NOHAR

Zeste de savoir

Un bot xkcd en Go - épisode 1

15 novembre 2020

Table des matières

Aujourd'hui, sur le serveur Discord de ZdS, quelqu'un a eu une idée…

> @Melcore : Il faut un bot xkcd.
> @Melcore : Genre tu tapes `=xkcd 2871` il t'affiche le bon, et il te donne celui du jour tous les jours.
> @entwanne : `=xkcd duty calls`
> @nohar : Une recherche par mots-clés serait encore meilleure.
> @nohar : `!xkcd someone wrong internet`

Une discussion sur le serveur Discord de ZdS

En fait, ça arrive tous les jours sur ce serveur que les gens aient des idées, mais celle-ci, aujourd'hui, a retenu mon attention car il s'agit d'un projet *parfait* pour illustrer les technologies que j'utilise au quotidien.

Dans ce billet (le premier d'une trilogie), nous allons développer un bot Discord en Go, qui interagit avec le contenu d'xkcd, et le déployer dans le cloud. Vous allez voir que mine de rien, en partant de cette idée toute simple, nous allons avoir l'occasion de découvrir des tas de choses!

1. Dégrossissons le projet

1.1. xkcd

Au cas où vous ne le connaîtriez pas encore, xkcd , c'est un célèbre site web sur lequel l'auteur Randall Munroe publie régulièrement de petits *strips* dessinés de façon très simpliste, à propos de sujets divers et variés. La notoriété d'xkcd est telle que nous autres, *geeks*, aimons particulièrement le citer au détour de discussions.

Voici un exemple que l'on pourrait ressortir pour expliquer aux gens qu'ils commencent à nous ennuyer à insister sur un sujet et à chercher à avoir le dernier mot:

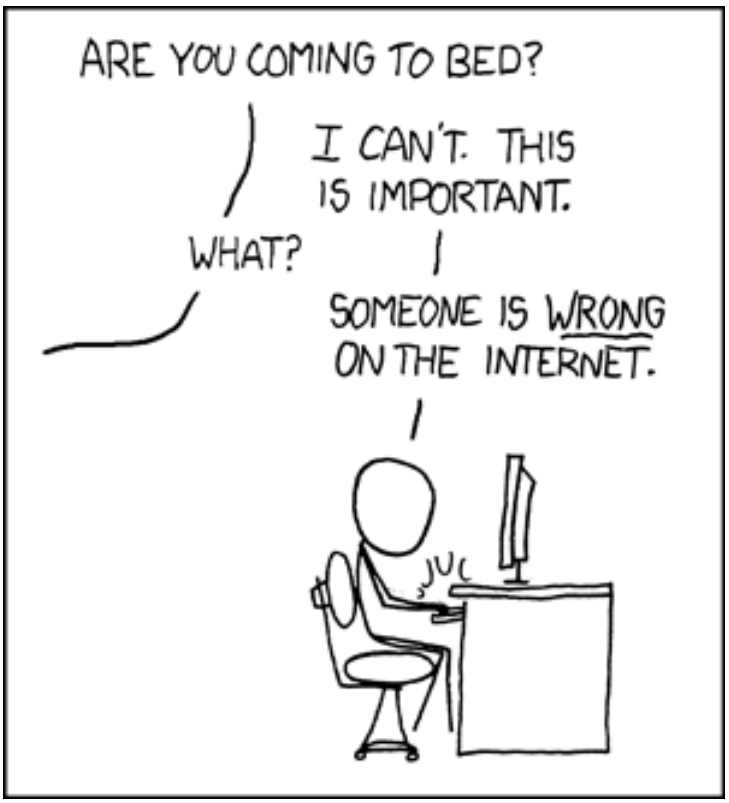


FIGURE 1.1. – xkcd #386 - Duty Calls (licence CC BY-NC)

1.2. Discord et les bots

Discord est une solution de *chat* dont la popularité a explosé ces dernières années, et plus particulièrement depuis le mois de mars dernier, suite aux confinements qui ont été mis en place un peu partout dans le monde. Il serait franchement étonnant que vous n'en ayez jamais entendu parler, puisque de nos jours, de très nombreux projets que nous voyons fleurir dans notre communauté tournent autour de serveurs Discord.

En plus de proposer et d'administrer facilement des serveurs de *chat*, Discord expose une API qui rend particulièrement commode la création de *bots*, c'est-à-dire de programmes avec lesquels nous pouvons interagir via Discord. Les jeunes développeurs sont particulièrement friands de ce

style de projets, le plus souvent réalisés avec Node.js ou Python. Ici, ce sera l'occasion pour moi de vous montrer que c'est pas mal non plus pour s'exercer en Go! 🍊

1.3. Classer les fonctionnalités par complexité

Dans le petit bout de discussion que j'ai cité plus haut, nous pouvons isoler quatre fonctionnalités du bot que nous désirons réaliser. Examinons-les pour nous faire une idée grossière de leur fonctionnement.

1.3.1. Récupérer un xkcd à partir de son ID

La première idée proposée par @Melcore est la suivante:

— On donne à notre bot l'identifiant unique d'un *strip* xkcd,
— Celui-ci va le chercher sur xkcd.com, récupère l'image (et pourquoi pas des méta-données) et l'affiche en réponse.

Je pense que l'on peut difficilement imaginer plus simple comme fonction. Il faut savoir que, connaissant l'identifiant d'un *strip*, il nous suffit de faire un appel HTTP GET à l'adresse `http://xkcd.com/<id>/info.0.json` pour obtenir toutes les données qui nous intéressent:

— Sa date de publication,
— Son titre,
— Son identifiant,
— L'adresse où récupérer l'image,
— Le texte alternatif de l'image,
— Une transcription qui décrit textuellement le contenu de l'image, y compris les dialogues.

Par exemple, voici ce que l'on obtient pour le xkcd #386 que j'ai montré plus haut (l'indentation et le formatage sont de moi):

```
{
  "month": "2",
  "num": 386,
  "link": "",
  "year": "2008",
  "news": "",
  "safe_title": "Duty Calls",
  "transcript":
      "[[A stick man is behind computer]]\nVoice outside frame: Are you coming
  "alt":
      "What do you want me to do?  LEAVE?  Then they'll keep being wrong!",
  "img": "https://imgs.xkcd.com/comics/duty_calls.png",
  "title": "Duty Calls",
  "day": "20"
}
```

Ainsi, le bot n'a qu'à récupérer le JSON correspondant à l'`id`, lire celui-ci pour obtenir l'URL de l'image, télécharger l'image et l'afficher avec éventuellement quelques infos tirées du JSON.

