

Setup Guide and Documentation of Catch Game Web Application

Project Engineer and Developer:

Name: **Md Mahia Maksud**

Email: **mahianmaksud@gmail.com**

Phone Number: **(+852) 97930025**

Project duration: **29th April 2024 – 7th May 2024**

Table of Contents

1. Setup Guide
2. Game Navigation
 - 2.1 Leaderboard Page
 - 2.2 Play Page
3. Documentation
 - 3.1 The Backend System
 - 3.1.1 Backend API Endpoints
 - 3.1.2 Backend Environment Variables
 - 3.2 The Frontend System and API Endpoints
4. Known Errors and Potential Improvements
 - 4.1 Known Errors
 - 4.2 Potential Improvements
5. Closing Thoughts

1. Setup Guide

The main tech stacks involved in this project are:

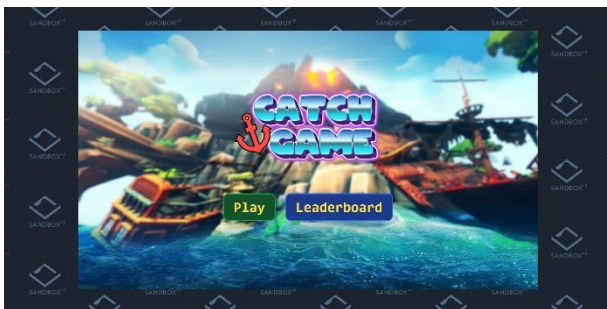
- Node.js with Express framework – for the backend
- PostgreSQL – for the database to persist data
- React.js template in Vite.js – for the frontend
- TypeScript – for code maintainability

Node and PostgreSQL are required to be installed on your computer. This project's development environment uses Node v20.12.2 and PostgreSQL version 16.2 with the operating system being Windows 10.

To get started, create a database in PostgreSQL named “social” on port 5432 as the project utilizes these specific database name and port. This can be changed in the environment variable file in the backend directory if needed, which will be discussed in later sections.

Afterwards, go into the environment variable file in the location “backend/.env” and put down the username next to “DB_USERNAME=” and password next to “DB_PASSWORD=” directly without any quotations for your database. The “APP_PORT=”, “CLIENT_PORT=”, “DB_NAME=”, and “DB_PORT=” are prefilled for getting started quickly but they can be changed to your preference and will be further discussed in the later sections. There is also an environment variable file for the front end in “frontend/.env” but it has already been prefilled and will be further discussed along with the backend environment variables later.

Afterwards, use a command line terminal to go into the backend directory and type “npm install” to install relevant dependencies and then type “npm run dev” to start the backend server. Then go into the frontend directory and type “npm install” to install relevant dependencies and then type “npm run dev” to start the frontend client. Assuming the prefilled environment variables haven't been changed, you should be able to go to your browser and go to the URL <http://localhost:5173/>. You have not entered the Catch Game application and should see the landing page below depending on your screen size. Below are two example screen sizes which will be used for demo:

