Appendix

# Shared code

## Protos

These proto files appear in both the client and server solutions, so I have only attacked them once for brevity.

### BuildMaze.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service MazeBuilder {  rpc BuildMaze (MazeRequest) returns (BuiltMaze);  }    message MazeRequest {  string algorithm = 1;  int64 width = 2;  int64 height = 3;  int64 removeWalls = 4;  string exitLocation = 5;  }    message BuiltMaze {  string maze = 1;  } |

### CheckIfUserExists.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service CheckerIfUserExists {    rpc CheckUser (Query) returns (Exists);  }    message Query {    string username = 1;  }    message Exists {    bool userExists = 1;  } |

### DeleteMaze.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service DeleterMazes {    rpc DeleteMaze (DeleteRequest) returns (SuccessAcknowledge);  }    message DeleteRequest {    int32 mazeID = 1;    int32 userID = 2;  }    message SuccessAcknowledge {    bool success = 1;  } |

### GetMazes.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service GetterMazes {    rpc GetMazes (Request) returns (MazesList);  }    message Request {    int32 userID = 1;  }    message MazesList {    string mazes = 1;  } |

### GetStats.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service StatsGetter {    rpc GetGlobalTimes (GetGlobalTimesRequest) returns (GlobalTimes);    rpc GetUserTimes (GetUserTimesRequest) returns (UserTimes);    rpc GetGlobalMazesGenerated (GetGlobalMazesGeneratedRequest) returns (GlobalMazesGenerated);    rpc GetUserMazesGenerated (GetUserMazesGeneratedRequest) returns (UserMazesGenerated);  }  message GetGlobalTimesRequest {    }  message GlobalTimes {  string time1DisplayTime = 1;  string time1Username = 2;  string time2DisplayTime = 3;  string time2Username = 4;  string time3DisplayTime = 5;  string time3Username = 6;  string time4DisplayTime = 7;  string time4Username = 8;  string time5DisplayTime = 9;  string time5Username = 10;  string time6DisplayTime = 11;  string time6Username = 12;  string time7DisplayTime = 13;  string time7Username = 14;  string time8DisplayTime = 15;  string time8Username = 16;  string time9DisplayTime = 17;  string time9Username = 18;  string time10DisplayTime = 19;  string time10Username = 20;  }    message GetUserTimesRequest {  int32 userID = 1;  }  message UserTimes {  string time1DisplayTime = 1;  string time2DisplayTime = 2;  string time3DisplayTime = 3;  string time4DisplayTime = 4;  string time5DisplayTime = 5;  string time6DisplayTime = 6;  string time7DisplayTime = 7;  string time8DisplayTime = 8;  string time9DisplayTime = 9;  string time10DisplayTime = 10;  }    message GetGlobalMazesGeneratedRequest {    }  message GlobalMazesGenerated {  int32 recursiveBacktrackMazesGenerated = 1;  int32 growingTreeMazesGenerated = 2;  int32 wilsonsMazesGenerated = 3;  }    message GetUserMazesGeneratedRequest {  int32 userID = 1;  }  message UserMazesGenerated {  int32 recursiveBacktrackMazesGenerated = 1;  int32 growingTreeMazesGenerated = 2;  int32 wilsonsMazesGenerated = 3;  } |

### greet.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Client";    package greet;    service Greeter {    rpc SayHello (HelloRequest) returns (HelloReply);  }    message HelloRequest {    string name = 1;  }    message HelloReply {    string message = 1;  } |

### HandleGlobalStats.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    import "google/protobuf/empty.proto";    package greet;    service GlobalStatHandler {    rpc IncrementMaze (MazeType) returns (google.protobuf.Empty);    rpc UploadTime (Time) returns (google.protobuf.Empty);  }    message MazeType {  string mazeType = 1;  }    message Time {  string username = 1;  string time = 2;  int32 timeMilliseconds = 3;  } |

### HandleUserStats.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    import "google/protobuf/empty.proto";    package greet;    service UserStatHandler {    rpc UserIncrementMaze (UserMazeType) returns (google.protobuf.Empty);    rpc UserUploadTime (UserTime) returns (google.protobuf.Empty);  }    message UserMazeType {  string mazeType = 1;  int32 userID = 2;  }    message UserTime {  string time = 1;  int32 timeMilliseconds = 2;  int32 userID = 3;  } |

### LoadMaze.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service LoaderMazes {    rpc LoadMaze (LoadRequest) returns (MazeToLoad);  }    message LoadRequest {    int32 userID = 1;    int32 mazeID = 2;  }    message MazeToLoad {    string maze = 1;    string mazeGenAlg = 2;  } |

### Login.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service LoginHandler {    rpc Login (Credentials) returns (Access);  }    message Credentials {    string username = 1;    string password = 2;  }    message Access {    bool loggedIn = 1;    int32 userID = 2;  } |

### Register.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service Registerer {    rpc Register (Account) returns (Acknowledgement);  }    message Account {    string username = 1;    string password = 2;    string salt = 3;  }    message Acknowledgement {    bool success = 1;  } |

### SaveMaze.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service Saver {    rpc SaveMaze (SaveRequest) returns (SuccessAck);  }    message SaveRequest {    string mazeName = 1;    string mazeType = 2;    string mazeJson = 3;    int32 userID = 4;  }    message SuccessAck {    bool success = 1;  } |

### SolveMaze.proto

|  |
| --- |
| syntax = "proto3";    option csharp\_namespace = "Server";    package greet;    service MazeSolver {  rpc SolveMaze (SolveRequest) returns (Path);  }    message SolveRequest {  string maze = 1;  string algorithm = 2;  string mazeGenerationAlgorithm = 3;  }    message Path {  string path = 1;  } |

# Client Code

## Classes

These are the skeletonised classes from the server that the client uses for building JSON mazes back into objects.

### Coordinate.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Client\_Mazes  {      internal class Coordinate      {          [JsonConstructor]          public Coordinate()          {            }          public Coordinate(int xPos, int yPos) {              this.xPos = xPos;              this.yPos = yPos;              visited = false;          }          public Coordinate(Tuple<int, int> pos) {              xPos = pos.Item1;              yPos = pos.Item2;              visited = false;          }            #region Properties          private int xPos;          public int Xpos {              get { return xPos; }              set { xPos = value; }          }            private int yPos;          public int Ypos {              get { return yPos; }              set { yPos = value; }          }            private bool visited;          public bool Visited {              get { return visited; }              set { visited = value; }          }          #endregion            #region Methods          public bool Equals(Coordinate target) {              return xPos == target.xPos && yPos == target.yPos;          }          #endregion      }  } |

### Globals.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;    namespace Client  {      internal static class Globals      {          public static string? g\_username = null;          public static int? g\_userID = null;            public const string g\_version = "v1.5";            public const int g\_cellWidth = 10;          public const int g\_cellHeight = 10;            public const int g\_keySize = 64;          public const int g\_iterations = 350000;      }  } |

### Growing Tree Generation.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Client\_Mazes  {      internal class GrowingTreeGeneration : Maze      {          List<Coordinate> cellsInMaze = new();            [JsonConstructor]          public GrowingTreeGeneration() {            }          public GrowingTreeGeneration(int cellWidth, int cellHeight) {              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();          }      }  } |

### Maze.cs

|  |
| --- |
| namespace Client\_Mazes  {      internal abstract class Maze {          #region Properties          private int mazeActualWidth;          public int MazeActualWidth {              get { return mazeActualWidth; }              set { mazeActualWidth = value; }          }            private int mazeActualHeight;          public int MazeActualHeight {              get { return mazeActualHeight; }              set { mazeActualHeight = value; }          }            private int mazeCellWidth;          public int MazeCellWidth {              get { return mazeCellWidth; }              set { mazeCellWidth = value; }          }            private int mazeCellHeight;          public int MazeCellHeight {              get { return mazeCellHeight; }              set { mazeCellHeight = value; }          }            private bool[,]? mazeWalls;          public bool[,]? MazeWalls {              get { return mazeWalls; }              set { mazeWalls = value; }          }            private Coordinate[,]? mazeCoordinates;          public Coordinate[,]? MazeCoordinates {              get { return mazeCoordinates; }              set { mazeCoordinates = value; }          }            private Coordinate? mazeEntranceCoordinate;          public Coordinate? MazeEntranceCoordinate {              get { return mazeEntranceCoordinate; }              set { mazeEntranceCoordinate = value; }          }            private Coordinate? mazeExitCoordinate;          public Coordinate? MazeExitCoordinate {              get { return mazeExitCoordinate; }              set { mazeExitCoordinate = value;}          }            protected Random rgen = new();          #endregion            #region Methods            public virtual void ResetVisited() {              foreach (Coordinate v in mazeCoordinates) {                  v.Visited = false;              }          }          #endregion      }  } |

### Recursive Backtrack Generation.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Client\_Mazes  {      internal class RecursiveBacktrackGeneration : Maze      {          [JsonConstructor]          public RecursiveBacktrackGeneration()          {            }            public RecursiveBacktrackGeneration(int cellWidth, int cellHeight){              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();          }      }  } |

### Wilsons Generation.cs

|  |
| --- |
| using Newtonsoft.Json;  namespace Client\_Mazes  {      internal class WilsonsGeneration : Maze      {          List<Coordinate> cellsInMaze = new();          bool exitAtBorder;            [JsonConstructor]          public WilsonsGeneration() {            }          public WilsonsGeneration(int cellWidth, int cellHeight, bool exitAtBorder) {              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();              this.exitAtBorder = exitAtBorder;          }      }  } |

### Program.cs

The main entry point for the Forms client.

|  |
| --- |
| //The client will operate over port 7178    namespace Client  {      internal static class Program      {          [STAThread]          static void Main()          {              ApplicationConfiguration.Initialize();              Application.Run(new frm\_mazeLogin());          }      }  } |

