Technical Documentation

# Program Flow and User Interface

|  |  |
| --- | --- |
|  | The client opens with a login interface. The user can enter a username and password. The password will be hidden by the dot character.  The user can also register a new account with the server by using the Register link. |
|  | Clicking the register link will open a new form which allows the user to enter new login credentials. There are several checks on the username and password to confirm they are suitable before the user will be allowed to register (See “Logging In and Registering” for details). |
|  | After registering an account, the register form will automatically close and the login form will refocus. Pressing Login will initiate the verification process (See “Logging In and Registering” for details). |
|  | This is the main window of the system. The window has several features. The selection boxes and number selectors on the left allow the user to select the parameters of the maze. Filling in all boxes whilst being connected to the server will unlock the Request Maze button which will send the information to the server where a maze will be generated (See “Maze Generation” for more detail.) Alternatively, by pressing Get Mazes, the selection box below the button will be populated with mazes the user has saved to the server. The user can load or delete the maze from the server with the relevant buttons. The panel on the right hosts dynamically generated charts showing local and global statistics for best times and mazes generated. |
|  | Here is the main form populated with data. The Request Maze button is now unlocked since each parameter has been filled in. Clicking the button will send the parameters to the server and a maze will be passed back to the client, being displayed in a new form. |
|  | Here is the Display form. The user can use WASD controls or the onscreen buttons to move the blue square through the maze. If they do this, a timer will start and be displayed, stopping when they reach the end of the maze. The Close button will then be unlocked, allowing access to the parameter form. The user can also use the selection box to select the algorithm they would like to request a server solve with. Pressing request solve will send the maze to the server which will send a solution back to the client before displaying the solution. The user also has the option to name their maze for a server save, or save it locally, which will open a File Explorer window where they can choose where to save the formatted image. |
|  | Here is the display form after requesting a solve using the Maze Routing algorithm. The solution is displayed with a purple line. The user can no longer attempt a manual solve. The close button has been enabled. Closing the window will refocus the parameter window and allow the user to begin the process again. |
|  | In the background, the server has been open the entire time. It logs all requests made to it in the console window. The client has exception handling on all server requests, since if the server closes unpredictably, there will be an unavoidable error which needs to be appropriately handled. (See “Network Exception Handling” for more details). |

# Project Structure

|  |  |
| --- | --- |
| Client | Server |
|  |  |
| The client’s code is mainly held within its form codebehind files. The Maze and Maze-derivative files it has contain only properties and [JsonConstructor] tagged constructors since they only need to build objects from the property structure to be read and displayed. This keeps the client lightweight. The client has a copy of every protocol file so it can send and receive the appropriate objects across the network. The client also makes use of a static class Globals (See “Global Variables” for more details.) | The server has many features in its extensive solution. It contains a folder of every protocol file so it can communicate correctly with clients. Each of these protocols have a dedicated service script in the Services folder, defining the management of the data in these requests, and what is sent back to the client. It also contains both the launchSettings.json and the appSettings.json files. These define how the server behaves (such as what port it operates across, whether it logs messages, what network protocol it uses, etc). It also contains the full definitions of all Maze classes and has the Solver and Solver-derivative classes, unlike the client. The server does not make use of global variables. |

# Techniques Used

## General Programming Techniques

## Network Exception Handling





Here is an example of how I have managed unpredictable server errors. When we make a network request, we can attach a deadline to it by providing a time: I have provided the current UCT time + 3 seconds for all deadlines, essentially forcing an exception after 3 seconds of server inactivity. Since we are turning an unpredictable server error into a predictable and specific error, we can catch that specific error by checking for the DeadlineExceeded status code, which is thrown when the deadline for a server request is exceeded. In the catch block, I have a procedure which is called whenever there is a server error that outputs an appropriate error message to a label on the form, as well as locking up buttons that interact with the server until a connection is re-established.

## Multithreading Exceptions

The code in my forms has many async and multithreaded methods. In a multithreaded form, threads cannot access objects from other threads without invoking a subroutine that interacts with it by proxy. However, the thread does not know when an object has been disposed, such as when a form closes, so an unpredictable error is thrown when trying to invoke access to a disposed object. Since this error causes no issue if it is ignored, we can stop a client crash if we specifically catch the ObjectDisposedException without handling it.

## Global Variables

See [PAGE] for the Globals.cs code.

Since excessive use of global variables makes code less robust, I have limited my usage to constant variables only. Furthermore, all global variables are prefixed with “g\_” to differentiate them from local variables. The only 2 non-constant globals are the username and userID, since these cannot be defined in the code as the system has many users. These are updated to the correct values at login and only read from afterwards. The version string is used to update the name of each form with the current client version. It is purely aesthetic but could have use in managing client updates. The cellWidth and cellHeight variables store the pixel width and height of cells in the maze, so the dynamically sized display form can autosize correctly. The keysize and iterations variables are used in hashing passwords (see “Logging In and Registering” for more details).

# Server Protocols

## Proto Files

See [PAGE] for the protocol files.

