Université de Montréal

DIRO

IFT3150 : Projet en Informatique

Rapport de Projet

Équipe CALQUE comprenant:

Camille Divisia - 20119289

Xia Benoit - 20214317

Emanuel Rollin - 20106951

Superviseur: Louis-Edouard Lafontant

Soumis au prof. Stefan Monnier le 09 août 2024

Table des matières

Introduction

S'orienter dans l'espace est un défi du quotidien. L'expérience gagne en complexité lorsqu'on explore un endroit inconnu. Les cartes demeurent un outil ubiquitaire pour surmonter ce défi, mais la nature statique de la majorité d'entre elles comporte plusieurs défauts qui affectent leur accessibilité. Cette limitation entrave la capacité des usagers à se repérer, à planifier des trajets et à obtenir des informations pertinentes.

On s'intéresse à offrir un outil de génération de cartes interactives, dans l'idée de faciliter l'accessibilité des cartes aux utilisateurs, mais aussi pour aider les designers de cartes à les créer de manière flexible et à les maintenir.

Plusieurs cartes interactives existent dans différents contextes. Notre proposition est de fournir l'outil de génération de carte interactive tel qu'un designer de carte pourrait fournir une carte interactive à son audience facilement et exporter celle-ci dans un format intéressant pour son établissement. Le designer pourrait représenter les points d'intérêts, ses propriétés et tracer les chemins entre les différents points d'intérêts afin de représenter son réseau.

Ce projet consiste à concevoir et développer une solution répondant à cette problématique. La solution devrait permettre de construire ou charger une carte avec des annotations, simuler des trajets (sur la carte) et proposer des chemins (itinéraires) ou destinations basés sur des requêtes.

Calque est un projet open source en cours de développement, dont le code source est accessible sur GitHub[1].

[1]Repertoire GitHub: https://github.com/protolabo/calque

Analyse

Dans l'extrême en termes de flexibilité, on peut penser à AllTrails (contexte de randonnée), qui offre aussi une application pour s'orienter sur les sentiers partout en Amérique du Nord et permet la création de nouveaux sentiers via les contributions de ses utilisateurs. Un autre exemple de carte interactive serait celles associées aux musées et centres d'achat qui proposent un site web interactif ou une borne contenant leur carte interactive, facilitant leur navigation sur leur site.

Nous souhaitons faire un outil accessible et consultable depuis n'importe quel écran, où n'importe quel utilisateur peut interagir avec les **entités**, dont:

- Les **nœuds** avec lesquels l'utilisateur peut interagir (points d'intérêt). Ils contiennent les informations qui seront affichées dans l'application.
- Les **arêtes** qui relient deux nodes et qui définissent les chemins possibles entre les nœuds. Elles sont aussi interactives selon le choix du designer.
- Les **images** qui peuvent servir comme base pour dessiner des chemins par-dessus, ou pour décorer la carte. Elles peuvent aussi être interactives selon le choix du designer.

L'ensemble des entités constitue le **Graphe**[2], et avec le canvas, on obtient la **carte**.

Besoins fonctionnels

Les besoins fonctionnels de la plateforme se divisent entre deux types d'utilisateurs: Designer et Utilisateur final

Designer

Le designer crée et modifie les cartes depuis l'application, en partant depuis zéro, ou en calquant des cartes statiques existantes. Il fait cela en:

- Ajoutant les entités sur le canvas.
- Manipulant les entités et leurs attributs
- Exportant la carte en un fichier .calque, permettant à un utilisateur final de s'en servir sur l'application.
- Sauvegardant l'état de la carte

[2] En Typescript, un objet Map contient des paires de clé-valeur et mémorise l'ordre dans lequel les clés ont été insérées. Cela n'a rien à voir avec Map = Carte dans notre application, c'est pourquoi Graph est utilisé pour spécifier la carte dans notre code.

Utilisateur final

L'utilisateur final, c'est celui qui va recevoir les cartes créées par le designer. Son but est d'interagir avec en cliquant sur les entités. En faisant cela, il peut:

- Tracer un itinéraire (Ici vers B, A vers B, A vers B vers C)
- Afficher les blocs d'information du nœud sélectionné
- Afficher / masquer certains éléments de la carte.
- Pouvoir réinitialiser la carte à son état de départ

Besoin non fonctionnels

- Flexibilité: Accommoder le designer s'il désire importer un style non implémenté par le builder
- Documentation: Donner les ressources nécessaires aux utilisateurs futurs pour mieux comprendre le système
- Portabilité: Le PWA permet la compatibilité maximale parmi les OS / systèmes
- Évolutivité: On veut que les développeurs puissent améliorer le projet de la façon la moins destructive possible
- Accessibilité/Utilisabilité: L'interface doit être intuitive et facile à utiliser pour tous les types d'utilisateurs, y compris ceux ayant des besoins spécifiques en matière d'accessibilité.
- Maintenabilité: Le code doit être structuré de manière modulaire pour faciliter les mises à jour et les modifications.
- Mode hors-ligne: Les cartes générées doivent être utilisables sans connexion internet.

Choix technologiques

Notre application web implémente la stack MERN (MongoDB, Express.js, React.js, Node.js).

Frontend

React est une bibliothèque JavaScript pour construire des interfaces utilisateur interactives, en utilisant des composants réutilisables. Tailwind CSS est un framework qui permet de donner du style facilement aux composants React de manière rapide et flexible, avec des classes CSS prédéfinies. Ensemble, ils permettent de créer l'interface moderne et réactive de l'application CALQUE.

TypeScript ajoute du typage statique à JavaScript. Au vu des différentes entités existantes, ainsi que les outils de manipulation du canvas, modes, pages et utilisateurs, nous pensons fortement qu'un typage strict permet de réduire les erreurs et bugs, et donc rendre l'application plus fiable et robuste.

Le canvas, ainsi que les entités, sont des éléments SVG. La librairie D3.js est très pratique pour manipuler les entités et modifier leurs attributs.

