AI Game Service (AIGS)

A web service for simple online games

Prof. Dr. Bradley Richards

http://javaprojects.ch

bradley.richards@fhnw.ch

Management Summary

More and more people play games online. Providing a platform for games is not only useful, but also covers many different aspects of software engineering. The AI Game Service (AIGS) is an extensible platform that supports grid-based games for one or two players. Examples of games could include: TicTacToe, Sudoku, Kakuro or Chess. If a game is for two players, the server provides an AI opponent.

AIGS is fully implemented, supporting one game: TicTacToe. This document describes the architecture of the existing server, as well as the TicTacToe game client.

Server repository: https://gitlab.fhnw.ch/bradley.richards/aigs-spring-server

Client repository: https://gitlab.fhnw.ch/bradley.richards/aigs-game-clients

This document also provides guidelines for student projects using AIGS. Student projects may either create new clients for existing games, or else add entirely new games to the system.

Table of Contents

Server Architecture

The server is implemented using SpringBoot. It uses the H2 “in memory” database. This means that each time the server is started, it begins with a freshly initialized database.

Packages

The server provides functionality for two entity types: users and games. These are implemented in the respective packages. Each of these packages contains the following files:

- User.java / Game.java – the classes used to represent the entities. Note that “game” is a generic container, not specialized to any particular game.
- Repository – the repository definition for this entity.
  - In addition to the standard methods, it is possible to define additional methods based on the defined attributes. For example, the UserRepository defines the method `findByToken` to find all users associated with a particular token (there will, of course, be only 0 or 1 such user).
- Controller – the mappings supported by the server for this entity.
- Exception – a class for exceptions raised by the Controller
- ExceptionHandler – a class for handling exceptions raised by the Controller.

The `gameEngine` package contains an interface that must be implemented by any game the system supports, as well as a subpackage for each implemented game.

The utility package contains helper classes used elsewhere in the system. Two of these classes are important

- Initializer – Provides initial data for the database. For example, one can initially define several users, to save time during system testing.
- Token – provides a method to generate random tokens, and a validation method that checks whether a token is currently in use.

User mappings

The available user mappings are:

- GET `/ping`: Tests the connection with the server. Returns “ping”:“success” if successful.
- POST `/users/register`: Register a new user. Requires a user-object containing user-name and password. Returns a user-object with a null token.
- POST `/users/login`: Login a user. Requires a user-object containing user-name and password. Returns a user-object, including a token.

- POST `/users/logout`: Logout a user. Requires a user-object containing a user-name. Returns a user-object with a null token.

Debug functionality: these mappings only exist, because this is an academic project.

- GET `/users`: Returns all information on all users, including their passwords.
- GET `/users/<user-name>`: Returns all information on one user, including their password.

Game mappings

The available game mappings are:

- POST `/game/new`: Create a new game. Requires a game-object containing token, game-type, difficulty and (if applicable) options. Returns a game object containing the initial position.
- POST `/game/quit`: Requires a game-object containing a token. Deletes the game from the server and returns a game-object with result set to true (indicated that the game has ended).
- POST `/game/move`: Requires a mapping of whatever information the specific game requires. Returns a game object with the new position.

Debug functionality: this mapping only exists, because this is an academic project.

- GET `/games`: Returns all information on all current games.

Game Engines

The supported games are declared in the enumeration `GameType` included in the game-package. Each game type must correspond to a subpackage in the `gameEngines` package, and that subpackage must contain a class of the same name.

The functionality required of a game-engine is defined by the `GameEngine` interface:

- A game engine must be able to create a new game, based on an incomplete game-object provided by the `GameController`. The primary task of this method is to return an initial game position to be sent to the player. In addition to the `GameType`, this object will include a desired `difficulty` (`long`) and the desired `options` (`String`). These may be interpreted differently for each `GameType`, and some may not use the `options` at all.
- A game engine must be able to accept a player move. A player move is a `HashMap` derived directly from the JSON parsed by the `GameController`. The contents of this `HashMap` may be different for each `GameType`.

AIGS takes the `GameType` from the user-provided game-object, and selects the game-engine accordingly (via reflection). A game engine is not a permanent object – a new instance is created for each move: the engine receives the current game position, along with the player’s move, and provides a new game position.