## Form Code

### MazeDisplay.cs

|  |
| --- |
| using Client\_Mazes;  using Grpc.Core;  using Grpc.Net.Client;  using Newtonsoft.Json;  using Server;  using System.Diagnostics;    namespace Client  {      public partial class frm\_mazeDisplay : Form      {          private readonly Maze maze;          private Coordinate player;          private List<Coordinate> solution;          private string mazeType;            private bool solved = false;          private bool startedManualSolve = false;          private Stopwatch sw = new();            //forces form to fully render before displaying, removing flickering.          protected override CreateParams CreateParams {              get {                  CreateParams @params = base.CreateParams;                  @params.ExStyle |= 0x2000000;                  return @params;              }          }            public frm\_mazeDisplay(string mazeToDisplay, string mazeType) {              InitializeComponent();                this.mazeType = mazeType;                switch (mazeType) {                  case "Recursive Backtrack":                      maze = JsonConvert.DeserializeObject<RecursiveBacktrackGeneration>(mazeToDisplay);                      break;                  case "Wilson's":                      maze = JsonConvert.DeserializeObject<WilsonsGeneration>(mazeToDisplay);                      break;                  case "Growing Tree":                      maze = JsonConvert.DeserializeObject<GrowingTreeGeneration>(mazeToDisplay);                      break;              }                player = new Coordinate(maze.MazeEntranceCoordinate.Xpos, maze.MazeEntranceCoordinate.Ypos);                btn\_requestSolve.Enabled = false;              cbx\_solveType.DropDownStyle = ComboBoxStyle.DropDownList;          }            private async void btn\_requestSolve\_Click(object sender, EventArgs e) {              player = null;              string mazeToSolve = JsonConvert.SerializeObject(maze);              using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new MazeSolver.MazeSolverClient(channel);              var reply = await client.SolveMazeAsync(new SolveRequest {                  Maze = mazeToSolve,                  Algorithm = cbx\_solveType.Text,                  MazeGenerationAlgorithm = mazeType              });                HandleSolveRender(reply);          }            private void HandleSolveRender(Server.Path reply) {              solution = JsonConvert.DeserializeObject<List<Coordinate>>(reply.Path\_);                solved = true;                btn\_close.Enabled = true;              btn\_requestSolve.Enabled = false;              btn\_left.Enabled = false;              btn\_right.Enabled = false;              btn\_up.Enabled = false;              btn\_down.Enabled = false;                sw = null;                tlp\_MazeDisplay.Refresh();          }            private void SetDisplaySize() {              Width = (45 + Globals.g\_cellWidth \* maze.MazeActualWidth > 510) ? 45 + Globals.g\_cellWidth \* maze.MazeActualWidth : 510;              Height = 145 + Globals.g\_cellHeight \* maze.MazeActualHeight;              pnl\_mazeContainer.Width = Globals.g\_cellWidth \* maze.MazeActualWidth + 5;              pnl\_mazeContainer.Height = Globals.g\_cellHeight \* maze.MazeActualHeight + 5;              pnl\_mazeContainer.Location = new Point(10, 80);          }            private void tlp\_MazeDisplay\_CellPaint(object sender, TableLayoutCellPaintEventArgs e) {              if (player != null && player.Equals(new Coordinate(e.Column, e.Row))) //Draw player.                  e.Graphics.FillRectangle(Brushes.Blue, e.CellBounds); //Blue                else if (maze.MazeEntranceCoordinate.Equals(new Coordinate(e.Column, e.Row))) //Draw entrance.                  e.Graphics.FillRectangle(Brushes.Red, e.CellBounds); //Red                else if (maze.MazeExitCoordinate.Equals(new Coordinate(e.Column, e.Row))) //Draw exit.                  e.Graphics.FillRectangle(Brushes.LawnGreen, e.CellBounds); //LawnGreen                else if (maze.MazeWalls[e.Row, e.Column]) //Draw wall.                  e.Graphics.FillRectangle(Brushes.Black, e.CellBounds); //Black                else if (solution != null) { //Draw solution.                  foreach (Coordinate c in solution) {                      if (c.Xpos == e.Column && c.Ypos == e.Row)                          e.Graphics.FillRectangle(Brushes.Purple, e.CellBounds); //Purple                  }              }                else //Draw path.                  e.Graphics.FillRectangle(Brushes.White, e.CellBounds); //White          }            private void frm\_mazeDisplay\_Load(object sender, EventArgs e) {              SetDisplaySize();                Text = $"MazeClient {Globals.g\_version}";                tlp\_MazeDisplay.ColumnStyles.Clear();              tlp\_MazeDisplay.RowStyles.Clear();                tlp\_MazeDisplay.RowCount = maze.MazeActualHeight;              tlp\_MazeDisplay.ColumnCount = maze.MazeActualWidth;                for (int i = 0; i < maze.MazeActualHeight; i++)                  tlp\_MazeDisplay.RowStyles.Add(new RowStyle(SizeType.Absolute, Globals.g\_cellHeight));              for (int i = 0; i < maze.MazeActualWidth; i++)                  tlp\_MazeDisplay.ColumnStyles.Add(new ColumnStyle(SizeType.Absolute, Globals.g\_cellWidth));            }          private void btn\_close\_Click(object sender, EventArgs e) {              Close();          }            private async void CheckSolved() {              if (!startedManualSolve) {                  startedManualSolve = true;                  HandleTimer();              }                if (!player.Equals(maze.MazeExitCoordinate)) return;                solved = true;              lbl\_solved.ForeColor = Color.Green;              lbl\_solved.Text = "Solved!";              btn\_close.Enabled = true;              btn\_requestSolve.Enabled = false;              btn\_left.Enabled = false;              btn\_right.Enabled = false;              btn\_up.Enabled = false;              btn\_down.Enabled = false;                using var channel = GrpcChannel.ForAddress("https://localhost:7178");                var clientGlobal = new GlobalStatHandler.GlobalStatHandlerClient(channel);              try {                  var replyGlobal = await clientGlobal.UploadTimeAsync(new Time {                      TimeMilliseconds = (int)sw.ElapsedMilliseconds,                      Time\_ = sw.Elapsed.ToString(),                      Username = Globals.g\_username                  });              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) { }                var clientUser = new UserStatHandler.UserStatHandlerClient(channel);              try {                  var replyUser = await clientUser.UserUploadTimeAsync(new UserTime {                      TimeMilliseconds = (int)sw.ElapsedMilliseconds,                      Time = sw.Elapsed.ToString(),                      UserID = (int)Globals.g\_userID                  });              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) { }          }            private void HandleTimer() {              sw.Start();                  ThreadPool.QueueUserWorkItem((state) => {                  try {                      while (!solved)                          Invoke(() => {                              if (sw != null)                                  lbl\_timer.Text = sw.Elapsed.ToString();                              else                                  lbl\_timer.Text = string.Empty;                          });                  }                  catch { }              });          }            private bool IsWall(Coordinate player, string direction) {              try {                  return direction switch {                      "Up" => !maze.MazeWalls[player.Ypos - 1, player.Xpos],                      "Down" => !maze.MazeWalls[player.Ypos + 1, player.Xpos],                      "Left" => !maze.MazeWalls[player.Ypos, player.Xpos - 1],                      "Right" => !maze.MazeWalls[player.Ypos, player.Xpos + 1],                      \_ => true                  };              }              catch { return false; }          }              private void btn\_left\_Click(object sender, EventArgs e) {              if (solved) return;                if (IsWall(player, "Left"))                  player = new Coordinate(player.Xpos - 1, player.Ypos);              tlp\_MazeDisplay.Refresh();              CheckSolved();          }            private void btn\_right\_Click(object sender, EventArgs e) {              if (solved) return;                if (IsWall(player, "Right"))                  player = new Coordinate(player.Xpos + 1, player.Ypos);              tlp\_MazeDisplay.Refresh();              CheckSolved();          }            private void btn\_up\_Click(object sender, EventArgs e) {              if (solved) return;                if (IsWall(player, "Up"))                  player = new Coordinate(player.Xpos, player.Ypos - 1);              tlp\_MazeDisplay.Refresh();              CheckSolved();          }            private void btn\_down\_Click(object sender, EventArgs e) {              if (solved) return;                if (IsWall(player, "Down"))                  player = new Coordinate(player.Xpos, player.Ypos + 1);              tlp\_MazeDisplay.Refresh();              CheckSolved();          }            #region WASD input listeners          private void frm\_mazeDisplay\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_left\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_right\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_up\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_down\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_requestSolve\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void cbx\_solveType\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_serverSave\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            private void btn\_localSave\_KeyDown(object sender, KeyEventArgs e) {              if (solved) return;              HandleKeyDown(e);          }            #endregion            private void HandleKeyDown(KeyEventArgs e) {              switch (e.KeyCode) {                  case Keys.W:                      if (IsWall(player, "Up"))                          player = new Coordinate(player.Xpos, player.Ypos - 1);                      break;                    case Keys.S:                      if (IsWall(player, "Down"))                          player = new Coordinate(player.Xpos, player.Ypos + 1);                      break;                    case Keys.A:                      if (IsWall(player, "Left"))                          player = new Coordinate(player.Xpos - 1, player.Ypos);                      break;                    case Keys.D:                      if (IsWall(player, "Right"))                          player = new Coordinate(player.Xpos + 1, player.Ypos);                      break;                    default:                      break;              }              if (new Keys[] { Keys.W, Keys.A, Keys.S, Keys.D }.Contains(e.KeyCode)) {                  tlp\_MazeDisplay.Refresh();                  CheckSolved();              }          }            private void cbx\_solveType\_SelectedIndexChanged(object sender, EventArgs e) {              if (cbx\_solveType.Text != string.Empty) {                  btn\_requestSolve.Enabled = true;              }              else {                  btn\_requestSolve.Enabled = false;              }          }            private void btn\_localSave\_Click(object sender, EventArgs e) {              Coordinate tempPlayer = player;              player = null;              tlp\_MazeDisplay.Refresh();                int width = tlp\_MazeDisplay.Size.Width;              int height = tlp\_MazeDisplay.Size.Height;                Bitmap mazeToSave = new(width, height);              tlp\_MazeDisplay.DrawToBitmap(mazeToSave, new Rectangle(0, 0, width, height));                SaveFileDialog sf = new();              sf.Filter = "JPEG Image (.jpeg)|\*.jpeg|Png Image (.png)|\*.png";              sf.ShowDialog();              var path = sf.FileName;                mazeToSave.Save(path);                player = tempPlayer;              tlp\_MazeDisplay.Refresh();          }            private async void btn\_serverSave\_Click(object sender, EventArgs e) {                if (txb\_mazeName.Text == string.Empty) {                  lbl\_error.ForeColor = Color.Red;                  lbl\_error.Text = "Maze needs a name!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(1000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return;              }                this.maze.ResetVisited();                string maze = JsonConvert.SerializeObject(this.maze);              using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new Saver.SaverClient(channel);              try {                  var reply = await client.SaveMazeAsync(new SaveRequest {                      MazeName = txb\_mazeName.Text,                      MazeType = mazeType,                      MazeJson = maze,                      UserID = (int)Globals.g\_userID                  },                  deadline: DateTime.UtcNow.AddSeconds(3));                    if (reply.Success) {                      lbl\_error.ForeColor = Color.Green;                      lbl\_error.Text = "Success!";                      ThreadPool.QueueUserWorkItem((state) => {                          Thread.Sleep(1000);                          Invoke(() => lbl\_error.Text = string.Empty);                      });                  }                  else {                      lbl\_error.ForeColor = Color.Red;                      lbl\_error.Text = "Error!";                      ThreadPool.QueueUserWorkItem((state) => {                          Thread.Sleep(1000);                          Invoke(() => lbl\_error.Text = string.Empty);                      });                  }              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_error.ForeColor = Color.Red;                  lbl\_error.Text = "Error!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(1000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });              }              catch (InvalidOperationException) { }          }      }  } |

