



Red Tetris

Tetris Network with Red Pelicans Sauce

Summary: The objective of this project is to develop a networked multiplayer Tetris game using a software stack that is exclusively Full Stack JavaScript.

Version: 5.2

Contents

I	Foreword	2
II	Introduction	3
III	Objectives	4
IV	General Instructions	5
V	Mandatory part	6
V.1	Tetris: The Game	6
V.1.1	Piece Movement	7
V.2	Tetris: Technical Details	7
V.2.1	Game Management	8
V.2.2	Server Setup	8
V.2.3	Client Setup	9
V.2.4	Boilerplate	10
V.2.5	Testing	10
VI	Bonus part	11
VII	Submission and Peer-Evaluation	12

Chapter I

Foreword

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.

Chapter II

Introduction

Everyone knows the game Tetris, and everyone knows JavaScript. All that's left is to build a Tetris game in JavaScript.

Yes, but...

Your Tetris game will be online and multiplayer. It will allow you to disrupt intergalactic gaming sessions during your late-night coding marathons. (There are still some WiFi issues on certain planets.)

Your Tetris will leverage cutting-edge JavaScript technologies, which are at the core of a fierce intellectual, industrial, and financial competition between Facebook and Google in their quest for global dominance.

Designing your Tetris will demand brainpower: you'll need to architect the application, define an asynchronous network protocol, implement it using functional programming principles, and animate and render everything using HTML!

Good luck, happy coding... and don't forget to test—and test again!

Chapter III

Objectives

The pedagogical goals are multifaceted. The main aim is to introduce the JavaScript language, explore its rich ecosystem, and implement several principles, techniques, and flagship tools of Full Stack JavaScript.

Everyone claims to know JavaScript, but very few truly grasp this multi-faceted language—which is partially functional, entirely prototype-based, dynamically typed, passionately asynchronous, and surprisingly efficient.

By building a networked Tetris game, you will:

- Apply functional programming principles (a requirement).
- Develop asynchronous client and server logic (by the nature of JavaScript).
- Implement reactive patterns (by the nature of the game and UI).

You will also write unit tests that meet industrial-level standards for continuous delivery pipelines.

Chapter IV

General Instructions

The project must be developed entirely in JavaScript, using the latest available version.



TypeScript is allowed, as it is a strict superset of JavaScript and compiles down to standard JavaScript.

Client-side code (browser) must be written without using the `this` keyword, to encourage the use of functional constructs over object-oriented ones. You may choose a functional library such as `lodash` or `ramda`, or none at all.

The game logic handling the board and pieces must be implemented using pure functions. The only exception to this rule: `this` may be used to define custom subclasses of `Error`.

Conversely, the server-side code must follow an object-oriented approach using prototypes. At minimum, you should define classes for `Player`, `Piece`, and `Game`.

The client application must be built using a modern JavaScript framework. HTML must not use `<TABLE />` elements. Layouts must use grid or flexbox.

The following are prohibited:

- DOM manipulation libraries (e.g., `jQuery`).
- Canvas.
- SVG (Scalable Vector Graphics).

There is no need to directly manipulate the DOM.

Unit tests must cover at least 70% of statements, functions, and lines, and at least 50% of branches.

Chapter V

Mandatory part

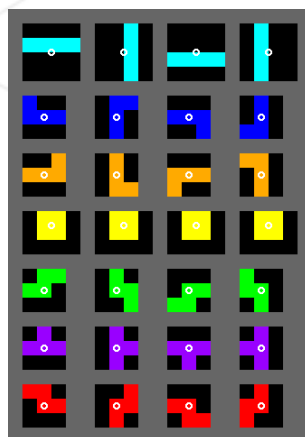
V.1 Tetris: The Game

Tetris is a puzzle game (see Wikipedia) where the goal is to fit falling pieces into a playing field for as long as possible. The game ends when a new piece can no longer enter the field. Completing a line causes it to disappear, extending the game.

Your version will maintain this core gameplay but will support multiple players. Each player has their own playing field and receives the same sequence of pieces. When a player clears lines, opponents receive $n - 1$ indestructible penalty lines at the bottom of their fields.

Each field is 10 columns wide and 20 rows tall. Players can see their opponents' names and a "spectrum" view of their fields. The spectrum shows the height of each column's highest block. This view updates in real-time.