While clients are expected to prevent illegal moves, the game-engines must also actively prevent illegal moves. There is no error reporting: illegal moves should simply be ignored: the game-engine simply returns the original game position to the client.

The TicTacToe Game Engine

This game-engine implements TicTacToe as simply as possible. The board is a 3x3 array of `long` integers. The traditional X and O pieces are represented by 1 and -1 respectively. A value of 0 indicates an empty square.

- The `newGame` method creates a new array containing all zeroes, and sets the game-result to false.
- The move method expects two values (`row` and `col`), for the player's move. It then calls an AI-player to make the computer's move, and returns the new game-object to the client.

There are two difficulty levels. Difficulty 1 uses a player who plays randomly. Difficulty 2 uses the Minimax algorithm to play.

Communication

Communication uses the JSON format exclusively. Where possible, `Game` and `User` objects are used, possibly with missing attributes. For example, when logging in, the client sends a `User` object containing the user-name and password (but no token or expiry). If the login is successful, the server replies with a `User` object containing a token and expiry, but it has deleted the password.

In cases where `User` and `Game` objects are not helpful, the JSON is simply interpreted as a map. For example, when making a move in TicTacToe, the client provides a row and a column, so these are the two attributes included in the JSON. Error messages are also handled this way.

Communication example using TicTacToe

Here is a complete game of TicTacToe, from initial user registration through the end of the game. The comments in red are not part of the communication.

Client sends	Server replies
`http://127.0.0.1:8080/users/register` `{ "userName":"Anna", "password":"anna" }`	`{ "password" : "", "token" : null, "userExpiry" : "2022-10-25T12:18:14.252767673", "userName" : "Anna" }`
`http://127.0.0.1:8080/users/login` `{ "userName":"Anna", "password":"anna2" }` ***incorrect password***	`{ "error" : "User error", "error_description" : "Wrong password for user 'Anna'" }`
`http://127.0.0.1:8080/users/login` `{ "userName":"Anna", "password":"anna" }`	`{ "password" : "", "token" : "64881692cd5fd400", "userExpiry" : "2022-10-25T12:28:18.481823789", "userName" : "Anna" }`
`http://127.0.0.1:8080/game/new` `{ "token":"64881692cd5fd400", "gameType":"TicTacToe", "difficulty" :"1" }`	`{ "board" : [ [0, 0, 0], [0, 0, 0], [0, 0, 0] ], "difficulty" : 1, "gameType" : "TicTacToe", "options" : null, "result" : false, "token" : "64881692cd5fd400" }`
`http://127.0.0.1:8080/game/move` `{ "token":"64881692cd5fd400", "row":"1", "col" :"2" }`	`{ "board" : [ [0, -1, 0], [0, 0, 1], [0, 0, 0] ], "difficulty" : 1, "gameType" : "TicTacToe", "options" : null, "result" : false, "token" : "64881692cd5fd400" }`
`http://127.0.0.1:8080/game/move` `{ "token":"64881692cd5fd400", "row":"1", "col" :"1" }`	`{ "board" : [ [0, -1, 0], [0, 1, 1], [0, -1, 0] ], "difficulty" : 1, "gameType" : "TicTacToe", "options" : null, "result" : false, "token" : "64881692cd5fd400" }`
`http://127.0.0.1:8080/game/move` `{ "token":"64881692cd5fd400", "row":"1", "col" :"0" }`	`{ "board" : [ [0, -1, 0], [1, 1, 1], [0, -1, 0] ], "difficulty" : 1, "gameType" : "TicTacToe", "options" : null, "result" : true,` ***game over*** `"token" : "64881692cd5fd400" }`
`http://127.0.0.1:8080/users/logout` `{ "userName":"Anna" }`	`{ "password" : "", "token" : null, "userExpiry" : null, "userName" : "Anna" }`

Miscellaneous topics

Security

Security should be considered in three areas:

- HTTPS: For simplicity, the server uses HTTP rather than HTTPS. Using HTTPS would not alter anything at all in the user and game handling.
- Passwords: Passwords are not encrypted. However, encrypting passwords on the client side would not change anything on the server side. The client would send the encrypted password to the server, instead of the unencrypted password.
- Tokens: Once a user has logged in, they authenticate their game commands with tokens. The token is only known to that specific user, and serves as a “secret” between the user and the server. The token is also used to identify the user’s current game. This is very similar to how commercial web services work (for example, using a Session-ID).

