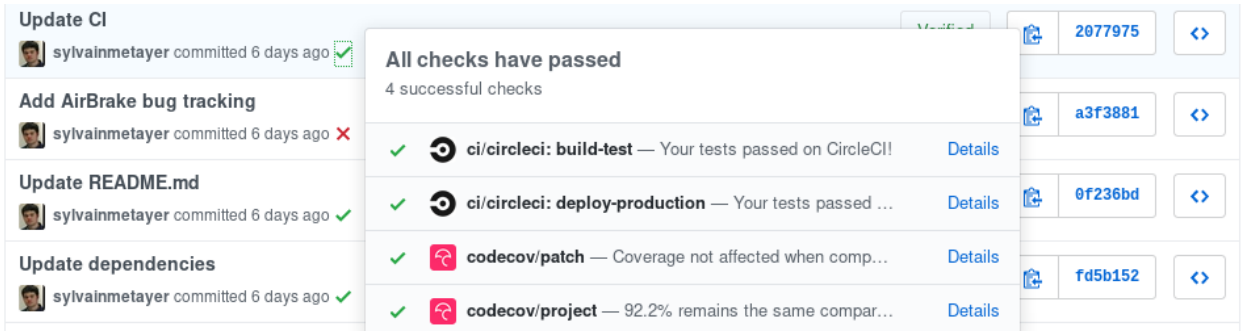

Illustration 1: Retour visuel sur Github de l'intégration de service externe

Circle CI

Ne pouvant pas utiliser Heroku comme plateforme de déploiement puisque ce dernier mettait en sommeil l'application après 5 minutes d'inactivité, nous avons choisi d'utiliser un serveur VPS privé pour héberger l'application. Néanmoins, il n'était pas question de déployer le chatbot à la main à chaque commit / release.

L'utilisation de CircleCI est la suivante :

- A chaque commit (quel que soit la branche), les tests sont déclenchés, ainsi que le linter et la couverture de code. Si l'une de ces étapes échoue, alors le build échoue.

- A chaque commit sur la branche master, le fonctionnement est le même, et si toutes les étapes sont réussies, le déploiement sur le serveur s'effectue.

CodeCov

Il s'agit d'un service de couverture de code qui permet de savoir quels sont les éléments restant à tester.

Depfu

Ce service permet de s'assurer de l'utilisation des dernières dépendances qui sont déclarées dans le fichier « package.json », en recevant automatiquement des pulls-request. Ces pulls requeset proposent la mise à jour des dépendances en déclenchant d'abord CircleCI afin de vérifier qu'aucune régression n'est apparue.

Malheureusement, l'interface d'administration ne propose pas d'accès public. Un badge (visible sur le README du dépôt) est néanmoins visible publiquement afin de savoir l'état des dépendances.

StandardJS

Cette librairie NodeJS permet d'assurer une convention de code commune (pas de guerre espace / indentation, ...) entre les membres du projet et d'assurer un niveau de qualité de code élevé. Une intégration avec un hook précommit a été mis en place afin d'assurer que le code sur le dépôt soit conforme aux standards.

AirBrake

Il s'agit d'un système de suivi de bug qui permet d'obtenir des retours sur les erreurs qui sont déclenchées par l'application sur l'environnement de production. Malheureusement, ce dernier n'est pas accessible publiquement puisqu'il pourrait remonter, selon les projets, des erreurs potentiellement confidentielles sur le projet, tel que des clés d'API ou autres. Ci-dessous se trouve une capture d'écran de l'interface principale qui permet de consulter les remontées d'erreurs.

