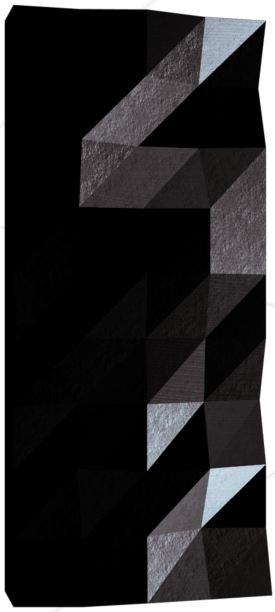


Red Tetris

Tetris Réseau à la Sauce de Pélicans Rouges

Résumé: L'objectif de ce projet est de développer un jeu de tetris multijoueurs en réseau à partir d'une pile logicielle exclusivement Full Stack Javascript

Table des matières

Chapitre I

Préambule

La société redpelicans est le sponsor de ce projet. Vous comprenez alors la tendance fortement marquée au rouge du jeu de Tetris que nous vous proposons de construire et l'envol de quelques pélicans sur vos terrains de jeu parmi une forêt de tetriminos.

redpelicans est spécialisée autour des 3 piliers techniques que sont :

- 1/ le Full Stack Javascript
- 2/ les bases de données NoSQL
- 3/ Docker

Suite au recrutement de collaborateurs de 42 et à leur formation sur les techniques du Full Stack JS, ils ont proposé de communiquer sur ces technos auprès de 42, en partant du constat que seul le modèle PHP était pratiqué : le projet `Red Tretis` était alors né ...

Chapitre II

Introduction

Tout le monde connaît le Jeu `Tetris` et tout le monde connaît `Javascript`, il ne reste donc plus qu'a construire un `Tetris en Javascript`.

Oui mais ...

Votre Tetris sera multijoueurs et en ligne. Il vous permettra ainsi de vous distraire dans des parties intergalactiques lors de vos longues nuits de code (reste quelques problèmes de WIFI sur certaines planètes).

Votre Tetris utilisera les toutes dernières technos `Javascript` qui sont au coeur d'une grande bataille intellectuelle, industrielle et financière entre `Facebook` et `Google` dont l'enjeu est d'être le maître du monde.

Votre Tetris nécessitera pas mal de jus de cerveau pour concevoir l'architecture, spécifier un protocole réseau asynchrone, implémenté, en `programmation fonctionnelle`, les algos d'animation de pièces et afficher graphiquement le tout en `HTML`!

Bon jeu, bon code ... et n'oubliez pas de tester et retester!!

Chapitre III

Objectifs

Les objectifs pédagogiques sont multiples, mais l'axe principal est d'introduire le langage `Javascript`, de découvrir son écosystème foisonnant et de mettre en oeuvre quelques uns des principes, techniques et outils phares du `Full Stack` Javascript.

Tout le monde dit connaître `Javascript`, mais très peu de personnes ont réellement une connaissance précise de ce langage aux multiples facettes à la fois partiellement fonctionnel, complètement orienté prototype, au type diaboliquement dynamique, passionnément asynchrone et redoutablement performant.

Au travers de l'écriture d'un jeu de Tetris en réseau, vous mettrez en oeuvre des principes fonctionnels (on vous l'impose), asynchrones client et serveur (par nature du langage) et réactifs (par nature du jeu et des `GUI`).

Vous aurez à écrire des tests unitaires qui devront être dignes d'une chaine industrielle de contiuous delivery.

Vous découvrirez aussi les derniers outils et librairies du Full Stack Javascript en vogue comme `Node.js`, `React.js` et `Redux.js`.

Chapitre IV

Consignes générales

Le projet doit être écrit totalement en `Javascript` et particulièrement dans sa version `es2015 (ES6)`.

