Bot Discord

Le but de ce TP est de créer un bot Discord en Node.js.

Création d'un bot

Rendez-vous sur le portail développeur de Discord.

Cliquez sur le bouton "New Application". Donnez un nom à votre bot (Node_NOM_Prénom) et cliquez sur "Create". Vous allez arriver sur la page de votre bot. A gauche, vous pouvez voir un onglet "Bot", cliquez sur le bouton "Add Bot". Un bouton "Reset Token" va apparaître, cliquez dessus **et copiez-collez votre token dans un document texte**, celui-ci est crucial.

Après ça, rendez-vous dans l'onglet "OAuth2", et "URL generator". Sélectionnez les options *bot* et *applications.commands*. Cliquez enfin sur "Copy" et collez l'URL dans le navigateur. Sélectionnez votre serveur et ajoutez-le dessus.

Code

Nous allons désormais voir la partie code du bot Discord.

conf.json

Dans ce fichier, vous devez inclure votre *clientId* (disponible dans l'onglet *OAuth2), votre guildId (renseignez-vous pour récupérer l'ID de votre serveur) et le token récupéré précédemment.

Notez que l'on ne touchera plus à ce fichier.

Instancier le fichier app.js

Ce fichier est le fichier principal de votre bot. C'est dans celui-ci que votre bot est instancié et que celui-ci sera capable d'appeler les commandes.

```
const { Client, Events, GatewayIntentBits } = require('discord.js');
const conf = require('../conf.json');
const TOKEN = conf.token;

// Créer un nouveau client
const client = new Client({ intents: [GatewayIntentBits.Guilds] });

// ...

// Le token permet à votre client de se connecter à Discord
client.login(TOKEN);
```

Nous allons créer un petit événement nous informant quand notre bot est bien en ligne. Pour cela, dans le fichier *events/ready.js* :

```
const { Events } = require('discord.js');

module.exports = {
    name: Events.ClientReady,
    once: true,
    execute(client) {
        console.log(`Ready! Logged in as ${client.user.tag}`);
    },
};
```

De plus, dans *app.js*, nous allons ajouter 2 modules :

```
const fs = require('node:fs');
const path = require('node:path');
```

Après l'instanciation de notre client, ajoutons le code suivant :

```
const eventsPath = path.join(__dirname, 'events');
const eventFiles = fs
    .readdirSync(eventsPath)
    .filter((file) => file.endsWith('.js'));

for (const file of eventFiles) {
    const filePath = path.join(eventsPath, file);
    const event = require(filePath);
    if (event.once) {
        client.once(event.name, (...args) => event.execute(...args));
    } else {
        client.on(event.name, (...args) => event.execute(...args));
    }
}
```

Celui-ci est très simple : tout d'abord, nous allons éplucher notre dossier *events* à la recherche d'événements (fichiers présents finissant par *.js*). Enfin, pour chaque événement trouvé, nous éxécutons celui-ci.

Notez que ce code aurait pu être écrit de manière légèrement différente dans notre *app.js*, cependant un projet bien architecturé permet de mieux s'y retrouver, donc nous allons déposer nos événements dans *events* et nos commandes dans *commands*.

Commandes

Nous allons désormais voir comment créer des commandes. Etant donné que nous allons toutes les ranger dans notre dossier *commands* sous la forme suivante **nomCommande.js**, nous allons créer une collection de commandes dans notre *app.js*. Allez chercher *Collection* de *discord.js*. Maintenant, instanciez **client.commands** comme étant une **Collection**. Après ça, reprenez notre code vu juste au dessus pour les *events*, changez *eventsPath* par *commandsPath* et *eventFiles* par *commandFiles* (faites attention à votre chemin !). La condition

de notre *if* devra vérifier si nous avons 'data' dans command ET 'execute' dans command. Si la condition est vérifiée, notre code éxécute

```
client.commands.set(command.data.name, command);
```

Sinon, le simple console.log suivant :

```
console.log(
    `[WARNING] The command at ${filePath} is missing a required "data" or
"execute" property.`
);
```

Après avoir fait ça, nous allons ajouter l'interaction avec la commande par le code suivant :

```
client.on(Events.InteractionCreate, async (interaction) => {
    if (!interaction.isChatInputCommand()) return;

    const command = interaction.client.commands.get(interaction.commandName);

    if (!command) {
        console.error(
            `Aucune commande ${interaction.commandName} n'a été trouvée.`
        );
        return;
    }

    try {
        await command.execute(interaction);
    } catch (error) {
        console.error(error);
        await interaction.reply({
            content: 'Erreur dans l\'exécution de la commande !',
            ephemeral: true,
        });
    }
});
```

Celui-ci vérifie si le client contient des commandes, si aucune n'est trouvée alors il le dit dans la console et quitte la fonction. Sinon, il exécute la commande en asynchrone (si celle-ci ne fonctionne pas, il capte l'erreur retournée par celle-ci).

Bien, nous en avons fini avec notre *app.js*, il est temps de créer nos fonctions !

Créez une fonction *ping*. Voici comment celle-ci est composée :

```
const { SlashCommandBuilder } = require('discord.js')

module.exports = {
    data: new SlashCommandBuilder()
        .setName('ping')
        .setDescription('Réponse : Pong!'),
    async execute(interaction) {
        await interaction.reply('Pong!')
        await interaction.followUp('Pong again!')
    },
}
```

Nous pouvons voir qu'elle est bien composée de 'data' : un nom 'ping' et une description, et d'un 'execute', le code exécuté lorsque celle-ci est lancée, ici le bot répond par 'Pong!'.

Pour information, chaque fonction est donc ce que l'on appelle un **module** (je vous laisse vous renseigner à ce sujet). *SlashCommandBuilder* permet d'utiliser la fonctionnalité **'/'** de Discord, je vous laisse essayer pour mieux comprendre.

Il est tant d'essayer ! Lancez un deploy pour déployer nos fonctions dans la commande '/' et lancez votre bot avec start (commandes présentes dans *package.json*).

A vous maintenant !

Créez les commandes suivantes :

- echo : prend un input en option et le renvoi à l'utilisateur
- server : sort le nom du serveur et son nombre de membres
- user : sort le nom de l'utilisateur de la commande et sa date d'arrivée sur le serveur
- info : commande avec sous-commande 'user' et 'server', si server est sélectionné sort les mêmes informations que la commande *server*, si user sort les mêmes infos que la commande *user* (si un pseudo est passé avec la sous-commande, sort les informations sur l'utilisateur passé en option, sinon sur l'utilisateur commanditaire)

Bien, après avoir créé ces commandes (et les avoir testé bien sûr !) vous devriez être à l'aise avec leur principe. Il est temps d'utiliser une API avec l'une d'elles.

Créez une commande 'cat'. Nous allons utiliser undici pour faire des requêtes.

```
const { request } = require('undici')
```

L'exécute de notre fonction va faire appel à l'API pour récupérer les données, puis aller chercher notre image de chat dans le JSON :

```
const catResult = await request('https://aws.random.cat/meow')
const { file } = await catResult.body.json()
```

```
await interaction.reply({ files: [file] })
```

Pour voir si vous avez compris le principe des commandes et des appels aux API dedans, créez une commande 'urban' qui utilise l'API urbandictionary (https://api.urbandictionary.com/v0/define? + votre terme à chercher).