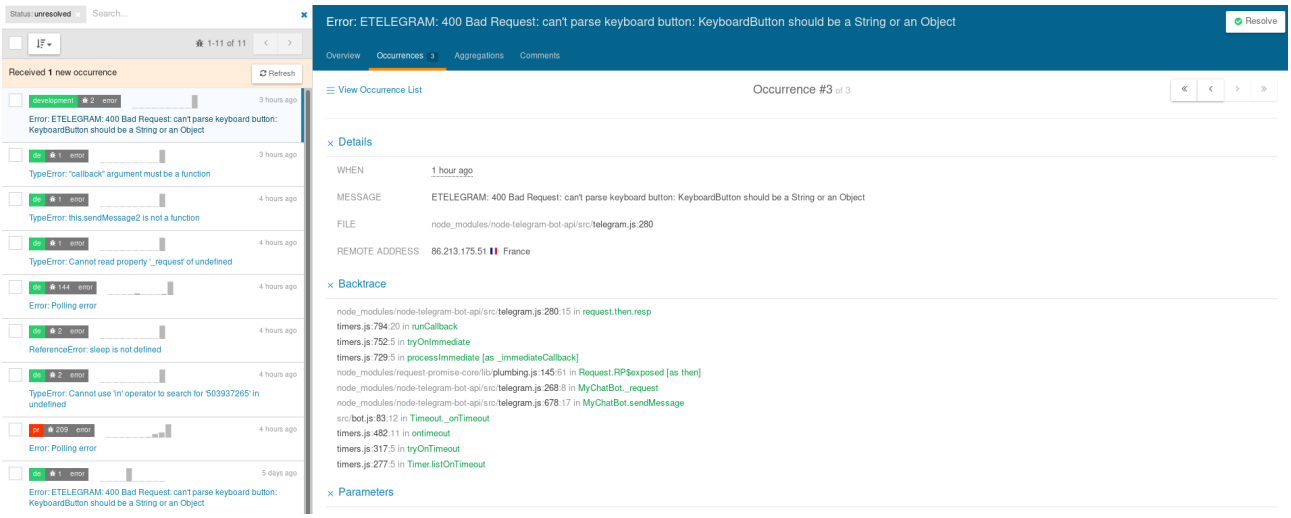

Illustration 2: Tableau de Bord Airbrake

Difficultés

Nous avons eu certaines difficultés sur ce projet mais la plupart sont désormais résolues. Nous allons dans cette partie énumérer les différents problèmes que nous avons rencontrés, ainsi que la réaction que nous avons adopté.

Déploiement continu

L'une des premières difficulté que nous avons rencontré fut de gérer le déploiement de notre application de façon continue, afin d'intégrer les développements au fur et à mesure de notre avancée. Nous étions initialement parti sur la solution Heroku, qui permet de déployer gratuitement des applications. Le soucis avec cette solution est que l'application se mettait en sommeil après 5 minutes d'inactivité, ce qui entrainait des taux de réponses trop long pour être acceptables.

Nous avons donc choisi d'abandonner Heroku pour mettre en place un déploiement sur un serveur privé, de façon automatique grâce à Circle CI dont nous avons parlé dans la section précédente.

Lors de la mise en place de notre déploiement continu, nous avons rencontré un bug sur l'exécution de nos tests sur la plateforme d'intégration continue. En effet, nos tests ne s'arrêtaient jamais, entrainant un échec constant du déploiement. La résolution fut longue, faute de connaissance sur le framework de test mocha, mais nous avons finalement corrigé l'erreur assez simplement, puisqu'il ne s'agissait que d'un paramètre manquant lors de l'exécution des tests puisqu'il fallait utiliser le paramètre « --exit » pour terminer le programme une fois les tests exécutés.

Pooling

La façon de récupérer les messages depuis Telegram peut s'effectuer de deux façons :

- Le pooling, qui va interroger le serveur Telegram toutes les X ms (en général, toutes les 500 ou 1000ms) afin de savoir si des mises à jour sont disponibles.

- Le webhook, qui permet de déclarer une URL sur laquelle Telegram enverra une requête avec les mises à jour. Dans ce cas, le serveur n'a aucune requête à effectuer, et se contente d'écouter, à l'attente de messages de la part de Telegram.

