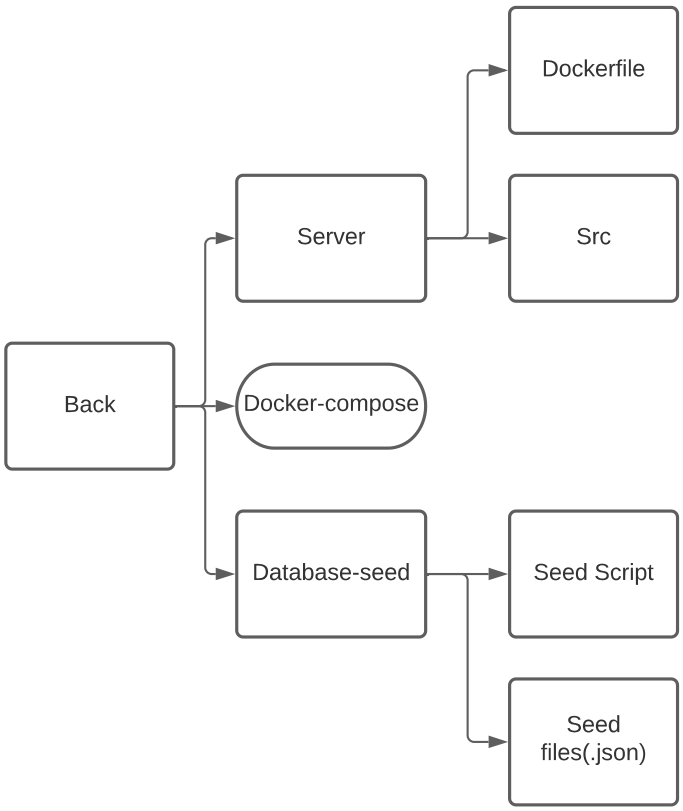
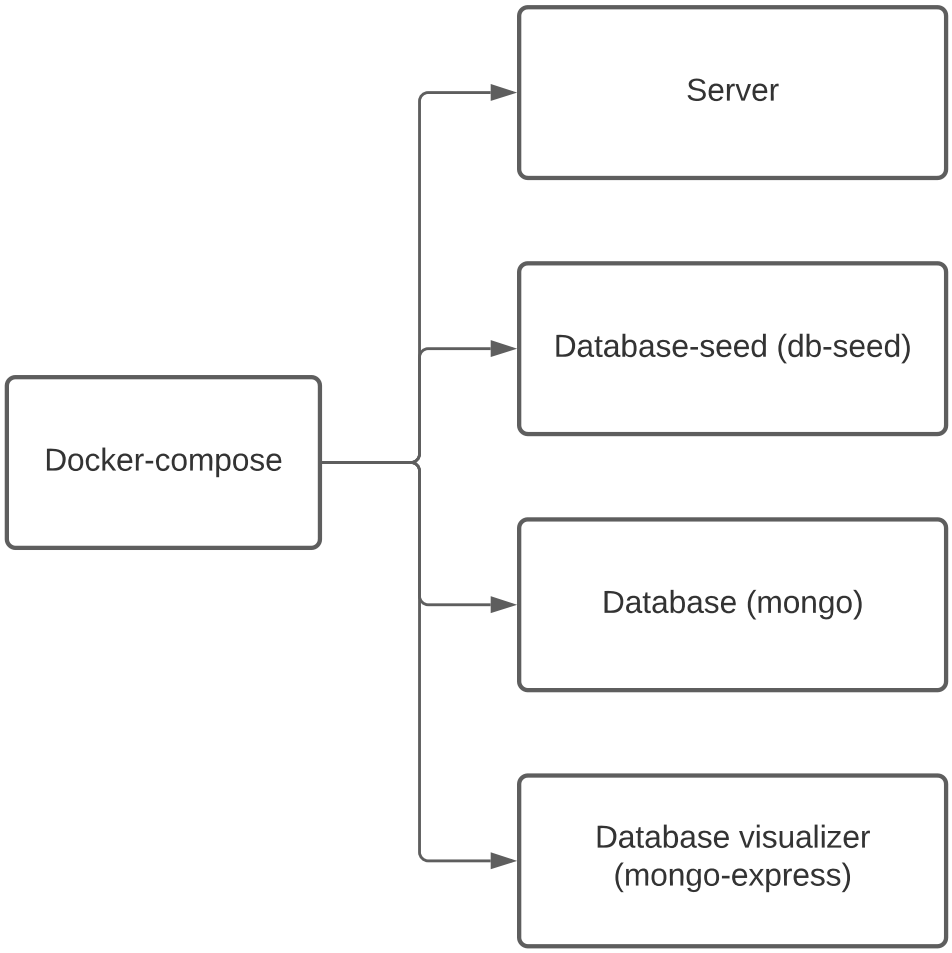
Documentation back