Avantages :

- ➔ Pour l'application, le rendu sélectif permet d'accélérer la mise à jour de la page, permet une compatibilité maximale entre les différents navigateurs en utilisant les méthodes de manipulation du DOM et les event listeners implémentés par React. Similairement, Tailwind utilise des classes de style qui sont adaptées pour les différents navigateurs et favorise la compatibilité. Tailwind supporte les Responsive designs et retire un poids sur les développeurs frontend.
- ➔ Pour le développement, React permet une approche moderne du développement web. Permet le rendu conditionnel des différents composants, facilite grandement la gestion des états d'une application, permet une approche modulaire au développement web qui aide à prévenir une structure monolithique.

Backend

Express.js est un framework minimaliste pour Node.js qui simplifie la création de serveurs web et d'API en fournissant une structure flexible pour gérer les requêtes et les réponses HTTP. Il permet de définir facilement des routes, de gérer les middlewares et d'intégrer des bases de données ou d'autres services. Express.js est largement utilisé pour développer des applications web backend rapides et modulaires.

MongoDB est une base de données NoSQL orientée document qui stocke les données sous forme de documents JSON flexibles. Mongoose est une bibliothèque ODM (Object Data Modeling) pour MongoDB qui impose des schémas stricts aux documents, facilitant la structuration, la validation et la manipulation des données en TypeScript.

Avantages :

- ➔ Pour l'application, le choix d'une base de données non relationnelle est adaptée à notre variété de données minimale et à leur relation : Nous sommes simplement intéressés à sauvegarder les informations des utilisateurs et de leurs projets.
- ➔ Pour le développement, la combinaison MongoDB-Mongoose-Express est celle enseignée en IFT3225 et fait partie des préférences de nos développeurs. La manipulation des données en utilisant les schémas et les abstractions fournies par Mongoose permet d'éviter de concevoir des requêtes SQL élaborées et susceptibles aux erreurs. Les options de filtrage et de sélection de collections sont simples et puissantes et simplifient la conception de notre API. Express.js permet une gestion de la connectivité, des routes et des contrôleurs qui est accessible et claire. Express dicte une structure épurée et une séquence claire dans l'ordonnancement des middlewares. Le transfert de données et le paramétrage à l'aide d'objets ou de JSON est clair et consistent dans toute l'application.

Conception

Prototype

Fig 1. Prototype de la page de création de cartes

Voici la page de création de cartes réalisée avec Figma[3]. Son apparence se rapproche beaucoup de Figma et Photoshop / Illustrator, étant eux aussi des applications de design.

[3] Prototype Figma: https://www.figma.com/design/M7quMfZrLLWfgbTmsTg0yl/CALQUE?node-id=25-2&t=WZJA1kvVXcml4tiw-1

Le prototype est divisé en 4 parties:

Navbar

La **Navbar** contient les outils principaux pour manipuler le canvas, ainsi que le bouton pour avoir un aperçu de la carte avec ses interactions, comme sur Figma. Il y a aussi le titre du projet (carte globale) et le titre de la carte (sous-carte-1) Enfin, il y a le changement de mode représenté (modifiable et aperçu) par un bouton *toggle* et un bouton *Play* qui permet de simuler les interactions possibles de la carte.

Canvas

Le **Canvas** permet de créer la carte à l'aide des outils situés sur la Navbar

Leftbar

La **Leftbar** ou Left Sidebar permettrait d'ajouter des étages ainsi que de définir des lignes et des tags. Cependant, ces fonctionnalités n'ont pas été développées, c'est pourquoi la Leftbar, qui est implémentée, n'est pas fonctionnelle.
Un projet calque aurait pu contenir plusieurs cartes (sous-cartes), et une carte aurait pu contenir plusieurs étages. La carte d'un bâtiment avec plusieurs étages, comme un centre commercial, une station de métro ou des bureaux d'entreprise, en est un bon exemple.
Les lignes auraient défini un chemin et imposé une apparence et des attributs spécifiques aux entités (nœud, edge, image).
Les tags auraient ajouté davantage de propriétés à une entité. Ils auraient été des marqueurs qui peuvent dicter le style ainsi qu'ajouter certains services. Un service aurait été un petit symbole, potentiellement un émoji, qui représente rapidement de l'information sur une certaine entité. L'utilisateur aurait pu consulter une description sommaire du service.

Rightbar

La **Rightbar** ou Right Sidebar pour changer l'apparence et les propriétés d'une entité (nœuds, edges et images). Pour les nœuds, il est possible de lui ajouter des lignes et des tags, ainsi que de les connecter à d'autres cartes. Si un nœud représente, par exemple, un bâtiment (ou une station sur la carte du métro de la STM), on pourrait cliquer sur ce nœud pour montrer la station en détail, et se diriger à l'intérieur de la station).

Pour résumer, la Leftbar définit les propriétés globales du projet et de ses cartes, alors que la Rightbar définit les propriétés d'une entité spécifique.

Diagramme d’architecture de haut niveau:

(Voir Figure 3 en Annexe)

Il existe une séparation nette entre le backend et le front-end, tel que les deux entités sont conçues avec l’idée que chaque partie pourrait être hébergée sur un système différent.

L’application React communique via le protocole HTTP avec le serveur pour effectuer les différentes requêtes CRUD. La principale communication entre les deux entités est l’échange d’informations concernant la sauvegarde et le chargement des projets de cartes interactives. Le corps de la requête d’une sauvegarde / chargement de projet est une version sérialisée du corps du canvas, comportant la structure imbriquée du SVG . La communication entre les deux entités suit l’architecture REST.

Les utilisateurs manipulent les éditeurs, qui spécifient les valeurs utilisées pour altérer le rendu des tags SVG comme <Circle/> , <Line/> et <Image/>. Les éditeurs s’affichent selon le type de l’élément sélectionné dans le contexte de l’application et dévoilent les attributs dans les champs éditables. Si l’éditeur altère la valeur des champs, alors le composant react associé va pouvoir mettre à jour le tag SVG relié en changeant l’attribut altéré.