1.3.2. Publier le dernier xkcd sorti dans un canal donné

La seconde idée suggérée par @Melcore est que chaque fois qu'un nouveau xkcd est publié, le bot le relaye dans un canal donné du serveur Discord.

Conceptuellement, ce n'est pas très difficile: il faut savoir que lorsque nous réalisons un appel HTTP GET à l'adresse https://xkcd.com/info.0.json , nous obtenons automatiquement les informations du tout dernier *strip* qui a été publié sur le site. Dans l'idée, il suffit de réaliser cet appel à intervalles réguliers, et dès que l'on constate que l'id du *strip* a changé (celui-ci est incrémenté de 1 à chaque nouvelle publication), afficher celui-ci dans un salon.

Cependant, cette fonctionnalité vient tout de même avec un élément de complexité supplémentaire par rapport à la précédente: déjà parce qu'il faut *garder en mémoire* l'id du dernier xkcd publié pour détecter quand il y a un changement, mais également, pour faire les choses proprement, parce qu'il faut permettre aux utilisateurs de spécifier dans quel canal ils veulent que le bot publie les nouvelles parutions. Pour faire ça bien, cela va nous demander d'utiliser une petite base de données afin que ces données (dernier xkcd publié et configuration du bot par serveur) soient persistantes, et survivent à un redémarrage du bot.

1.3.3. Récupérer un xkcd à partir de son titre

La proposition d'@entwanne rajoute un élément de complexité supplémentaire. En effet, l'API d'xkcd ne nous permet pas de retrouver facilement un *strip* à partir de son titre. Il faut donc que **nous** maintenions cette correspondance dans une base de données à nous.

Nous pourrions donc imaginer sauvegarder les json des différents xkcd dans une base de données, et indexer ceux-ci suivant le titre du *strip*.

Faire cela va nous demander de *scraper* les données de tous les xkcd parus à ce jour, et donc, ce faisant, faire attention à ne pas pourrir bêtement le site en réalisant trop d'appels d'un coup. 🍊

1.3.4. Réaliser une recherche textuelle sur le contenu des xkcd

Il s'agit d'une évolution par rapport à la précédente fonctionnalité: au lieu d'indexer une correspondance entre le titre et le numéro d'un xkcd, il s'agit ici de créer un indexe de recherche textuelle sur les transcriptions des images.

Ainsi, si l'on se souvient de la *punchline* d'une BD, il nous suffit d'en taper les mots-clés pour retrouver le xkcd correspondant. Cela va donc nous demander de faire appel à un moteur d'indexation.

1.4. Décider d'un MVP

Un MVP, c'est *le programme le plus simple que l'on puisse réaliser* et qui soit déjà utile à quelqu'un. Dans notre cas, le but que je me propose d'atteindre dans ce billet c'est:

— Un bot qui supporte uniquement la première fonctionnalité ci-dessus,
— Qui soit correctement packagé et versionné,
— Qui tourne sur une infrastructure quelque part dans le cloud.

C'est également cette approche que l'on appelle un *tracer bullet*, c'est-à-dire "faire tourner un MVP en prod". Tout un programme! Alors, vous êtes prêts? 🍊

2. Communiquer avec xkcd

2.1. La création d'un nouveau projet en Go

En Go, créer un nouveau projet demande de suivre une procédure un petit peu particulière.

En effet, le système de gestion de dépendances de Go présuppose, par défaut, un couplage assez fort entre un projet et le dépôt git où celui-ci est hébergé. J'ai donc commencé par créer un dépôt sur gitlab . Dès lors, tous les packages que j'écrirai dans ce tuto seront préfixés par `gitlab.com/neuware/xkcd-bot`.

Voici la façon dont mon dépôt est organisé dès le début, et qui suit mon "organisation standard":

```
$GOPATH/gitlab.com/neuware/xkcd-bot/
├── app/      <package où est implémenté l'exécutable qui servira à
lancer le bot>
├── pkg/      <package qui contiendra tout le code "métier">
│   ├── bot/  <package où les fonctions du bot seront implémentées>
│   └── xkcd/ <package qui sert à interagir avec xkcd.com>
├── go.mod    <fichier "standard" décrivant le projet (module) et
ses dépendances)>
├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
MIT)>
└── README.md <le README du dépôt>
```

D'autres fichiers et répertoires s'ajouteront à la racine de ce dépôt (notamment, dans ce billet, `bin/` `docker/` et `deploy/`, ainsi qu'un `Makefile`), mais nous y reviendrons en temps voulu.

2.2. Lire les informations d'xkcd

Allez, il est temps de commencer à coder!

En règle générale, la première chose que j'ai tendance à implémenter dans un projet, ce n'est pas du tout le code du bot, mais **le modèle des données**. Ici, notre modèle pour communiquer avec xkcd est très simple, il s'agit d'une seule et unique structure en JSON.

En Go, cela va se traduire par une simple structure qui va en reproduire les champs.

```
$GOPATH/gitlab.com/neuware/xkcd-bot/
├── app/      <package où est implémenté l'exécutable qui servira à
    lancer le bot>
├── pkg/      <package qui contiendra tout le code "métier">
│   ├── bot/  <package où les fonctions du bot seront implémentées>
│   └── xkcd/ <package qui sert à interagir avec xkcd.com>
├── go.mod    <fichier "standard" décrivant le projet (module) et
    ses dépendances)>
├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
    MIT)>
└── README.md <le README du dépôt>
```

Vous remarquerez que chaque champ de cette structure est annotée avec un *label* sous la forme `json:"..."`. Ces annotations sont une façon d'expliquer au package standard `json` comment nous voulons que cette structure soit traduite en/depuis JSON. Notez que j'aurais pu uniquement mettre dans cette structure les champs qui vont nous intéresser dans l'immédiat…

2.2.1. Représentation de la date

La première chose que je ne peux m'empêcher de remarquer, c'est que la date tient sur trois champs distincts. Ajoutons une petite méthode à cette structure, de manière à convertir cette date en un `time.Time` standard. Ce sera l'occasion de remarquer la façon rigolote dont on utilise les fonctions de formatage des dates de Go.