1. Structure du projet



Le projet s'articule autour de docker.
Dans le dossier back : nous avons un docker-compose et 2 dossiers contenant des parties contenairisees du projet. Server et Database-seed (ou db-seed)

1.1 Docker-compose



Le docker-compose contient le necessaire pour deployer :

- le server contenu dans le dossier server / sur le port 8084
- le db-seed contenu dans le dossier db-seed
- la database (mongo) par son image docker
- le visualiseur de base de donnee (mongo-express) par son image docker. / sur le port 8081

1.2 Db-seed

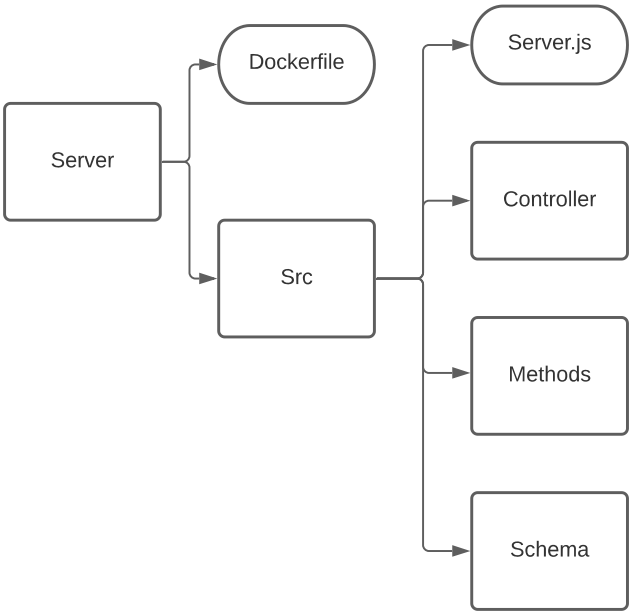
Le db-seed est simplement un script, import.sh qui va permettre de pré-remplir la base de donnée. La base de donnée etant dockerisée, on doit aussi appliquer le db-seed dans un conteneur pour que les conteneurs puissent communiquer entre eux. C'est pourquoi db-seed est deployé par le docker-compose, remplit la database, puis se ferme tout seul.

Le script compris dans db-seed remplit la db avec les informations qu'on lui donne dans des fichiers .json situés dans le même dossier.
Un fichier par schema dans la db, les .json ressemblent au prototype suivant :

```
[{
      "_id": { "$oid": "5fff3620a0efdc70962fefa4" },
      "name": "post posted by",
      "type": "facebook-post-posted",
      "description": "something is posted",
      "service": "5fff3620c37c77753705f766"
   },
   {
      "_id": { "$oid": "60018a13cab6dd49d2ba5a95" },
      "name": "post removed by",
      "description": "something is removed",
      "service": "5fff3620c37c77753705f766"
   },
   {
      "_id": { "$oid": "601033c4779c01000abdc6b5" },
      "name": "gmail: email received",
      "type": "gmail-mail-received",
      "description": "You received an email on your gmail account",
      "service": { "$oid": "601034a8779c01000abdc6b7" }
   },
   {
      "_id": { "$oid": "6034ed97f6a933000952dcf4" },
      "name": "gmail: email received that match",
      "type": "gmail-mail-received-match",
      "description": "You received an email on your gmail account that matches words",
      "service": { "$oid": "601034a8779c01000abdc6b7" }
   }
]
```

On ci-dessus la liste des actions qui doivent etre pré -rentrées dans la db. C'est simplement une liste d'objets, chaque object correspondant a une action dans la db.
Pour les id, ne pas oublier de les noter comme : « id » : {« $oid » : « …. »} pour permettre a db-seed d'identifier l'id comme un id.

1.3 Server



Le serveur a d'abord, un dockerfile, pour être deploye par le docker-compose.
Ensuite, nous avons un dossier src qui contient tout le code du serveur.
Nous les enumerer ci-dessous

1.3.1 Server.js

C'est le fichier principal du server, le 'entry-point', qui appelle tout le reste du code du server.