Lorsque les utilisateurs utilisent les différents outils dans la barre de menu, l’état de l’application est changé en accordance. Les composants représentant les éléments interactifs de l’application sont aussi ceux qui réagissent aux clics et aux entrées. Plusieurs responsabilités leur sont accordées, comme un rendu conditionnel particulier s’ils sont sélectionnés et le déclenchement de certains évènements (ajout d’une arête).

Lorsque les utilisateurs interagissent avec le canvas, ils peuvent entre autres déplacer des nœuds, sélectionner des éléments, ajouter des éléments et supprimer des éléments.

Les changements apportés par les utilisateurs sont stockés dans leur composant React, qui contient aussi l’élément SVG et les attributs associés à son affichage. L’entièreté du canvas peut être représenté par son contexte GraphContext, qui est aussi stocké dans le localStorage du client pour permettre une certaine persistance. Lorsqu’on exporte le projet, on exporte le contenu du GraphContext pour y extraire le SVG. L’utilisateur peut télécharger l’image ou la sauvegarder dans la base de données. Éventuellement, l’utilisateur pourra utiliser le nom de son projet pour le retrouver et l’afficher sur l’application.

L’architecture avait initialement été conçue pour permettre davantage de possibilités d’améliorations et ajouts futurs. Un couplage plus restrictif et moins modulaire a été choisi au final, afin de permettre un développement plus rapide. Les avantages du couplage plus intime avec les fonctionnalités de React sera discuté dans le diagramme de bas niveau.

Diagramme d’architecture de bas niveau :

(Voir Figure 4 en Annexe)

1. Ajouter un Node / Edge / Image

Pour l’utilisateur, l’ajout d’éléments interactifs sur la carte dépend des outils dans la barre de menu. Ils sont représentés par des boutons qui mettent à jour l’état du système “tool” en utilisant un contexte React appelé “AppContext”. AppContext stocke aussi les états “page” et “mode” qui aident à spécifier certaines conséquences et adapter le rendu conditionnel de certains composants react. Lorsque l’état “tool” est spécifié à la valeur “node”, le canva réagit différemment au déclencheur “onClick” et répond en appelant le module “insertNode”, qui s’occupe de mettre à jour les éléments de notre tag <SVG> avec id = “canvas”. InsertNode accomplit ce travail en déconstruisant l’interface GraphState et en ajoutant à son attribut “NodeState” (tableau de Nodes) le nouveau Node. L’ajout d’images et d’arêtes fonctionne de la même manière, mais on mettra plutôt à jour EdgeState et ImageState.

2. Modifier un Node / Edge / Image

Pour modifier les paramètres affectant le rendu d’un élément SVG comme <Circle/>, <Line/> et <Image/>, les éditeurs sont utilisés. Ils mettent à disposition aux utilisateurs des champs qui, lorsque changés, déclencheront l’évènement “onChange” qui sera interprété par leur composant React respectif. Le composant NodeEditor répond avec la fonction “updateField”, qui récupère l’attribut et la valeur du champ. Le module updateNode pourra ensuite modifier le nœud en utilisant une méthode similaire à l’ajout d’un nœud, soit en déconstruisant et reconstruisant GraphState, accessible globalement grâce au contexte GraphContext.

3. Déplacer un Node / Edge / Image

Les éléments interactifs contribuent à informer les contextes de la “sélection” actuelle. Lorsqu’on active le déclencheur “onClick” d’un élément interactif de la carte, le handler du composant React de l’élément change les états des contextes SelectedEntityContext et CanvasContext pour informer de la sélection et de l’action du canvas à prévoir. Lorsqu’on interagit avec le canvas, les déclencheurs onMouseMove et onMouseUp s’occupent d' effectuer un déplacement (en utilisant le module updateField) et de l’arrêter.

4. Exporter un projet

Le projet est imbriqué dans la balise HTML <SVG id=”canvas”>. Il peut être accédé en manipulant le DOM (document.getElementById("canvas")), en utilisant le contexte GraphContext ou en chargeant le contenu du local storage (mémoire du navigateur). Lorsque l’utilisateur active le bouton associé à l’export ou à la sauvegarde, la méthode appropriée du composant ModeSwitcher.tsx est appelée. À l’export, le contenu de l’élément SVG est sérialisé, puis automatiquement téléchargé à l’aide d’un élément <a> créé temporairement. Le document SVG sera téléchargé sous le nom ‘myMap.calque.svg’, pour qu’à l’import on puisse en vérifier le format avec le substring ‘.calque’. Le corps d’une requête HTTP (POST) est chargé avec le contenu sérialisé du projet. Une vérification est faite pour constater si le projet existe déjà avec un nom similaire, si oui on change pour une requête PUT à la place en utilisant le API call de la route de la collection “projet”.

Application

Spécification de l'implémentation

Frontend

Pour implémenter le frontend de l'application, nous avons utilisé React avec Vite. C'est l'un des outils le plus répandu pour exécuter un serveur local de développement et compiler les fichiers TypeScript. Vite permet aussi le "hot-reloading", qui recharge l'application à chaque changement dans le code en fonction des modifications, de manière dynamique et instantanée. Cela permet de voir les changements en temps réel sans avoir à relancer l'application, ou rafraîchir la page. Il est aussi possible de détecter les erreurs à travers l'application. Elles sont affichées sur la page de l'application, ou la page devient toute blanche.

L'interface graphique se divise en plusieurs composants principaux:

- Le header, contenant les contrôles de mode d'édition.
- Le canvas, offrant une représentation visuelle du réseau, et permettant d'interagir avec ce dernier.
- La rightbar, présentant les propriétés de la dernière entité sélectionnée.

De par son interactivité, l'application possède un état complexe. Ainsi, nous avons choisi d'utiliser un état unique pour centraliser les données décrivant le réseau (nœuds, arêtes, coordonnées de ces derniers…). Nous utilisons également quelques états auxiliaires pour gérer d'autres fonctionnalités de l'éditeur (mode d'édition, dernière entité sélectionnée).

