



Red Tetris

Tetris Network with Red Pelicans Sauce

Eric BASLEY eric.basley@redpelicans.com
Red_Pelicans red_tetris@redpelicans.com
42 Staff pedago@42.fr

Summary: Summary: The objective of this project is to develop a networked multiplayer tetris game from a stack of software exclusively Full Stack Javascript

Contents

I	Preamble	2
II	Introduction	3
III	Objectives	4
IV	General instructions	5
V	Mandatory part	7
V.1	Tetris: the game	7
V.1.1	Moving the pieces	8
V.2	Tetris: the technique	8
V.2.1	Game Management	9
V.2.2	Server Construction	9
V.2.3	Client Construction	10
V.2.4	Boilerplate	11
V.2.5	Tests	11
VI	Bonus part	12
VII	Rendering and peer-evaluation	13

Chapter I

Preamble

[Redpelicans](#) is the sponsor of this project. You understand the trend strongly highlighted in red in the Tetris game - that we propose to build and flight some pelicans on your playgrounds in a forest of tetriminos.

[redpelicans](#) specializes in 3 technical pillars that are:

- 1/ Full Stack Javascript
- 2/ NoSQL databases
- 3/ Docker

Following the recruitment of 42 employees and their training in Full Stack JS, they proposed to communicate on these technos with 42, starting the fact that only the PHP model was practiced: the **Red Tretis** project was born ...

Chapter II

Introduction

Everyone knows the `Tetris` Game and everyone knows `Javascript`, it only remains to build a `Tetris` in `Javascript`.

Yes, but ...

Your `Tetris` will be multiplayer and online. It will allow you to disturb intergalactic parties during your long coding nights (There are still some WIFI issues on some planets).

Your `Tetris` will use the latest technologies `Javascript` which are at the heart of a great intellectual, industrial and financial battle between `Facebook` and `Google` whose challenge is to be the master of the world.

Your `Tetris` will require a lot of brain juice to design the architecture, specify an asynchronous network protocol, implemented in `functional programming`, create an algorithm of pieces' animation and display everything graphically in `HTML`!

Good game, good code ... and do not forget to test and retest !!

Chapter III

Objectives

The pedagogical objectives are multiple, but the main axis is to introduce the language **Javascript**, to discover its abundant ecosystem and to implement some of the principles, techniques and Flagship tools of **Full Stack Javascript**.

Everyone says they know **Javascript**, but very few people have a really precise knowledge of this multi- faceted language which is at the same time partially functional, completely prototype oriented, of a diabolically dynamic type, passionately asynchronous and frighteningly efficient.

Through the writing of a network Tetris game, you'll implement functional principles (which is required), asynchronous client and server (by nature of the language) and reagents (by nature of the game and **GUI**).

You will have to write unit tests that will have to be worthy of an industrial chain of [continuous delivery](#).

You will also discover the latest popular tools and libraries the Full Stack Javascript like `Node.js`, `React.js` and `Redux.js`.

Chapter IV

General instructions

The project must be written totally in Javascript and especially in its **es2015** (ES6) version.

The client code (browser) must be written without a call to **"this"** in the purpose of pushing you to use functional constructs and not object. You have the choice of the functional library ([lodash](#), [ramda](#), ...) to use it or not.

The handling logic of the heap and pieces must be implemented as **"pure functions"**. An exception to this rule: **"this"** can be used to define its own subclasses of **"Error"**.

On the opposite, the server code must use object-oriented programming (prototype). We want to find there at least the (ES6) **Player**, **Piece** and **Game** classes.

The client application must be built from the [React](#) and [Redux](#) libraries.

HTML code must not use **"<TABLE />"** elements, but must be built exclusively from a layout [flexbox](#).

Prohibition to use:

- A DOM manipulation library like **jQuery**
- **Canvas**
- **SVG** (Scalable Vector Graphics)

There is no need to directly manipulate the DOM.

Unit tests must cover at least 70 of the statements, functions, lines and at least 50 of branches (see below).