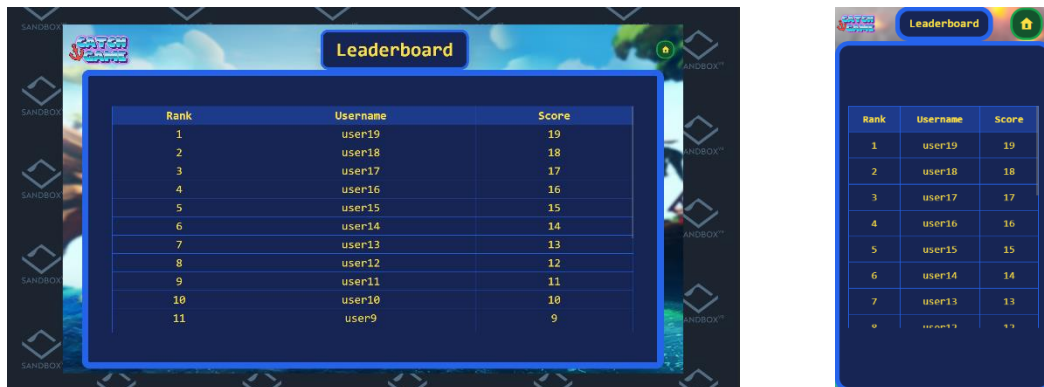


left: 2269px by 1192px, right: 375px by 667px

2. Game Navigation

2.1 Leaderboard Page

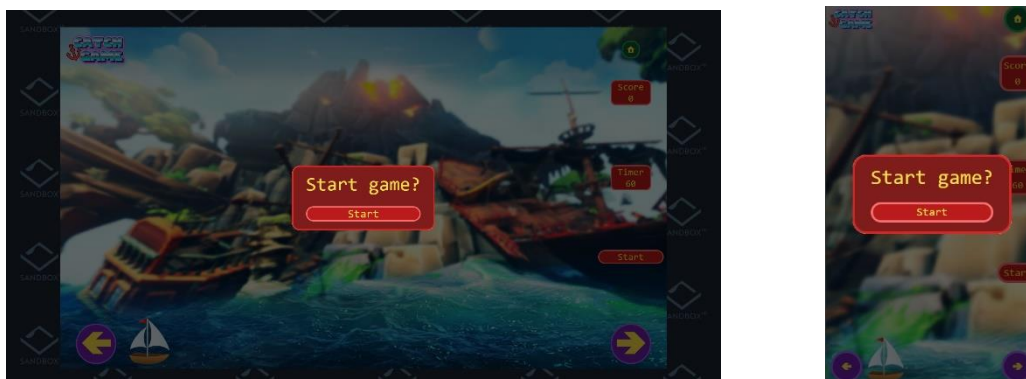
In the landing page, clicking the “Leaderboard” button will navigate to the ranking page, which is the Leaderboard application where the top 100 scores are displayed in real time.



The scores seen in the example image are just dummy data for demonstration purposes. Removing the dummy data and database initialization will be further discussed in later sections. The top right green and yellow home icon can be used to go back to the landing page.

2.2 Play Page

Clicking the “Play” button on the landing page will navigate to the catch game application and initially will have a popup for you to get ready before starting the game.

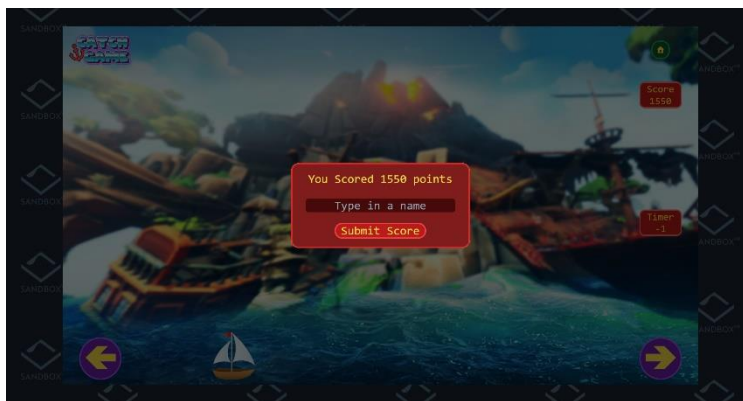


When you are ready, click “Start” and the game will begin.