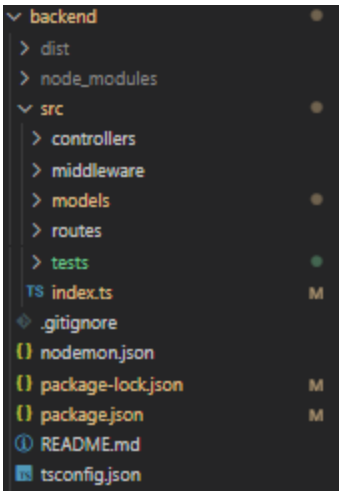
Les composants de l'interface échangent les identifiants des entités qu'ils décrivent, et utilisent des contextes React, pour faire appel à l'état central et obtenir les propriétés de ces entités. Ainsi, chaque état utilisé dans de nombreuses parties de l'application possède son propre contexte, qui peut être lu et écrit par les composants utilisant ce contexte.

Les principaux états et contextes sont:

- La carte, contenant les informations des nœuds, arêtes et lignes.
- Le mode d'édition de l'interface (sélectionner, créer un nœud, créer une arête).
- L'entité sélectionnée (assigné lorsque l'on clique sur une entité de la carte, lu par la rightbar.
- L'action en cours sur la carte (bouger un nœud ou une image, créer une arête à partir d'un point).
- L'état de l'application (Les pages, mode édition et mode aperçu / preview)

Les événements sont gérés par des fonctions internes aux composants qui appellent les logiques de lecture et d'écriture de l'état.
Voir Annexe pour des exemples (Fig. 5 à 7)

Backend



Dans un backend utilisant Express.js, Mongoose et MongoDB, plusieurs éléments de l'implémentation sont typiques et l'architecture demeure quelque peu variée entre les serveurs. La structure du répertoire inclut le fichier d'entrée (index.ts), un dossier pour les routes, un dossier pour les middlewares (fonctions intermédiaires), un dossier pour les contrôleurs (fonctions end-point), un dossier pour les modèles Mongoose et un dossier pour les tests unitaires. Cette structure peut être altérée après la conversion du Typescript en Javascript (déploiement), mais la logique demeure la même.

Le fichier index.ts est responsable de connecter l'application express à la base de données MongoDB via Mongoose, il spécifie aussi le port de communication avec le frontend et communique les requérants autorisés avec une whitelist spécifiée par les paramètres CORS. On peut aussi y spécifier des paramètres additionnels comme la taille maximale d'un payload à l'aide de dépendances comme bodyParser. Les routes de base de l'API et les middlewares globaux comme les loggers y sont spécifiés Express.js.

```typescript
import express from 'express';
import ProjectController from '../controllers/project.controller';

const router = express.Router();

// Route: GET all projects
router.get('/', ProjectController.getAllProjects);

// Route: GET project by ID
router.get('/:id', ProjectController.getProjectById);

// Route: POST create a new project
router.post('/', ProjectController.createProject);

// Route: PUT update project by ID
router.put('/:id', ProjectController.updateProject);

// Route: DELETE project by ID
router.delete('/:id', ProjectController.deleteProject);

export default router;
```

Les routes principales aident à rediriger les requêtes concernant une collection vers le contrôleur responsable d'un type particulier de requête. Une route pour toutes requêtes CRUD habituelles y sont définies et la méthode appropriée du contrôleur affilié sera appelée. On peut aussi utiliser les routes pour ajouter des middlewares, notamment dans les routes nécessitant des privilèges ou une forme d'authentification. Pour l'instant, l'authentification n'est pas implémentée dans le backend, car il n'existe pas les éléments du frontend pour le justifier, mais le token d'authentification est habituellement présent dans le header de la requête et peut être vérifié par un middleware pour permettre de finaliser la requête.

```typescript
// POST create a new project
public async createProject(req: Request, res: Response): Promise<void> {
    try {
        // Destructure the properties from the request body
        const { title, description, content, creator } = req.body;
        //create the document
        const newProject: IProject = new Project({
            title,
            description,
            content,
            creator // Assuming creator is the _id of a user
        });
        const savedProject: IProject = await newProject.save();
        res.status(201).json(savedProject);
    } catch (err:any) {
        res.status(400).json({ message: err.message });
    }
}
```

Les endpoints sont les contrôleurs de chaque collection. Ils utilisent le contenu du corps de la requête pour spécifier les entrées nécessaires à ajouter, retirer ou modifier. Le corps de la requête peut aussi représenter un filtre pour acquérir une entrée particulière. Le routing et la communication avec le client par le protocole HTTP est dicté par les méthodes de Express.js et la manipulation des collections MongoDB est faite par l'ODM Mongoose. On peut conséquemment utiliser les méthodes de mongoose pour faciliter le filtrage ou la manipulation des collections (tables) sans avoir à formuler des requêtes complexes. L'envoi d'une réponse et d'un statut complète la requête.

Gestion des données

Frontend

Dans la version finale de notre application, les données sont fortement liées à une implémentation en composants React.

1. La carte interactive

Canvas.tsx est le composant react responsable du rendu du tag <SVG id=”canvas”> agissant comme parent de tous les éléments SVG imbriqués. Il sert aussi à informer de l’état du système “CanvasContext” en encapsulant le canvas, spécifiant l’action à effectuer lorsque le canvas est manipulé par les entrées de l’utilisateur.

2. Les éléments de la carte

Node.tsx, Edge.tsx, Image.tsx sont responsables de gérer les déclencheurs sur leur élément svg associé. Ils contribuent aussi à changer certains états, comme la sélection (“SelectedEntitycontext”) et les actions (“CanvasContext”).

3. Les états du système

Les divers contextes (CanvasContext, SelectedEntitycontext, AppContext, GraphContext, …) aident à communiquer les états globaux du système et fournissent une structure inspirée du design patterns Provider-Consumer. Les contextes fournis par React aident à définir les actions à prendre lors des entrées des utilisateurs et à afficher les pages appropriées.

