Battleship
(deutsch: «Schiffe versenken»)

Prof. Dr. Bradley Richards

http://brad-home.ch

Management Summary

Battleship is a classic game originating around World War I, or possibly even earlier. In this project, two clients communicate with a server (which already exists) in order to play. There are variations of the game rules; this project uses the basic rules explained in the official rulebook (see references).

The goal of this project is to create a client that can play the game. Two players start the client, enter their game-key, and play until the game ends. This requires not only network communication with a RESTful web service, but also local data management and a user interface. As long as the client communicates correctly with the server, there are no restrictions on how the client looks or works. For example, one student sank professors instead of ships.

The server source code is available at: https://gitlab.com/kri/battleship-server

A sample (JavaFX) client is available at: https://gitlab.com/kri/battleship-client

Table of Contents

Student Project

Implement a client

Create a client app that can play Battleship using the server http://brad-home.ch:50003. The client must support all API functions of the server listed in this document.

Individual contributions

This is a group project for 2-3 people.

Each team member must program a significant part of the project. Add your name in comments to identify the code that you have written. Pair programming is fine – the point is that everyone works on the project.

Project documentation

Title page: Give your project a name, and list the team members.

Description: Describe your project. What does it look like? How does it work?

Design: How is the code structured? How do the parts work together? Suppose that your reader has no time to study the code itself: Explain the most important ideas in your implementation. (1-2 pages of text, plus diagrams)

Usage: Explain how to use your project, if this is not obvious. Test your project – not on your computer, but on someone else's – be sure it works on a "fresh" computer.

Using code from external sources

There is nothing wrong with using code snippets from external sources or AI. However, any significant amount of code must be referenced in a comment in the code. Using external code without referencing the source is plagiarism.

Communications protocol

The messaging protocol is RESTful and uses JSON for data transfer. The first section below provides an overview of how the game works. Following sections describe the messages in more detail.

Overview

To start a game, two clients must send the same game-key, along with their initial ship positions, to the server. The game-key is simply a string, and two clients that send the same string are placed into the same game. The server will reject any other clients that attempt to connect using the same game-key. After sending a game-key (assuming no error) the client will either receive an immediate error message, or will receive (after a delay) a game update.

After both players have registered, one player will be randomly selected to start. This player receives an initial game update, and can then make their move. Then the other player will receive a game update, and be asked for their move.

The game then continues: Whenever a player receives a game update, they send their move. The reply to their move will be another game update. There will be a delay before this reply – whatever time the other player takes to make their move. This delay may be quite long, so the clients should not timeout quickly. Game updates and moves are exchanged until the game is over.

If a player sends an invalid move, they will receive an immediate error message. They can then try again to send a correct move. An invalid move is either a move that is outside the game-grid, or one that repeats a previous move.

When the game is finished, after the final update to both players, the game is deleted from the server. Any unfinished game that has not had a move made in a long time will also be deleted. The server stores no information except for the game positions of games still in progress.

To keep the implementation simple, there is no security. There is no login, there are no passwords or session tokens, data is not encrypted.

Interface specification

If the server is sent an invalid mapping, it will return a status 418. If the mapping is valid, it will return a status 200. *That does not mean that the request succeeded.* Status 200 only means that the path is valid; the result may still be an error. For example, a move may be invalid. If an error occurs in a valid mapping, the returned JSON will contain an error message like this:

```
{ "Error" : "Something went wrong" }
```

The error message will generally be informative. This specification does not mention error messages in detail, but you may assume they exist as needed.