### MazeLogin.cs

|  |
| --- |
| using Grpc.Core;  using Grpc.Net.Client;  using Server;    namespace Client  {      public partial class frm\_mazeLogin : Form      {          public frm\_mazeLogin() {              InitializeComponent();          }            private void frm\_mazeLogin\_Load(object sender, EventArgs e) {              Text = $"MazeClient {Globals.g\_version}";              lbl\_error.ForeColor = Color.Red;          }            private void llb\_register\_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e) {              Form register = new frm\_mazeRegister();              register.ShowDialog();          }            private async void btn\_login\_Click(object sender, EventArgs e) {              using var channel = GrpcChannel.ForAddress("https://localhost:7178");                var clientGreet = new Greeter.GreeterClient(channel);              try {                  var replyGreet = await clientGreet.SayHelloAsync(new HelloRequest                  { Name = Environment.MachineName },                      deadline: DateTime.UtcNow.AddSeconds(3));              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_error.Text = "Cannot connect to\nserver!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return;              }                var clientLogin = new LoginHandler.LoginHandlerClient(channel);              var replyLogin = await clientLogin.LoginAsync(new Credentials {                  Username = txb\_username.Text,                  Password = txb\_password.Text              });                if (replyLogin.LoggedIn) {                  Globals.g\_userID = replyLogin.UserID;                  Globals.g\_username = txb\_username.Text;                  Form mazeParams = new frm\_mazeParams();                  Hide();                  mazeParams.Closed += (s, args) => Close();                  mazeParams.Show();              }              else {                  lbl\_error.Text = "Username or Password\nincorrect!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });              }          }      }  } |

### MazeParameter.cs

|  |
| --- |
| using Client\_Mazes;  using Grpc.Core;  using Grpc.Net.Client;  using Newtonsoft.Json;  using Server;  using System.Windows.Forms.DataVisualization.Charting;    namespace Client  {      public partial class frm\_mazeParams : Form      {          private CancellationTokenSource cts = new CancellationTokenSource();          private bool connected = false;          private bool algorithmSelected = false;          private bool exitSelected = false;            public frm\_mazeParams() {              InitializeComponent();          }            private void MazeClient\_Load(object sender, EventArgs e) {              btn\_requestMaze.Enabled = false;                cbx\_algorithm.DropDownStyle = ComboBoxStyle.DropDownList;              cbx\_whereExit.DropDownStyle = ComboBoxStyle.DropDownList;              cbx\_statType.DropDownStyle = ComboBoxStyle.DropDownList;              cbx\_loadedMazes.DropDownStyle = ComboBoxStyle.DropDownList;                Text = $"MazeClient {Globals.g\_version} : Welcome {Globals.g\_username}";                ThreadPool.QueueUserWorkItem(async (state) => {                  while (true) {                        using var channel = GrpcChannel.ForAddress("https://localhost:7178");                      var client = new Greeter.GreeterClient(channel);                      try {                          var reply = await client.SayHelloAsync(new HelloRequest {                              Name = Environment.MachineName                          },                              deadline: DateTime.UtcNow.AddSeconds(3));                          Invoke(() => {                              lbl\_connectionError.Text = "Connected to server!";                              lbl\_connectionError.ForeColor = Color.Green;                              connected = true;                              HandleAllowSend();                          });                        }                      catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                          Invoke(() => {                              HandleServerError();                          });                      }                      catch (ObjectDisposedException) { }                        Thread.Sleep(10000);                  }              }, cts.Token);          }            private void HandleServerError() {              lbl\_connectionError.Text = "Not connected to server!";              lbl\_connectionError.ForeColor = Color.Red;              connected = false;              HandleAllowSend();          }            private void HandleAllowSend() {              if (connected &&                  algorithmSelected &&                  exitSelected) {                  btn\_requestMaze.Enabled = true;              }              else {                  btn\_requestMaze.Enabled = false;              }          }            private async void btn\_requestMaze\_Click(object sender, EventArgs e) {              btn\_requestMaze.Enabled = false;              string mazeToDisplay = await RequestMaze();              if (mazeToDisplay != string.Empty)                  ChangeForm(mazeToDisplay, cbx\_algorithm.Text);          }            private async Task<string> RequestMaze() {              using var channel = GrpcChannel.ForAddress("https://localhost:7178");                var clientBuild = new MazeBuilder.MazeBuilderClient(channel);              BuiltMaze replyBuild;              try {                  replyBuild = await clientBuild.BuildMazeAsync(new MazeRequest {                      Algorithm = cbx\_algorithm.Text,                      Width = (int)nud\_mazeWidth.Value,                      Height = (int)nud\_mazeHeight.Value,                      RemoveWalls = (int)nud\_removeWalls.Value,                      ExitLocation = cbx\_whereExit.Text                  }, deadline: DateTime.UtcNow.AddSeconds(3));              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_connectionError.Text = "Not connected to server!";                  lbl\_connectionError.ForeColor = Color.Red;                  HandleServerError();                  return string.Empty;              }                var clientStatsGlobal = new GlobalStatHandler.GlobalStatHandlerClient(channel);              try {                  var replyStatsGlobal = await clientStatsGlobal.IncrementMazeAsync(new MazeType {                      MazeType\_ = cbx\_algorithm.Text                  }, deadline: DateTime.UtcNow.AddSeconds(3));              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) { }                var clientStatsUser = new UserStatHandler.UserStatHandlerClient(channel);              try {                  var replyStatsUser = await clientStatsUser.UserIncrementMazeAsync(new UserMazeType {                      MazeType = cbx\_algorithm.Text,                      UserID = (int)Globals.g\_userID                  },                      deadline: DateTime.UtcNow.AddSeconds(3));              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) { }                return replyBuild.Maze;          }            private void ChangeForm(string maze, string algorithm) {              Form mazeDisplay = new frm\_mazeDisplay(maze, algorithm);              mazeDisplay.FormClosed += (s, args) => HandleAllowSend();              mazeDisplay.ShowDialog();              mazeDisplay.Focus();          }            private void frm\_mazeParams\_FormClosing(object sender, FormClosingEventArgs e) {              cts.Cancel();              cts.Dispose();          }            private async void btn\_getMazes\_Click(object sender, EventArgs e) {              cbx\_loadedMazes.Items.Clear();                using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new GetterMazes.GetterMazesClient(channel);              try {                  var reply = await client.GetMazesAsync(new Request {                      UserID = (int)Globals.g\_userID                  }, deadline: DateTime.UtcNow.AddSeconds(3));                  var mazes = JsonConvert.DeserializeObject<List<(int mazeID,string mazeName)>>(reply.Mazes);                  foreach (var maze in mazes) {                      cbx\_loadedMazes.Items.Add($"{maze.mazeID}: {maze.mazeName}");                  }                  lbl\_loadError.Text = $"Found {mazes.Count} mazes!";              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_loadError.Text = "Error fetching\nmazes!";              }          }            private async void btn\_loadMaze\_Click(object sender, EventArgs e) {              if (cbx\_loadedMazes.Text == string.Empty) return;                using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new LoaderMazes.LoaderMazesClient(channel);              try {                  var reply = await client.LoadMazeAsync(new LoadRequest {                      UserID = (int)Globals.g\_userID,                      MazeID = Convert.ToInt32(cbx\_loadedMazes.Text.Split(':')[0])                  },                      deadline: DateTime.UtcNow.AddSeconds(3));                    ChangeForm(reply.Maze, reply.MazeGenAlg);                }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_loadError.Text = "Error loading\nmazes!";              }          }            private async void btn\_deleteMaze\_Click(object sender, EventArgs e) {              if (cbx\_loadedMazes.Text == string.Empty) return;                using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new DeleterMazes.DeleterMazesClient(channel);              try {                  var reply = await client.DeleteMazeAsync(new DeleteRequest {                      UserID = (int)Globals.g\_userID,                      MazeID = Convert.ToInt32(cbx\_loadedMazes.Text.Split(':')[0])                  },                      deadline: DateTime.UtcNow.AddSeconds(3));                  if (reply.Success) {                      lbl\_loadError.Text = "Deleted maze\nsuccessfully!";                      cbx\_loadedMazes.Items.RemoveAt(cbx\_loadedMazes.SelectedIndex);                      cbx\_loadedMazes.SelectedIndex = -1;                  }                  else {                      lbl\_loadError.Text = "Error Deleting\nmaze!";                  }                }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_loadError.Text = "Error Deleting\nmaze!";              }          }            private async void btn\_displayStats\_Click(object sender, EventArgs e) {              if (cbx\_statType.Text == string.Empty) return;                using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new StatsGetter.StatsGetterClient(channel);                if (cbx\_statType.Text == "Mazes Generated" && chbx\_globalStats.Checked) {                  try {                      var reply = await client.GetGlobalMazesGeneratedAsync(                          new GetGlobalMazesGeneratedRequest { },                          deadline: DateTime.UtcNow.AddSeconds(3));                        Chart chrt\_mazesGenerated = HandleMazesGeneratedStatsView();                      Series series = new("Maze Types Generated");                      series.ChartType = SeriesChartType.Pie;                        string[] segmentNames = {                          "Recursive Backtrack",                          "Growing tree",                          "Wilson's"                      };                      double[] segmentValues = {                          reply.RecursiveBacktrackMazesGenerated,                          reply.GrowingTreeMazesGenerated,                          reply.WilsonsMazesGenerated                      };                      series.Points.DataBindXY(segmentNames, segmentValues);                        chrt\_mazesGenerated.Series.Add(series);                  }                  catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                      HandleServerError();                  }              }              else if (cbx\_statType.Text == "Mazes Generated" && !chbx\_globalStats.Checked) {                  try {                      var reply = await client.GetUserMazesGeneratedAsync(new GetUserMazesGeneratedRequest {                          UserID = (int)Globals.g\_userID                      },                          deadline: DateTime.UtcNow.AddSeconds(3));                        Chart chrt\_mazesGenerated = HandleMazesGeneratedStatsView();                      Series series = new("Maze Types Generated");                      series.ChartType = SeriesChartType.Pie;                        string[] segmentNames = {                          "Recursive Backtrack",                          "Growing Tree",                          "Wilson's"                      };                      double[] segmentValues = {                          reply.RecursiveBacktrackMazesGenerated,                          reply.GrowingTreeMazesGenerated,                          reply.WilsonsMazesGenerated                      };                      series.Points.DataBindXY(segmentNames, segmentValues);                        chrt\_mazesGenerated.Series.Add(series);                  }                  catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                      HandleServerError();                  }              }              else if (cbx\_statType.Text == "Best Times" && chbx\_globalStats.Checked) {                  try {                      var reply = await client.GetGlobalTimesAsync(new GetGlobalTimesRequest { },                          deadline: DateTime.UtcNow.AddSeconds(3));                        RichTextBox rtb\_times = HandleTimeStatsView();                      rtb\_times.Text += "Global Best Times\n";                      rtb\_times.Text += $"1st:  {reply.Time1Username} : {reply.Time1DisplayTime}\n" +                          $"2nd:  {reply.Time2Username} : {reply.Time2DisplayTime}\n" +                          $"3rd:  {reply.Time3Username} : {reply.Time3DisplayTime}\n" +                          $"4th:  {reply.Time4Username} : {reply.Time4DisplayTime}\n" +                          $"5th:  {reply.Time5Username} : {reply.Time5DisplayTime}\n" +                          $"6th:  {reply.Time6Username} : {reply.Time6DisplayTime}\n" +                          $"7th:  {reply.Time7Username} : {reply.Time7DisplayTime}\n" +                          $"8th:  {reply.Time8Username} : {reply.Time8DisplayTime}\n" +                          $"9th:  {reply.Time9Username} : {reply.Time9DisplayTime}\n" +                          $"10th: {reply.Time10Username} : {reply.Time10DisplayTime}";                  }                  catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                      HandleServerError();                  }              }              else if (cbx\_statType.Text == "Best Times" && !chbx\_globalStats.Checked) {                  try {                      var reply = await client.GetUserTimesAsync(new GetUserTimesRequest {                          UserID = (int)Globals.g\_userID                      },                          deadline: DateTime.UtcNow.AddSeconds(3));                        RichTextBox rtb\_times = HandleTimeStatsView();                      rtb\_times.Font = new Font("Calibri", 20, FontStyle.Bold);                      rtb\_times.Text += "Your Best Times\n";                      rtb\_times.Font = DefaultFont;                      rtb\_times.Text += $"1st:  {reply.Time1DisplayTime}\n" +                          $"2nd:  {reply.Time2DisplayTime}\n" +                          $"3rd:  {reply.Time3DisplayTime}\n" +                          $"4th:  {reply.Time4DisplayTime}\n" +                          $"5th:  {reply.Time5DisplayTime}\n" +                          $"6th:  {reply.Time6DisplayTime}\n" +                          $"7th:  {reply.Time7DisplayTime}\n" +                          $"8th:  {reply.Time8DisplayTime}\n" +                          $"9th:  {reply.Time9DisplayTime}\n" +                          $"10th: {reply.Time10DisplayTime}";                  }                  catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                      HandleServerError();                  }              }          }            private RichTextBox HandleTimeStatsView() {              RichTextBox rtb\_times;              pnl\_graph.Controls.Clear();              pnl\_graph.Controls.Add(rtb\_times = new());                rtb\_times.Dock = DockStyle.Fill;                return rtb\_times;          }            private Chart HandleMazesGeneratedStatsView() {              Chart chrt\_generatedStats;              pnl\_graph.Controls.Clear();              pnl\_graph.Controls.Add(chrt\_generatedStats = new());                chrt\_generatedStats.Dock = DockStyle.Fill;              chrt\_generatedStats.ChartAreas.Add(new ChartArea("MazeChartArea"));              chrt\_generatedStats.ChartAreas["MazeChartArea"].Area3DStyle.Enable3D = true;                  return chrt\_generatedStats;          }            private void cbx\_algorithm\_SelectedIndexChanged(object sender, EventArgs e) {              algorithmSelected = true;              HandleAllowSend();          }            private void cbx\_whereExit\_SelectedIndexChanged(object sender, EventArgs e) {              exitSelected = true;              HandleAllowSend();          }      }  } |