4. La persistence

Le navigateur fournit quelques megabytes afin de préserver de l’information à court terme, même si l’on ferme l’application. Le contenu du GraphContext y est sauvegardé afin de revenir sur le projet rapidement, sans nécessairement avoir à stocker le contenu sur la database.

Backend

1. Les utilisateurs

Les collections des utilisateurs sont rudimentaires et n’utilisent que les adresses courriel pour indexer le contenu. Le concept d'authentification n’est pas implémenté dans le frontend, alors les routes et contrôleurs du backend ne sont pas mobilisés dans notre contexte.

```
const ProjectSchema: Schema = new Schema({
    title: {
        unique:true,
        type: String,
        required: true
    },
    description: {
        type: String,
        default:''
    },
    content: {
        type: String,
        required: true
    },
    creator: {
        type: String, //type = object schema User
        default:'unknown'
    },
    createdAt: {
        type: Date,
        default: Date.now
    }
});
```

2. Le projet contenant la carte interactive

Le contenu (content) du projet correspond au <SVG id=”canvas”> et tout son contenu imbriqué. Le projet est stocké dans la collection sous forme sérialisée. Le titre du projet est spécifié par l’utilisateur dans le frontend à l’aide d’un champ contenu dans le composant associé au titre du projet. Lors de la sauvegarde, on spécifie les champs dans le corps de la requête.

Évaluation/Tests et documentation

Tests

Les tests unitaires implémentés concernent le backend, plus précisément le retour prévu après l'envoi d'une requête HTTP. Considérant la structure rudimentaire de nos schémas Mongoose, peu de restrictions autre que le type des attributs (colonnes) et de l'unicité des clés importe.

Chaque collection ("project" et "user") contient au moins 5 routes/contrôleurs à tester.

Ce qu'on peut tester sur un backend Express-Mongoose-MongoDB:

- Est-ce que la réponse (res) est valide?
    - Bon code HTTP?
    - Bon response body?
- Middleware (non implémenté pour le middleware)
    - Pertinent pour l'authentification (nécessaire à certaines routes)
- Est-ce que nos schémas mongooses sont bien structurés?
    - Typage
    - Champs requis
    - Relation entre les collections
- Contrôle des entrées / requêtes (req)
    - Prévenir les vulnérabilités & Contrôle des requérants (CORS)
    - Format / Structure de la requête (Header & Body)

Technologie utilisée pour les tests unitaires:

Les tests unitaires ne sont pas finalisés en la date de remise du rapport, mais Jest provenant de la dépendance supertest permet de compléter quelques-uns des points listés plus haut. On peut notamment tester les routes avec des entrées artificielles et tester les codes d'erreurs prévus pour les différentes routes. Les tests sont mieux documentés dans le README.md du backend.

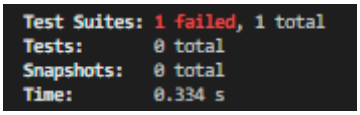


Figure 2.
Exemple de test sur une route.

Discussion

Une des difficultés rencontrées est l'inexpérience de certains membres de l'équipe avec React. La structure des states, contextes et différents hooks peut porter à confusion, et trouver quelle section du code produit un certain résultat attendu peut s'avérer être un casse-tête. Une autre difficulté rencontrée est un bug lorsqu'on essaie de coller une image dans Chrome, qui ne survient pas avec Firefox. Le problème survient lorsqu'on essaie de coller l'image alors que le canvas est vide. Pour le contourner, on doit d'abord créer un nœud sur le canvas, puis avec l'outil de sélection, s'assurer que ce nœud n'est plus sélectionné. Alors, les images peuvent être collées normalement.

Il y a quelques fonctionnalités qui n'ont pas pu être implémentées, dû au temps limité du trimestre. Une de ces fonctionnalités est le concept d'étages, similaires au layers qu'on retrouve dans d'autres applications de design. Cela permettrait, en quelque sorte, de relier des nœuds appartenant à différentes cartes. Le concept d'étiquettes a aussi été discuté. Cela permettrait de déterminer certaines valeurs pour certains attributs d'un nœud, et ensuite de pouvoir l'assigner à plusieurs nœuds, plutôt qu'avoir à aller changer les mêmes valeurs à chaque fois pour chaque nœud. Une autre piste pour pallier ce problème serait de conserver en mémoire l'état des éditeurs, permettant ainsi d'attribuer les mêmes valeurs aux attributs des nouveaux nœuds créés, plutôt que de toujours revenir aux valeurs par défaut.

Au niveau de la Navbar, la gestion du contexte selon un lien modifié à la main ou un clic sur "revenir en arrière" n'a pas été implémenté, causant un bug sur l'état de la navbar, qui n'est pas la navbar attendu pour la page affichée.

Un autre point qui pourrait être amélioré avec notre code est le découplage. Certains composants, dont Node.tsx, ont trop de responsabilités. En respectant mieux les *design patterns* existants, ce problème pourrait être facilement réglé.

Une potentielle amélioration pourrait aussi impliquer l'introduction d'un système global et centralisé de gestion des états, comme celui proposé par Redux, qui faciliterait l'accès aux états de l'application et leur manipulation avec une architecture en flux. L'architecture redux implémente les *reducers* et leurs *actions* qui dictent les changements sur les états, préviennent les collisions et optimisent le code en diminuant le re-rendering.

Conclusion

En conclusion, notre projet permet la création facile de cartes variées, et permet à l'utilisateur d'interagir minimalement avec ces dernières. Il serait intéressant de pousser ce projet à de futures itérations, durant lesquelles on pourrait ajouter des informations supplémentaires avec lesquelles interagir. Par exemple, le temps de déplacement sur une certaine arête, ou les services offerts à certains nœuds, seraient des informations utiles à ajouter. De plus, il serait intéressant d'offrir un outil pour accélérer le stylage des nœuds et des arêtes dans l'éditeur, tels les tags et les lignes mentionnés plus haut. En outre, pour étendre l'échelle du projet, l'implémentation du concept d'étages est à considérer. En fin de compte, on termine le trimestre avec une solide base pour notre projet, qui est d'une envergure trop grande pour être complété en un seul été.

