RoboTanks Design Document

Mar 12, 2016 - Version 2.0

Table of Contents

Introduction

Purpose

The purpose of this document is to outline the technical design of the RoboTanks game and provide an overview for the implementation.

Document Organization

This document is organized into the following sections:

Introduction	Provides information relating to this document (purpose, organization, audience, etc.)
Components Overview	Gives an overview of the main components which form the system
Data Design	Describes the underlying data structures for RoboTanks and how they are used and accessed in various parts of the project.
Architecture Design	Gives details of the system architecture and technologies used

References throughout this document are made to specifications from the requirements document using parentheses. E.g. (5.1.2) or (7.3, 7.4)

Components Overview

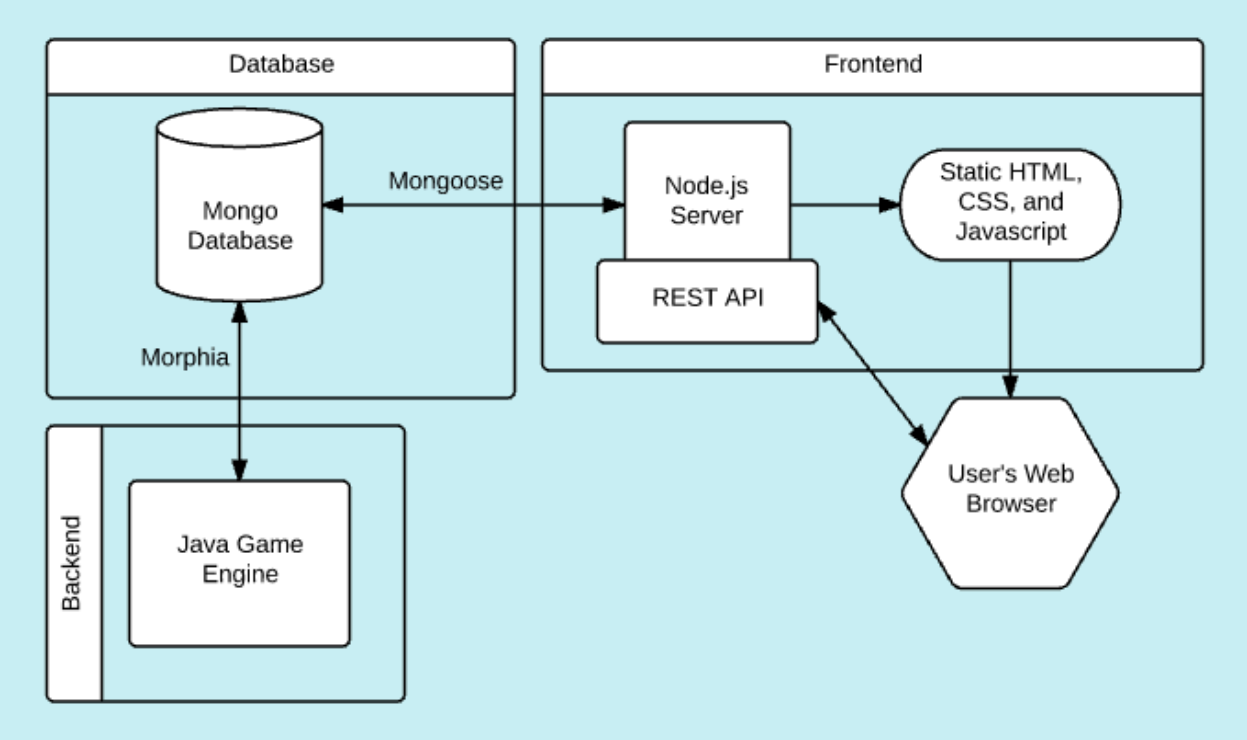
Website

The purpose of the RoboTanks project is to create an online system where users can code tanks and watch them fight in an arena.

The only access point for users will be a website hosted using Amazon Web Services. The site domain will be *robotanks.tk*. There will be a landing page with general information (5) and authentication will be required (1) for further access. After authenticating, pages will allow the user to create and manage tanks (2), create, join and watch games (3), view statistics for their tanks (6) and look up useful reference material for how to code tanks (5).

More details for the pages on the website are given in the Architecture Design section.

The system consists of three major components detailed below: the frontend, backend and database.



Frontend

The frontend portion of the system will perform the following functions:

- Handling registering of new users (1.1)
- Providing authentication tokens and recognition so users can log in (1.2)
- Allow users to create tanks and register them to the database (2.1)
- Provide an interface where users may upload code they've created locally (2.1.3, 2.2.3)
- Allow users to edit (2.2), delete (2.3) and manage (2.4) their tanks, including persisting any relating user actions to the database
- Allow users to create games (3.1) or join existing ones (3.2)

- Provide an interface where users can watch any of their games (3.3)
- Display game mechanics (5.2) and reference material for use when coding a tank (5.1)

The frontend is split into two components, the user interface and server.