Le code client (navigateur) doit être écrit sans appel à `"this"` dans le but de vous pousser à utiliser des constructions fonctionnelles et non objet. Vous avez le choix de la librairie fonctionnelle (lodash, ramda, ...) à utiliser ou pas. La logique de manipulation du tas et des pièces doit être implémentée sous forme de `"pure functions"`. Une exception à cette règle : `"this"` peut être utilisé pour définir ses propres sous classes d'`"Error"`.

A l'inverse le code serveur doit utiliser de la programmation orientée objet (prototype). Nous voulons y retrouver au minimum les classes (ES6) `Player`, `Piece` et `Game`.

L'application client doit être construite à partir des librairies `React` et `Redux`.

Le code HTML ne doit pas utiliser d'éléments `"<TABLE/>"`, mais doit être exclusivement construit à partir d'un layout `flexbox`.

Interdiction d'utiliser :

- Une librairie de manipulation du DOM comme `jQuery`
- Des canvas
- Du SVG (Scalable Vector Graphics)

Il est d'ailleurs inutile de manipuler directement le `DOM`.

Les tests unitaires devront couvrir au minimum 70% des statements, functions, lines et au minimum 50% des branches (voir ci-après).

Respecter l'organisation des répertoires du `boilerplate` (voir ci-dessous), soit :

test : tests unitaires
src : toutes les sources du projet
src/client : sources relatives au client
src/server : sources relatives au server
src/client/actions : actions redux
src/client/components : composants react
src/client/containers : containeurs react
src/client/middlewares : middlewares redux
src/client/reducers : reducers redux

Clients et serveur communiquent via `socket.io`, nous vous demandons d'utiliser un middleware Redux pour gérer l'envoi et la réception des messages. Une première version, qu'il vous faudra adapter, est disponible dans le `boilerplate`.

Chapitre V

Partie obligatoire

V.1 Tetris : le jeu

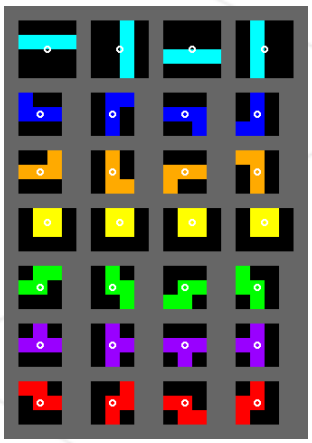
`Tetris` est un jeu de puzzle (voir Wikipedia), dont l'objet est de contenir le plus longtemps possible des pièces en chute dans un terrain de jeu. La partie est terminée lorsque le terrain n'offre plus suffisamment de place à une nouvelle pièce pour tomber. Lorsqu'une ou plusieurs lignes du terrain sont complètes, elles disparaissent, permettant ainsi de repousser l'échéance de fin de jeu.

Le jeu que vous allez construire reprend ces bases mais se joue entre plusieurs joueurs.

Chaque joueur possède son terrain de jeu, tous les joueurs subissent la même série de pièces. Dès lors qu'un joueur détruit des lignes sur son terrain, les joueurs adverses recoivent en malus n - 1 lignes, alors indestructibles, qui s'insèrent en bas de leur terrain.

Un terrain est formé de `10 colonnes` et `20 lignes`. Chaque joueur peut observer graphiquement la liste de ses adversaires (nom) et le spectre de leur terrain. Un spectre indique pour chaque colonne la première ligne occupée par une pièce sans détailler l'occupation des lignes suivantes. Dès mise à jour d'un terrain, tous les adversaires doivent visualiser l'évolution de son spectre.

Le jeu reprend les tetriminos historiques et leurs principes de rotation :



Il n'y a pas de score, le dernier joueur d'une partie est le gagnant. Le jeu doit être multi parties, mais doit aussi permettre de jouer en solitaire.

V.1.1 Déplacement des `pièces`

Les pièces "descendent" à pas et fréquence constantes (vitesse fixe). On appelle "tas" l'amas de pièces posées sur la partie inférieure du terrain. Sauf en cas de chute, une piéce ne s'intègre au tas (cesse d'être mobile) non pas au contact de ce dernier mais au pas suivant, cela permet d'ajuster sa position en contact du tas.