Ce projet nous aura permis d'explorer toutes les étapes de création d'une application, de l'élaboration des exigences au code en soi, en passant par le prototypage, le travail d'équipe et la communication. Ces apprentissages sont des qualités cruciales pour le marché du travail.

Bibliographie

- [Anson the Developer]. (2024). *Express JS Full Tutorial Playlist 2024* [vidéo]. YouTube. https://www.youtube.com/watch?v=P6RZfI8KDYc&list=PL_cUvD4qzbkwjmjy-KjbieZ8J9cGwxZpC
- [ByteGrad]. (2023). *STOP using useState, instead put state in URL (in React & Next.js)* [vidéo]. YouTube. https://www.youtube.com/watch?v=ukpgxEemXsk
- [Codevolution]. (2021). *React Redux Tutorial* [vidéo]. YouTube. https://www.youtube.com/watch?v=9boMnm5X9ak&list=PLC3y8-rFHvwheJHvseC3I0HuYI2f46oAK
- Cook, P. (2023). d3indepth.com. https://www.d3indepth.com/
- *Data Visualization Certification*. (s. d.). freecodecamp.org. https://www.freecodecamp.org/learn/data-visualization/
- *Documents - MongoDB Manual v7.0*. (2024). mongodb.com. https://www.mongodb.com/docs/manual/core/document/
- [freeCodeCamp.org]. (2024). *CRUD API Tutorial – Node, Express, MongoDB* [vidéo]. YouTube. https://www.youtube.com/watch?v=_7UQPve99r4
- Herrington, J. [Jack Herrington]. (2022). *Mastering React Context: Do you NEED a state manager?* [vidéo]. YouTube. https://www.youtube.com/watch?v=MpdFj8MEuJA
- *Installation - Tailwind CSS*. (s. d.). tailwindcss.com. https://tailwindcss.com/docs/installation useContext – React
- *JavaScript and HTML DOM References*. (2024). w3schools.com. https://www.w3schools.com/jsref/
- *Map - JavaScript | MDN*. (s. d.). developer.mozilla.org. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Map
- *Mongoose v8.5.2: Schemas*. (s. d.). mongoosejs.com. https://mongoosejs.com/docs/guide.html
- [Programming with Mosh]. (2022). *TypeScript Tutorial for Beginners* [vidéo]. YouTube. www.youtube.com/watch?v=d56mG7DezGs
- *The starting point for learning TypeScript*. (2024). typescriptlang.org. https://www.typescriptlang.org/docs/
- *TypeScript Tutorial*. (2024). w3schools.com. https://www.w3schools.com/typescript/
- *useContext - React*. (s. d.). react.dev. https://react.dev/reference/react/useContext
- *useState - React*. (s. d.). react.dev. https://react.dev/reference/react/useState
- [Web Dev Simplified]. (2019). *Build A REST API With Node.js, Express, & MongoDB - Quick* [vidéo]. YouTube. https://www.youtube.com/watch?v=fgTGADljAeg
- [Web Dev Simplified]. (2020). *Learn Express Middleware In 14 Minutes* [vidéo]. YouTube. https://www.youtube.com/watch?v=lY6icfhap2o
- [Web Dev Simplified]. (2021). *Learn Express JS In 35 Minutes* [vidéo]. YouTube. https://www.youtube.com/watch?v=SccSCuHhOw0
- [Web Dev Simplified]. (2023). *Learn TypeScript Generics In 13 Minutes* [vidéo]. YouTube. https://www.youtube.com/watch?v=EcCTIExsqmI
- [Web Dev Simplified]. (2024). *State Managers Are Making Your Code Worse In React* [vidéo]. YouTube. https://www.youtube.com/watch?v=VenLRGHx3D4

