SOI: 1.1/TAS DOI: 10.15863/TAS

International Scientific Journal

Theoretical & Applied Science

p-ISSN: 2308-4944 (print) **e-ISSN:** 2409-0085 (online)

Year: 2022 **Issue:** 06 **Volume:** 110

Published: 05.06.2022 **http://T-Science.org**

Issue Article

Vadim Andreevich Kozhevnikov
Peter the Great St.Petersburg Polytechnic University
Senior Lecturer
vadim.kozhevnikov@gmail.com

Abdulla Salimovich Akhmedov
Peter the Great St.Petersburg Polytechnic University
Student
abdulla.akhmedov431@gmail.com

DEVELOPMENT OF THE SERVER PART OF A MULTIPLAYER ONLINE GAME USING WEBSOCKET

Abstract: *This article is devoted to the development of the server part of the online game. A comparative analysis of various algorithms for generating labyrinths has been carried out. The implementation of the software product, the main structures, their methods and interactions between the main modules of the application, and the process of orchestrating services are described.*

Key words: *server development, administration, continuous integration and delivery, websocket protocol, mazes, maze generation algorithms, Golang.*

Language: *English*

Citation: Kozhevnikov, V. A., & Akhmedov, A. S. (2022). Development of the server part of a multiplayer online game using websocket. *ISJ Theoretical & Applied Science, 06 (110),* 43-49.

Soi: http://s-o-i.org/1.1/TAS-06-110-7 ***Doi***: crossref https://dx.doi.org/10.15863/TAS.2022.06.110.7

Scopus ASCC: *1700.*

Introduction

Recently, video games have evolved greatly - with the advent of mobile phones for video games, completely new opportunities have opened up. With the development of network technologies and the general increase in the average speed of information transfer between two devices, new ideas and tools for developing network games have appeared. This article discusses the development of the server part of a mobile online game, the main idea of which is to pass the labyrinth at speed. Players start the game in two opposite cells of a procedurally generated random maze. The game ends when one of the players reaches the cell from which his opponent started the game.

The relevance of the work lies in the creation of a unique game in the theme of passing labyrinths. The uniqueness lies in the fact that the mazes are procedurally generated, and each time the user is offered a new maze to play, instead of one of the pre-designed finite set of mazes. It is also based on a competitive idea - users play in the same maze against each other in real time and see the movements of the opponent.

Problem statement, mazes theory

The aim of the work was to develop the server part of the game, as well as delivering the finished application to the server and creating an environment for its administration.

To achieve this goal, it was necessary to perform a number of tasks:

1. Analyze similar games, identify the main shortcomings;
2. To study the theory of labyrinths and algorithms for creating labyrinths;
3. Determine the technology stack that will be used in the development of the application;
4. Create a planned architecture design;
5. Implement the software part of the application;
6. Implement a bot. Make the bot behave like a human;

7. Compile documentation for public interfaces;

8. Set up CI/CD tools;

9. Configure a reverse proxy, set up a production and test environment;

10. Secure access to the server.

When reviewing existing games with labyrinths, the following common features of applications related to the theme of labyrinths were noted: as a rule, they differ from each other cosmetically and in the way of control, in all applications the set of labyrinths for passing was generated in advance, and these games, as a rule, are intended for a single game.

The labyrinth of the ancient Greeks and Romans meant a more or less vast space, consisting of numerous halls, chambers, courtyards and passages, arranged according to a complex and intricate plan, in order to confuse and prevent a person ignorant of the labyrinth from escaping. Labyrinth is a Greek word that, in a stricter definition, means any structure (usually in two or three dimensions) consisting of intricate paths to the exit (and / or paths leading to a dead end). An exit is any vertex, cell or face of the maze.

Labyrinths (and therefore algorithms for creating labyrinths) can be organized into seven different classifications [1]. These are: dimension, hyperdimension, topology, tessellation, routing, texture, and focus. The maze can take one item from each class in any combination.

In this work, we considered two-dimensional, ideal labyrinths and algorithms for their generation for various types of tessellation. Two-dimensionality means two-dimensionality in the mathematical sense of the word. A "perfect" maze means a maze without any loops or closed circuits, and also without any inaccessible areas, it is also called a simply connected maze. From every point there is exactly one path to any other point, the maze has exactly one solution. In computer science terms, such a maze can be described as a spanning tree over a set of cells or vertices.