Respect the organization of the directories of **boilerplate** (see below), is :

test: unit tests

src: all sources of the project

src / client: customer-related sources

src / server: sources related to the server

src / client / actions: redux actions

src / client / components: react components

src / client / containers: react containers

src / client / middlewares: redux middlewares

src / client / reducers: redux reducers

Clients and server communicate via **socket.io**, we will ask you to use a [Redux middleware](#) to manage the sending and receiving of messages. The first version, which you will have to adapt, is available in the **boilerplate**.

Chapter V

Mandatory part

V.1 Tetris: the game

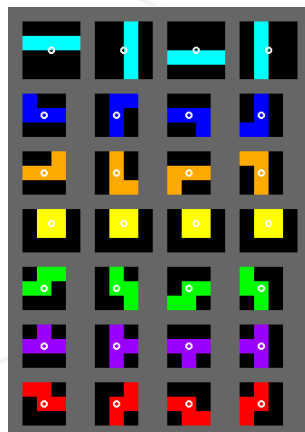
Tetris is a puzzle game (see [Wikipedia](#)), whose subject is to contain falling pieces as long as possible in a Playground. The game is over when the field no longer offers enough room for a new piece to fall. When one or more lines of land are complete, they disappear, allowing to postpone the expiry of the game.

The game you are going to build is based on these basics but is played between several players.

Each player has his own playing field, all players undergo the same series of pieces. As soon as a player destroys lines on his ground, the opposing players receive $n - 1$ lines in penalty, then indestructible, which fit at the bottom of their playground.

A terrain consists of 10 columns and 20 lines. Each player can graphically observe the list of his opponents (name) and the specter of their land. For each column, a spectrum indicates the first line occupied by a piece without providing any details about occupation of the following lines. As soon as the terrain is updated, all opponents must visualize the evolution of their spectrum.

The game takes the historical tetriminos and their principles of rotation:



If there is no score, the last player of the game is the winner. The game must be multi-part, but must also allow to play solo.

V.1.1 Moving the pieces

The pieces "go down" at a constant speed and frequency. The pile of pieces placed on the lower part of the terrain is called a "pile". Except in the event of a fall, a piece sticks to the pile (stops being mobile) not at the moment of contact with the pile, but at the next step, this makes it possible to adjust its position in contact with the pile.

The movements initiated by the player for a piece are as follows:

Left and right arrows: Horizontal move to the right or left

Top arrow: Rotation (only one direction is enough)

Down arrow: Fall towards the pile

Spacebar: Vertical move to position a piece in a hole in the pile

V.2 Tetris: the technique

The game relies on a **client** / **server** architecture. The client runtime environment is a browser (we recommend a version [evergreen](#)). The server will be written with **NodeJS**. Clients and server communicate via **http**.

The server will be in charge of:

- Management of games and players.
- Distribution of pieces for each game.
- Scattering of spectra



Reminder: each player of the same game must receive, may be in different times, the same parts in the same positions and coordinates.

The communication between the server and the clients is **event** and **bi-directional**, you will use [socket.io](#) for its implementation.

No persistence system is needed.

The client implements a [Single Page Application](#) type architecture.

V.2.1 Game Management

Each player connects to a game via a [hash-based](#) url type:

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

room: name of the game to join

player__name: name of the player

The first to join a game, will be the person in charge, will have control of the game, he can launch it as he pleases. In the end, he will be the only one to be able to restart it. At the moment of starting, one of the remaining players will take this role.

A player can not join a game during the game. He must wait until the end, he can then join and participate when the leader decides to launch it.

A game is over when there is only one player left, it's then the winner.

A game can be played with one player.

Several games can be organized simultaneously..

V.2.2 Server Construction

In the game, the server is in charge of the `game management`, of the `coin distribution` and `scattering the pieces` players' fields. We invite you to very precisely identify from the beginning the sharing of responsibilities between the clients and the server and specify the network protocol of the game.

Technically, the server is an [asynchronous loop](#) in charge of processing events issued by customers. `Socket.io` allows you to receive and send events to one or more players simultaneously for the same game.