### MazeRegister.cs

|  |
| --- |
| using Grpc.Net.Client;  using System.Text.RegularExpressions;  using System.Security.Cryptography;  using System.Text;  using Server;  using Grpc.Core;    namespace Client  {      public partial class frm\_mazeRegister : Form      {          HashAlgorithmName hashAlgorithm = HashAlgorithmName.SHA512;            public frm\_mazeRegister() {              InitializeComponent();          }          private void frm\_mazeRegister\_Load(object sender, EventArgs e) {              Text = $"MazeClient {Globals.g\_version}";              lbl\_error.ForeColor = Color.Red;          }            private async void btn\_register\_Click(object sender, EventArgs e) {              using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var clientGreet = new Greeter.GreeterClient(channel);              try {                  var replyGreet = await clientGreet.SayHelloAsync(new HelloRequest {                      Name = Environment.MachineName                  },                      deadline: DateTime.UtcNow.AddSeconds(3));              }              catch (RpcException ex) when (ex.StatusCode == StatusCode.DeadlineExceeded) {                  lbl\_error.Text = "Cannot connect to server!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return;              }                if (await CheckCrendentials()) {                    var password = HashPasword(txb\_password.Text, out var salt);                    var clientRegister = new Registerer.RegistererClient(channel);                  var replyRegister = await clientRegister.RegisterAsync(new Account {                      Username = txb\_username.Text,                      Password = password,                      Salt = Convert.ToHexString(salt),                  });                    if (replyRegister.Success) {                      lbl\_error.ForeColor = Color.Green;                      lbl\_error.Text = "Registered!";                      Task.Delay(1000).Wait();                      Close();                  }                  else {                      lbl\_error.ForeColor = Color.Red;                      lbl\_error.Text = "Failed to register!";                      Task.Delay(1000).Wait();                      lbl\_error.Text = string.Empty;                  }              }          }              private async Task<bool> CheckCrendentials() {              using var channel = GrpcChannel.ForAddress("https://localhost:7178");              var client = new CheckerIfUserExists.CheckerIfUserExistsClient(channel);              var reply = await client.CheckUserAsync(new Query {                  Username = txb\_username.Text              });                if (reply.UserExists) {                  lbl\_error.Text = "Username already taken!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return false;              }              if (txb\_password.Text.Length < 7) {                  lbl\_error.Text = "Password must be at least\n 7 characters!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return false;              }              if (!Regex.IsMatch(txb\_password.Text, @"[!-**\/**:-@[-`{-~]")) {                  lbl\_error.Text = "Password must contain at\n least 1 special character!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return false;              }              if (txb\_password.Text != txb\_confirm.Text) {                  lbl\_error.Text = "Passwords do not match!";                  ThreadPool.QueueUserWorkItem((state) => {                      Thread.Sleep(3000);                      Invoke(() => lbl\_error.Text = string.Empty);                  });                  return false;              }                return true;          }            // SOURCE: https://code-maze.com/csharp-hashing-salting-passwords-best-practices/          string HashPasword(string password, out byte[] salt) {              salt = RandomNumberGenerator.GetBytes(Globals.g\_keySize);              var hash = Rfc2898DeriveBytes.Pbkdf2(                  Encoding.UTF8.GetBytes(password),                  salt,                  Globals.g\_iterations,                  hashAlgorithm,                  Globals.g\_keySize);              return Convert.ToHexString(hash);          }      }  } |

# Server Code

## Service Definitions

### CheckerIfUserExistsService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class CheckerIfUserExistsService : CheckerIfUserExists.CheckerIfUserExistsBase      {          public override Task<Exists> CheckUser(Query request, ServerCallContext context) {              using SQLiteConnection conn = new(                  "Data Source=mazeData.db; " +                  "Version=3; " +                  "New=True; " +                  "Compress=True; ");              conn.Open();                using SQLiteCommand cmd = conn.CreateCommand();              cmd.CommandText = "SELECT COUNT(\*) FROM User WHERE Username = @username";              cmd.Parameters.AddWithValue("@username", request.Username);                int rowCount = Convert.ToInt32(cmd.ExecuteScalar());                if (rowCount > 0) {                  return Task.FromResult(new Exists { UserExists = true });              }              else {                  return Task.FromResult(new Exists { UserExists = false });              }          }      }  } |

### DeleteMazeService

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class DeleteMazeService : DeleterMazes.DeleterMazesBase      {          public override Task<SuccessAcknowledge> DeleteMaze(DeleteRequest request, ServerCallContext context) {              try {                  using (SQLiteConnection conn = new(                      "Data Source= mazeData.db; " +                      "Version = 3; " +                      "New = True; " +                      "Compress = True; ")) {                      conn.Open();                      using SQLiteCommand cmd = conn.CreateCommand();                        cmd.CommandText = @"DELETE FROM Mazes                                          WHERE @MazeID = MazeID                                          AND @UserID = UID";                      cmd.Parameters.AddWithValue("@MazeID", request.MazeID);                      cmd.Parameters.AddWithValue("@UserID", request.UserID);                      cmd.ExecuteNonQuery();                  }                    return Task.FromResult(new SuccessAcknowledge { Success = true });                }              catch (Exception) {                  return Task.FromResult(new SuccessAcknowledge { Success = false });              }          }      }  } |

### GetMazesService.cs

|  |
| --- |
| using Grpc.Core;  using Newtonsoft.Json;  using System.Data.SQLite;    namespace Server.Services  {      public class GetMazesService : GetterMazes.GetterMazesBase      {          public override Task<MazesList> GetMazes(Request request, ServerCallContext context) {              List<(int mazeID, string mazeName)> mazes = new();                using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                    using SQLiteCommand cmd = conn.CreateCommand();                  cmd.CommandText = "SELECT MazeID, MazeName FROM Mazes WHERE @UID = UID";                  cmd.Parameters.AddWithValue("@UID", request.UserID);                  using SQLiteDataReader reader = cmd.ExecuteReader();                  while (reader.Read()) {                      mazes.Add((reader.GetInt32(0), reader.GetString(1)));                  }              }                return Task.FromResult(new MazesList {                  Mazes = JsonConvert.SerializeObject(mazes)              });          }      }  } |