The user interface, written using the React javascript framework consists of the html, css and javascript files necessary to create the web pages.

The server is implemented using node.js and will handle serving the static files as well as providing the necessary endpoints to perform the functions of the frontend. The functionality of endpoints will generally consist of handling REST API requests, performing necessary queries on the database and returning a result through the HTTP response.

Exact specifications for html, css, javascript and node.js endpoints are being left up to individual developers as they see fit to implement functionality.

Backend

The backend will perform the following functions:

- Compile tanks and provide feedback to the user if a tank fails to compile (2.1.4, 2.2.4)
- Process games that are ready to be played and record tank movements throughout the game (3.3.3)
- Record any errors that take place during the game (3.3.8)
- Implement the full tank interface (4)

Database

The database will store information pertaining to users, games and tanks. It will provide the only point of communication between the frontend and backend.

Data Design

The data will be stored in a Mongo database.

Connecting to the Database

Backend Connection

The backend mainly takes java code from the user and compiles it to run the game. The results are stored in the Mongo database. Java classes from the backend are converted into Mongo Schemas using Morphia.

The backend will only write data to the game object in the database. It will not be used for managing users.

Frontend Connection

The frontend then accesses the mongo database through the Node.js server endpoints using the mongoose library. The endpoints generate the appropriate JSON files to manage and reenact the game.

The frontend will use a REST API to manage the data structures. For example, to add a user, it would make the request (replace ":username" with the actual username)

> POST to /api/users/:username

To list a user's tanks, the request would be made:

> GET to /api/users/:username/tanks

Endpoints would follow this pattern with GET returning the data, POST used to create new objects, PUT to update the objects, and DELETE to remove objects.

All endpoints will have security put on them such that only authorized users will be allowed to make requests and they will only be able to make requests that correspond to their user.

Data Structures

The database will be comprised as two main structures, Games (2, 3) and Users (1).

Games will hold necessary information for the backend to calculate the results of a game and store the moves made by each tank during the game.

```
Game {
  id:ObjectId,
  users:User [{
        userID:ObjectId,
        userName:String,
        tankName:String,
        tankID:ObjectID
  }],
  name:String,
  tanks:Tank [{
        coord:Coordinate {
              x:int,
```

```
            y:int
        },
        tankId:ObjectId,
        tankName:String,
        health:int,
        dir:TANK_DIR,
        actionPoints:int,
    }],
    winnerID:ObjectId,
    moves:MoveTracker {
        curTurn:int,
        listOfMoves:Map<Integer, TANK_MOVES> []
    },
    ready:boolean,
    status:int,
    errors:LogItem [{
        type:LOG_TYPE,
        timestamp:Date,
        user:ObjectId,
        message:String
    }]
}
```

User objects will hold information relating to the user, their tanks, and their statistics for wins and losses.

```
DBUser {
    id:ObjectId,
    username:String,
    password_hash:String,
    kills:int,
    deaths:int,
    wins:int
    tanks:DBTank [{
        id:ObjectId,
        name:String,
        code:String,
        status:TANK_STATUS,
        type:TANK_TYPE,
        skin:TANK_SKIN,
        kills:int,
        deaths:int,
        wins:int
        errors:LogItem [{
            type:LOG_TYPE,
            timestamp:Date,
            user:ObjectId,
            message:String
        }]
    }]
```

```
}
```

A list of possible values is provided for the Java end in the form of enumerations. These include possible tank directions, tank moves, tank status and log type.

```
_________
| ENUMS |
_________

TANK_DIR (4.1):
 N,E,S,W

TANK_MOVES (4.2):
 TURN_LEFT, TURN_RIGHT, MOVE_FORWARD, MOVE_BACKWARD, WAIT, SHOOT

TANK_STATUS (2.1.4, 2.2.4):
 COMPILE,ERROR,SUCCESS

LOG_TYPE (3.3.8):
 GAME_ERROR,TANK_ERROR
```

Architecture Design

Frontend

Home Page

Displayed when a user enters the website.

The home page serves three purposes:

1. Allow existing users to login.
2. Allow new users to register for a new account.
3. Introduce the game to new users.

The login and registration block is placed above the introduction block.

User Login (1.2)

Form for the user to authenticate with the server.

This is placed before the Game Introduction to allow easy access to users. By default, the form prompts the user to login. A link below the login form allows a user to register instead, if the user does not already have an existing account.

User Registration (1.1)

Form for the user to register for a new account.

This is the same block as User Login. Clicking 'Create new account' just below the login form toggles the registration view.

Registration form fields:

1. Username
2. Password
3. Confirmation of password
4. Email address

Game Introduction (5.2.1)

A description of how RoboTanks is played.

This is a brief and succinct description that is no more than 3 scroll heights long, meaning that the user will not have beyond scroll 3 full pages. The description includes screenshots of how the different components of RoboTanks work together to allow the user to play a game.

In particular, these phases are described:

1. Create a new tank
2. Join a battle
3. View personal statistics

Armory

An interface for a user to manage tanks and view their statistics.