Protocol files end in the extension .proto and have their own language and syntax to define a service which operates across the network and the related request and response objects the service uses. The files have a set structure: first, the syntax is set. All my protocols are written in the proto3 syntax, so the top line of every file will set this. Next, the namespace the protocol will be implemented in is set. My project uses the Server namespace and Client namespace, so all my protocols are passed the Server namespace. Next, the service is defined. The service can contain many operations, represented by the rpc keyword. The operation is made up of the method name, the request object name, and the reply object name. Once all the operations have been defined in the service, the request and reply objects must be defined with what data they carry. This is done using the message keyword followed by the object name. Within the braces each variable is defined using a simple data type and an identifier. The number assigned to them is the order in which they must be passed to the collection when creating a request. Some of my protocols do not need to return a reply message to the client since they happen discreetly, such as incrementing stat values. In this case, you can import the google/protobuf/empty.proto, which when set as the reply object in the rpc definition, allows you to create a void network service. Once these protocols are defined, they have to be integrated into the project to allow the service code to be written (See “Project File References” for more details).

## Project File References

See [PAGE] for the client and server project files.

Since protocol files generate obfuscated code to facilitate the low-level transfer of data across the internet, they need to be referenced in the project file, so the compiler knows to create these files when the project is built. This is done by including the path to the file in a protobuf element within an itemgroup in the project file.

# Algorithms

## Database Management

### General information

Across all SQL commands, I have implemented the same techniques.

* I have used the using keyword on all SQLite objects, so they are properly disposed after use, preventing memory leaks.
* I have used the @ symbol before all SQL commands to make them into multiline verbatim string literals. This allows me to put each component of the SQL command onto a new line, vastly increasing readability.
* I have used the tag system to manually add parameters rather than interpolating them in. This helps prevent SQL injection attacks.

### Table creation

See [PAGE] for the table creation scripts.

I have used the pragma command to turn foreign key constraints on. This means that SQL commands that would cause the primary and foreign keys across 2 tables to become mismatched are ignored.

The GlobalStats table records 3 pieces of information for each time: an integer time in milliseconds, a string display time, and the username of the user who set it. The latter 2 are used in displaying the best times on the client. The millisecond time is used in a serverside algorithm which orders the times, which would be made harder if the times were formatted strings instead.

### Triggers

See [PAGE] for the trigger body.

My database uses 1 trigger: CreateStatsRecord. The trigger creates a record in the stats table when a user registers a new account and fills it with default data. It also sets the userID of the record to be the userID of the new user. This is necessary since the stats record is not otherwise automatically created when the user registers, so many of the stat related services would break.

The code below this is similar: we need exactly 1 global stat record, so the SQL checks if there are any records in the GlobalStats table, and only adds a record if there is one. This record is filled with default data when it is initialized.

## Maze Generation

### Recursive Backtrack Algorithm

### Growing Tree Algorithm

### Wilson’s Algorithm

## Maze Solving

### Depth-First Search

### Maze-Routing Algorithm

## Logging in and Registering

See [PAGE] for the implementation of these features.

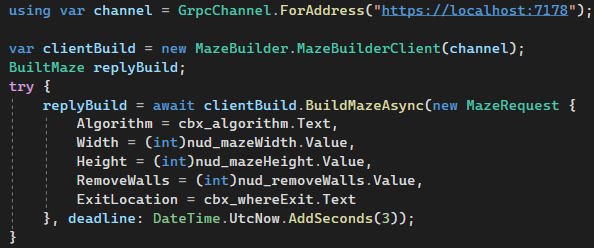
When the user presses the Register button on the registration form, their credentials are not immediately sent to the server. First, their credentials are checked:

* The server is queried as to whether the username is taken.
* The password length is checked.
* The password is checked for special characters using Regex. It must contain at least 1
* The password is checked if it matched the Confirm Password box.

The credentials are processed only if these checks are passed. The first step to processing the credentials is hashing the password. We call the HashPassword subroutine [[1]](#footnote-1)to do this. The salt is randomly generated with a length defined by the keySize global constant. The hash is then produced by the PBKDF2 algorithm with a number of iterations defined by the iterations global constant. This hash is then passed to the server as a hexadecimal string, as well as the hexadecimal conversion of the salt. The hex conversion is necessary as the hash and salt are both byte arrays, which are not supported types in the protocol files. The server creates a new record for the new user and stores their username, password, and salt under a new userID.

When the user attempts a login, the username and password entered are sent to the server’s LoginHandler service. The server uses SQL to find the stored password and salt hashes associated with the entered username. It then hashes the password it received using the same salt and algorithm parameters. It compares the hashes using the FixedTimeEquals method, so the state of the server cannot be guessed via the length of time the comparison takes. If the hashes are the same, the user is granted access and the clientside globals username and userID are updated. If the credentials are incorrect, the message “Username or Password incorrect!” is displayed. The message is ambiguous so the user cannot guess usernames and passwords and get information from it.

## Client-Server Interactions



Here is an example of a client call to the server. There are 3 steps to making a server call, and these are the same regardless of which service the client is requesting. Firstly, a communication channel is established with the server, using the address specified in the server’s launchSettings.Json file. The using keyword is used so the channel is properly disposed after use, to avoid memory leaks. Next, using the channel, the service to request is specified. Finally, the client calls the function of the service, passing in the request object with the relevant parameters. The await keyword is used here, and the method encompassing this code is an async method. This is because the time the server will take to receive, process, and transfer the data is uncertain, and we don’t want the client interface to hang while it is waiting. The async and await keywords allow the user interface to be interactive whilst the server manages data.

## Other Algorithms

1. Sourced from https://code-maze.com/csharp-hashing-salting-passwords-best-practices/ [↑](#footnote-ref-1)