Pour comprendre la fonction qui suit, il faut savoir que pour spécifier un format de date à Go, il suffit de lui montrer comment "la date magique" de référence s'écrit, et cette date magique est:

`01/02 03:04:05PM '06 -0700`

Le 2 Janvier 2006 à 15h04 et 5 secondes, sur le fuseau GMT-7

```
$GOPATH/gitlab.com/neuware/xkcd-bot/
├── app/      <package où est implémenté l'exécutable qui servira à
    lancer le bot>
├── pkg/      <package qui contiendra tout le code "métier">
```

```
4 │   ├── bot/  <package où les fonctions du bot seront implémentées>
5 │   └── xkcd/ <package qui sert à interagir avec xkcd.com>
6 ├── go.mod    <fichier "standard" décrivant le projet (module) et
    ses dépendances)>
7 ├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
    MIT)>
8 └── README.md <le README du dépôt>
```

Remarquez que cette méthode retourne à la fois une date (`time.Time`) et une erreur. Lorsque tout va bien, l'erreur est nulle (`nil`).

2.2.2. L'URL de la BD sur le site

Ajoutons une autre méthode à cette structure, de manière à facilement récupérer l'URL du Comic sur le site d'xkcd. Aucune difficulté à signaler ici:

```
1 $GOPATH/gitlab.com/neuware/xkcd-bot/
2 ├── app/      <package où est implémenté l'exécutable qui servira à
    lancer le bot>
3 ├── pkg/      <package qui contiendra tout le code "métier">
4 │   ├── bot/  <package où les fonctions du bot seront implémentées>
5 │   └── xkcd/ <package qui sert à interagir avec xkcd.com>
6 ├── go.mod    <fichier "standard" décrivant le projet (module) et
    ses dépendances)>
7 ├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
    MIT)>
8 └── README.md <le README du dépôt>
```

2.2.3. Récupérer une BD sur xkcd

Écrivons maintenant une fonction pour récupérer un xkcd à partir de son identifiant.

En fait, faisons mieux que ça. Nous allons écrire une fonction `Get` telle que:

— `xkcd.Get("386")` nous retourne le `Comic` représentant le xkcd #386,
— `xkcd.Get(xkcd.Latest)` nous retourne le tout dernier xkcd publié.

Voici comment ce genre de choses peut s'écrire en Go:

```
1 $GOPATH/gitlab.com/neuware/xkcd-bot/
2 ├── app/      <package où est implémenté l'exécutable qui servira à
    lancer le bot>
```

```
├── pkg/      <package qui contiendra tout le code "métier">
│   ├── bot/  <package où les fonctions du bot seront implémentées>
│   └── xkcd/ <package qui sert à interagir avec xkcd.com>
├── go.mod    <fichier "standard" décrivant le projet (module) et
    ses dépendances)>
├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
    MIT)>
└── README.md <le README du dépôt>
```

2.2.4. Testons ce code !

Écrivons un petit programme de test pour vérifier que tout fonctionne comme prévu:

```
$GOPATH/gitlab.com/neuware/xkcd-bot/
├── app/      <package où est implémenté l'exécutable qui servira à
    lancer le bot>
├── pkg/      <package qui contiendra tout le code "métier">
│   ├── bot/  <package où les fonctions du bot seront implémentées>
│   └── xkcd/ <package qui sert à interagir avec xkcd.com>
├── go.mod    <fichier "standard" décrivant le projet (module) et
    ses dépendances)>
├── LICENSE   <le fichier de licence, classique (ici j'ai choisi
    MIT)>
└── README.md <le README du dépôt>
```

Résultat:

```
2008-02-20 #386  "Duty Calls"
    https://imgs.xkcd.com/comics/duty_calls.png
2020-11-06 #2382 "Ballot Tracker Tracker"
    https://imgs.xkcd.com/comics/ballot_tracker_tracker.png
```

Parfait! Nous allons pouvoir implémenter le bot, maintenant.

3. Premiers pas avec discordgo

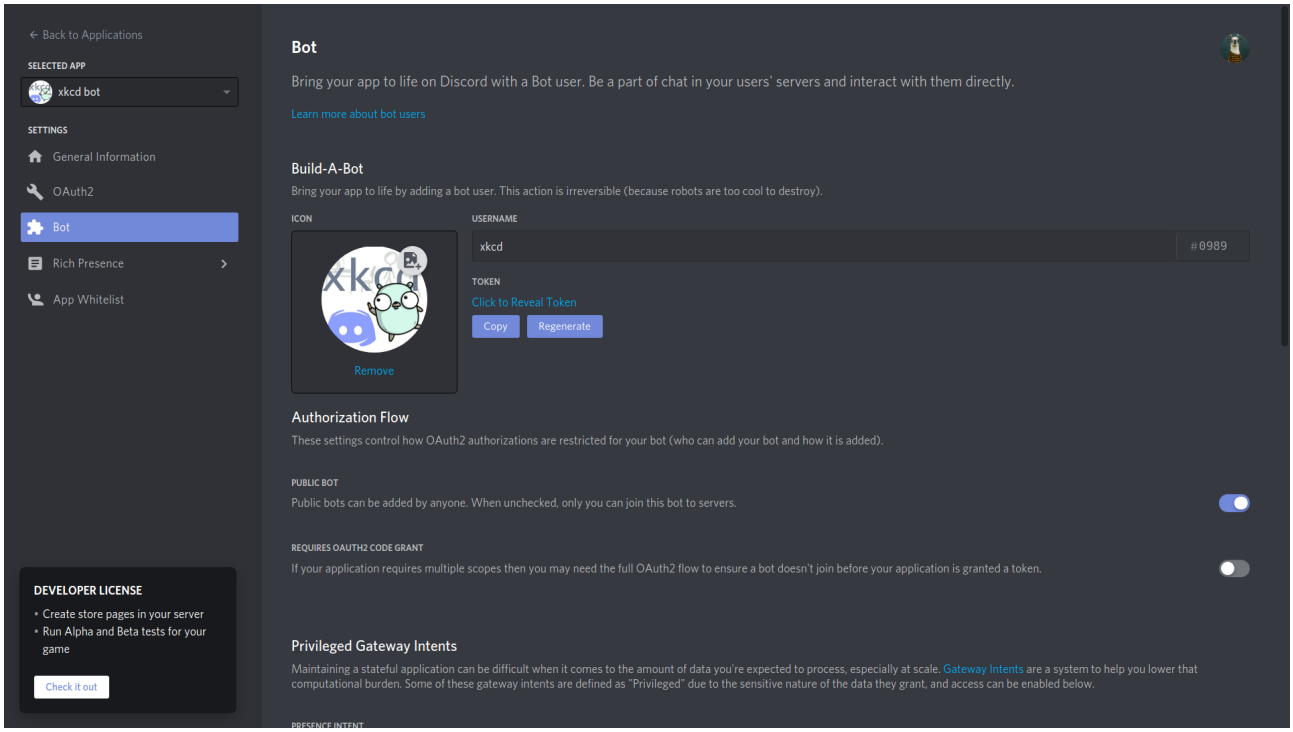
Pour interagir avec l'API de Discord, il existe une excellente bibliothèque en Go: discordgo . C'est celle que nous allons utiliser dans ce projet.