1.3.2 Controller

Le controller est un dossier qui contient toutes les routes du server. Autrement dit, il contient tous les requetes http qui peuvent etre faites sur le server.
Elles sont divisées par fichiers, et pour un fichier, on a une route differente et une resource differente.
Par exemple : pour recuperer les services, on fera des requete à <url>/service/. Et les fonctions se trouveront dans le dossier back/server/controller/service.js.
Il y a actuellement les routes/fichiers :

- action/action.js
- admin/admin.js
- auth/auth.js
- reaction/reaction.js
- script/script.js
- service/service.js
- user/user.js

1.3.3 Methods

Les methods constituent toutes les fonctions du serveur. Le but de ce dossier est de ne pas ranger les fonctions au meme endroit que les routes. Ainsi, nous avons des fonctions qui correspondent au routes mentionnees plus haut, et correspondent aux resources du server. Nous avons un. Dossier par resources.
Par example : Pour effectuer des fonctions sur les services, on ira dans le dossier /methods/service/<fichier> pour appeler les fonctions. Ces fichiers se trouvent donc dans back/server/src/methods/service/<fichier>

1.3.4 Schema

Ce sont tout simplement les models des resources qui seront ajoutees dans la db.

2. Les types de resources

2.1. Account

L'endroit où sont stockés les comptes d'un user. Un User peut avoir plusieurs account pour plusieurs services. Un account stocke toutes les informations necessaires pour se connecter a un service de la part d'un utilisateur : email, mot de passe, username, access_token, refresh_token, authorization_code…, selon la methode d'auth du service.
Voir dans back/server/schema/schemaAccount

```
const schemaAccount = mongoose.Schema({
    service: {
        type: mongoose.Types.ObjectId,
        ref: "Service"
    },
    user: {
        type: mongoose.Types.ObjectId,
        ref: "User"
    },
    access_token: {
        type: String,
        require: false
    },
    refresh_token: {
        type: String,
        require: false
    },
    autorization_code: {
        type: String,
        require: false
    },
    username: {
        type: String,
        require: false
    },
    password: {
        type: String,
        require: false
    }
});
```

2.2 Action

On stocke tous les types d'actions. Ils ne seront pas modifies pendant le run-time. Il sont seulement ajoutes par le db-seed au debut du deploiuement.

```
const schemaAction = mongoose.Schema({
    name: {
        type: String,
        require: true
    },
    type: {
        type: String,
        require: true
    },
    description: {
        type: String,
        require: true
    },
    service: {
        type: mongoose.Schema.Types.Objectl
        ref: "Service"
    },
    img: {
        type: String,
        require: false
    }
});

module.exports = Action = mongoose.model("A
```

2.3 Reaction

On stocke tous les types de reactions. Ils ne seront pas modifies pendant le run-time. Il sont seulement ajoutes par le db-seed au debut du deploiuement.

```
const schemaReaction = mongoose.Schema({
    name: {
        type: String,
        require: true
    },
    description: {
        type: String,
        require: true
    },
    type: {
        type: String,
        require: true
    },
    service: {
        type: mongoose.Schema.Types.Objectl
        ref: "Service"
    },
    img: {
        type: String,
        require: true
    }
});

module.exports = Reaction = mongoose.model(
```

2.4. Script

Les scripts sont la partie centrale du server : Il correspondent à une combinaison d'une action et une reaction. Ils comportent en plus, des parametres action, des parametres reactions, et un timer qui stocker le moment de la derniere utilisation.

```
const schemaScript = mongoose.Schema({
    name: {
        type: String,
        required: true,
    },
    variables: {
        type: Object,
        require: false
    },
    action_parameters: {
        type: Object
    },
    action: {
        type: mongoose.Types.ObjectId,
        ref: "Action"
    },
    reaction_parameters: {
        type: Object
    },
    reaction: {
        type: mongoose.Types.ObjectId,
        ref: "Reaction"
    },
    activated: {
        type: Boolean,
        require: false,
        default: true
    },
    last_activation: {
        type: Number,
        require: false,
        default: 0
    }
});
```

2.5. Service

On stocke tous les types de services. Ils ne seront pas modifies pendant le run-time. Il sont seulement ajoutes par le db-seed au debut du deploiement. Il contiennent les actions et reactions qui leurs sont associés, et si ilya besoin, ou pas, d'un compte.

La Parse-Map : Il y a aussi une parse-map : C'est un objet dont le but est de traduire les champs donnees par le services en champs de notre base de donnee dans account.