Housekeeping

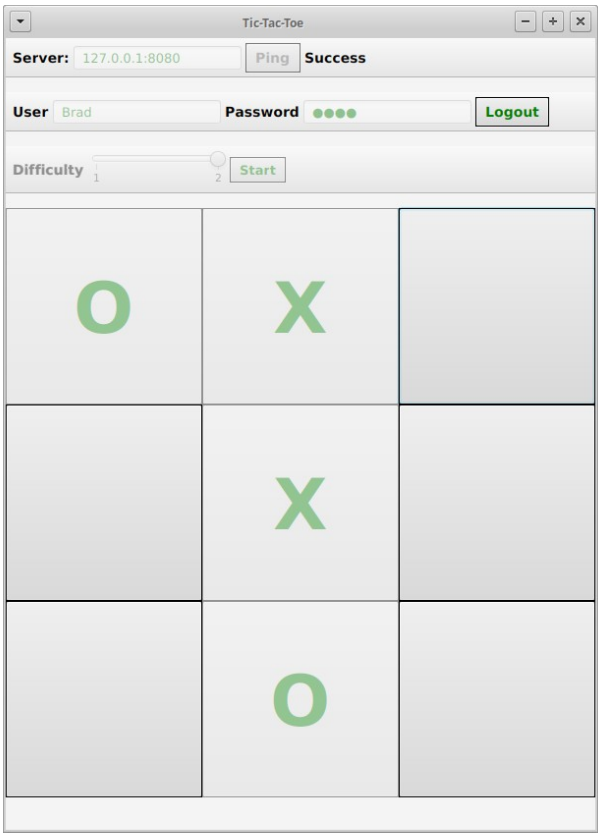
The server currently does not discard old records from the database. However, user objects are provided with an expiry date: this is intended to allow a housekeeping task to run. If implemented, a housekeeping task should:

- Delete users who are past their expiry date.
- Delete games associated with tokens that do not exist in any user record.

Architecture of the JavaFX TicTacToe Client

The TicTacToe client follows the MVC architecture:

- The Model contains the logic. In this case, since the actual processing is on a server, the model contains the code that communicates with the server.
- The View defines what is on the screen, but is fundamentally static.
- The Controller handles all user interaction, communicating with the View and Model as needed.



Server: 127.0.0.1:8080 Success

User Brad Password ●●●●

Difficulty 1 2

The client interface consists of four general parts, as shown in the screenshot on the right. From top to bottom, these are:

- Server address entry and validation
- User and password entry and validation
- Game setup and start
- Game play

In order to make this sample client easy to understand, all three architectural components (model, view and controller) have been split into classes corresponding to these parts. The topmost classes have names beginning with TTT, and they create instances of all of the associated classes.

This means that each individual class is quite small. In a student project implementing a similar functionality, each class can be worked on independently of the others.

View

We describe the view components first, since they are what appear on the screen. The JavaFX code for creating each of the screen regions is simple and largely self-explanatory. The one section that does require explanation is the game board.

The View-class for game play defines a TicTacToe board consisting of a 3x3 `GridPane` of buttons. However, there is no easy way to reach into a `GridPane` and access individual `Button` objects. For this reason, the buttons are also referenced in a 3x3 array:

```
public final Button[][] buttons = new Button[3][3];
```

In this array, the first dimension is the row, and the second is the column. This may be confusing, because the axes are reversed when adding items to a `GridPane`. However, the usage is consistent everywhere else in the application.

The View class for game play also defines a method to allow the Controller to easily update the game board. This method takes a `Game` object (provided by the Model) and updates the text displayed on the buttons. It also disables all buttons which have already been played.

Controller

The controller components include a variety of events and listeners, in order to demonstrate different types of user interaction. Button clicks are handled with the usual `ActionEvent`, whereas text fields are dynamically validated using their `textProperty`.

Some changes to the user interface are driven by properties stored in the Model. For example, it is the Model that receives a token after a successful login. Making changes to the View as a result of a model property-change requires use of `Platform.runLater`, in order to place UI changes under the control of JavaFX.

As each section is completed, the following section is enabled: You can only login after successfully pinging the server. You can only start a new game after login. You can only play after creating a game. This process is also partially reversible: you can logout at any time, which will disable the game setup and game play sections.