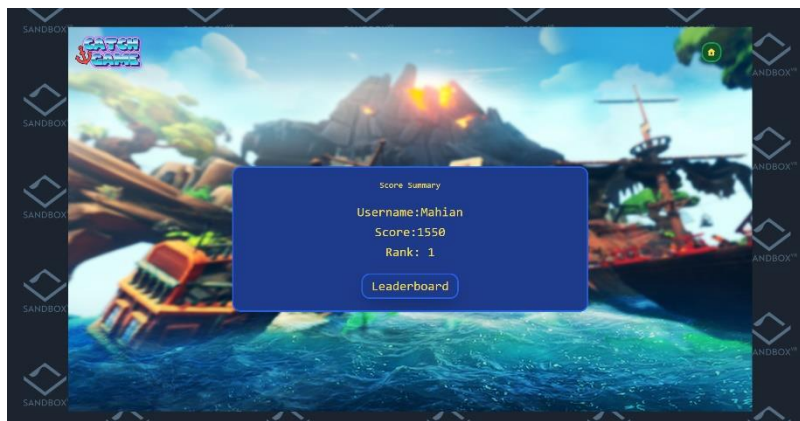


During the game play, you have 60 seconds to catch the falling items. You can click on the arrow buttons on the screen to move the boat left and right. You can also use arrow keys on your keyboard if your device takes in keyboard inputs and is connected to a keyboard. Some of the falling items will give you 50 points while others will lose you 100 points as mentioned in the “Take home questions” outline. The green and yellow home icon on the top right will stop the game and take you back to the landing page without any prompts.

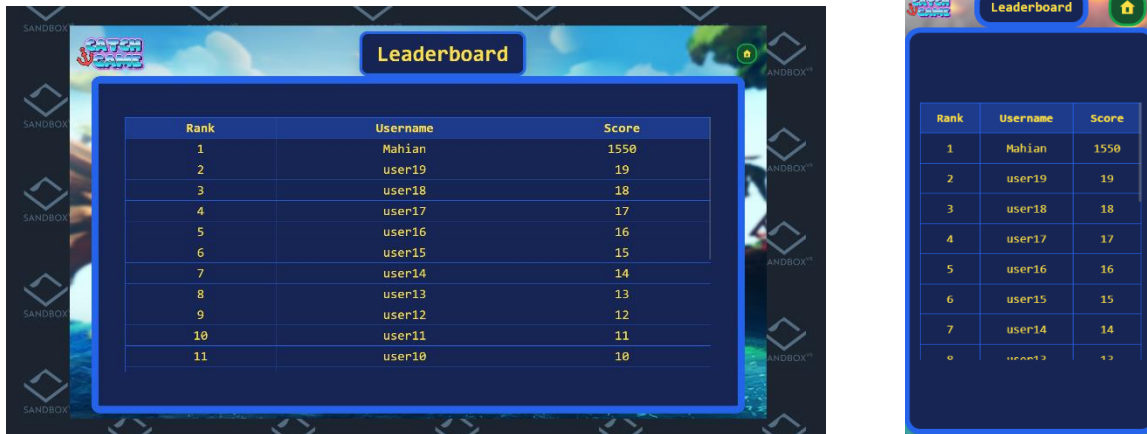
Once the time runs out, the game will stop, and a popup will show prompting you to type in a name for you to store your score in the database and join the rankings.



After you type in your name and click “Submit Score”, your score will be stored in the database with your name, and you will be navigated to the summary page where you can see what rank the score you just got is in the leaderboards. I used my name for demo.



Afterwards, you click the top right green and yellow home icon to go back to the landing page or you can click the “Leaderboard” button under your rank to go to the ranking page for the most updated leaderboard.



Rank	Username	Score
1	Mahian	1550
2	user19	19
3	user18	18
4	user17	17
5	user16	16
6	user15	15
7	user14	14
8	user13	13
9	user12	12
10	user11	11
11	user10	10

My score appeared on the leaderboard and is ranked first as the it is the highest score in the database. That is how you can navigate the application and play the game.

3. Documentation

3.1 The Backend System

The backend is built using Node.js and PostgreSQL database. The Express web application framework for Node.js is used to simplify the development process with its robust set of features for web and mobile applications.

TypeScript is used instead of JavaScript to improve maintainability and bug detection.

To connect Node.js to PostgreSQL, the TypeScript query builder, Kysely, was used due to it being light weight and more performant than Object Relational Mappers (ORMs) while still being more developer friendly than raw SQL. The configurations for Kysely can be seen in the types.ts file in “backend/src/types.ts” for table manipulation and the database.ts file in “backend/src/database.ts” for connecting to the PostgreSQL database.