3.1. Généralités sur les bots Discord

Pour créer un bot Discord, il faut commencer par créer **un compte** pour ce bot. C'est ce compte que les administrateurs pourront inviter sur leurs serveurs en utilisant une URL spéciale.

C'est ici que tout démarre.

Il faut donc commencer par créer une "application", puis cliquer sur la section "Bot", et ajouter un nouveau bot. On finit avec une page comme celle-ci:



Pour pouvoir connecter notre bot à l'API de Discord, nous devons copier son **Token**. Il s'agit d'une valeur *secrète* grâce à laquelle notre programme va s'authentifier.

3.2. Se connecter à Discord grâce à discordgo

Commençons à implémenter notre bot. Nous allons pour cela écrire une fonction `Run` qui prend le token du bot en argument, s'authentifie, puis ouvre une connexion websocket sur Discord et commence à écouter les nouveaux messages qu'il voit passer.

Puisque notre bot tournera sous un environnement Linux (plus généralement un unixoïde), nous allons également nous mettre à l'écoute de signaux d'interruption (SIGTERM et SIGKILL), de manière à pouvoir quitter proprement lorsque le programme est interrompu.

```
1 2008-02-20 #386  "Duty Calls"
      https://imgs.xkcd.com/comics/duty_calls.png
2 2020-11-06 #2382 "Ballot Tracker Tracker"
      https://imgs.xkcd.com/comics/ballot_tracker_tracker.png
```

Lorsque l'on enregistre un nouveau callback avec `AddHandler`, `discordgo` va inspecter la signature de notre fonction pour déterminer de quels types d'événements nous souhaitons être notifiés. Ici, nous nous trouvons dans le cas le plus classique: notre bot va simplement réagir à des messages qui sont postés sur les canaux qu'il a le droit de lire. Il s'agit donc des événements `MessageCreate`.

3.3. Réagir à des commandes simples

Bien qu'il existe des bibliothèques qui se greffent à `discordgo` pour gérer un nombre arbitrairement complexe de commandes auxquelles un bot peut répondre, dans ce billet, nous allons nous contenter d'implémenter celles-ci à la main. Nous allons ici en implémenter trois:

— `xkcd!<id>` (par exemple `xkcd!389`) va afficher le xkcd ayant un ID donné,
— `xkcd!latest` va afficher le dernier xkcd.
— `xkcd!help` affichera les commandes disponibles.

Pour afficher une BD, nous allons utiliser la fonctionnalité "*Embed*" de Discord, qui permet de créer des cadres bien intégrés dans lesquels nous pouvons préciser une source, une description, une image en aperçu, etc. Enfin, si une erreur se produit lorsque le bot traite une commande, celui-ci réagira à la commande en question avec un , pour indiquer à l'utilisateur que sa commande a bien été reçue mais que quelque chose s'est mal passé.

Allez, codons!

```
2008-02-20 #386  "Duty Calls"
    https://imgs.xkcd.com/comics/duty_calls.png
2020-11-06 #2382 "Ballot Tracker Tracker"
    https://imgs.xkcd.com/comics/ballot_tracker_tracker.png
```

3.4. Créer un exécutable

Il est maintenant temps de créer notre exécutable. Pour cela, voici le contenu du fichier `app/main.go`:

```
2008-02-20 #386  "Duty Calls"
    https://imgs.xkcd.com/comics/duty_calls.png
2020-11-06 #2382 "Ballot Tracker Tracker"
    https://imgs.xkcd.com/comics/ballot_tracker_tracker.png
```

Comme vous le remarquez, celui-ci ne contient pratiquement aucune intelligence. On se contente ici de récupérer les arguments en ligne de commande (le Token est passé grâce au flag `-t` de la ligne de commande), et de lancer le bot.

Une pratique que j'aime bien mettre en œuvre dans mes projets, est de créer un `Makefile` à la racine. Voici comment nous pourrions démarrer le nôtre.

```
all: bot

.PHONY: bot

bot:
	CGO_ENABLED=0 go build -o bin/bot ./app
```

On lui rajoutera de nouvelles règles au fur et à mesure que nous avancerons. Pour l'heure, compilons et lançons notre bot:

```
$ make
CGO_ENABLED=0 go build -o bin/bot ./app
$ bin/bot -t <mon token secret>
2020/11/09 00:15:27 Up and running. Press Ctrl-C to exit.
```