The controller for game play makes use of the array defined in the view. All buttons send their events to the same method, which then searches this array to discover which button has been clicked. The row and column of this button are sent to the Model.

The controller monitors the Model’s `GameProperty`. Whenever this changes, an update has been received from the server: the Controller updates the View, and awaits the user’s move.

Model

According to the MVC principle, the model should know nothing of the user interface. Nonetheless, to make the sample client easily understandable, the model defines classes corresponding to the different UI sections, the same as the View and the Controller. Each of these classes contains the methods that send corresponding requests to the server.

In addition to these classes, the model also contains a `User` class and a `Game` class. These are nearly identical to the classes of the same names on the server, except that they omit the Spring annotations. Also, the `Game`-class uses a `String` for game-type, since we have no knowledge of the server’s `GameType` enumeration.

HTTP is a fundamentally simple protocol, and could be implemented in any number of ways. This sample client uses the Java HTTP client from `java.net.http`. Since the server uses Spring, we communicate with JSON. The client uses the Jackson library to process JSON. Obviously, we must follow the server’s specification for how communication works (see the Communication section of the server documentation.

In order to explain how communication works, let us consider creating a new game:

Code from class SetupModel	Explanation

`public Game newGame(String serverAddress,`
`                  String token, String options, long difficulty) {`
	Information to create the game-class

`  Game game = null;`	Default return value is null

`  Game gameQuery = new Game();`
`  gameQuery.setToken(token);`
`  gameQuery.setOptions(options);`
`  gameQuery.setDifficulty(difficulty);`
`  gameQuery.setGameType("TicTacToe");`
`  gameQuery.setOptions("");`
`  try {`
	We send the information in an incomplete game object, constructed here

`    ObjectMapper objectMapper = new ObjectMapper();`
`    String sreq = objectMapper.writeValueAsString(gameQuery);`
	Jackson's object mapper turns an object into a JSON-formatted String

`    BodyPublisher json_req = BodyPublishers.ofString(sreq);`
`    HttpRequest request = HttpRequest.newBuilder()`
`      .uri(new URI("http://" + serverAddress + "/game/new"))`
`      .setHeader("Content-type", "application/json")`
`      .timeout(Duration.of(3, ChronoUnit.SECONDS))`
`      .POST(json_req)`
`      .build();`
	Build an HttpRequest
	- to this server
	- telling it to expect JSON
	- timeout after 3 seconds
	- this is a POST request
	- build the request

`    HttpResponse<String> response = HttpClient.newBuilder()`
`      .build()`
`      .send(request, BodyHandlers.ofString());`
	We will receive an HttpResponse
	- which we build here, from...
	- the reply from sending the request

`    if (response.statusCode() == 200) {`
`      String body = response.body();`
`      ObjectMapper objectMapper2 = new ObjectMapper();`
`      game = objectMapper2.readValue(body, Game.class);`
`    }`
	A statusCode of 200 is success
	- So get the content of the response
	- Create an object from the JSON

`  } catch (Exception e) {`
`    // Do nothing - fall through`
`  }`
	If anything goes wrong, we return null

`  return game;`
`}`
	Return the new game object

In a very few places, such as "ping", we have no data to transmit, so we use HTTP-GET instead of HTTP-POST. This eliminates the first half of the example above, where we are preparing data to transmit. Ping is also special in another way: We have no suitable class to hold the response, so we use Jackson's ObjectMapper to produce a `HashMap` of the JSON contents:

```
myMap = objectMapper.readValue(body, HashMap.class);
```

For making game moves, we could follow the reverse procedure: place row and col values into a HashMap, and then translate this into a String. It is simpler just to build our own JSON manually:

```
String sreq = "{ \"token\":\"" + token + "\", \"row\":\""
                                  + row + "\", \"col\":\"" + col + "\" }";
```

Any of these techniques can be used by other games and other game clients.

Student Projects

General requirements

- Your project will either implement a new client or add a new game to AIGS. The client may be programmed using web technologies (e.g., Angular or React), or in Java using JavaFX. *If you implement a JavaFX client, it must be completely different from the sample client!*
- Each project must be managed in a Git repository, and the instructor must have read-permission (in GitLab that means "Reporter" privileges). All project code *and documentation* must be in Git at the end of the project.
- *Each team member must program a significant part of the project.* Add your name in a comment above methods or classes that you have written. Note that your contributions will be visible in the Git history.