Notre solution a été d'utiliser le pooling, bien que cela ne soit évidemment pas la solution la plus efficace. Notre choix a été motivé par le fait qu'utiliser les webhook nécessite un nom de domaine et un certificat HTTPS. De plus, lors de nos développements, nos adresses locale (localhost), n'étaient pas routable sur internet.

De ce fait et étant donné les prérequis nécessaire pour utiliser les webhook, nous avons choisi de garder le pooling sur la version de production.

Messages reçus dans le désordre par le client

La dernière difficulté que nous avons rencontré, et que nous rencontrons toujours au moment de la rédaction de ce rapport, est le fait que les messages Telegram ne soient pas toujours reçus dans le bon ordre par le client, lors de tests réels. Le cas ne se se répétait pas sur nos tests, ce qui nous a poussés à chercher à faire échouer notre test, afin de comprendre l'origine du problème.

Le problème est toujours existant à l'heure actuelle, mais nous sommes confiant sur sa résolution d'ici la présentation. Nous penchons actuellement sur une limitation du nombre de caractères maximum pour un message Telegram, qui pourrait entrainer une erreur sur le client Telegram, et entrainer des retard de réception.

Diagramme des questions

Voir fichier PDF « Diagramme_questions.pdf » joint.

Scénarios d’acceptation

Les tests sont répartis en plusieurs catégories. Ces dernières sont visibles lors de l’exécution des tests.

Au moment de l’écriture de ce rapport, tous nos tests n’étaient pas encore écrits / ont été réécrits. Cette section est donc à titre informative. Pour de réelles données, merci de lancer les tests et de regarder le fichier de tests qui sera la version à jour.

Onboarding

Devrait accueillir l’utilisateur

- L’utilisateur démarre la conversation, le bot le salue et lui propose de commencer.
- L’utilisateur accepte, et le bot lui indique que le test commence

Devrait accueillir l’utilisateur, mais l’utilisateur refuse de commencer

- L’utilisateur démarre la conversation, le bot le salue et lui propose de commencer.
- L’utilisateur refuse, et le bot lui indique que le test ne commencera que lorsqu’il aura donné son accord.

Tests avec mots clés

Ces tests démarre tous avec des mots clés, afin de tester unitairement le bon fonctionnement des fonctionnalités du bot. Pour ce faire, une chaine aléatoire est utilisée pour lancer chaque test, d’une longueur suffisante pour qu’un utilisateur ne puisse pas la deviner et lancer directement une partie de la conversation. Néanmoins, il ne s’agit pas d’une sécurité parfaite, mais étant donné le contexte, nous estimons qu’elle est suffisante.

Devrait demander le nom et le prénom

- Le bot demande le nom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement
- Le bot demande le prénom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement

Devrait demander le nom, prénom et un email valide

- Le bot demande le nom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement
- Le bot demande le prénom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement

- Le bot demande l’email, l’utilisateur répond avec un email valide.

Devrait demander le nom, prénom et un email invalide

- Le bot demande le nom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement
- Le bot demande le prénom, l’utilisateur répond
- Le bot demande confirmation, l’utilisateur répond affirmativement
- Le bot demande l’email, l’utilisateur répond avec un email invalide.
- Le bot réponds que l’email est invalide, et demande une nouvelle saisie.

Devrait répondre correctement à une question de développement de type « vrai/faux »

Pour ces types de tests, et afin d’éviter de « subir » la question aléatoire, nous définissons à l’aide d’une méthode spécifique à nos tests la question posée.

- Le bot pose une question de type « vrai/faux » à laquelle l’utilisateur répond vrai
- Le bot indique à l’utilisateur qu’il a répondu correctement.

Devrait répondre correctement à une question de développement de type «évaluation de code»

- Le bot pose une question de type «évaluation de code» à laquelle l’utilisateur répond correctement.
- Le bot indique à l’utilisateur qu’il a répondu correctement.

Devrait répondre correctement à une question de développement de type «QCM»

- Le bot pose une question de type «QCM» à laquelle l’utilisateur répond correctement.
- Le bot indique à l’utilisateur qu’il a répondu correctement.