There are three main types of tessellation or tiling:

An orthogonal maze is a standard rectangular grid in which cells have passages that intersect at right angles.

Delta Maze - The delta maze consists of interconnected triangles, where each cell can have up to three passages connected to it.

Sigma Maze - The sigma maze consists of interconnected hexagons, where each cell can have up to six passages connected to it.

There are other types of tessellation, but in the framework of this work, we considered the types of tessellation described above. It can be noted that the types of tiling differ in the number of faces of one cell of the labyrinth or the number of passages from this cell.

There are several ways to create perfect mazes, each of which has its own characteristics. Most of these are described as creating a maze by cutting out passages, however, unless otherwise noted, each can also be done by adding walls.

The main algorithms for generating labyrinths:

- Kruskal's algorithm. This algorithm is based on the creation of a minimum spanning tree. It's interesting because it doesn't "grow" the maze like a tree, but rather carves out passage segments throughout the maze randomly, but still ends up with a perfect maze.

- Prim's algorithm. This algorithm is based on creating a minimum spanning tree that works with unique random edge weights. Kruskal's algorithm, which also creates a minimum spanning tree, can be considered the best because it is faster. In fact, given the same initial value of random numbers, it is possible to create identical mazes using Prim's and Kruskal's algorithms.

- Aldous-Broder algorithm. The most interesting thing about this algorithm is that it is universal, which means that it generates all possible mazes of a given size with equal probability. The bad side of this algorithm is that it is very slow, as it does not perform any reasonable search for the last cells. In fact, it's not even guaranteed to be completed.

- Wilson's algorithm. This is an improved version of the Aldous-Broder algorithm because it creates mazes with exactly the same texture as the Aldous-Broder algorithm (algorithms are unified with all possible mazes generated with equal probability), but Wilson's algorithm is much faster.

- Eller's algorithm. This algorithm is special because not only is it faster than all the others with no obvious offsets or flaws, but its creation is also the most efficient in terms of memory usage.

Technology stack

The compiled multi-threaded programming language Golang [2], developed internally by Google, was chosen as the main programming language. The main advantages of the language are its speed and ease. Golang is a fast compiled language [3]. Using the Go language, it is convenient to write network applications. The standard distribution of the language contains a large number of convenient tools and libraries for working with the network, in particular, the “net/http” package covers all the functionality of the HTTP protocol.

Object-oriented programming tools are limited to interfaces. This allows programming with abstractions. Golang provides tools for multi-threaded programming. The language has a “goroutines” mechanism, which is a lightweight thread, as well as a means for communication between threads: channels.

The Java language and the Spring framework were considered as the main alternative. The main disadvantage is the high entry threshold. Since Spring is a full-fledged framework, to work with it, you must

be fully familiar with the principles of working with this tool. Also in Java, working with multithreading and networking is more complex compared to similar work in Golang.

"Clean Architecture" was chosen as the main design pattern. The main ideas of this pattern are to separate the logic into the immediate logic (or business logic) and the transport layer. Thus, with the help of dependency injection, we abstract from the way we receive data from the client.

The choice of communication protocol was dictated by the main objectives of the online game. To support the completion of the maze in real time, it is necessary to open a tunnel with the client and keep it open for the required time, for example, until one of the players has completed the maze. Ultimately, a full-duplex communication protocol over a TCP connection was chosen - WebSocket [4].

The convenience lies in the fact that to search for a game and for the game itself, it is enough to open one websocket and send messages with different commands to it. All that is needed to support this approach is command handlers on both sides: the client side and the server side.

The Wire library was chosen for dependency injection. This library allows you to conveniently inject all the necessary dependencies. The main tool is code generation. To implement the necessary dependency, it is enough to write the so-called provider of this dependency, which creates and configures it. The very task of introducing this dependency into all the necessary consumers is taken over by Wire.

Architecture overview, software implementation

The central objects of the system are: client, game, queue (Fig. 1).