Annexe

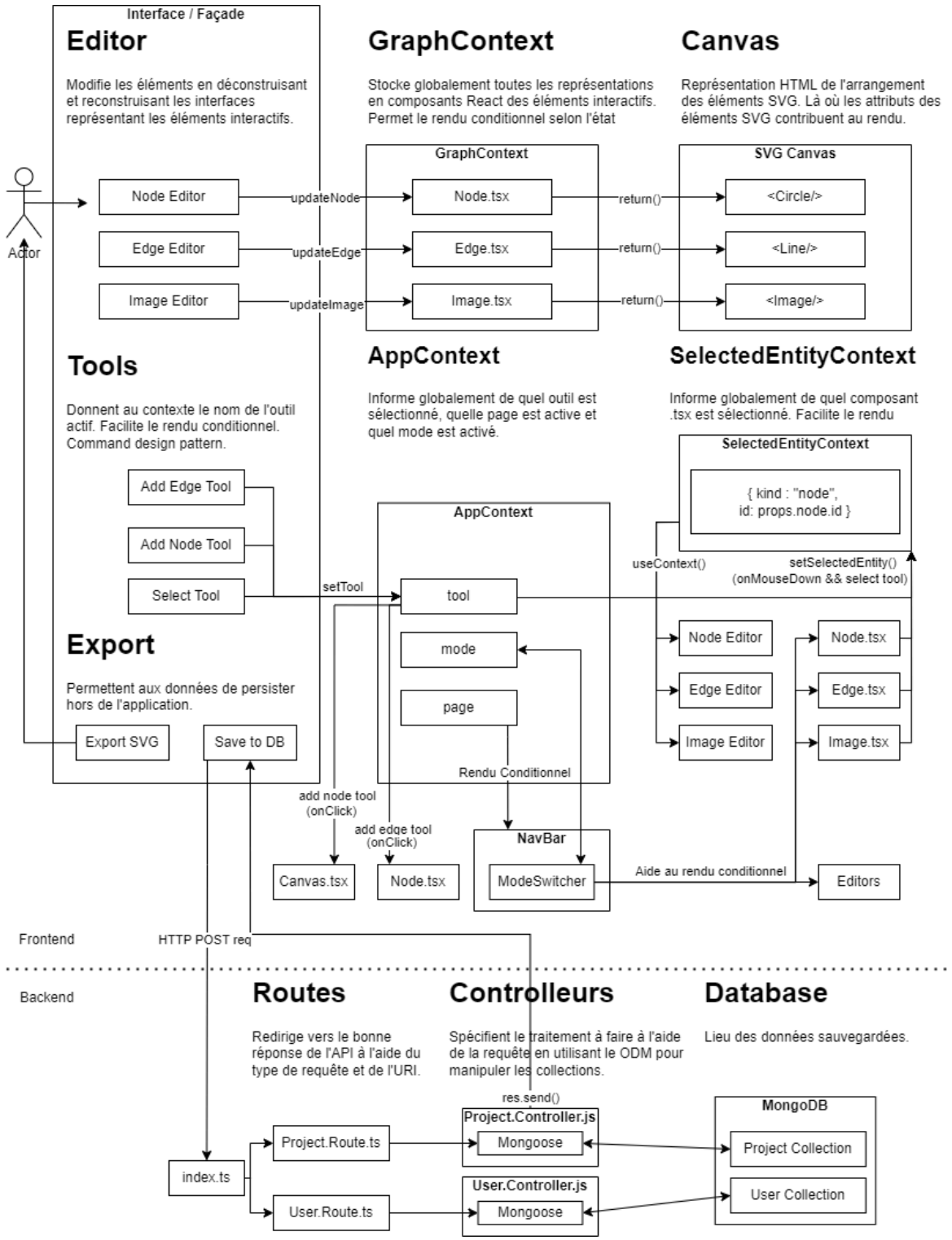


Figure 3: Diagramme de Haut Niveau

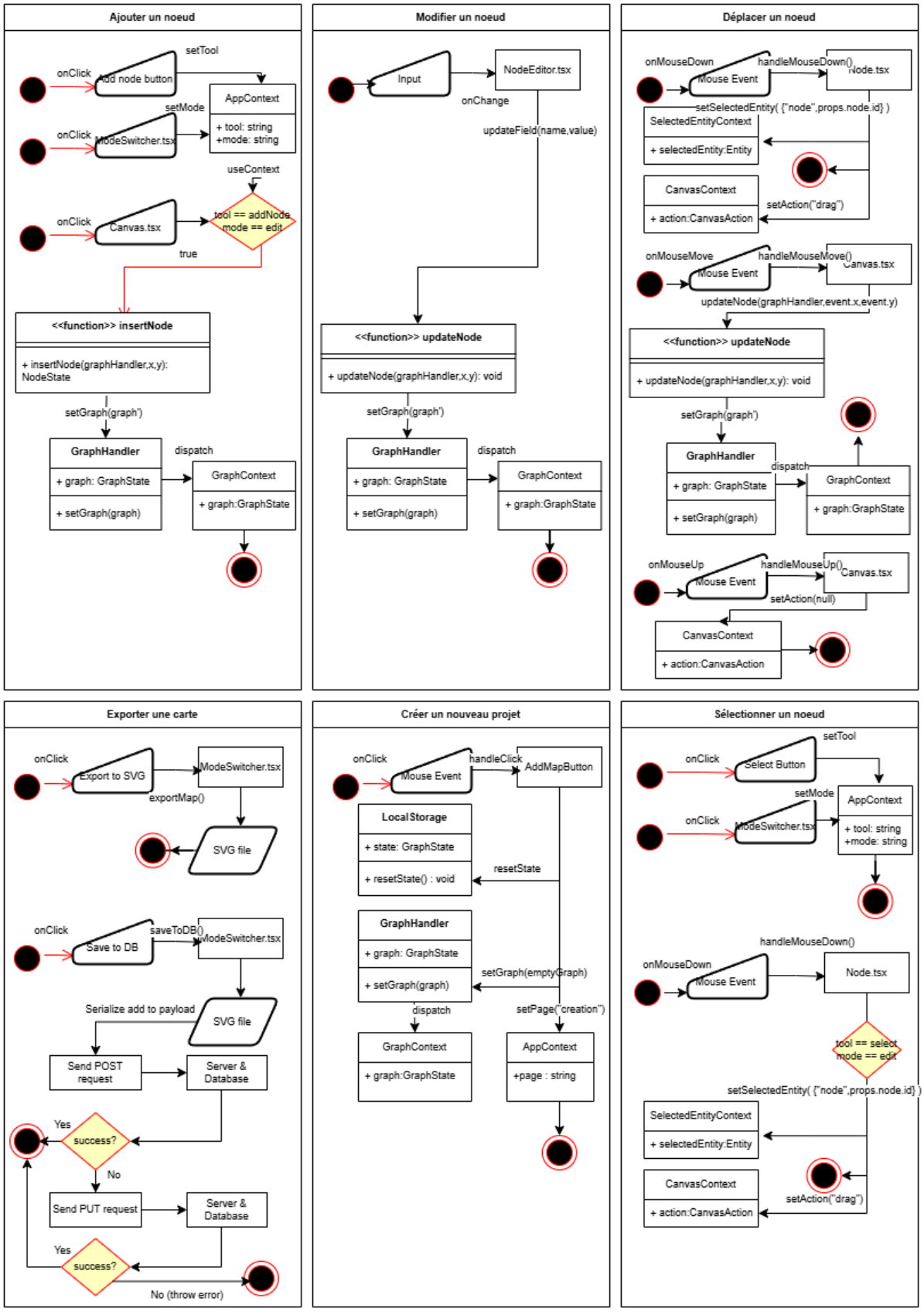


Figure 4: Diagramme de bas niveau

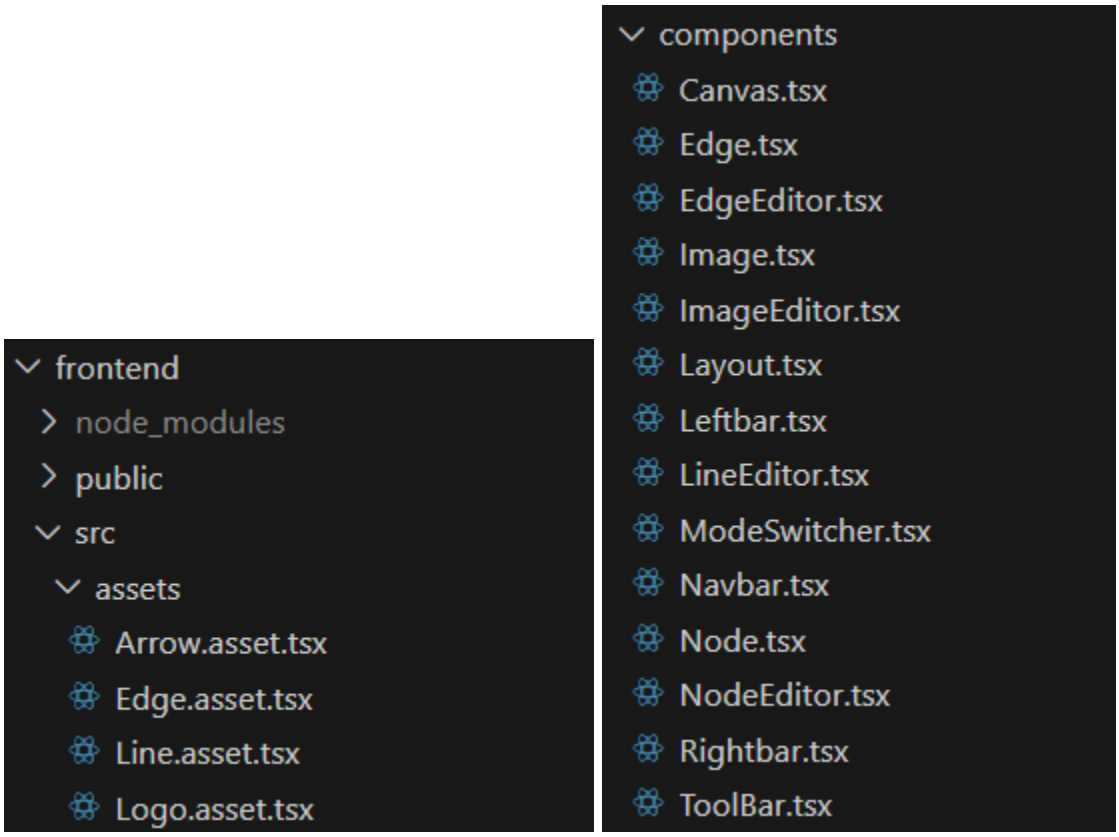


Fig. 5 Structure du code (assets et components)

```tsx
import { GraphHandler } from "../components/Layout";

interface GraphState {
  autoIncrement: number;
  nodes: NodeState[];
  edges: EdgeState[];
  images: ImageState[];
}

interface NodeState {
  id: number;
  name: string;
  x: number;
  y: number;
  color: string;
  size: number;
```

```
  stroke: string;
  strokeWidth: number;
  description: string;
}

function getNode(graph: GraphState, nodeId: number): NodeState {
  const node = graph.nodes.find(node => node.id === nodeId);
  if (node === undefined) {
    throw `No node ${nodeId} found in the graph.`;
  }

  return node;
}

function insertNode(handler: GraphHandler, x: number, y: number): NodeState {
  const node: NodeState = {
    id: handler.graph.autoIncrement,
    name: `node-${handler.graph.autoIncrement}`,
    x,
    y,
    size: 15,
    color: 'white',
    stroke: 'blue',
    strokeWidth: 10,
    description: '',
  };

  const graph = {
    ...handler.graph,
    autoIncrement: handler.graph.autoIncrement + 1,
    nodes: [...handler.graph.nodes, node],
  };

  handler.setGraph(graph);
  return node;
}


function updateNode(handler: GraphHandler, updatedNode: NodeState) {
```

```
  const graph = {
    ...handler.graph,