Les mouvements initiés par le joueur pour une pièce sont les suivants :

Flèches gauche et droite : Déplacement horizontal à droite ou gauche

Flèche du haut : Rotation (un seul sens est suffisant)

Flèche du bas : Chute en direction du tas

Barre d'espace : Déplacement vertical afin de positionner une piéce dans un trou du tas

V.2 Tetris : la technique

Le jeu s'appuie sur une architecture `client/serveur`. L'environnement d'exécution client est un navigateur (nous vous recommandons une version evergreen). Le serveur sera écrit avec `NodeJS`. Clients et serveur communiquent via `http`.

Le serveur sera en charge :

- De la gestion des parties et des joueurs.
- De la distribution des pièces pour chaque partie.
- De la diffusion des spectres

`Rappel : chaque joueur d'une même partie doit recevoir, peut être dans des temps différents, les mêmes pièces dans les mêmes positions et aux mêmes coordonnées.`

La communication entre le serveur et les clients est `événementielle` et `bi-directionnelle`, vous utiliserez socket.io pour sa mise en oeuvre.

Aucun système de persistance n'est nécessaire.
Le client implémente une architecture de type Single Page Application.

V.2.1 Gestion des parties

Chaque joueur se connecte à une partie via une hash-based url de type :

- `http://<server_name_or_ip>:<port>/#<room>[<player_name>]`

room : nom de la partie à rejoindre
player_name : nom du joueur

Le premier à rejoindre une partie, en sera le responsable, il en aura le contrôle, il pourra la lancer à sa guise. A la fin, il sera le seul à pourvoir la relancer. En cas de départ, un des joueurs restant prendra ce rôle.

Un joueur ne peut rejoindre une partie en cours de jeu. Il doit attendre sa fin, il pourra dès lors la rejoindre et y participer quand le responsable décidera de la lancer.

Une partie est terminée lorsqu'il ne reste plus qu'un seul jouer, c'est alors le gagnant.

Une partie peut se jouer avec un seul joueur.

Plusieurs parties peuvent être organisées simultanément.

V.2.2 Construction du Serveur

Dans le jeu, le serveur est en charge de la `gestion des parties`, de la `distribution des pièces` et de `la diffusion les spectres` des terrains des joueurs. Nous vous invitons à identifier dès le début très précisément le partage de responsabilités entre clients et serveur afin de spécifier le protocole réseau du jeu.

Techniquement, le serveur est une boucle asynchrone en charge du traitement des événements émis par les clients. `Socket.io` permet de recevoir et d'emettre les événements vers un ou plusieurs joueurs simultanément pour une même partie.

Il offre un service `HTTP` (en plus de socket.io) dont la seule finalité est de fournir, au lancement de la connexion du client, les fichiers `index.html` et `bundle.js`, voire quelques ressources supplémentaires.

V.2.3 Construction du Client

Le client s'exécute au sein d'un navigateur dans une architecture de type `Single Page Application` :

- A la première requête, le navigateur récupère du serveur un fichier `index.html` qui fait référence via une balise "<script/>" à un fichier `Javascript` (bundle.js) qui contient l'ensemble du code de l'application cliente.

- Le navigateur exécute bundle.js et dès lors il n'y a plus d'échanges de fichiers HTML entre serveur et client, ce dernier est totalement autonome pour le rendu graphique et pour la gestion de la logique applicative. Seules des données seront échangées avec le serveur, échanges bi-directionnels dans notre cas via `socket.io`.

Nous vous demandons d'utiliser les deux librairies suivantes pour la construction du client :

- React : C'est le V de l'acronyme MVC qui vous permettra de construire l'interface graphique

- Redux : Librairie qui vous permettra de gérer les états de votre application. Elle offre une réponse fonctionnelle à une problématique réactive qui permet d'organiser la diffusion des événements entre les vues et le modèle.