### GetStatsService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class GetStatsService : StatsGetter.StatsGetterBase      {          public override Task<GlobalMazesGenerated> GetGlobalMazesGenerated(GetGlobalMazesGeneratedRequest request, ServerCallContext context) {              int recursiveBacktrackMazesGenerated = 0;              int growingTreeMazesGenerated = 0;              int wilsonsMazesGenerated = 0;                using SQLiteConnection conn = new SQLiteConnection(                  "Data Source=mazeData.db; " +                  "Version=3; " +                  "New=True; " +                  "Compress=True; ");              conn.Open();              using SQLiteCommand cmd = conn.CreateCommand();              cmd.CommandText = @"SELECT RecursiveBacktrackMazesGenerated,                                         GrowingTreeMazesGenerated,                                         WilsonsMazesGenerated                                  FROM GlobalStats";              using SQLiteDataReader reader = cmd.ExecuteReader();              while (reader.Read()) {                  recursiveBacktrackMazesGenerated = reader.GetInt32(0);                  growingTreeMazesGenerated = reader.GetInt32(1);                  wilsonsMazesGenerated = reader.GetInt32(2);              }              conn.Close();                return Task.FromResult(new GlobalMazesGenerated {                  RecursiveBacktrackMazesGenerated = recursiveBacktrackMazesGenerated,                  GrowingTreeMazesGenerated = growingTreeMazesGenerated,                  WilsonsMazesGenerated = wilsonsMazesGenerated              });          }            public override Task<GlobalTimes> GetGlobalTimes(GetGlobalTimesRequest request, ServerCallContext context) {                string time1DisplayTime = string.Empty;              string time2DisplayTime = string.Empty;              string time3DisplayTime = string.Empty;              string time4DisplayTime = string.Empty;              string time5DisplayTime = string.Empty;              string time6DisplayTime = string.Empty;              string time7DisplayTime = string.Empty;              string time8DisplayTime = string.Empty;              string time9DisplayTime = string.Empty;              string time10DisplayTime = string.Empty;              string time1Username = string.Empty;              string time2Username = string.Empty;              string time3Username = string.Empty;              string time4Username = string.Empty;              string time5Username = string.Empty;              string time6Username = string.Empty;              string time7Username = string.Empty;              string time8Username = string.Empty;              string time9Username = string.Empty;              string time10Username = string.Empty;                using SQLiteConnection conn = new("" +                  "Data Source=mazeData.db; " +                  "Version=3; " +                  "New=True; " +                  "Compress=True; ");              conn.Open();              using SQLiteCommand cmd = conn.CreateCommand();              cmd.CommandText = @"SELECT Time1Display,                                         Time2Display,                                         Time3Display,                                         Time4Display,                                         Time5Display,                                         Time6Display,                                         Time7Display,                                         Time8Display,                                         Time9Display,                                         Time10Display,                                         Time1Name,                                         Time2Name,                                         Time3Name,                                         Time4Name,                                         Time6Name,                                         Time7Name,                                         Time8Name,                                         Time9Name,                                         Time10Name                                  FROM GlobalStats";              using SQLiteDataReader reader = cmd.ExecuteReader();                while (reader.Read()) {                  time1DisplayTime = reader.GetString(0);                  time2DisplayTime = reader.GetString(1);                  time3DisplayTime = reader.GetString(2);                  time4DisplayTime = reader.GetString(3);                  time5DisplayTime = reader.GetString(4);                  time6DisplayTime = reader.GetString(5);                  time7DisplayTime = reader.GetString(6);                  time8DisplayTime = reader.GetString(7);                  time9DisplayTime = reader.GetString(8);                  time10DisplayTime = reader.GetString(9);                  time1Username = reader.GetString(10);                  time2Username = reader.GetString(11);                  time3Username = reader.GetString(12);                  time4Username = reader.GetString(13);                  time5Username = reader.GetString(14);                  time6Username = reader.GetString(15);                  time7Username = reader.GetString(16);                  time8Username = reader.GetString(17);                  time9Username = reader.GetString(18);                  time10Username = reader.GetString(19);              }                return Task.FromResult(new GlobalTimes {                  Time1DisplayTime = time1DisplayTime,                  Time2DisplayTime = time2DisplayTime,                  Time3DisplayTime = time3DisplayTime,                  Time4DisplayTime = time4DisplayTime,                  Time5DisplayTime = time5DisplayTime,                  Time6DisplayTime = time6DisplayTime,                  Time7DisplayTime = time7DisplayTime,                  Time8DisplayTime = time8DisplayTime,                  Time9DisplayTime = time9DisplayTime,                  Time10DisplayTime = time10DisplayTime,                  Time1Username = time1Username,                  Time2Username = time2Username,                  Time3Username = time3Username,                  Time4Username = time4Username,                  Time5Username = time5Username,                  Time6Username = time6Username,                  Time7Username = time7Username,                  Time8Username = time8Username,                  Time9Username = time9Username,                  Time10Username = time10Username              });          }            public override Task<UserMazesGenerated> GetUserMazesGenerated(GetUserMazesGeneratedRequest request, ServerCallContext context) {              int recursiveBacktrackMazesGenerated = 0;              int growingTreeMazesGenerated = 0;              int wilsonsMazesGenerated = 0;                using SQLiteConnection conn = new(                  "Data Source=mazeData.db; " +                  "Version=3; " +                  "New=True; " +                  "Compress=True; ");              conn.Open();              using SQLiteCommand cmd = conn.CreateCommand();              cmd.CommandText = @$"SELECT RecursiveBacktrackMazesGenerated,                                          GrowingTreeMazesGenerated,                                          WilsonsMazesGenerated                                  FROM UserStats                                  WHERE UID = {request.UserID}";              using SQLiteDataReader reader = cmd.ExecuteReader();              while (reader.Read()) {                  recursiveBacktrackMazesGenerated = reader.GetInt32(0);                  growingTreeMazesGenerated = reader.GetInt32(1);                  wilsonsMazesGenerated = reader.GetInt32(2);              }              conn.Close();                return Task.FromResult(new UserMazesGenerated {                  RecursiveBacktrackMazesGenerated = recursiveBacktrackMazesGenerated,                  GrowingTreeMazesGenerated = growingTreeMazesGenerated,                  WilsonsMazesGenerated = wilsonsMazesGenerated              });          }            public override Task<UserTimes> GetUserTimes(GetUserTimesRequest request, ServerCallContext context) {                string time1DisplayTime = string.Empty;              string time2DisplayTime = string.Empty;              string time3DisplayTime = string.Empty;              string time4DisplayTime = string.Empty;              string time5DisplayTime = string.Empty;              string time6DisplayTime = string.Empty;              string time7DisplayTime = string.Empty;              string time8DisplayTime = string.Empty;              string time9DisplayTime = string.Empty;              string time10DisplayTime = string.Empty;                using SQLiteConnection conn = new("" +                  "Data Source=mazeData.db; " +                  "Version=3; " +                  "New=True; " +                  "Compress=True; ");              conn.Open();              using SQLiteCommand cmd = conn.CreateCommand();              cmd.CommandText = @$"SELECT Time1Display,                                          Time2Display,                                          Time3Display,                                          Time4Display,                                          Time5Display,                                          Time6Display,                                          Time7Display,                                          Time8Display,                                          Time9Display,                                          Time10Display                                  FROM UserStats                                  WHERE UID = {request.UserID}";              using SQLiteDataReader reader = cmd.ExecuteReader();                while (reader.Read()) {                  time1DisplayTime = reader.GetString(0);                  time2DisplayTime = reader.GetString(1);                  time3DisplayTime = reader.GetString(2);                  time4DisplayTime = reader.GetString(3);                  time5DisplayTime = reader.GetString(4);                  time6DisplayTime = reader.GetString(5);                  time7DisplayTime = reader.GetString(6);                  time8DisplayTime = reader.GetString(7);                  time9DisplayTime = reader.GetString(8);                  time10DisplayTime = reader.GetString(9);              }                return Task.FromResult(new UserTimes {                  Time1DisplayTime = time1DisplayTime,                  Time2DisplayTime = time2DisplayTime,                  Time3DisplayTime = time3DisplayTime,                  Time4DisplayTime = time4DisplayTime,                  Time5DisplayTime = time5DisplayTime,                  Time6DisplayTime = time6DisplayTime,                  Time7DisplayTime = time7DisplayTime,                  Time8DisplayTime = time8DisplayTime,                  Time9DisplayTime = time9DisplayTime,                  Time10DisplayTime = time10DisplayTime              });          }      }  } |

### GlobalStatHandlerService.cs

|  |
| --- |
| using Google.Protobuf.WellKnownTypes;  using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class GlobalStatHandlerService : GlobalStatHandler.GlobalStatHandlerBase      {          public override Task<Empty> IncrementMaze(MazeType request, ServerCallContext context) {              using (SQLiteConnection conn = new("" +                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                  using SQLiteCommand cmd = conn.CreateCommand();                  switch (request.MazeType\_) {                      case "Recursive Backtrack":                          cmd.CommandText = @"UPDATE GlobalStats                                              SET RecursiveBacktrackMazesGenerated = RecursiveBacktrackMazesGenerated + 1";                          cmd.ExecuteNonQuery();                          break;                      case "Wilson's":                          cmd.CommandText = @"UPDATE GlobalStats                                              SET WilsonsMazesGenerated = WilsonsMazesGenerated + 1";                          cmd.ExecuteNonQuery();                          break;                      case "Growing Tree":                          cmd.CommandText = @"UPDATE GlobalStats                                              SET GrowingTreeMazesGenerated = GrowingTreeMazesGenerated + 1";                          cmd.ExecuteNonQuery();                          break;                  }              }              return Task.FromResult(new Empty());            }            public override Task<Empty> UploadTime(Time request, ServerCallContext context) {              List<int> milliseconds = new();              List<string> displayTimes = new();              List<string> usernames = new();                using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                  using SQLiteCommand cmd = conn.CreateCommand();                    cmd.CommandText = @"SELECT Time1Milliseconds,                                             Time2Milliseconds,                                             Time3Milliseconds,                                             Time4Milliseconds,                                             Time5Milliseconds,                                             Time6Milliseconds,                                             Time7Milliseconds,                                             Time8Milliseconds,                                             Time9Milliseconds,                                             Time10Milliseconds                                      FROM GlobalStats";                  using SQLiteDataReader readerMilliseconds = cmd.ExecuteReader();                  while (readerMilliseconds.Read()) {                      for (int i = 0; i < 10; i++) {                          milliseconds.Add(readerMilliseconds.GetInt32(i));                      }                  }                  readerMilliseconds.Close();                    cmd.CommandText = @"SELECT Time1Display,                                             Time2Display,                                             Time3Display,                                             Time4Display,                                             Time5Display,                                             Time6Display,                                             Time7Display,                                             Time8Display,                                             Time9Display,                                             Time10Display                                      FROM GlobalStats";                  using SQLiteDataReader readerDisplay = cmd.ExecuteReader();                  while (readerDisplay.Read()) {                      for (int i = 0; i < 10; i++) {                          displayTimes.Add(readerDisplay.GetString(i));                      }                  }                  readerDisplay.Close();                    cmd.CommandText = @"SELECT Time1Name,                                             Time2Name,                                             Time3Name,                                             Time4Name,                                             Time5Name,                                             Time6Name,                                             Time7Name,                                             Time8Name,                                             Time9Name,                                             Time10Name                                      FROM GlobalStats";                  using SQLiteDataReader readerUsername = cmd.ExecuteReader();                  while (readerUsername.Read()) {                      for (int i = 0; i < 10; i++) {                          usernames.Add(readerUsername.GetString(i));                      }                  }                  readerUsername.Close();                    int place = -1;                  for (int i = 0; i < 10; i++) {                      if (request.TimeMilliseconds < milliseconds[i] || milliseconds[i] == -1) {                          milliseconds.Insert(i, request.TimeMilliseconds);                          place = i;                          if (milliseconds.Count > 10) milliseconds.RemoveAt(milliseconds.Count - 1);                          break;                      }                  }                    if (place == -1) { return Task.FromResult(new Empty()); }                    displayTimes.Insert(place, request.Time\_);                  if (displayTimes.Count > 10) displayTimes.RemoveAt(displayTimes.Count - 1);                    usernames.Insert(place, request.Username);                  if (usernames.Count > 10) usernames.RemoveAt(usernames.Count - 1);                        for (int i = 0; i < 10; i++) {                      cmd.CommandText = $@"UPDATE GlobalStats                                           SET Time{i + 1}Milliseconds = {milliseconds[i]}";                      cmd.ExecuteNonQuery();                      cmd.CommandText = $@"UPDATE GlobalStats                                           SET Time{i + 1}Display = '{displayTimes[i]}'";                      cmd.ExecuteNonQuery();                      cmd.CommandText = $@"UPDATE GlobalStats                                           SET Time{i + 1}Name = '{usernames[i]}'";                      cmd.ExecuteNonQuery();                  }              }              return Task.FromResult(new Empty());          }      }  } |

### GreeterService.cs

|  |
| --- |
| using Grpc.Core;  using Server;    namespace Server.Services  {      public class GreeterService : Greeter.GreeterBase      {          private readonly ILogger<GreeterService> \_logger;          public GreeterService(ILogger<GreeterService> logger)          {              \_logger = logger;          }            public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)          {              return Task.FromResult(new HelloReply              {                  Message = "Hello " + request.Name              });          }      }  } |