The view is organized into two blocks, with the user's list of tanks on the left and the statistics of the selected tank on the right. The right block also doubles as the code editor for the selected tank when in edit mode.

Tank List (2.4.1)

List view of all tanks created by the user.

This component is comprised of the list of tanks and an 'Add Tank' button below the list.

Tank Card (2.2.1, 2.2.2)

A visual representation of each tank.

Each tank is visually represented in the form of a card. A card has 4 elements:

1. Tank name
2. Tank image
3. Tank description
4. Tank win/loss ratio

The tank name is placed above the tank image. These two elements are wrapped in a block that spans 3 column widths within the card. The next 6 columns are occupied by the block that wraps the tank description and win/loss ratio.

Constraints:

Tank name – No longer than 20 characters
Tank image – 64 x 64 pixels
Tank description – No longer than 200 characters

When a card is selected, a 'Edit' and 'Download' button appears to the right of the card. These buttons are wrapped in a block that occupies the final 3 columns of the tank card.

When the 'Edit' button is clicked, the editor is toggled on, the statistics toggled off and the button changed to 'Stats'. The reverse happens when the 'Stats' button is clicked.

Statistics (6.2)

Statistics describing the selected tank.

Each tank has the following statistics:

1. Wins
2. Losses
3. Draws
4. Games played
5. Tanks destroyed

Each statistic has a corresponding image and a number. Statistics are arranged vertically down the height of the block.

Editor (2.2.3, 2.2.4)

An editor that allows the user to modify a tank.

When the 'Edit' button is clicked on a selected tank card, the editor is toggled on and the statistics are toggled off. The editor displays the code of the selected tank. Any changes made in the editor is not saved until the 'Save' button is clicked.

Battle

An interface for a user to create and join battles.

The view is comprised of two parts: The top block allows a user to create a new game while the bottom allows the user to join an open game.

Create Game (3.1.1)

A form to create a new game.

The form consists of only one field: Game Name. If the user clicks 'Create' after filling out this field, a game will be created and listed under 'Open Games'.

Open Games (3.2)

A list of open games that a user can join.

Each row of the list displays the information of a specific game, namely the name and open spots left in the game. The last column consists of a 'Join' button.

Upon clicking 'Join', a modal appears prompting the user to select one of the tanks listed in the Armory. Once a tank is selected, the number of open spots is updated.

Watch

An interface for a user to watch the progression of a battle.

Games (3.3)

A list of games available to view.

Each row of the list displays the information of a specific game, namely the name and status of the game. The last column consists of a 'Watch' button.

Upon clicking 'Watch', a modal appears, allowing the user to watch the progression of the game.

Backend

Database

Where all of the game data is stored (Tanks, Games, Users)

Game (3.3)

All of the information and code that goes into making a game function. Once the game has started, tanks make a move based on who has spent the least amount of 'Action Points.' Each action costs a separate amount of 'Action Points.'

In the game class, the turn of the selected tank is evaluated. Based on their action, the area affected by their action is also evaluated, for example, to see if another tank is hit by a shot.

There are a number of classes that help the game function. They fit under these categories:

- Board : Coordinates and size of board used in calculations to determine positions and vulnerability of tanks.
- Board Elements : All of the pieces of the board
    - Walls: These are obstacles that do but prevent the tank from moving to their squares.
    - Tanks: There is a base Tank class that holds most of the basic information and getters and setters. There are three abstract classes: Heavy, Basic, and Light, which all have different stats.
    - Different Tanks have differing required action points in order to carry out their respective actions. For example, it costs the heavy tank more to move than the light tank, but the heavy tank also does more damage than the light tank.
- Users : The registered players of the game. Their losses, wins, draws, and tanks killed are stored. This also stores a list of their Tanks.
- Util : How moves are processed in the game. Possible tank moves and directions are stored in classes as enums. There is also a MoveTracker that records past moves. There is also a class that evaluates the user's string of code and turns it into functional Java.

Poller

Checks periodically to see if the amount of games have changed and if anything needs to be compiled. If that is affirmative, then it passes the game to the PreGame Processor.

PreGame Processor (7.1, 2.1.4, 2.2.4)

Before a game can be calculated and recorded, tank code must be processed. This is called pregame processing and consists of the following components:

- GameProcessor: Takes the game from the Poller and runs it through UserLookup and TankProcessor.
- UserLookup: This examines the list of tanks that have been passed into the game by the Poller and will then populate a list of users (owners of said tanks) to coincide with the list of tanks.

- TankProcessor: This runs through the four tanks and runs the code of each tank through the TankCodeLoader.
- TankCodeLoader: The TankCodeLoader class parses through the class to make sure that it contains no malicious code (7.1). If it passes those tests, then it compiles the code immediately prior to playing the game. If compiled, the completed tank is returned to the TankProcessor. Otherwise, an error is returned and the game does not compile.

Pregame Sequence Diagram