Devrait répondre faux à deux questions de développement afin que son score soit égal à 0

Pour ce type de test, nous n’avons pas besoin de savoir la question. Nous gardons donc l’aspect aléatoire des questions, et nous contentons de répondre une réponse impossible, afin qu’elle soit fausse.

- Le bot pose une question de développement à laquelle l’utilisateur répond faux
- Le bot indique à l’utilisateur qu’il a mal répondu
- Le bot pose une question de développement à laquelle l’utilisateur répond faux
- Le bot indique à l’utilisateur qu’il a mal répondu
- Le score de l’utilisateur est de 0

Devrait répondre vrai à toutes les questions de réseau afin que son score soit égal à 3

Pour ces types de tests, et afin d'éviter de « subir » la question aléatoire, nous définissons à l'aide d'une méthode spécifique à nos tests la question posée.

- Le bot pose une question à laquelle l'utilisateur répond juste
- Le bot indique à l'utilisateur qu'il a répondu juste
- Le bot pose une question à laquelle l'utilisateur répond juste
- Le bot indique à l'utilisateur qu'il a répondu juste
- Le score de l'utilisateur est de 3

Test global / Workflow

Ces tests représente les cas réels d'utilisation.

Devrait être un test d'intégration

- L'utilisateur démarre la conversation, le bot le salue et lui demande s'il est prêt à commencer
- L'utilisateur dis « oui » et le bot pose une question de développement
- L'utilisateur répond faux, le bot lui signale
- Le bot pose une question de développement
- L'utilisateur répond faux, le bot lui signale
- Le bot pose une question de développement
- L'utilisateur répond faux, le bot lui signale
- Les questions de développement sont terminées, l'utilisateur est informé que les questions de réseau vont bientôt commencer.

Devrait répondre correctement à toutes les questions mais ne veut pas de job

- L’utilisateur démarre la conversation, le bot le salue et lui demande s’il est prêt à commencer
- L’utilisateur dis « oui » et le bot pose une question de développement
- L’utilisateur répond correctement, le bot lui signale
- Le bot pose une question de développement
- L’utilisateur répond correctement, le bot lui signale
- Le bot pose une question de développement
- L’utilisateur répond correctement, le bot lui signale et lui indique que les questions de développement sont terminées. Les questions de réseaux vont commencer
- Le bot pose une question de réseau
- L’utilisateur répond correctement, le bot lui signale
- Le bot pose une question de réseau
- L’utilisateur répond correctement, le bot lui signale
- Le bot indique que le test est terminé, et qu’il va calculer ses résultats. Le bot propose les jobs à l’utilisateur
- L’utilisateur ne choisit aucun job
- Le bot demande si l’utilisateur souhaite laisser ses coordonnées.
- L’utilisateur refuse.
- Le bot met fin à la conversation et remercie l’utilisateur.

Tests divers

Devrait pouvoir réinitialiser la conversation à tout moment

- Le bot accueille l’utilisateur et lui demande son prénom
- L’utilisateur réponds « Recommencer » ou « Je veux recommencer ».
- Le bot indique que la conversation a été réinitialisée et indique à l’utilisateur peut recommencer avec la commande « start »

Installation / Utilisation du chatbot.

Bien que l'utilisation du chatbot soit détaillée dans le README du projet, vous trouverez ci-joint des détails sur l'installation et l'usage de ce dernier.

Pré-requis

- NodeJS & npm

Installation

- Git clone git@github.com:sylvainmetayer/Telegram_Bot.git
- npm install
- cp .env.sample .env
- vim .env
    - Remplir le champ TOKEN. Pour obtenir un TOKEN, se référer à la documentation Telegram disponible ici : https://core.telegram.org/bots#3-how-do-i-create-a-bot

Usage

- Démarrer le serveur
    - npm start
- Lancer les tests
    - npm test

Parler au bot

https://t.me/EPSI_UsainBot