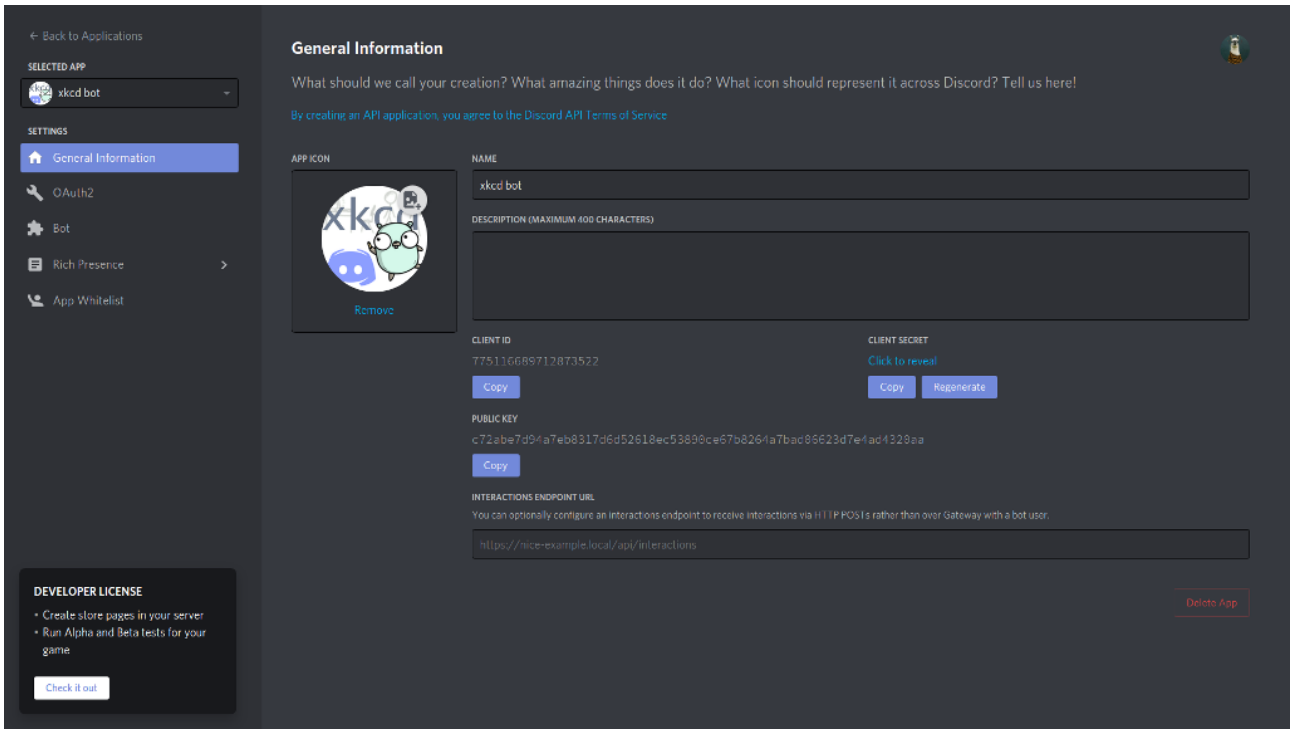
Ça y est, notre bot est lancé! Nous n'avons plus qu'à l'inviter sur un serveur pour le tester.

3.5. Testons !

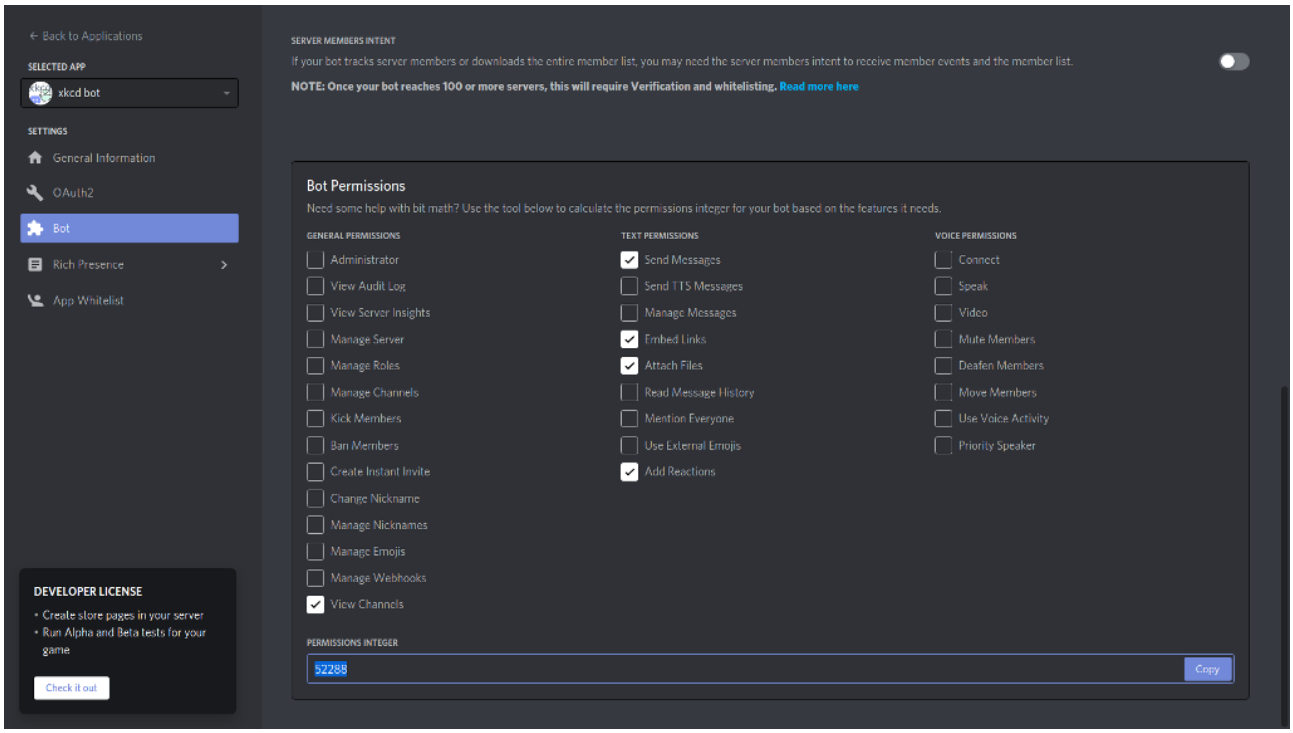
Pour inviter notre bot sur un serveur, nous avons besoin de deux éléments:

— Son **Client ID**
— Les permissions dont celui-ci aura besoin par défaut

Ces deux éléments se récupèrent sur la page de gestion de l'application Discord. Le client ID sur cet onglet:



Et les permissions se calculent via un tableau de l'onglet Bot:

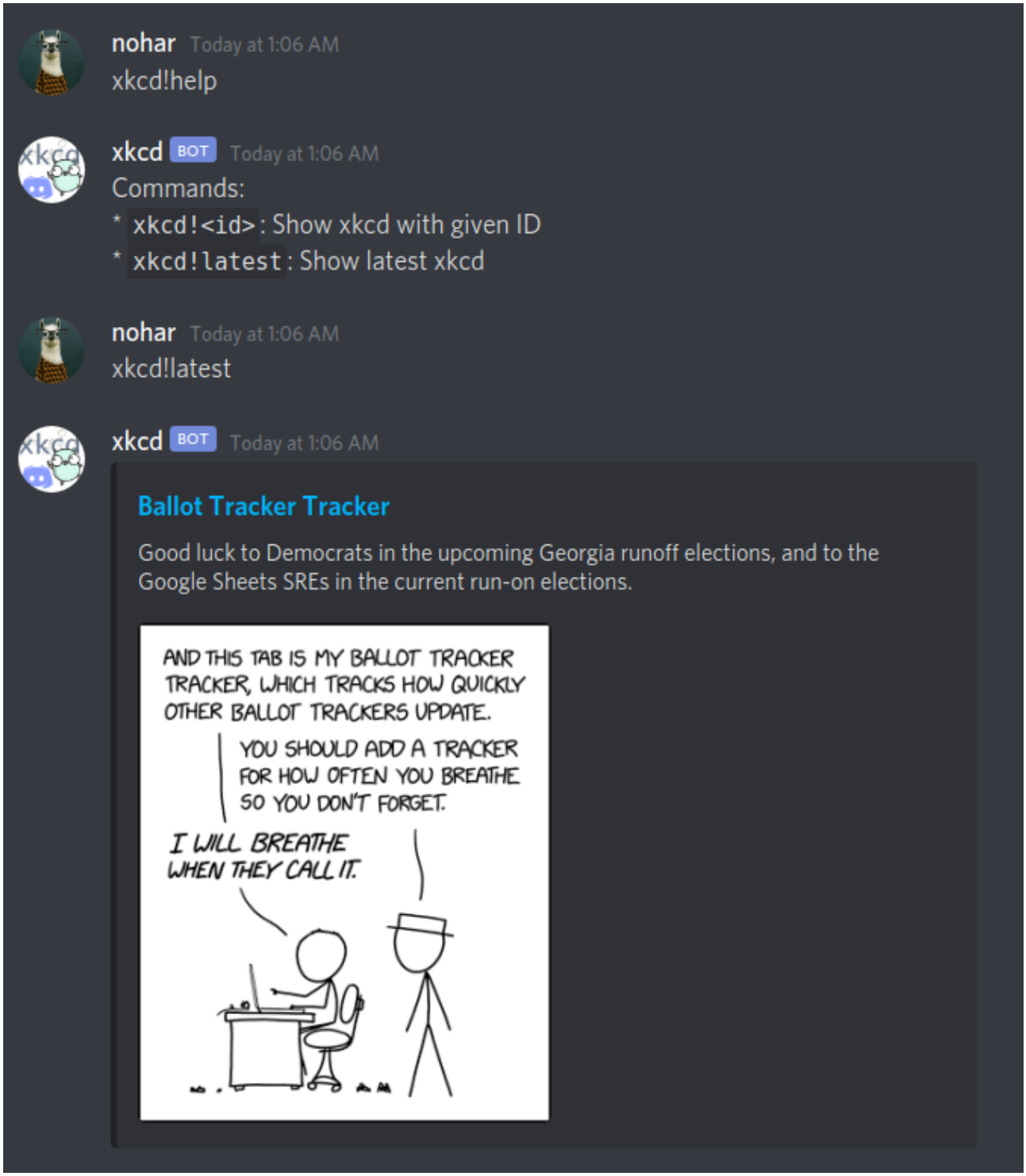


Une fois munis de ces données, nous pouvons former l'URL suivante:

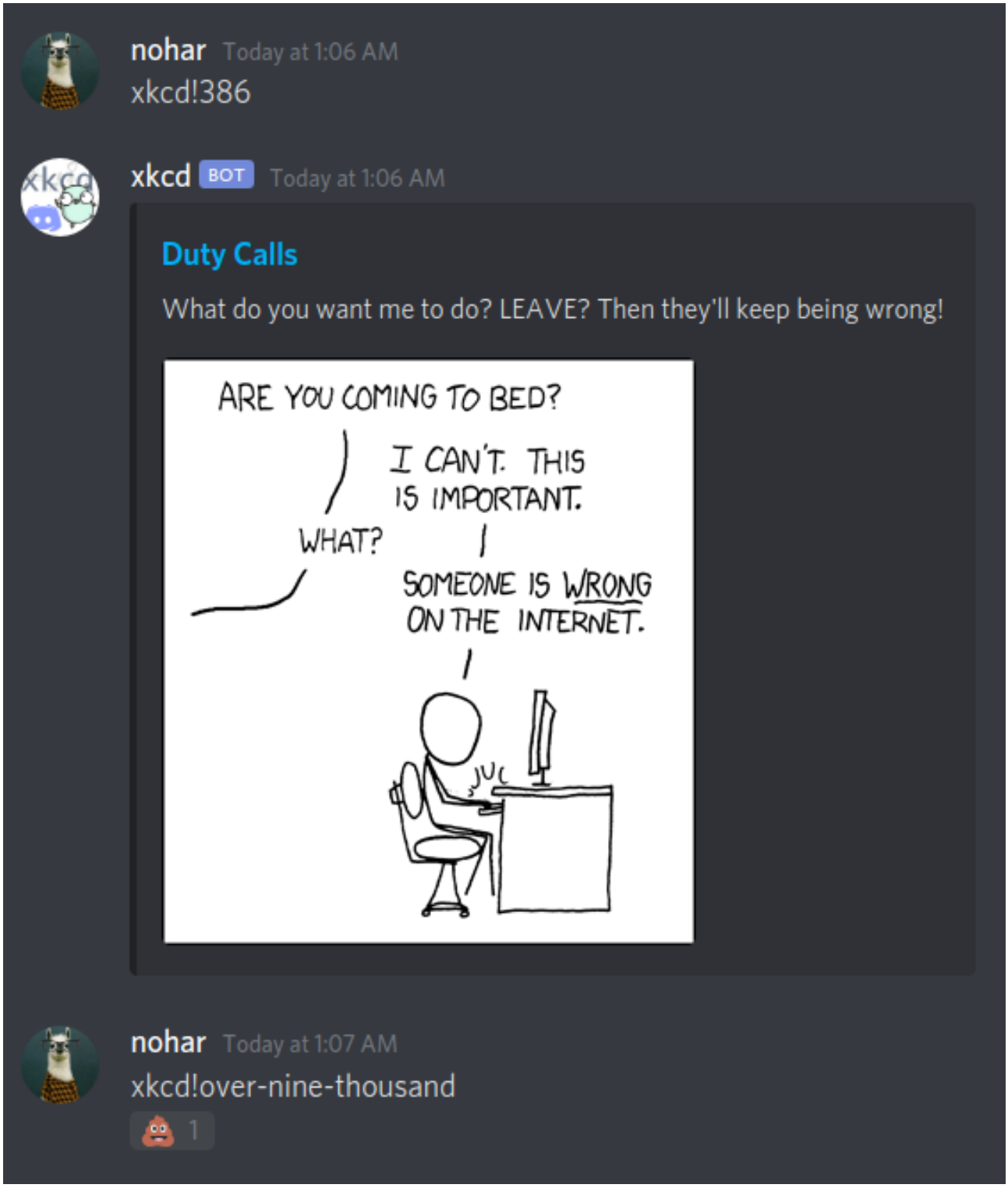
```
1 https://discord.com/oauth2/authorize?client_id=<CLIENT_ID>&scope=bot&permissio
```