It offers an HTTP service (in addition to `socket.io`) whose only purpose is to provide, at the launch of the connection from the client, the files `index.html` and `bundle.js`, and some additional resources.

V.2.3 Client Construction

The client runs within a browser in a **Single Page Application** type architecture :

- At the first request, the browser retrieves a file from the server `index.html` that references a "`<script />`" tag to a file `Javascript` (`bundle.js`) which contains the entire code of the client application.
- The browser runs `bundle.js` and then there are no more exchanges of HTML files between server and client, the latter is totally standalone for graphical rendering and for application logic management. Only data will be exchanged with the server, bi-directional exchanges in our case are done via `socket.io`.

We ask you to use the following two libraries for the customer build :

- **React**: This is the V of acronym **MVC** which will allow you to build the GUI
- **Redux**: Library which will help you manage the states of your application. It offers a functional response to a reactive problem that allows to organize the diffusion of the events between the views and the model.

In addition to these two libraries you can totally call upon the ecosystem of **modules** Javascript:

- **Functional** : `lodash`, `ramda` are very popular, but not essential, solutions, ES6 offers as standard several set operators (`map`, `reduce`)
- **Immutability**: One of the principles of `redux` is the immutability of its model. Like the functional, `Immutable.js` it offers immutable structures, but ES6 also offers some **syntactic sugars** to avoid edge effects.
- **Dyssynchrony** : The dissemination of events in `Redux` is exclusively synchronous, or your client / server protocol is strongly asynchronous, so you will need some help on the side of a **middleware** like `redux-thunk` or `redux-promise`

V.2.4 Boilerplate

Configuring a **Full Stack Javascript** application is still laborious because the technologies used are still immature (and change every 6 months).

We propose you a starter kit which will help you avoid spending many hours setting up the configuration of base and allowing you to:

- Run the server
- Set up a bundle of JS files for the browser
- Run unit and coverage tests.

The **boilerplate** is available via the github repository [red_tetris_boilerplate](#)
The documentation is present in the [Readme](#).

V.2.5 Tests

The object of the tests is:

- Increase the reliability of the delivered versions
- To reduce the "time to market" by facilitating / automating the recipe for each new version
- Reinforce customer satisfaction and the relevance of the delivered versions by allowing longer development cycles.

Javascript and its ecosystem are now fully mature to be at the heart of a company's strategy as well as **.Net** or **Java** in the past, we are talking about d'**Enterprise Javascript**. One of the key elements of the solution will be the definition of a textttpipeline of tests for rejecting within an automatic workflow a faulty software version.

We propose you to familiarize yourself with the unit tests (the first ones encountered in the pipeline) and impose as constraints that these tests must cover 70 of the lines of code.

More precisely, when running the tests you will get 4 metrics :

- **Statements** : statement coverage rate
- **Functions** : functions coverage rate
- **Lines** : coverage rate of lines of code
- **Branches** : coverage rate of code execution paths

Your goal is to cover at least 70 of the statements, functions, lines and at least 50 of the branches.

Boilerplate includes a chain of measure of coverage and unit tests with 3 examples of unit tests (see [documentation](#))

Chapter VI

Bonus part

As Red_tetris is, basically, a video game, the possibility of adding bonuses is great. We offer, without limitation, the following:

- Add a scoring system during the game.
- Have a persistence of these scores for each player.
- Have several game modes (invisible parts, increased gravity, etc ...)

Also, several technical passages have been imposed on you, including the use of couple `React / Redux`.

The objective was:

- To make you discover this binome that is very used and sought after in the professional environment
- To facilitate the project as well as its correction

An alternative solution would have been to use a library [FRP](#) (Functional Reactive Programming). Exciting paradigm to discover and perfectly adapted to the context. We invite you to discover [flyd](#) which is a minimalist library, but with a very interesting API

Chapter VII

Rendering and peer-evaluation

Push your code to the `GiT` repository as usual. Only the work present on your deposit will be assessed in defense.

The game must be fully operational.