En complément de ces deux librairies vous pouvez totalement faire appel à l'écosystème de modules Javascript :

- `Fonctionnel` : lodash, ramda sont des solutions très pratiquées, mais pas indispensables, `ES6` offre en standard plusieurs opérateurs ensemblistes (map, reduce)

- `Immutabilité` : l'un des principes de redux est l'immutabilité de son modèle. A l'instar du fonctionnel, Immutable.js vous offre des structures immutables, mais ES6 propose aussi quelques syntactic sugars pour éviter les effets de bords.

- `Asynchronisme` : La diffusion des événements dans `Redux` est exclusivement synchrone, or votre protocole client / serveur est fortement asynchrone, il vous faudra donc un peu d'aide du coté d'un middleware comme redux-thunk ou bien redux-promise

V.2.4 Boilerplate

La configuration d'une application `Full Stack Javascript` reste laborieuse car les technologies utilisées sont encore immatures (changement tous les 6 mois).

Nous vous proposons donc un starter kit vous évitant de consacrer de nombreuses heures à la mise en place de la configuration de base et vous permettant :

- D'exécuter le serveur
- De mettre en place un bundle des fichiers JS pour le navigateur
- D'exécuter les tests unitaires et de couverture.

Le `boilerplate` est disponible via le dépôt github red_tetris_boilerplate

La documentation est présente dans le Readme.

V.2.5 Tests

L'objet des tests est :

- D'accroître la fiabilité des versions livrées
- De diminuer le "time to market" en facilitant/automatisant la recette de chaque nouvelle version
- De renforcer la satisfaction du client et la pertinence des versions livrées en autorisant des cycles de développement plus cours

`Javascript` et son écosystème sont aujourd'hui totalement matures pour être au coeur de la stratégie d'une entreprise au même titre que `.Net` ou `Java` par le passé, on parle alors d'`Entreprise Javascript`. Un des éléments clef de la solution sera la définition d'un `pipeline` de tests permettant de rejeter au sein d'un workflow automatique une version logicielle défaillante.

Nous vous proposons ici de vous familiariser avec les tests unitaires (les premiers rencontrés dans le pipeline) et vous imposons comme contraintes que ces tests doivents couvrir 70% des lignes de code.

Précisement lors de l'exécution des tests vous obtiendrez 4 métriques :

- `Statements` : taux de couverture des déclarations
- `Functions` : taux de couverture des fonctions
- `Lines` : taux de couverture des lignes de code
- `Branches` : taux de couverture des chemins d'exécution du code

Votre objectif est de couvrir au minimum 70% des statements, functions, lines et au minimum 50% des branches.

Le boilerplate inclut une chaine de mesure de la couverture et de tests unitaires avec 3 exemples de tests unitaire (voir la documentation)

Chapitre VI

Partie bonus

Comme Red_tetris est, basiquement, un jeu vidéo, la possibilitée de rajouter des bonus est grande.
Nous vous proposons, a titre non exhaustif, les suivants :

- Ajouter un systeme de scoring durant la partie.
- Avoir une persistence de ces scores pour chaque joueur.
- Avoir plusieurs modes de jeu (pièces invisible, gravité augmentée, etc...)

Aussi, plusieurs passages techniques vous ont été imposé, dont l'utilisation du couple `React` / `Redux`.
L'objet était :

- De vous faire découvrir ce bînome très usité et recherché dans le milieu professionnel
- De faciliter le projet ainsi que sa correction

Une solution alternative aurait été d'utiliser une librairie FRP (Functional Reactive Programming). Paradigme passionnant à découvrir et parfaitement adapté au contexte. Nous vous proposons de découvrir flyd qui est une librairie minimaliste, mais avec une API très intéressante

Chapitre VII

Rendu et peer-évaluation

Rendez votre travail sur votre dépot `GiT` comme d'habitude. Seul le travail présent sur votre dépot sera évalué en soutenance.

Le jeu doit être totalement opérationnel.