Dans mon cas, l'URL sera donc https://discord.com/oauth2/authorize?client_id=775116689712873522&scopmissions=52288 . Pour inviter ce bot sur un serveur dont vous êtes l'administrateur, il vous suffit de cliquer sur ce lien, sélectionner le serveur sur lequel vous désirez inviter le bot, et confirmer. Et… c'est tout!

Il ne nous reste plus qu'à vérifier que le bot fonctionne. Cela pourrait donner ceci pour `xkcd!help` et `xkcd!latest`:



Et maintenant, essayons pour un xkcd précis, ainsi qu'une commande inconnue:



On remarquera que cette dernière commande s'accompagne d'un message de log dans la console:

```
2020/11/09 01:07:17 Error processing command
    "xkcd!over-nine-thousand": couldn't get
    'https://xkcd.com/over-nine-thousand/info.0.json': 404 Not
    Found
```

Tout fonctionne exactement comme prévu!

4. Déployer le bot dans le cloud

Bon, c'est bien joli, mais ce bot ne tourne pour l'instant que sur mon laptop. Si ferme mon ordinateur, le bot ne répondra plus à personne. Ce n'est pas très pratique!

Pour cette raison, nous allons maintenant voir comment un tel bot peut être déployé dans le cloud. Si vous me connaissez bien, vous ne serez pas surpris de découvrir que la solution que j'ai choisie pour cela est… une offre Kubernetes gratuite. Voyons comment cela se passe.

4.1. Conteneuriser l'application

Il n'y a rien de plus facile que de conteneuriser une application écrite en Go. Il nous suffit de créer une image de conteneur qui embarque notre exécutable. Celui-ci ayant été compilé et *lié statiquement* grâce à l'option `CGO_ENABLED=0`, nous n'avons besoin de rien de plus que d'une image composée uniquement de ce binaire…

Quoique, pas si vite!

4.1.1. Passer des arguments à un programme conteneurisé

Juste au-dessus, nous avons implémenté le passage du token comme un argument en ligne de commande. C'est certes pratique lorsque nous souhaitons lancer le binaire tel quel, mais pas tellement pour une application conteneurisée: même si ce n'est pas impossible, c'est *moche* de forcer les utilisateur d'un conteneur à surcharger la ligne de commande qui sert à lancer celui-ci. Quand on interagit avec des conteneurs, il y a deux façons privilégiées de leur passer des valeurs de configuration:

— Quand il y a beaucoup de valeurs à configurer, on va monter un fichier de configuration dans un volume, au *runtime*, que le binaire est implémenté pour aller lire par défaut.
— Quand on n'a qu'un petit nombre de valeurs, ou que ces valeurs sont *secrètes*, on privilégie l'utilisation de variables d'environnement.

Une chose importante à retenir, c'est qu'entre ces trois façons de passer des valeurs à notre programme, il existe une "hiérarchie" que respectent la plupart des applications modernes:

1. Par défaut, le programme utilise les valeurs présentes dans son fichier de configuration,
2. Si une variable d'environnement est présente, celle-ci *surcharge* les valeurs du fichier de configuration,

3. Si un argument est passé en ligne de commande, celui-ci *surcharge* à la fois les variables d'environnement et le fichier de configuration.

Dans notre cas, nous allons simplement faire en sorte que le programme lise la variable d'environnement `BOT_TOKEN`, et utilise sa valeur comme valeur par défaut lorsqu'il va parser les arguments en ligne de commande.

Cela à l'air compliqué, dit comme ça. En fait, c'est bête comme tout, il suffit de modifier la ligne 14 de notre fichier `main.go`:

```
2020/11/09 01:07:17 Error processing command
    "xkcd!over-nine-thousand": couldn't get
    'https://xkcd.com/over-nine-thousand/info.0.json': 404 Not
    Found
```

Nous pouvons vérifier que cela marche en lançant notre bot comme ceci:

```
$ export BOT_TOKEN=<mon token secret>
$ ./bin/bot
2020/11/09 12:17:45 Up and running. Press Ctrl-C to exit.
```

4.1.2. Construire l'image du conteneur

Puisque notre bot va devoir se connecter en https à l'API de discord, ainsi qu'à xkcd, nous n'allons pas pouvoir partir d'une image toute nue: nous allons avoir besoin d'une image qui dispose au minimum des certificats SSL racine communs. La façon la plus courante d'obtenir une telle image la plus légère possible est de baser notre image sur *alpine linux*, et d'installer dedans ces fameux certificats racine.