    nodes: handler.graph.nodes.map(node => node.id === updatedNode.id ?
updatedNode : node),
  };

  handler.setGraph(graph);
}

function deleteNode(handler: GraphHandler, nodeId: number) {
  const graph = {
    ...handler.graph,
    nodes: handler.graph.nodes.filter(node => node.id !== nodeId),
    edges: handler.graph.edges.filter(edge => edge.node1id !== nodeId &&
edge.node2id !== nodeId),
  };

  handler.setGraph(graph);
}

const emptyGraph = {
  autoIncrement: 0,
  nodes: [],
  edges: [],
  images: []
};
```

Fig. 6 - Extrait de State.ts qui définit le graphe et les entités (Exemple pour les noeuds)

```
type Page = 'menu' | 'creation' | 'enduser'
type Mode = 'view' | 'edit';
type Tool = 'select' | 'node' | 'edge' | 'pan';

type Entity =
  | { kind: 'node', id: number }
  | { kind: 'edge', id: number }
  | { kind: 'image', id: number }
```

```tsx
interface AppHandler {
  page: Page
  setPage: React.Dispatch<React.SetStateAction<Page>>;
  mode: Mode;
  setMode: React.Dispatch<Mode>;
  tool: Tool;
  setTool: React.Dispatch<Tool>;
}

interface GraphHandler {
  graph: GraphState;
  setGraph: React.Dispatch<GraphState>;
}

interface SelectedEntityHandler {
  selectedEntity: Entity | null;
  setSelectedEntity: React.Dispatch<Entity | null>;
}

const AppContext = createContext<AppHandler>(undefined as any);
const GraphContext = createContext<GraphHandler>(undefined as any);
const SelectedEntityContext = createContext<SelectedEntityHandler>(undefined as
any);
```

Fig. 7 - Contextes (dans Layout.tsx)