Ping mapping

The following mapping let you test server connectivity:

- GET `/ping`: Tests the connection with the server. Returns `{"ping":true}`

Game mappings

To create or join a game:

- POST **`/game/join`**: Join a game. Requires **`player`**, **`gamekey`**, and **`ships`**. The first two parameters are strings and must be at least 3 characters in length. The third parameter specifies the position of this player's ships (details discussed below). If there is no error, the reply is the same as described below for **`/game/enemyFire`**, i.e., either null coordinates (for the first player), or with the coordinates of the enemy's fire (for the second player).
- POST **`/game/fire`**: To make a move, a player sends the position on the opponent's grid that they wish to attack. This requires **`player`**, **`gamekey`**, **`x`** and **`y`** coordinates. The reply contains the result **`true`** if a hit or **`false`** if a miss), and an array of enemy ships sunk.
  Example: `{ "player" : "brad", "gamekey" : "game1", "x" : 2, "y" : 3 }`
  Reply: `{ "hit" : true, "shipsSunk" : [ "Carrier", "Submarine"] }`
- POST **`/game/enemyFire`**: After each move with **`/game/fire`** a client must send this request with **`player`** and **`gamekey`**. The server will reply with the enemy's move, which contains **`x`** and **`y`** coordinates and the **`gameOver`**. Three important notes:
  - The reply will wait for the enemy to move. This may take a while.
  - The **`gameOver`** is boolean: if **`true`**, the game is over, no further moves will be accepted, and the game has already been deleted from the server. If **`false`**, the client can immediately send its next move using **`/game/fire`**.
  - The first move of the game, the reply to this request will contain *no* coordinates, because there was no preceding enemy move. It will simply contain `{ "gameover" : false }`

Example: `{ "player" : "brad", "gamekey" : "game1" }`

Reply: `{ "x" : 2,  "y" : 3, "gameover" : false }`

Initial ship positions

The initial ship positions sent with **`/game/join`** are an array containing one entry for each ship. Each ship is named, followed by the coordinate of it's upper-left corner (assuming 0,0 is the top-left), and then either the word "horizontal" or "vertical" to give the ship's orientation. Example:

```
"ships" : [
  { "ship" : "Carrier",    "x" : 0, "y" : 3, "orientation" : "horizontal" },
  { "ship" : "Battleship", "x" : 1, "y" : 1, "orientation" : "vertical"   },
  { "ship" : "Destroyer",  "x" : 2, "y" : 4, "orientation" : "horizontal" },
  { "ship" : "Submarine",  "x" : 3, "y" : 3, "orientation" : "vertical"   },
  { "ship" : "PatrolBoat", "x" : 5, "y" : 5, "orientation" : "horizontal" }
]
```