### HandleUserStatsService.cs

|  |
| --- |
| using Google.Protobuf.WellKnownTypes;  using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class HandleUserStatsService : UserStatHandler.UserStatHandlerBase      {          public override Task<Empty> UserIncrementMaze(UserMazeType request, ServerCallContext context) {              using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                  using SQLiteCommand cmd = conn.CreateCommand();                  switch (request.MazeType) {                      case "Recursive Backtrack":                          cmd.CommandText = $@"UPDATE UserStats                                               SET RecursiveBacktrackMazesGenerated = RecursiveBacktrackMazesGenerated + 1                                               WHERE UID = {request.UserID}";                          cmd.ExecuteNonQuery();                          break;                      case "Wilson's":                          cmd.CommandText = $@"UPDATE UserStats                                               SET WilsonsMazesGenerated = WilsonsMazesGenerated + 1                                               WHERE UID = {request.UserID}";                          cmd.ExecuteNonQuery();                          break;                      case "Growing Tree":                          cmd.CommandText = $@"UPDATE UserStats                                               SET GrowingTreeMazesGenerated = GrowingTreeMazesGenerated + 1                                               WHERE UID = {request.UserID}";                          cmd.ExecuteNonQuery();                          break;                  }              }              return Task.FromResult(new Empty());          }            public override Task<Empty> UserUploadTime(UserTime request, ServerCallContext context) {              List<int> milliseconds = new();              List<string> displayTimes = new();                using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                  using SQLiteCommand cmd = conn.CreateCommand();                    cmd.CommandText = @$"SELECT Time1Milliseconds,                                              Time2Milliseconds,                                              Time3Milliseconds,                                              Time4Milliseconds,                                              Time5Milliseconds,                                              Time6Milliseconds,                                              Time7Milliseconds,                                              Time8Milliseconds,                                              Time9Milliseconds,                                              Time10Milliseconds                                      FROM UserStats                                      WHERE UID = {request.UserID}";                  using SQLiteDataReader readerMilliseconds = cmd.ExecuteReader();                  while (readerMilliseconds.Read()) {                      for (int i = 0; i < 10; i++) {                          milliseconds.Add(readerMilliseconds.GetInt32(i));                      }                  }                  readerMilliseconds.Close();                    cmd.CommandText = @$"SELECT Time1Display,                                              Time2Display,                                              Time3Display,                                              Time4Display,                                              Time5Display,                                              Time6Display,                                              Time7Display,                                              Time8Display,                                              Time9Display,                                              Time10Display                                      FROM UserStats                                      WHERE UID = {request.UserID}";                  using SQLiteDataReader readerDisplay = cmd.ExecuteReader();                  while (readerDisplay.Read()) {                      for (int i = 0; i < 10; i++) {                          displayTimes.Add(readerDisplay.GetString(i));                      }                  }                  readerDisplay.Close();                        int place = -1;                  for (int i = 0; i < 10; i++) {                      if (request.TimeMilliseconds < milliseconds[i] || milliseconds[i] == -1) {                          milliseconds.Insert(i, request.TimeMilliseconds);                          place = i;                          if (milliseconds.Count > 10) milliseconds.RemoveAt(milliseconds.Count - 1);                          break;                      }                  }                    if (place == -1) { return Task.FromResult(new Empty()); }                    displayTimes.Insert(place, request.Time);                  if (displayTimes.Count > 10) displayTimes.RemoveAt(displayTimes.Count - 1);                          for (int i = 0; i < 10; i++) {                      cmd.CommandText = $@"UPDATE UserStats                                           SET Time{i + 1}Milliseconds = {milliseconds[i]}                                           WHERE UID = {request.UserID}";                      cmd.ExecuteNonQuery();                      cmd.CommandText = $@"UPDATE UserStats                                           SET Time{i + 1}Display = '{displayTimes[i]}'                                           WHERE UID = {request.UserID}";                      cmd.ExecuteNonQuery();                  }              }              return Task.FromResult(new Empty());          }      }  } |

### LoadMazeService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class LoadMazeService : LoaderMazes.LoaderMazesBase      {          public override Task<MazeToLoad> LoadMaze(LoadRequest request, ServerCallContext context) {              string maze;              string mazeGenAlg;                using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3;" +                  " New = True; " +                  "Compress = True; ")) {                  conn.Open();                    using SQLiteCommand cmd = conn.CreateCommand();                  cmd.CommandText = @"SELECT MazeObject, MazeGenAlg                                      FROM Mazes                                      WHERE @MazeID = MazeID                                      AND @UserID = UID";                  cmd.Parameters.AddWithValue("@MazeID", request.MazeID);                  cmd.Parameters.AddWithValue("@UserID", request.UserID);                    using SQLiteDataReader reader = cmd.ExecuteReader();                  if (reader.Read()) {                      maze = reader.GetString(0);                      mazeGenAlg = reader.GetString(1);                  }                  else {                      maze = string.Empty;                      mazeGenAlg = string.Empty;                  }              }                return Task.FromResult(new MazeToLoad {                  Maze = maze, MazeGenAlg = mazeGenAlg              });          }      }  } |

### LoginService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;  using System.Security.Cryptography;  using System.Text;    namespace Server.Services  {      public class LoginService : LoginHandler.LoginHandlerBase      {          public const int keySize = 64;          public const int iterations = 350000;          HashAlgorithmName hashAlgorithm = HashAlgorithmName.SHA512;          public override Task<Access> Login(Credentials request, ServerCallContext context) {                string password;              string salt;              int userID;                using (SQLiteConnection conn = new(                  "Data Source= mazeData.db; " +                  "Version = 3; " +                  "New = True; " +                  "Compress = True; ")) {                  conn.Open();                    using SQLiteCommand cmd = conn.CreateCommand();                  cmd.CommandText = $@"SELECT UID, Password, Salt                                       FROM User                                       WHERE Username = @username";                  cmd.Parameters.AddWithValue("@username", request.Username);                    using SQLiteDataReader reader = cmd.ExecuteReader();                  if (reader.Read()) {                      userID = reader.GetInt32(0);                      password = reader.GetString(1);                      salt = reader.GetString(2);                  }                  else {                      return Task.FromResult(new Access { LoggedIn = false });                  }              }                return Task.FromResult(new Access {                  LoggedIn = VerifyPassword(request.Password, password, Convert.FromHexString(salt)),                  UserID = userID              });          }            // SOURCE: https://code-maze.com/csharp-hashing-salting-passwords-best-practices/          bool VerifyPassword(string password, string hash, byte[] salt) {              var hashToCompare = Rfc2898DeriveBytes.Pbkdf2(                  password,                  salt,                  iterations,                  hashAlgorithm,                  keySize);              return CryptographicOperations.FixedTimeEquals(hashToCompare, Convert.FromHexString(hash));          }      }  } |

### MazeBuilderService.cs

|  |
| --- |
| using Grpc.Core;  using Newtonsoft.Json;  using System.Text;    namespace Server.Services  {      public class MazeBuilderService : MazeBuilder.MazeBuilderBase      {          public override Task<BuiltMaze> BuildMaze(MazeRequest request, ServerCallContext context) {                Maze maze = null;                switch (request.Algorithm) {                  case "Recursive Backtrack":                      maze = new RecursiveBacktrackGeneration(                          (int)request.Width,                          (int)request.Height);                      maze.InitMaze();                      maze.BuildMaze(maze.MazeCoordinates[1, 1]);                      maze.RemoveWalls((int)request.RemoveWalls);                      maze.CreateEntranceExit(request.ExitLocation == "Border");                      break;                  case "Wilson's":                      maze = new WilsonsGeneration(                          (int)request.Width,                          (int)request.Height,                          request.ExitLocation == "Border");                      //algorithm requires exitlocation initialised before other algorithms                      //so it knows where the exit will be for the intial loop erased walk.                      maze.InitMaze();                      maze.CreateEntranceExit(request.ExitLocation == "Border");                      maze.BuildMaze(maze.MazeCoordinates[1, 1]);                      maze.RemoveWalls((int)request.RemoveWalls);                      break;                  case "Growing Tree":                      maze = new GrowingTreeGeneration(                          (int)request.Width,                          (int)request.Height);                      maze.InitMaze();                      maze.BuildMaze(null); //startCoordinate unnecessary for this algorithm                      maze.RemoveWalls((int)request.RemoveWalls);                      maze.CreateEntranceExit(request.ExitLocation == "Border");                      break;                  }                string jsonMaze = JsonConvert.SerializeObject(maze);                return Task.FromResult(new BuiltMaze { Maze = jsonMaze });          }      }  } |

### MazeSolverService.cs

|  |
| --- |
| using Grpc.Core;  using Newtonsoft.Json;    namespace Server.Services  {      public class MazeSolverService : MazeSolver.MazeSolverBase      {          public override Task<Path> SolveMaze(SolveRequest request, ServerCallContext context)          {              SolvingAlgorithm solver = null;              Maze maze = null;                switch (request.MazeGenerationAlgorithm) {                  case "Recursive Backtrack":                      maze = JsonConvert.DeserializeObject<RecursiveBacktrackGeneration>(request.Maze);                      break;                  case "Wilson's":                      maze = JsonConvert.DeserializeObject<WilsonsGeneration>(request.Maze);                      break;                  case "Growing Tree":                      maze = JsonConvert.DeserializeObject<GrowingTreeGeneration>(request.Maze);                      break;              }                  switch (request.Algorithm) {                  case "Depth First":                      solver = new DepthFirstSolve();                      break;                  case "Maze Routing":                      solver = new MazeRoutingSolve();                      break;                  case "Breadth First":                      solver = new BreadthFirstSolve();                      break;              }                List<Coordinate> path = solver.SolveMaze(maze);                return Task.FromResult(new Path { Path\_ = JsonConvert.SerializeObject(path) });          }      }  } |

### RegisterService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;      namespace Server.Services  {      public class RegisterService : Registerer.RegistererBase      {          public override Task<Acknowledgement> Register(Account request, ServerCallContext context) {              try {                  using (SQLiteConnection conn = new(                      "Data Source= mazeData.db; " +                      "Version = 3; " +                      "New = True; " +                      "Compress = True; ")) {                      conn.Open();                        using SQLiteCommand cmd = conn.CreateCommand();                      cmd.CommandText = @"INSERT INTO User(Username, Password, Salt)                                          VALUES(@Username, @Password, @Salt)";                      cmd.Parameters.AddWithValue("@Username", request.Username);                      cmd.Parameters.AddWithValue("@Password", request.Password);                      cmd.Parameters.AddWithValue("@Salt", request.Salt);                        cmd.ExecuteNonQuery();                  }                    return Task.FromResult(new Acknowledgement { Success = true });              }              catch (Exception) {                  return Task.FromResult(new Acknowledgement { Success = false });              }          }      }  }    // test code: retrieves all usernames and password hashes    //cmd.CommandText = "SELECT \* FROM Login";  //SQLiteDataReader sqliteDataReader = cmd.ExecuteReader();    //while (sqliteDataReader.Read()) {  //    string myReader = $"{sqliteDataReader.GetString(0)}, {sqliteDataReader.GetString(1)}";  //    Console.WriteLine(myReader);  //}  //sqliteDataReader.Close(); |

### SaveMazeService.cs

|  |
| --- |
| using Grpc.Core;  using System.Data.SQLite;    namespace Server.Services  {      public class SaveMazeService : Saver.SaverBase      {          public override Task<SuccessAck> SaveMaze(SaveRequest request, ServerCallContext context) {              try {                  using (SQLiteConnection conn = new(                      "Data Source= mazeData.db; " +                      "Version = 3; " +                      "New = True; " +                      "Compress = True; ")) {                      conn.Open();                      using SQLiteCommand cmd = conn.CreateCommand();                        cmd.CommandText = @"INSERT INTO Mazes(MazeObject, MazeGenAlg, MazeName, UID)                                          VALUES(@MazeObject, @MazeGenAlg, @MazeName, @UID)";                      cmd.Parameters.AddWithValue("@MazeObject", request.MazeJson);                      cmd.Parameters.AddWithValue("@MazeGenAlg", request.MazeType);                      cmd.Parameters.AddWithValue("@MazeName", request.MazeName);                      cmd.Parameters.AddWithValue("@UID", request.UserID);                      cmd.ExecuteNonQuery();                  }                    return Task.FromResult(new SuccessAck { Success = true });                }              catch (Exception) {                  return Task.FromResult(new SuccessAck { Success = false });              }          }      }  } |