Possible projects

Implement a new client for an existing game

Take an existing AIGS game and create a new client to play the game. For example, if the current client is JavaFX-based, create a web client. The client must have the following features:

- Register, Login and Logout. The AIGS server requires players to have accounts, and to be logged in, in order to play. The client provides to login to an existing account, and to logout.
- The client must be able to start a new game, play the game to the end, and then start a new game.
  - The client must support all relevant features of the game, in order to allow the user to play the game.
  - If the game is a two-player game, the client must allow selection of the skill level of the AI player.
  - The client must validate data before sending it to the server. Depending on the game selected, some moves will be impossible – the client should not allow these moves to be made.
- The client should provide at least minimal styling – not be totally ugly.

Implement an entirely new game

Provide a new rule-module, and (if a two-player game) an AI player with skill levels.

- The rules module must provide full validation: it must not accept illegal moves.
  - Invalid moves should already be prohibited by the client, but servers must protect themselves from hacking and rogue clients.
- If the game is a two-player game, the AI module provides the second player.

    - The AI player must accept a skill-level setting, and provide at least two different skill levels.
- You will need to test your server implementation with a client. *This is an excellent opportunity to coordinate with another group that implements a full client for your game.* See the previous section “Implement a new client for an existing game”.
    - If no other group implements a client for your game, you must implement at least a simple client. This can even be a console-based client.

Implement your own server

A very ambitious project would be to completely re-implement the server, using a different technology. For example, instead of using Spring, implement the server using pure Java. For this project:

- The server must communicate in exactly the same way as the current server – it must be fully compatible with existing clients.
- The server must store `User` and `Game` objects in a database. For this purpose, the new server will have to provide its own ORM (Object-Relational Mapping) code.
- From this foundation, add capabilities and features to your server as you wish.

Project documentation

Title page: List the type of project you have chosen, and the names of all team members.

Usage: Explain how to clone, compile and run your project. *Be sure to test your project on a “fresh” computer, to ensure that these instructions 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)

Using code from external sources

There is nothing wrong with using code snippets from external sources. However, the source must be referenced in a comment in the code. Using external code without referencing the source is plagiarism. For examples: see the methods “hash” and “bytesToHex” in class Account of the existing server.

Handing in the project

To hand in your project, just hand in a link to the project repository in Moodle. Only one person needs to hand in the project.

Helpful tools and references

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/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 is:

```
[
   {
      "password" : "woof",
      "token" : "1046135aeb8a5000",
      "userExpiry" : "2022-10-25T09:58:08.587686",
      "userName" : "Brad"
   }
]
```

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

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/users/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/users/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 object, by returning the object:

```
{
   "password" : "meow",
   "token" : null,
   "userExpiry" : "2022-10-25T10:17:03.546134142",
   "userName" : "Marble"
}
```

Helpful websites

Spring and SpringBoot

There are many helpful websites discussing Spring and SpringBoot. Here are a few:

- Spring home: https://spring.io → see in particular the guides https://spring.io/guides
- Joining tables with foreign keys: https://www.baeldung.com/spring-jpa-joining-tables
- Avoiding infinite recursion in JSON with joined tables: https://javacrafters.com/spring-boot-jpa-infinite-recursion/ or https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion

Jackson

Jackson is the most well-known Java library for handling JSON. It can automatically translate between JSON and POJOs (plain old java objects) or between JSON and HashMaps of keys and values.

- Helpful examples: https://www.digitalocean.com/community/tutorials/jackson-json-java-parser-api-example-tutorial
- Information about JSON: https://www.w3schools.com/js/js_json_intro.asp
- Baeldung Jackson tutorial: https://www.baeldung.com/jackson

Java HTTP client (java.net.http)

HTTP is an easy protocol to implement yourself: you need nothing more than sockets and text. However, if you want to make use of the many options available in HTTP, you can also use the HTTP client build into Java.

- API documentation: https://docs.oracle.com/en/java/javase/17/docs/api/java.net.http/java/net/http/package-summary.html
- Baeldung examples: https://www.baeldung.com/java-9-http-client