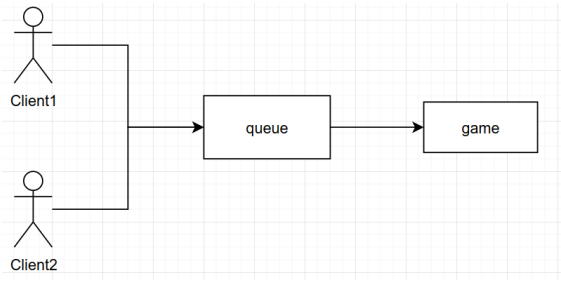

Fig. 1. Architecture diagram.

Client is a structure responsible for the player. This structure stores information about the player: name, color, as well as other technical information for the correct rendering of the labyrinth on the client side. To search for a game, the client sends a request to another object of the system - a queue.

Queue is a framework for creating games. This structure handles client requests for queuing, as well as creating a game for clients.

Game is a game structure between two clients. Handles all the information that clients provide during the game, such as their coordinates. This structure is also responsible for calculating whether any client went through the maze, that is, ended up in the cell he needed.

The central structure in an application is the Application structure. The Wire library from Google is responsible for dependency injection. As a product develops, the number of code blocks that are linked together grows. At the same time, dependency management becomes more difficult. Dependency injection systems come to the rescue. They can be divided into two types: based on code generation and based on reflection. Systems based on reflection provide the necessary structures during program execution. This negatively affects the speed of the program, since reflection is a slow tool. Products based on code generation generate all the code necessary for the program to work at the compilation stage. This approach also has its disadvantage: the program compilation time increases.

Wire is a tool based on code generation. To work with Wire, you need to write a dependency provider function for each of the dependencies, such a function is called Provider. Providers must have a specific signature. They must return strictly 3 parameters: any structure (interface), a resource release function, and an error.

Next, you need to collect all the providers together by calling the wire.Build () function. Providers should be passed to this function in a strictly defined order: first, all providers that do not require other dependencies to initialize the structure are passed. Next, all functions are passed to consumers of dependencies provided by providers. In Wire terms, such functions are called Consumer.

Multiplexing or routing of requests is carried out using the Gochi library [8]. There are multiplexers in the standard distribution of GoLang, but their functionality is very limited.

There are only two endpoints registered in the application. The /generate endpoint takes the size of the maze as input and generates it. The response to the request comes in json format. The second endpoint is used to open a WebSocket. This happens with the help of the Upgrade mechanism [10]. A web socket connection begins with an HTTP GET request from the client to the server. The request contains a special header to tell the server that it wants to create a WebSocket connection. The client should send a Sec-WebSocket-Key header containing base64-encoded random bytes, and the server responds with a hash of the key in the Sec-WebSocket-Accept header [11].

The Client structure represents a client in terms of a client-server interaction. The game also has bots. The Client layer abstraction allows you to organize the code in such a way that it does not matter to the application what type of client is participating in the game. The part of the code directly responsible for the game does not distinguish between a real person and a bot. The Client structure has two methods: read() and write(). The read() method tries to read messages from the WebSocket in an infinite loop. If successful, it send this message to the Game structure for further processing. The write() method reads messages from the Send channel in an infinite loop and sends them to the WebSocket. When creating a client, you need to run two goroutines: on read() and write().

The GameQ framework is for registering clients and creating games between clients. The main method of this structure is the Run() method. The Run() function works according to a simple principle: if a client was sent to the register channel and there is a client in Clients, then we create a game, otherwise we put the client in the queue. The Run() method must be run in the goroutine after the GameQ structure has been initialized.

The Game structure is the central structure of a game between two clients. This is where all messages from clients are processed. All the main logic for processing commands, like the rest of the structures described above, is in the Run () method.

The Message structure is used to send messages from the client to the server and back.