Example : On a un service google, on log in sur google et on recoit, un access token.., et des informations sur le profil de la personne : username, email… On veut maintenant ajouter ces informations dans un Account dans la db. Le problème ? Les noms donnés aux champs dans la db ne sont pas toujours les meme que ceux que les services comme google leur donne. Ce qu'on appelle 'username' sur la db, pourra etre appele 'name', 'user', 'email', ou meme 'login' par le service. La parse-map, stocke que 'email' sur google correspond a 'username' pour nous, et on peut stocker tranquilement toutes les infos.

```
const schemaService = mongoose.Schema({
    name: {
        type: String,
        required: true,
    },
    type: {
        type: String,
        require: true
    },
    account: {
        type: Boolean,
        require: false,
        default: false
    },
    actions: [{
        type: mongoose.Types.ObjectId,
        ref: "Action"
    }],
    reactions: [{
        type: mongoose.Types.ObjectId,
        ref: "Reaction"
    }],
    img: {
        type: String,
        require: false
    },
    parse_map: {
        type: Object,
        require: false
    }
});
```

2.6. User

Nous stockons les utilisateurs dans users. Un untilisateur a ses informations de connections, ses infos persos (prenom, nom…). Il a une liste de Scripts. Il a aussi une liste de Account.

```
const schemaUser = mongoose.Schema({
    username: {
        type: String,
        required: true,
    },
    password: {
        type: String,
        required: true,
    },
    email: {
        type: String,
        required: false,
    },
    firstname: {
        type: String,
        required: false,
    },
    lastname: {
        type: String,
        required: false
    },
    accounts: [{
        type: mongoose.Types.ObjectId,
        ref: "Account"
    }],
    scripts: [{
        type: mongoose.Types.ObjectId,
        ref: "Script",
    }],
    img: {
        type: String,
        require: false
    }
});
```

3. Les routes

Voir routes et leurs explications sur postman : voir sur postman

4. Les actions/reactions

4.1 Ajouter des actions/reactions

Pour ajouter une action :

- Ajouter l'action dans le db-seed : dans back/db-seed/init_actions.json, ajouter un object action dans la liste. Dans cet objet : un nouvel id, qu'on peut generer sur localhost :8081 dans mongo-express, un id de service auquel appartient cette action, un nom, une description et un type (nom de code pour reconnaitre l'action)

- Ajouter l'id de cette action dans la liste d'action du service auquel elle correspond dans back/db-seed/init_services.json

- Ajouter toutes les fonctions necessaires pour faire marcher l'action dans back/server/action/action_triggers/<nouveau_fichier>.js

- Ajouter le type de l'action dans le switch case, dans la fonction 'filterAction' situee dans back/server/methods/action/action_functions.js. Dans ce switch case, on appellera la nouvelle fonction ajoutee a l'etape precedente.

Pour ajouter une reaction :

- Ajouter la reaction dans le db-seed : dans back/db-seed/init_reactions.json, ajouter un object reaction dans la liste. Dans cet objet : un nouvel id, qu'on peut generer sur localhost :8081 dans mongo-express, un id de service auquel appartient cette reaction, un nom, une description et un type (nom de code pour reconnaitre la reaction)

- Ajouter l'id de cette reaction dans la liste de reactions du service auquel elle correspond dans back/db-seed/init_services.json

- Ajouter toutes les fonctions necessaires pour faire marcher la reaction dans back/server/reaction/reactions /<nouveau_fichier>.js

- Ajouter le type de la reaction dans le switch case, dans la fonction 'filterReaction' situee dans back/server/methods/reaction/reaction_functions.js. Dans ce switch case, on appellera la nouvelle fonction ajoutee a l'etape precedente.

C'est bon !

4.2 Executer des actions/reactions

Les scripts qui contiennent actions/reactions sont executes par les fonctions dans le dossier /methods/activation.

La fonction activate(), situee dans /methods/activation/activation.js, va lancer un flow, qui activera tous les scripts de tous les users, et chechera, pour chacun, si l'action a eu lieu, et activera alors la reaction.

Pour executer une action ou une reaction, on lance la fonction qui lui est associee. Cette fonction peut obtenir comme parametres :

```
(account, parameters, script_vars, last_activation)
```

- Account est le compte associe au service associe a laction.

- Parameters est les parametres donnees lors de l'initilisation du script, par example : pour l'action 'gmail :verifier si un mail a été recu de la part de <expediteur>, on aura en parametre {author : <expediteur> }.

- Script_vars contient des variables stockes pendant l'activation du script, par example le dernier mail recu, le dernier expediteur… On peut y stocker ce que l'on veut. Simplement separer les vars destinnees aux action et celle aux reactions :
  Script_vars : {action : {<vars>}, reaction : {<vars>}}

  Tout ce qu'on stocke dans script_vars apres l'execution de l'action/reaction sera sauvegardee. On peut donc par example avoir l'action 'recevoir un mail', stocker le contenu du mail dans script_vars, et reutiliser ce contenu dans la reaction qui sera declenchee.

- Last_activation
  Dernier temps d'activation du script. Le but est, pour les actions, de comparer le moment ou sont arrives les donnees mises a jours (par example, le dernier mail recu), et le moment ou on a active le script poiur la derniere fois. Ainsi, on evite d'exeecuter 2 fois la meme action alors qu'elle n'a eu lieu qu'une fois.