## Algorithm Classes

### Bread First Solve.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Runtime.Serialization.Formatters.Binary;  using System.Text;  using System.Threading.Tasks;    namespace Server {      internal class BreadthFirstSolve : SolvingAlgorithm      {          List<(Coordinate cell, Coordinate parentCell)> paths = new List<(Coordinate cell, Coordinate parentCell)>();          List<(Coordinate cell, Coordinate parentCell)> activeCells = new List<(Coordinate cell, Coordinate parentCell)>();            public override List<Coordinate> SolveMaze(Maze maze) {                  List<Coordinate> cellsNeighbouringEntrance = GetUnvisitedNeighbours(maze.MazeEntranceCoordinate, maze);              Coordinate startCell = new Coordinate(cellsNeighbouringEntrance[0].Xpos, cellsNeighbouringEntrance[0].Ypos);              paths.Add((maze.MazeCoordinates[startCell.Ypos, startCell.Xpos], maze.MazeEntranceCoordinate));              activeCells.Add((maze.MazeCoordinates[startCell.Ypos, startCell.Xpos], maze.MazeEntranceCoordinate));              maze.MazeCoordinates[startCell.Ypos, startCell.Xpos].Visited = true;                bool finished = false;              while(!finished) {                  (Coordinate cell, Coordinate parentCell) = activeCells[0];                  List<Coordinate> neighbourCells = GetUnvisitedNeighbours(cell, maze);                    foreach (Coordinate neighbourCell in neighbourCells) {                      paths.Add((maze.MazeCoordinates[neighbourCell.Ypos, neighbourCell.Xpos], maze.MazeCoordinates[cell.Ypos, cell.Xpos]));                      activeCells.Add((maze.MazeCoordinates[neighbourCell.Ypos, neighbourCell.Xpos], maze.MazeCoordinates[cell.Ypos, cell.Xpos]));                      maze.MazeCoordinates[neighbourCell.Ypos, neighbourCell.Xpos].Visited = true;                      if (neighbourCell.Equals(maze.MazeExitCoordinate)) finished = true;                  }                    activeCells.RemoveAt(0);                }                return GetSolution(paths, maze);          }            private List<Coordinate> GetSolution(List<(Coordinate cell, Coordinate parentCell)> paths, Maze maze) {              List<Coordinate> solution = new();                Coordinate parentCoordinate = null;              foreach (var cell in paths) {                  if (cell.cell.Equals(maze.MazeExitCoordinate)) {                      solution.Add(cell.cell);                      parentCoordinate = cell.parentCell;                      break;                  }              }                while (!parentCoordinate.Equals(maze.MazeEntranceCoordinate)) {                  foreach (var cell in paths) {                      if (cell.cell.Equals(parentCoordinate)) {                          solution.Add(cell.cell);                          parentCoordinate = cell.parentCell;                          break;                      }                  }              }                return solution;          }            private List<Coordinate> GetUnvisitedNeighbours(Coordinate cell, Maze maze) {                List<Coordinate> cells = new();                if (cell.Ypos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos - 1, cell.Xpos]                  && !IsInPath(maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos]))                  cells.Add(maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos]);                if (cell.Xpos + 1 < maze.MazeActualWidth                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos + 1]                  && !IsInPath(maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1]))                  cells.Add(maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1]);                if (cell.Ypos + 1 < maze.MazeActualHeight                  && !maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos + 1, cell.Xpos]                  && !IsInPath(maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos]))                  cells.Add(maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos]);                if (cell.Xpos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos - 1]                  && !IsInPath(maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1]))                  cells.Add(maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1]);                return cells;          }            private bool IsInPath(Coordinate cell) {                foreach (var coordinate in paths) {                  if (coordinate.cell.Equals(cell) || coordinate.parentCell.Equals(cell)) return true;              }              return false;          }      }  } |

### Coordinate.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Server  {      internal class Coordinate      {          [JsonConstructor]          public Coordinate() {            }          public Coordinate(int xPos, int yPos)          {              this.xPos = xPos;              this.yPos = yPos;              visited = false;          }          public Coordinate(Tuple<int, int> pos)          {              xPos = pos.Item1;              yPos = pos.Item2;              visited = false;          }            #region Properties          private int xPos;          public int Xpos          {              get { return xPos; }              set { xPos = value; }          }            private int yPos;          public int Ypos          {              get { return yPos; }              set { yPos = value; }          }            private bool visited;          public bool Visited          {              get { return visited; }              set { visited = value; }          }          #endregion            #region Methods          public (int x, int y) GetCartesianCoordinates(Maze maze)          {              return (xPos + 1, maze.MazeActualHeight - yPos + 1);          }          public int GetManhattanDistance(Coordinate targetCoordinate)          {              return Math.Abs(Xpos - targetCoordinate.Xpos) + Math.Abs(Ypos - targetCoordinate.Ypos);          }            public bool Equals(Coordinate target)          {              return xPos == target.xPos && yPos == target.yPos;          }          #endregion      }  } |

### DepthFirstSolve.cs

|  |
| --- |
| namespace Server  {      internal class DepthFirstSolve : SolvingAlgorithm      {          public override List<Coordinate> SolveMaze(Maze maze) {              Coordinate solver = new(maze.MazeEntranceCoordinate.Xpos, maze.MazeEntranceCoordinate.Ypos);              List<Coordinate> path = new();              path.Add(solver);              maze.MazeCoordinates[solver.Ypos, solver.Xpos].Visited = true;                while (!solver.Equals(maze.MazeExitCoordinate)) {                  List<Coordinate> unvisitedNeighbours = GetUnvisitedNeighbours(solver, maze);                    if (unvisitedNeighbours.Count > 0) { //if paths exist, take the first one.                      solver = new(unvisitedNeighbours[0].Xpos, unvisitedNeighbours[0].Ypos);                      path.Add(solver);                      maze.MazeCoordinates[solver.Ypos, solver.Xpos].Visited = true;                  }                  else { //otherwise, backtrack.                      solver = path[^2];                      path.RemoveAt(path.Count - 1);                  }              }                return path;          }          private List<Coordinate> GetUnvisitedNeighbours(Coordinate cell, Maze maze) {                List<Coordinate> cells = new();                if (cell.Ypos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos - 1, cell.Xpos])                  cells.Add(maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos]);                if (cell.Xpos + 1 < maze.MazeActualWidth                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos + 1])                  cells.Add(maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1]);                if (cell.Ypos + 1 < maze.MazeActualHeight                  && !maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos + 1, cell.Xpos])                  cells.Add(maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos]);                if (cell.Xpos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos - 1])                  cells.Add(maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1]);                return cells;          }      }  } |

### Growing Tree Generation.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Server  {      internal class GrowingTreeGeneration : Maze      {          List<Coordinate> cellsInMaze = new();            [JsonConstructor]          public GrowingTreeGeneration() {            }          public GrowingTreeGeneration(int cellWidth, int cellHeight) {              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();          }            public override void BuildMaze(Coordinate startCell) {              List<Coordinate> activeCells = new();              List<Coordinate> visitedCells = new();              activeCells.Add(cellsInMaze[rgen.Next(cellsInMaze.Count)]);                while (activeCells.Count > 0) {                  Coordinate cell = activeCells[rgen.Next(activeCells.Count)];                  List<Coordinate> unvisited = GetUnvisitedNeighbours(cell, activeCells, visitedCells);                    if (unvisited.Count > 0) {                      Coordinate targetCell = unvisited[rgen.Next(unvisited.Count)];                      DestroyWall(cell, targetCell);                      activeCells.Add(targetCell);                  }                  else {                      activeCells.Remove(cell);                      visitedCells.Add(cell);                      continue;                  }              }          }          private void DestroyWall(Coordinate cell1, Coordinate cell2) {              int midX = Math.Min(cell1.Xpos, cell2.Xpos) + Math.Abs(cell1.Xpos - cell2.Xpos) / 2;              int midY = Math.Min(cell1.Ypos, cell2.Ypos) + Math.Abs(cell1.Ypos - cell2.Ypos) / 2;              MazeWalls[midY, midX] = false;          }            private List<Coordinate> GetUnvisitedNeighbours(Coordinate cell, List<Coordinate> activeCells, List<Coordinate> visitedCells) {              List<Coordinate> unvisited = new();                if (cell.Ypos - 2 >= 0)//N                  if (!activeCells.Contains(MazeCoordinates[cell.Ypos - 2, cell.Xpos]) &&                      !visitedCells.Contains(MazeCoordinates[cell.Ypos - 2, cell.Xpos]))                      unvisited.Add(MazeCoordinates[cell.Ypos - 2, cell.Xpos]);                if (cell.Xpos + 2 < MazeActualWidth)//E                  if (!activeCells.Contains(MazeCoordinates[cell.Ypos, cell.Xpos + 2]) &&                      !visitedCells.Contains(MazeCoordinates[cell.Ypos, cell.Xpos + 2]))                      unvisited.Add(MazeCoordinates[cell.Ypos, cell.Xpos + 2]);                if (cell.Ypos + 2 < MazeActualHeight)//S                  if (!activeCells.Contains(MazeCoordinates[cell.Ypos + 2, cell.Xpos]) &&                      !visitedCells.Contains(MazeCoordinates[cell.Ypos + 2, cell.Xpos]))                      unvisited.Add(MazeCoordinates[cell.Ypos + 2, cell.Xpos]);                if (cell.Xpos - 2 >= 0)//W                  if (!activeCells.Contains(MazeCoordinates[cell.Ypos, cell.Xpos - 2]) &&                      !visitedCells.Contains(MazeCoordinates[cell.Ypos, cell.Xpos - 2]))                      unvisited.Add(MazeCoordinates[cell.Ypos, cell.Xpos - 2]);                return unvisited;          }              public override void CreateEntranceExit(bool atBorder) {              MazeWalls[1, 0] = false; //entrance              MazeEntranceCoordinate = new Coordinate(0, 1);                if (atBorder) //border exit              {                  Maze​Walls[MazeActualHeight - 2, MazeActualWidth - 1] = false; //exit                  MazeExitCoordinate = new Coordinate(MazeActualWidth - 1, MazeActualHeight - 2);              }              else //central exit              {                  int centerX, centerY;                  centerX = MazeActualWidth / 2;                  centerY = MazeActualHeight / 2;                  MazeWalls[centerY, centerX] = false;                  MazeExitCoordinate = new Coordinate(centerX, centerY);              }                ResetVisited();          }            public override void InitMaze() {              MazeActualHeight = 2 \* MazeCellHeight + 1;              MazeActualWidth = 2 \* MazeCellWidth + 1;                MazeWalls = new bool[MazeActualHeight, MazeActualWidth];              MazeCoordinates = new Coordinate[MazeActualHeight, MazeActualWidth];                for (int y = 0; y < MazeActualHeight; y++) {                  for (int x = 0; x < MazeActualWidth; x++) {                      MazeCoordinates[y, x] = new Coordinate(x, y);                        if (y % 2 != 0 && x % 2 != 0) {                          MazeWalls[y, x] = false;                          cellsInMaze.Add(MazeCoordinates[y, x]);                      }                        else                          MazeWalls[y, x] = true;                  }              }          }            public override void RemoveWalls(int wallsToRemove) {              int wallsRemoved = 0;                while (wallsRemoved < wallsToRemove) {                  int xPos = rgen.Next(1, MazeActualWidth - 1);                  int yPos = rgen.Next(1, MazeActualHeight - 1);                  Coordinate cellToRemove = new(xPos, yPos);                    if (IsWall(cellToRemove)) {                      MazeWalls[yPos, xPos] = false;                      wallsRemoved++;                  }              }          }            private bool IsWall(Coordinate cell) {              if (MazeWalls[cell.Ypos + 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == true                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == true) {                  return true;              }              else if (MazeWalls[cell.Ypos + 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == false                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == false) {                  return true;              }              else return false;          }      }  } |