The Bot structure is used to design a bot. In the application, the bot is designed to mimic the game of a real person. This is done so that the user does not know whether he is playing with a bot or with a real user. The client starts the game with the bot after waiting for a certain period of time specified in the config, from the moment of queuing. This logic was implemented to ensure that the client does not stay in the queue for too long. To simulate the humanoid movements of the bot, it was decided to add random fluctuations to the bot's trajectory. Also, logic was added that deploys the bot if there is a dead end in front of it with a length equal to the value of the variable in the config. The main method of the bot functioning is the solveMaze() method. This method takes as input the generated maze in the form of an array of cells. A cell is characterized by indices X and Y in the maze matrix, as well as an array of length N, where N is the number of edges in the maze cell (takes values 3, 4, 6).

First, using the depth-first search, the bot solves the maze. In the course of the search in depth, the cells in which the search took place are written to the array, thus forming the trajectory of the bot in the maze. It remains to walk along the already formed trajectory of movement.

Since the clients in the system have usernames, it was necessary to implement the generation of random names for the bot. It was decided to implement a simple scheme with a set of adjectives and nouns.

There are many different approaches to application programming. One such approach is TDD, which stands for Test Driven Development. The main idea of this approach is to formulate the specification, define interfaces, write tests with the expected behavior of code sections, and only after that write the code that executes the logic. It is difficult to overestimate the importance of writing tests during development. Properly formulated test cases (a set of code usage scenarios) allow you to reduce a lot of time for debugging and fixing errors in the future.

As part of the development of this product, it was decided to use unit testing. A feature of unit testing is the testing of small, independent sections of code or modules. Typically, units of testing are functions, methods, procedures, modules, or objects. A typical test function structure has two parts. The first part is setup or preparing test input for the unit of code being tested. The second part is asserting or checking the correctness of the output of the unit of code being tested.

Unit testing in Go is just as arbitrary as any other aspect of the language, such as formatting or naming [9]. The syntax deliberately avoids the use of asserts and places the responsibility of checking values and behavior on the developer. Go has a built-in test command called go test and batch testing, which together provide a minimal but complete testing tool.

For tests, a separate file is usually created with the name of the file being tested and the _test suffix. Since the standard distribution of the language does not provide tools for checking values, the testify library was used. This library provides a large set of out-of-the-box functions for checking the values of structures and primitives.

Developing an application in a client-server architecture implies development from two sides: from the client side and from the server side. This entails some organizational problems, especially in terms of direct interaction between parts of the system. In the course of development, requirements, specifications, functionality may change. This is why

it is so important to write and maintain consistent documentation for all public interfaces.

The Swagger tool was chosen to compile the documentation. This is a code generation based product that allows you to create a description of requests, as well as various information transfer models that are used between the client and the server. The convenience lies in the fact that the creation of documentation occurs by commenting on sections of code that the developer wants to document.

In the modern technological world, there are a large number of different devices. Different architectures, operating systems, versions of installed libraries - all this dictates new standards in application delivery. To simplify and speed up the delivery of the product, Docker containerization technology was invented. A rough definition of a container sounds like a lightweight virtual machine. This approach allows you to write a product assembly description once (this description is called an image) and use it as many times as you like without thinking about the OS, architecture, and other characteristics of the host machine.

There are two main resources that separate the containers and the host machine - the network and the file system. Docker provides the ability to flexibly manage both. This allows you to create a large number of virtual networks for a secure, isolated connection to a network of different containers.

Running a single container on a server is not a big deal, but when the number of services (and with it the containers) that need to be started and administered at the same time increases, so does the complexity of management. To solve this problem, orchestration systems were invented. The description of the container interaction process is a file with the .yml extension. It describes all the containers, the order in which they are launched, the virtual networks used by the containers, the areas of the file system to be shared between the host machine and the container. Using the orchestration system, it is convenient to set environment variables for the container, as well as set the rules for restarting the container in case of an unexpected termination of work.

When developing with small changes and a large number of commits, there is a need for frequent manual testing of the application. To do this, you need to deploy the program in a test environment. The principles of continuous integration involve automating the process of delivering an application to the required environments. In reality, manually integrating and configuring the application would take a significant amount of time.

Github Actions [7] was chosen as the CI/CD tool. The integration process is executed on the Github servers. To describe the rules, you need to create a file with the .yml extension in the .github/workflows directory. A pipeline consists of a list of tasks or jobs. Tasks run in parallel. Each task consists of a list of steps. Steps within a single task are performed sequentially. Also in this file, you must specify the event trigger - an event that will launch the pipeline. Github offers a huge variety of all kinds of events. For the pipeline that deploys the application to the test environment, an event was selected on the push of the testing tag.