The game will use the original Tetrimino shapes and rotation rules:



There is no scoring system; the last remaining player is the winner. The game supports both solo and multiplayer modes.

V.1.1 Piece Movement

Pieces fall at a constant speed. Once a piece touches the pile (existing pieces), it becomes immobile only on the next frame—allowing last-moment adjustments.

Available player inputs:

Left/Right arrows: Move piece horizontally.

Up arrow: Rotate piece.

Down arrow: Soft drop.

Spacebar: Hard drop to fill a gap.

V.2 Tetris: Technical Details

The game architecture follows a `client/server` model. Clients run in modern browsers, while the server runs on `Node.js`. Communication uses `HTTP` and `socket.io` for bi-directional events.

The server handles:

- Game and player management
- Piece distribution
- Spectrum updates



Each player in the same game must receive the same pieces in the same positions and coordinates—even if at different times.

No data persistence is necessary.

You are encouraged to use functional components (e.g., React [Hooks](#)).

The client must be a [Single Page Application](#).

V.2.1 Game Management

Players join games via a URL like:

- `http://<server_name_or_ip>:<port>/<room>/<player_name>`.

room: Name of the game to join.

player_name: Player's name.



Use appropriate routers such as `BrowserRouter` or `MemoryRouter` for URL handling.

The first player to join becomes the host and controls when to start or restart the game. If the current host leave the game, one of the remaining players will take this role. Once started, no new players can join until the next round.

Games end when one player remains.

A game can be played with one player.

Multiple concurrent games are supported.

V.2.2 Server Setup

The server manages the game logic and communication. Identify and define clearly the responsibilities shared between clients and the server, and specify the network protocol.

The server runs an asynchronous loop handling events via `socket.io`. It must also serve `index.html`, `bundle.js`, and any static assets via HTTP.

V.2.3 Client Setup

The client is a browser-based SPA (Single Page Application):

- The browser loads `index.html`, which includes a reference to `bundle.js` containing the full application.
- No further HTML files are exchanged. All rendering and logic is managed client-side.
- Communication with the server is via `socket.io`.

We recommend using:

- A JS framework like [React](#) or [Vue](#) for the view layer (MVC)
- [Redux](#) to manage application state.

You can enhance your app using packages from [npm](#):

- **Functional:** [lodash](#), [ramda](#) (optional)
- **Immutability:** [Immutable.js](#), or use ES features (e.g., object spread)
- **Asynchronous:** Redux is synchronous by default; use [redux-thunk](#) or [redux-promise](#) for async workflows



Avoid using outdated libraries. If necessary, find modern alternatives.

V.2.4 Boilerplate

To help you avoid tedious setup, we provide a boilerplate to:

- Run the server.
- Build JS bundles for the browser.
- Run unit and coverage tests.

GitHub repo: [red_tetris_boilerplate](#)

Documentation: [README](#)

V.2.5 Testing

Testing helps:

- Improve release reliability.
- Accelerate delivery via automation.
- Ensure product quality and long-term maintainability.

JavaScript is now enterprise-ready. Like **.NET** or **Java** in the past, it's the foundation of "Enterprise JavaScript". Testing pipelines are a core part of that, ensuring faulty versions are caught automatically.

Unit tests must cover at least 70% of statements, functions, and lines, and 50% of branches.

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

[Boilerplate](#) includes a test pipeline and sample tests (see [documentation](#)).



For obvious security reasons, do NOT store credentials, API keys, or env variables in the repository. Use a .env file and ensure it's gitignored. Exposing secrets will result in project failure.

Chapter VI

Bonus part

Red_Tetris is a video game, so there are plenty of opportunities to add extras.

Here are a few suggestions:

- Add a scoring system.
- Persist player scores.
- Introduce new game modes (e.g., invisible pieces, increased gravity).

By default, this project proposes using React. However, if you wish to explore more advanced concepts, you are encouraged to look into [Functional Reactive Programming](#) (FRP), an exciting paradigm particularly well-suited to this type of project. Check out [flyd](#), a minimalist library with a compelling API.



Bonus features will only be considered if the mandatory requirements are COMPLETED AND FUNCTIONAL. Otherwise, they will not be evaluated.

Chapter VII

Submission and Peer-Evaluation

Submit your project in your `Git` repository as usual. Only the contents of your repository will be evaluated. Double-check your folders and filenames to ensure everything is correctly named.

The game must be fully functional.