The Nodemon utility is also used to automate some of the development process, such as auto restart of the server when changes are made to the code.

index.ts()

The index.ts file at “backend/index.ts” is the entry point of the application where the database and application are initialized. The application first run the main() function where the database is initialized before server start. The main function implementation is as such:

```
async function main() {
  try {
    //removes any table in the database
    //can comment out if data needs to be persisted after shutting down the server
    await dropAllTable();

    //create the score table if it doesn't exist
    await createTable();

    //adds dummy data to the database
    //can comment out if no dummy data needed
    await dummyScore();
  } catch (error) {
    console.error("Error:", error);
  }
}
```

All of these functions are ran by default but the dropAllTable() and dummyScore() functions can be commented out depending on the needs as mentioned in the comments above each function. The function implementations are in “backend/initialized.ts”.

After the main function, the server and socket connection is started. Socket.IO is used for the API endpoints as real time bi-directional communication is required for this application. The logs for server startup and socket connection/disconnection are temporarily logged to the console.

```
const PORT = process.env.APP_PORT || 3000;

main().then(() => {

  //start the server (on port 3000 by default)
  const server = app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
  });

  //connect the socket to communicate and exchange data between server and client
  const io = require("socket.io")(server, {
    cors: {
      origin: `http://localhost:${clientPort}`,
    },
  });

  //backend socket API definitions
  io.on("connection", async (socket) => {
    console.log("connected socket id ", socket.id);
    socket.emit("rankingSocket", await top100Ranking());
    socket.on("addingScore", async (scoreData: NewScore) => {
      const savedData = await saveScore(scoreData);
      socket.emit("savedScore", savedData);
      io.emit("rankingSocket", await top100Ranking());
    });
    socket.on("getRank", async (score_id: number) => {
      const rank = await getRank(score_id);
      socket.emit("rankCallBack", rank);
    });
    socket.on("disconnect", () => {
      console.log("A client disconnected");
    });
  });
});
```

3.1.1 Backend API Endpoints

```
//backend socket API definitions
io.on("connection", async (socket) => {
  //logs a new connection to the console
  console.log("connected socket id ", socket.id);

  //sends top 100 score data to the client
  socket.emit("rankingSocket", await top100Ranking());

  //recieves a new score data from the client and then
  //sends back the new saved score object back to the sender client
  //the top 100 score data also gets resent to all other clients
  //who are currently connected to get the most updated top 100 score
  //in real time
  socket.on("addingScore", async (scoreData: NewScore) => {
    const savedData = await saveScore(scoreData);
    socket.emit("savedScore", savedData);
    io.emit("rankingSocket", await top100Ranking());
  });

  //recieve the score id from the client and send back
  //the rank of the particular score
  socket.on("getRank", async (score_id: number) => {
    const rank = await getRank(score_id);
    socket.emit("rankCallBack", rank);
  });

  //logs a message to the console if a client disconnected
  socket.on("disconnect", () => {
    console.log("A client disconnected");
  });
});
```

The API endpoints for the backend are in the index.ts file in “backend/index.ts”. The explanation of the backend endpoints are commented above each endpoint as seen in the screenshot above and the source code. There is an endpoint that sends the top 100 scores to the client. There is an endpoint that receives a new score entry from the client and resends the saved score back to the relevant client and resending the top 100 scores to all clients that are connected to make sure the leaderboard always displays the top 100 scores in real time. And there is an endpoint that receives a score ID from the client and sends back the current rank of the specific score back to the client.

There are functional methods throughout the endpoints and their implementations are in “backend/src/controller/ScoreController.ts”. The purpose of the controller was to have middleware between the entry point and database for validation purposes but due to time limitations, the validations were not implemented, which will be further discussed later.

The controller functions make function calls to the repository, which contains functions to perform CRUD operations on the database. The implementations of the repository functions can be seen in “backend/src/repository/ScoreRepo.ts”. In ScoreRepo.ts, the functionality and the SQL statements being built by the Kysely query builder can be seen above each method.

As mentioned previously, the validation for incoming data was not implemented due to limited time. However, potential validation methods have been written for potential future expansion and their demo implementations can be seen in “backend/src/service/ScoreService.ts”.