A reverse proxy server is used to relay requests from the external network to any servers / services of the internal network (for example, web servers, databases or file storages) and allows (Fig. 2):

- to ensure the concealment of the structure of the internal network and the details of the services located in it;

- perform load balancing between instances of the same service or servers with the same tasks;

- provide an encrypted (HTTPS) connection between the client and any service, in this case an SSL session is created between the client and the proxy, and an unencrypted HTTP connection is established between the proxy and the service on the internal network, if the service supports HTTPS, then you can organize an encrypted connection on the internal network;

- organize access control to services (client authentication), as well as install a firewall.

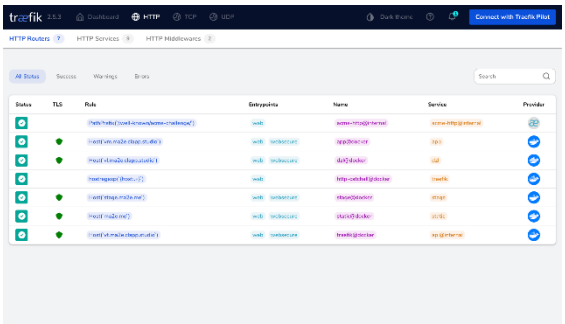

Fig. 2. Reverse proxy control panel

During the development of the application, there was a need to set up two isolated environments: a working (production) and a test (dev) one. The purpose of this event is to keep the stable version of

the product working as the application is in release. Testing new functionality and fixing bugs can serve as a potential threat to performance. To prevent possible problems, some restrictions on system administration have been introduced, namely:

- Separation of application versions at the image level. The main image ma2e (production environment) with the latest tag is the actual release build of the application. A mobile client located in the Google Play Market application store at the api.ma2e.me domain sends its requests to the container of this image.

- The image for testing functionality (test environment) has a "-stage" suffix. The container of this image is not used by the actual users of the application, but is needed only for debugging by developers. Requests to the data container are proxied through the stage.ma2e.me domain.

- To update the release build of the application, you must first test its performance in a test environment. You need to manually make sure that there are no critical errors, namely, the application was successfully compiled and launched. Next, you need to run all the tests and make sure that they all pass successfully. Only after these steps can you update the release build.

It often happens that users find a bug in a working stable version of an application. In order to effectively correct such errors, it is necessary to understand what user steps led to this error. To do this, it is important to configure entries in the event log. Logs (log files) are files containing system information about the operation of a server or computer, in which certain actions of a user or program are recorded. The open source program Dozzle was chosen as the log collection tool. This product is especially useful when building logs from docker containers. Dozzle works on the principle of serving a web page to view the contents of container event logs. For ease of development, a subdomain was also created for the Dozzle service. Requests are proxied for log.ma2e.me domains. Since information about the operation of the program is classified as sensitive, as well as the fact that Dozzle is hosted on a public network, it is necessary to protect access to this resource. This was done using traefik tools [12]. In particular, the bauth interceptor was used, which requires the presence of the Authorization header for all requests to this container.

There is a set of rules that can help secure the connection to the server via SSH [6].

- Change the default SSH port. Port 22 is reserved for the SSH protocol, but you can use any of the available ones. It's worth noting that changing the port won't make connections secure, but it can prevent automated attacks that typically assume the server is listening on port 22.

- Using a pair of keys for authorization. The standard SSH authentication method is a password, but the password can be easily compromised, for example, through a man in the middle attack. To avoid this, you can use asymmetric encryption algorithms and perform authorization using a key pair.

- Prohibition of connection to the server with superuser rights.

- Prohibition of connections, the means of authorization of which is a password.

- Restrictions on IP addresses from which you can connect to the server.

The ability to connect to the server via root has been disabled (for this, you need to edit the ssh configuration file). In the same file, the ability to log in with a password was disabled.

The system has two panels designed for private developer access, however, the presence of one server does not allow proper protection of the panels at the network level. The correct way to do this is to keep all internal services in the gray subnet. The gray subnet refers to addresses on the local network, that is, services do not listen on public interfaces.