Sample communication

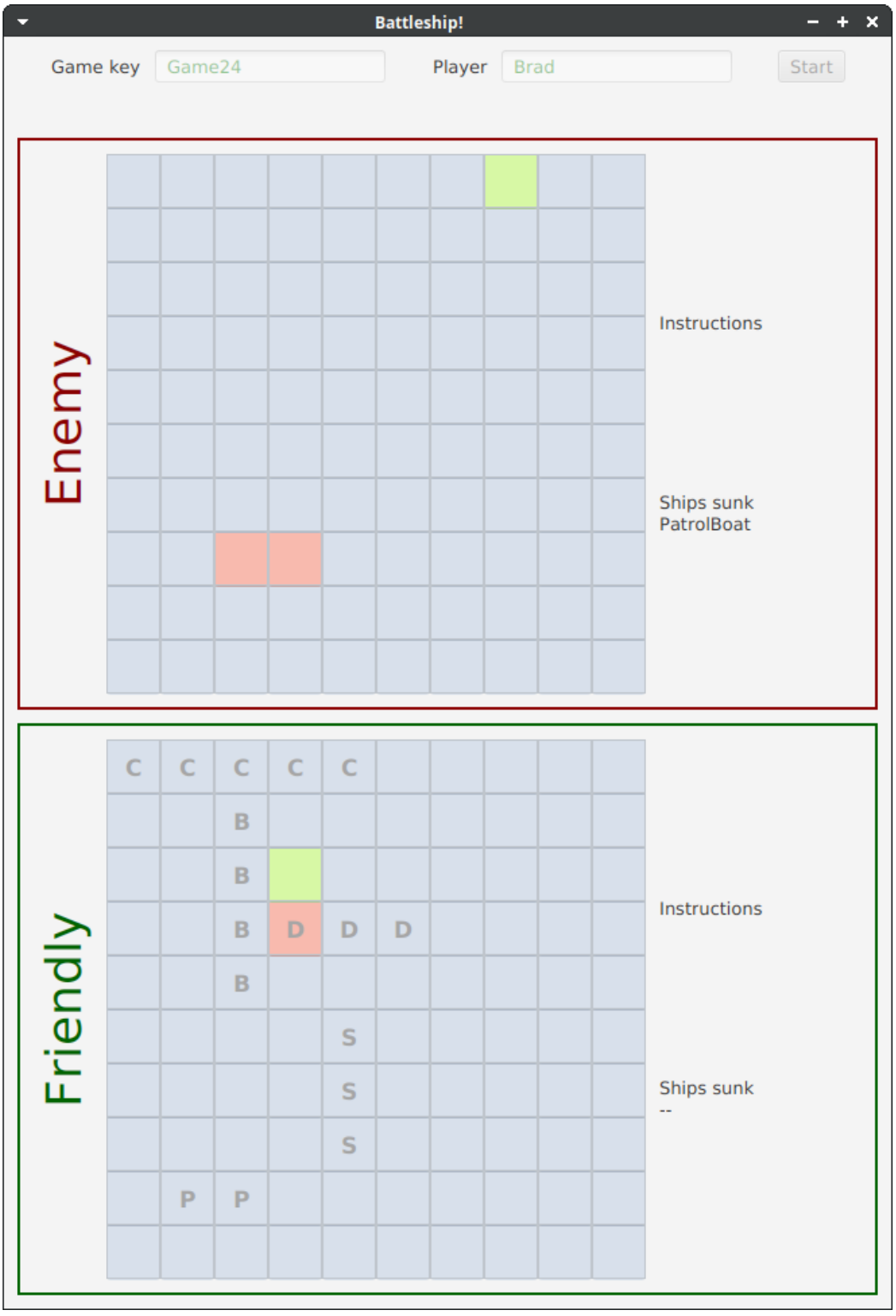
Here is a sample log of communication with the server. Comments about the messages are in *red*.

Client sends		Server replies

`{ "ships" :`
`[ {"orientation":"horizontal","x":3,"ship":"Destroyer","y":3},`
`{"orientation":"horizontal","x":0,"ship":"Carrier","y":0},`
`{"orientation":"vertical","x":2,"ship":"Battleship","y":1},`
`{"orientation":"vertical","x":4,"ship":"Submarine","y":5},`
`{"orientation":"horizontal","x":1,"ship":"PatrolBoat","y":8}`
`],`
`"gamekey" : "Game24",`
`"player" : "Brad"`
`}`
		To start, the client sends location of all ships, the game-key, and the player's name.
		Note: ship attributes are sorted alphabetically, although this is unimportant.

We are the first player to move, so there are no x and y coordinates for the opponent's shot.	`{"gameover":false}`	
`{ "x" : 7, "y" : 0, "gamekey" : "Game24" , "player":"Brad" }`		*We must always send the game-key and player name. We fire on field (7,0)*
Our shot missed. We have not sunken any ships.	`{ "hit" : false, "shipsSunk" : [] }`	
`{ "gamekey" : "Game24", "player" : "Brad" }`		*Request information on the enemy's move*
Enemy fired on field (3,7). *The game is not finished.*	`{ "x" : 3, "y" : 3, "gameover" : false}`	
`{ "x" : 2, "y" : 7, "gamekey" : "Game24", "player" : "Brad" }`		*We fire on field (2,7)*
We hit a ship! We have not sunken any ships.	`{ "hit" : true, "shipsSunk" : [] }`	
`{ "gamekey" : "Game24", "player" : "Brad" }`		*Request information on the enemy's move*
Enemy fired on field (3,2). *The game is not finished.*	`{ "x" : 3, "y" : 2, "gameover" : false }`	
`{ "x" : 3, "y" : 7, "gamekey" : "Game24", "player" : "Brad" }`		*We fire on field (3,7)*
We hit a ship! We have sunken the PatrolBoat.	`{ "hit" : true, "shipsSunk" : [ "PatrolBoat" ] }`	