3.1.2 Backend Environment Variables

A few environment variables have been defined in the .env file at “backend/.env” as such:

- APP_PORT – the port number of where the backend application will be run
- CLIENT_PORT – the port number of where the client will be running
- DB_USERNAME – username for your database
- DB_PASSWORD – password for you database
- DB_NAME – name of the database
- DB_PORT – port number of where the database is in

Other than the username and password, the other definitions have been prefilled as the project was built around those definitions. It is recommended to change the prefilled definitions for quickly starting the application. However, if a change to the prefilled variables is still needed, then some notes need to be accounted for as such:

- APP_PORT – the client-side environment variable have to reflect this change
- CLIENT_PORT – the port number of the client and browser URL need to reflect this
- DB_NAME – the name of your database have to reflect this
- DB_PORT – the port number your database is running on have to reflect this

3.2 The Frontend System and API Endpoints

The frontend is built using the React.js template of Vite.js to easily set up the development environment.

TypeScript is also used instead of JavaScript to improve maintainability and bug detection.

For the visual aspect of the frontend development, a combination of raw CSS and Tailwind is used. Tailwind is used for a more developer friendly approach to implement CSS. However, there are some limitations for Tailwind especially when it comes to dynamically setting CSS properties, for which raw CSS is used.

The entry point of the application is the main.tsx file in “frontend/src/main.tsx”. A router is implemented to handle the routing of the application for which the implementation can be seen in routes.tsx file at “frontend/src/router/routes.tsx”.

```
export const router = createBrowserRouter([
  {
    path: "/",
    element: (
      <ScoreProvider>
        <App />
      </ScoreProvider>
    ),
    errorElement: <NotFoundPage />,
    children: [
      { path: "/play", element: <Play /> },
      { path: "/ranking", element: <Ranking /> },
      { path: "/summary", element: <Summary /> },
    ],
  },
]);
```

The project is organized in a way where there is a parent component <App/>, three child components, <Play/>, <Ranking/>, and <Summary/>, and a page to handle routing errors <NotFoundPage/>. There is also a context implemented which is provided by <ScoreProvider> for temporarily storing the user’s score to see the rank of the specific score.

<App/> Component

The implementation of this component is in the App.tsx file at “frontend/src/App.tsx”. This component is the landing page at the path “/”. It looks at the current URL path and then displays either the elements of the <App/> component or the child components. It includes the background and game title elements, a “Play” button to render the <Play/> component for the game application, and the “Leaderboard” button to render the <Ranking/> component for the leaderboard application.

<Play/> Component

The implementation of this component is in the Play.tsx file at “frontend/src/pages/Play.tsx”. This is the component that renders the game application. Other than the game title image and a link to the <App/> component, it contains the <GameArea/> component which includes all the game logic and renders. Game logic and renderings in the file include starting the game, ending the game, moving the boat, generating falling items, collision detection, displaying game start/end modals, and more. The implementation of the <GameArea/> component is in the GameArea.tsx file at “frontend/src/component/play/GameArea.tsx”.

After the game ends, a modal containing API endpoints will pop up and prompt the user to submit their username, which sends the data to the server and the server will return the saved score data, which then is stored in a context to temporarily store the score data and

then the user is navigated to `<Summary/>` component for a summary of their game performance.

The score context is implemented in the file `ScoreContext.tsx` at `“frontend/src/context/ScoreContext.tsx”`, which have been further converted to a custom hook in `“frontend/src/context/useScore.tsx”`. This allows the score data to be temporarily accessible on other pages.

Majority of the logic is stored in one file as a lot of states are dependent on each other and due to time limitations. If time more time was available, then the game logic would be organized in a better way for readability and maintainability.

`<Ranking/> Component`

The implementation of this component is in the `Ranking.tsx` file at `“frontend/src/pages/Ranking.tsx”`. Other than the game title image and a link to the `<App/>` component, this component is mainly used to store the `<Leaderboard/>` component at `“frontend/src/component/leaderboard/Leaderboard.tsx”`, which is the leaderboard application. The `<Leaderboard/>` component contains a API socket endpoint that retrieves the top 100 highest ranked scores from the database and displays the rank, username, and score in a table in real time.