To protect technical panels within one virtual private server, a protection method was chosen using a login and password for authorization. Authorization occurs by means of a traefik reverse proxy. In the standard distribution there is a bauth request interceptor. This interceptor checks the Authorization header for each request to the required service. The value of the Authorization header is a base64 encoded combination of username and password separated by a colon. Valid login and password combinations are specified in the userfile. For security purposes, the password has been hashed.

Conclusion

We have developed a server part for an online game. Based on the results of the work, the application is deployed on the server, configured and ready to receive and process client requests.

To achieve this goal, a number of tasks were performed:

1. The most popular maze-themed apps in the Google Play Market app store were analyzed. The analysis revealed the main shortcomings of the games.

2. The theory of labyrinths, their classification, the main types of labyrinths according to the type of tiling have been studied. A comparative analysis of the labyrinth generation algorithms was also carried out, the main criteria were identified and determined when comparing the generation algorithms.

3. Technological stack for application development formulated and justified.

4. The design of the application architecture has been created in terms of the interaction between the modules and structures of the program.

5. The software part of the game was implemented. All the main structures in the application were described, their methods, as well as

interaction with other structures. A tool for configuring game parameters has been created.

6. Implemented a bot to reduce the user's expectation of finding a new game. The bot was designed in such a way that it was difficult to distinguish it from a real person. A fluctuation was added to the movements of the bot, the bot turned around at the sight of a dead end, and did not rest against it. A random name generator for the bot was also implemented.

7. All public interfaces were documented using the Swagger tool. All models for communication between server and client have been described. Access to the documentation in the form of a web page has been created.

8. CI/CD Github Actions tools were used to automate the delivery process and integrate new versions of the software product.

9. To route requests to the right containers, a traefik reverse proxy was deployed. To isolate the release and test versions of the program, a semblance of a production and test environment was created.

10. Access to the server is via SSH. Access to the server was protected, as well as access to technical web panels with public access.

At the moment, the main steps for monetizing the application are being considered, since this requires users to have accounts, and technical aspects are also being considered. There is a certain set of tasks that have been formed to technically improve the game, such as implementing client move validation to make sure the user does not go through the walls of the maze.

References:

1. (n.d.). *Lists of Maze generation methods, Maze solving methods, and classes of Mazes in general*. Retrieved 29.05.2022 from http://www.astrolog.org/labyrnth/algrithm.htm
2. (n.d.). *The Go Programming Language*. Retrieved 29.05.2022 from https://go.dev/
3. (n.d.). *Golang dlya samyh malenkih*. [in Russian]. Retrieved 29.05.2022 from https://habr.com/ru/post/588743/
4. (n.d.). *The WebSocket Protocol*. Retrieved 29.05.2022 from https://datatracker.ietf.org/doc/html/rfc6455
5. (n.d.). *Docker Get Started*. Retrieved 29.05.2022 from https://docs.docker.com/get-started/
6. (n.d.). *Naskolko horosho zaschicheny vashi SSH-sessii?* [in Russian]. Retrieved 29.05.2022 from https://habr.com/ru/company/pentestit/blog/336726/
7. (n.d.). *Github Actions documentation*. Retrieved 29.05.2022 from https://docs.github.com/en/actions
8. (n.d.). *Gochi Routing*. Retrieved 29.05.2022 from https://go-chi.io/#/pages/routing
9. (n.d.). *Golang Unit Testing*. Retrieved 29.05.2022 from https://golangdocs.com/golang-unit-testing
10. (n.d.). *Protocol upgrade mechanism*. Retrieved 29.05.2022 from https://developer.mozilla.org/en-US/docs/Web/HTTP/Protocol_upgrade_mechanism
11. (n.d.). *Websocket Connection Handshake Under The Hood*. Retrieved 29.05.2022 from https://medium.com/easyread/websocket-connection-handsake-under-the-hood-560ab1ceaff5
12. (n.d.). *Vvedenie v Traefik 2.0*. [in Russian]. Retrieved 29.05.2022 from https://habr.com/ru/post/508636/