Voici le Dockerfile que j'utilise:

```
# docker/Dockerfile
FROM alpine
RUN apk update && apk add ca-certificates && rm -rf
    /var/cache/apk/*
ADD ./bin/bot /bin/bot
ENTRYPOINT ["/bin/bot"]
```

Alternativement, de façon à permettre aux gens qui ne disposent pas d'un environnement de développement Go de pouvoir construire notre image, nous pouvons créer un Dockerfile *staged*:

— Une première image, basée sur `golang` va nous permettre de compiler le binaire.
— Une seconde image, finale, permet de récupérer le binaire et de le placer dans un environnement `alpine`.

```
# docker/nogo.Dockerfile
FROM golang AS builder
RUN mkdir -p $GOPATH/src/gitlab.com/neuware/xkcd-bot
WORKDIR $GOPATH/src/gitlab.com/neuware/xkcd-bot
ADD . .
RUN CGO_ENABLED=0 go build -o /bin/bot ./app

FROM alpine
RUN apk update && apk add ca-certificates && rm -rf
    /var/cache/apk/*
COPY --from=builder /bin/bot /bin/bot
ENTRYPOINT ["/bin/bot"]
```

L'avantage du premier Dockerfile est que la construction de l'image est beaucoup plus rapide. L'avantage de la seconde est que l'on n'a besoin d'aucune dépendance (à part docker) pour construire l'image finale.

Ajoutons les règles correspondantes à notre Makefile:

```
all: bot

.PHONY: bot

bot:
        CGO_ENABLED=0 go build -o bin/bot ./app

image: bot
        docker build -f docker/Dockerfile -t neuware/xkcd-bot .

image-nogo:
        docker build -f docker/nogo.Dockerfile -t neuware/xkcd-bot
            .

publish:
        docker push neuware/xkcd-bot
```

Testons:

```
$ make image
$ docker run -e BOT_TOKEN=$BOT_TOKEN neuware/xkcd-bot
2020/11/09 11:18:09 Up and running. Press Ctrl-C to exit.
```

Un petit `make publish` plus tard, et mon image se retrouve poussée sur Dockerhub .

4.2. Déployer le conteneur dans un cluster Kubernetes

Pour ce projet, j'ai opté pour l'offre gratuite de Okteto cloud qui est LARGEMENT suffisante pour héberger ce projet. Je ne détaillerai pas ici comment installer kubectl ni ne me lancerai dans une longue présentation de Kubernetes. Toutes ces infos sont trouvables sur le net (et un gros tutoriel est en cours d'écriture…). Je vais me contenter dans ces billets de vous montrer *comment on l'utilise*.

Ici, notre bot est encore le genre d'application le plus simple possible que l'on puisse vouloir faire tourner dans un cluster Kubernetes: du moment que le conteneur tourne, c'est gagné, nous n'avons pas besoin qu'il soit accessible depuis l'extérieur (c'est lui qui se connecte à Discord), et pour le moment, il n'interagit pas avec une base de données ni aucun autre composant. Dans ce cas précis, nous avons juste besoin de comprendre trois abstractions de Kubernetes: le *pod*, le *deployment*, et le *secret*.

Un pod, c'est un ensemble de conteneurs indivisible, qui doivent tous tourner sur la même machine pour communiquer efficacement entre eux. Dans notre cas, notre pod ne contiendra qu'un seul conteneur. Mais nous n'allons pas créer notre pod explicitement (d'ailleurs, on ne crée jamais de pod explicitement avec Kubernetes), au lieu de cela, nous allons décrire à Kubernetes le pod que nous désirons qu'il crée au moyen d'une ressource appelée *deployment*. Quand notre *deployment* sera envoyé à Kubernetes, celui-ci (ou plutot l'un de ces contrôleurs) va faire en sorte qu'il y ait toujours un pod correspondant, à tout instant.

Autrement dit: si le bot crashe, il le redémarre et s'il y a trop d'instances du pod, il en supprime.

Mais avant de créer ce déploiement, nous avons un petit détail préliminaire à régler: il faut que nous stockions la configuration (le token) de notre bot quelque part. Pour ce faire, nous allons créer un *secret*, c'est-à-dire une valeur de configuration qui sera stockée chiffrée. Voici comment faire avec kubectl:

```
$ kubectl create secret generic bot-secret --from-literal
    token=$BOT_TOKEN
secret/bot-secret created
```

Remarquez que cette commande fonctionne uniquement parce que j'ai défini la variable d'environnement `BOT_TOKEN` plus haut (`export BOT_TOKEN=...`).

Maintenant que ce secret et défini, il ne nous reste plus qu'à définir le *deployment* du bot, qui va utiliser ce secret. Cela passe par un fichier de configuration en yaml:

```yaml
# deploy/deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: xkcd-bot
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: xkcd-bot
  template:
    metadata:
      labels:
        app: xkcd-bot
    spec:
      containers:
        - name: bot
          image: neuware/xkcd-bot
          env:
            - name: BOT_TOKEN
              valueFrom:
                secretKeyRef:
                  name: bot-secret
                  key: token
```

Donc ce fichier nous dit que l'on crée un déploiement, dont le nom est `xkcd-bot`. Celui-ci spécifie un pod, dont une seule réplique est autorisée à tourner. La partie `spec/selector` et `spec/template/metadata` permet au déploiement de retrouver aisément les pods dont il a la charge. Ce qui est plus intéressant, c'est la partie `spec/template/spec/containers`, dans laquelle on décrit le conteneur que l'on désire faire tourner. Le conteneur s'appellera `bot`, sera créé à partir de notre image publiée plus haut, et on précise que la variable d'environnement `BOT_TOKEN` sera affectée en fonction de la valeur `token` du secret `bot-secret`.

Appliquons cette configuration:

```
$ kubectl apply -f deploy/deployment.yml
deployment.apps/xkcd-bot configured
$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
xkcd-bot   1/1     1            1           5s
$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
xkcd-bot-5ff98cd999-wj6wt   1/1     Running   0          49s
```

Mission accomplie, ça tourne dans le cloud!

Et voilà, comment d'une idée anodine, on aboutit en quelques heures à un bot Discord minimaliste qui tourne dans le cloud.

Dans le prochain épisode, nous irons plus loin dans le développement de ce bot, et apprendrons à déployer dans Kubernetes une application *stateful*, dont nous voulons rendre les données persistantes. 🍊

Liste des abréviations

MVP Minimum Viable Product. 1, 5