After the sequence of message above, the game state (from the perspective of player “Brad”) is as shown below:



Notes about server behavior

When sending **`/game/join`**:

- **`player`** or **`gamekey`** too short → Error
- **`gamekey`** of a game that already exists → Error
- Initial ship positions incorrect → Error

Ship types are defined in the server as an enumeration

- Ship types and their sizes are: Carrier (5), Battleship (4), Destroyer (3), Submarine (3), PatrolBoat (2)
- Names of ships are sent in JSON as strings.

The game grid is always 10x10 (x and y coordinates 0-9)

When sending **`/game/fire`** the server checks the move for validity

- Coordinates have already been fired upon → Not smart, but *not* an error.
- Coordinates outside the game grid → Error
- **`player`** or **`gamekey`** incorrect → Error
- Not this player's turn → Error

When sending **`/game/enemyFire`**

- **`player`** or **`gamekey`** incorrect → Error
- The server does not send a list of *your* ships sunk – the client can determine this by itself

There are numerous other error conditions not listed here. For example, sending a move for a game that does not exist, or sending incorrect JSON.

Games will be deleted after an hour of no activity.

Please don't *try* to break the server. However, if you find a legitimate error in server behavior, please report it so that the server can be fixed.

Helpful tools and references

References

Official rules for “Battleship”: https://www.manualslib.com/manual/580174/Milton-Bradley-Battleship.html

Tools: Curl and Invoke-RestMethod

These commands provide command-line access to web resources. Under MacOS or Linux use `curl` (you may need to install the command). Under Windows[1] PowerShell use `Invoke-RestMethod`.

For this project, these commands are useful to send GET and POST commands to the server and see the replies.

GET commands

GET commands simply request a particular page (mapping) from the server. For example, to fetch a web page under MacOS or Linux, execute

```
curl javaprojects.ch
```

Under Windows, execute

```
Invoke-RestMethod javaprojects.ch
```

In both cases, you will be presented with the home page from that website.

For this project, to see the list of all registered users, we send a GET command to the mapping defined by the server:

```
curl http://127.0.0.1:8080/users
```

Under Windows

```
Invoke-RestMethod http://127.0.0.1:8080/users
```

Under Windows, this displays detailed information about the response, as well as the content of the reply (JSON-formatted text). Under MacOS/Linux, the command only displays the content of the reply, however, you can get information about the response by including the “verbose” option:

```
curl -v http://127.0.0.1:8080/users
```

Also under MacOS/Linux, you can format the JSON into a more readable format by piping it through the `json_pp` command (again, you may need to install this):

```
curl -v http://127.0.0.1:8080/users | json_pp
```

The resulting output looks like this:

```
{
   "users" : [
```

1*The `curl` command also exists in Windows, but does not work as described here.*

```
        "sunnu",
        "marble"
    ]
}
```

POST commands

More complex server commands require sending information to the server using a POST command. For example, we can register a new user, using the **`/users/register`** mapping. The following command is entered *all on one line*:

```
curl --header "Content-type: application/json"
http://127.0.0.1:8080/user/register
-d '{ "username":"Marble", "password":"meow" }'
```

Under Windows, again *all on one line*:

```
Invoke-RestMethod -Method POST -ContentType "application/json"
http://127.0.0.1:8080/user/register
-Body '{ "username":"Marble", "password":"meow" }
```

Note the following parts of this POST command:

- A header stating that the data will be in JSON format
- POST data attached using the `-d` (MacOS/Linux) or `-Body` (Windows) option
- The data is enclosed in single-quotes, to avoid a conflict with the double-quotes in JSON

The reply confirms the creation of a new user by returning the username:

```
{ "username" : "marble" }
```