`<Summary/> Component`

The implementation of this component is in the `Summary.tsx` file at `“frontend/src/pages/Summary.tsx”`. This component contains endpoints that send the score data stored in context, if there is any stored, and receives the rank of the score. This component is navigated to after the user submits their score and it displays a summary of the user’s score data, which includes the username, score, and score rank. The user can navigate to the leaderboard page afterwards.

`<NotFoundPage/> Component`

This component is a simple fall back for when there are errors with navigations and routing. It’s a simple page the displays `“Page Not Found”` and provides links back to the landing page (`“/”`).

Environment Variable

There is just one environment variable `“VITE_SERVER_PORT”` in the `.env` file at `“frontend/.env”` which is used to get the port of the server. If the port number of the server is changed, then it should be reflected here.

4. Known Errors and Potential Improvements

4.1 Known Errors

Due to time limitations, some errors and vulnerabilities had to be overlooked.

Lack of Validation

Firstly, input validations were not implemented on both the frontend and backend so the current state of the project assumes that only correct inputs will be passed through the API socket endpoints. This could lead to data transfer errors and even possible SQL injections but if given more time, the validations could be implemented with some examples already prototyped at “backend/src/service/ScoreService.tsx”. A temporary fix for the purpose of demonstrating the project would be to avoid putting complex usernames and only input short and basic usernames with letters and numbers when storing score data.

<Leaderboard/> Component CSS Issue

Some of the current CSS property in the <Leaderboard/> component at “frontend/src/component/leaderboard/Leaderboard.tsx” the height property of the <div/> element containing the <table/> element has been hardcoded as setting properties that are more responsive seem to have unexpected behaviors. This results in the top section of the leaderboard page to be unreachable on screens that are too short but wide as those elements get pushed out of the screen. A temporary fix to leave the page it just directly using the browser URL and going back to the landing page.

Modifying the Screen in the Middle of Game Play

Changing the screen size or changing windows or tabs in the middle of game play can sometimes cause the game elements, such as the boat and falling items, to be located at unexpected places. A temporary fix for this is to just refresh the page and the new dimensions of elements will be registered. To properly fix this, an event listener would need to listen for window changes, which can be implemented if more time was available.

No Form of Rate Limiting Set in the Backend

As there are no forms of rate limiting for the socket endpoints, a malicious user could easily overload the system by making a lot of API calls. A solution to this is to implement a rate limiter, such as Express Rate Limiter for this project, and limit the number of connections from any one client or IP address at a certain time.

4.2 Potential Improvements

There are other improvements that could be made along with the ones mentioned in section 4.1.

Hitbox Dimensions

The hit box of the boat and the falling items during the game are just rectangles based on the dimensions of the files. If time permitted, a more precise shape for the elements could be calculated for a more realistic collision detection.

Storing User Data

Currently, the only method of user data storage is the score entry with the username. However, that entry won't be accessible to the user once the user leaves the summary page as it is just temporarily stored in the React context and will be removed as the only other way to re-access the summary page is through the browser URL and that would remove the score context. To fix this, the context could instead be stored in the user's device, whether it be through local storage, cookies, etc. Another method would be to have a login system for users. This would unlock a lot more customizable form of data storage for specific users but needs time to implement.

Better UI Design and Fair Gameplay

The current UI design could be improved by a lot of given more time. More animations could be added to buttons, game elements, and page transitions. Better colors could be used for a more seamless design. And better CSS properties could be used to display the desired interface more efficiently and fair gameplay as thinner screen has less distance to cover when traversing the game field. These improvements would increase the user experience and responsiveness of the application but a lot and can be implemented with more time available.

5. Closing Thoughts

This project has been a great experience and an adventure to see it come to life as the development continued. Although improvements could be made, the implementation of such improvements can easily be achieved with more time.

I would like to thank you for this opportunity to test my skills against your challenge and would love to discuss more about my project implementation and your insights. I can easily be reached through email (mahianmaksud@gmail.com) or by phone ((+852) 97930025) at your convenience.

- Md Mahian Maksud