### Maze Routing Solve.cs

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;    namespace Server  {      internal class MazeRoutingSolve : SolvingAlgorithm      {          public override List<Coordinate> SolveMaze(Maze maze) {              List<Coordinate> solution = new List<Coordinate>();              Coordinate solver = new Coordinate(maze.MazeEntranceCoordinate.Xpos, maze.MazeEntranceCoordinate.Ypos);                int bestMD = solver.GetManhattanDistance(maze.MazeExitCoordinate);              solution.Add(solver);                maze.MazeCoordinates[solver.Ypos, solver.Xpos].Visited = true;                while (!solver.Equals(maze.MazeExitCoordinate)) {                  List<(Coordinate coordinate, char direction)> unvisited = GetUnvisitedNeighbours(maze, solver);                    if (unvisited.Count == 0) {                      do {                          solution.Remove(solution.Last());                          solver = new Coordinate(solution[^1].Xpos, solution[^1].Ypos);                          unvisited = GetUnvisitedNeighbours(maze, solver);                      } while (unvisited.Count == 0);                      continue;                  }                  else if (unvisited.Count == 1) {                      solver = new Coordinate(unvisited[0].coordinate.Xpos, unvisited[0].coordinate.Ypos);                      solution.Add(maze.MazeCoordinates[solver.Ypos, solver.Xpos]);                      maze.MazeCoordinates[solver.Ypos, solver.Xpos].Visited = true;                  }                  else {                      char directionToMove = TryPaths(maze, unvisited);                      Coordinate cellToMoveTo = null;                      foreach (var cell in unvisited) {                          if (cell.direction == directionToMove) {                              cellToMoveTo = cell.coordinate;                              break;                          }                      }                      solver = new Coordinate(cellToMoveTo.Xpos, cellToMoveTo.Ypos);                      solution.Add(maze.MazeCoordinates[solver.Ypos, solver.Xpos]);                      maze.MazeCoordinates[solver.Ypos, solver.Xpos].Visited = true;                  }              }                  return solution;          }            private char TryPaths(Maze maze, List<(Coordinate coordinate, char direction)> paths) {              Coordinate tempSolver;              List<Coordinate> cellsVisited = new List<Coordinate>();              (char direction, int BestMD) bestPath = (' ', 99999999); //effectively infinity for these mazes                foreach (var path in paths) {                  List<(Coordinate coordinate, char direction)> unvisited;                    do {                      tempSolver = new Coordinate(path.coordinate.Xpos, path.coordinate.Ypos);                      unvisited = GetUnvisitedNeighbours(maze, tempSolver);                        if (unvisited.Count == 1) {                          tempSolver = new Coordinate(unvisited[0].coordinate.Xpos, unvisited[0].coordinate.Ypos);                          cellsVisited.Add(maze.MazeCoordinates[tempSolver.Ypos, tempSolver.Xpos]);                          maze.MazeCoordinates[tempSolver.Ypos, tempSolver.Xpos].Visited = true;                      }                      else {                          int MD = tempSolver.GetManhattanDistance(maze.MazeExitCoordinate);                            if (MD < bestPath.BestMD) {                              bestPath = (path.direction, MD);                          }                      }                  } while (unvisited.Count == 1);              }                foreach (var cell in cellsVisited) {                  cell.Visited = false;              }                return bestPath.direction;          }            private List<(Coordinate coordinate, char direction)> GetUnvisitedNeighbours(Maze maze, Coordinate cell) {              List<(Coordinate coordinate, char direction)> cells = new();                if (cell.Ypos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos - 1, cell.Xpos])                  cells.Add((maze.MazeCoordinates[cell.Ypos - 1, cell.Xpos], 'N'));                if (cell.Xpos + 1 < maze.MazeActualWidth                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos + 1])                  cells.Add((maze.MazeCoordinates[cell.Ypos, cell.Xpos + 1], 'E'));                if (cell.Ypos + 1 < maze.MazeActualHeight                  && !maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos].Visited                  && !maze.MazeWalls[cell.Ypos + 1, cell.Xpos])                  cells.Add((maze.MazeCoordinates[cell.Ypos + 1, cell.Xpos], 'S'));                if (cell.Xpos - 1 >= 0                  && !maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1].Visited                  && !maze.MazeWalls[cell.Ypos, cell.Xpos - 1])                  cells.Add((maze.MazeCoordinates[cell.Ypos, cell.Xpos - 1], 'W'));                return cells;          }      }  } |

### Maze.cs

|  |
| --- |
| namespace Server  {      internal abstract class Maze      {          #region Properties          private int mazeActualWidth;          public int MazeActualWidth          {              get { return mazeActualWidth; }              set { mazeActualWidth = value; }          }            private int mazeActualHeight;          public int MazeActualHeight          {              get { return mazeActualHeight; }              set { mazeActualHeight = value; }          }            private int mazeCellWidth;          public int MazeCellWidth          {              get { return mazeCellWidth; }              set { mazeCellWidth = value; }          }            private int mazeCellHeight;          public int MazeCellHeight          {              get { return mazeCellHeight; }              set { mazeCellHeight = value; }          }            private bool[,]? mazeWalls;          public bool[,]? MazeWalls          {              get { return mazeWalls; }              set { mazeWalls = value; }          }            private Coordinate[,]? mazeCoordinates;          public Coordinate[,]? MazeCoordinates          {              get { return mazeCoordinates; }              set { mazeCoordinates = value; }          }            private Coordinate? mazeEntranceCoordinate;          public Coordinate? MazeEntranceCoordinate          {              get { return mazeEntranceCoordinate; }              set { mazeEntranceCoordinate = value; }          }            private Coordinate? mazeExitCoordinate;          public Coordinate? MazeExitCoordinate          {              get { return mazeExitCoordinate; }              set { mazeExitCoordinate = value; }          }            protected Random rgen = new();          #endregion            #region Methods          public abstract void InitMaze();          public abstract void BuildMaze(Coordinate startCell);          public abstract void CreateEntranceExit(bool atBorder);          public abstract void RemoveWalls(int wallsToRemove);          protected virtual bool CellVisited(Coordinate cellPos)          {              return cellPos.Visited;          }          protected virtual void ResetVisited()          {              foreach (Coordinate v in mazeCoordinates)              {                  v.Visited = false;              }          }          #endregion      }  } |

### Program.cs

The main entry point for the server, containing definitions for all database tables and the code which maps available services to the server request pipeline.

|  |
| --- |
| using Server.Services;  using System.Data.SQLite;    //initialize database  SQLiteConnection conn = new(      "Data Source= mazeData.db; " +      "Version = 3; " +      "New = True; " +      "Compress = True; ");  conn.Open();  using (SQLiteCommand cmd = conn.CreateCommand()) {      cmd.CommandText = "PRAGMA foreign\_keys = ON;";      cmd.ExecuteNonQuery();        cmd.CommandText = @"CREATE TABLE IF NOT EXISTS User(                              UID INTEGER PRIMARY KEY,                              Username VARCHAR,                              Password VARCHAR,                              Salt VARCHAR                          )";      cmd.ExecuteNonQuery();        cmd.CommandText = @"CREATE TABLE IF NOT EXISTS Mazes(                              MazeID INTEGER PRIMARY KEY,                              MazeObject VARCHAR,                              MazeGenAlg VARCHAR,                              MazeName VARCHAR(10),                              UID INTEGER,                              FOREIGN KEY(UID) REFERENCES User(UID)                          )";      cmd.ExecuteNonQuery();        cmd.CommandText = @"CREATE TABLE IF NOT EXISTS UserStats(                              UID INTEGER PRIMARY KEY,                              Time1Display VARCHAR,                              Time1Milliseconds INTEGER,                              Time2Display VARCHAR,                              Time2Milliseconds INTEGER,                              Time3Display VARCHAR,                              Time3Milliseconds INTEGER,                              Time4Display VARCHAR,                              Time4Milliseconds INTEGER,                              Time5Display VARCHAR,                              Time5Milliseconds INTEGER,                              Time6Display VARCHAR,                              Time6Milliseconds INTEGER,                              Time7Display VARCHAR,                              Time7Milliseconds INTEGER,                              Time8Display VARCHAR,                              Time8Milliseconds INTEGER,                              Time9Display VARCHAR,                              Time9Milliseconds INTEGER,                              Time10Display VARCHAR,                              Time10Milliseconds INTEGER,                              RecursiveBacktrackMazesGenerated INTEGER,                              GrowingTreeMazesGenerated INTEGER,                              WilsonsMazesGenerated INTEGER,                              FOREIGN KEY(UID) REFERENCES User(UID)                          )";      cmd.ExecuteNonQuery();      //ON DELETE CASCADE deletes the relevent record if the user is deleted        cmd.CommandText = @"                      CREATE TABLE IF NOT EXISTS GlobalStats (                          Time1Display VARCHAR,                          Time1Milliseconds INTEGER,                          Time1Name VARCHAR,                            Time2Display VARCHAR,                          Time2Milliseconds INTEGER,                          Time2Name VARCHAR,                            Time3Display VARCHAR,                          Time3Milliseconds INTEGER,                          Time3Name VARCHAR,                            Time4Display VARCHAR,                          Time4Milliseconds INTEGER,                          Time4Name VARCHAR,                            Time5Display VARCHAR,                          Time5Milliseconds INTEGER,                          Time5Name VARCHAR,                            Time6Display VARCHAR,                          Time6Milliseconds INTEGER,                          Time6Name VARCHAR,                            Time7Display VARCHAR,                          Time7Milliseconds INTEGER,                          Time7Name VARCHAR,                            Time8Display VARCHAR,                          Time8Milliseconds INTEGER,                          Time8Name VARCHAR,                            Time9Display VARCHAR,                          Time9Milliseconds INTEGER,                          Time9Name VARCHAR,                            Time10Display VARCHAR,                          Time10Milliseconds INTEGER,                          Time10Name VARCHAR,                            RecursiveBacktrackMazesGenerated INTEGER,                          GrowingTreeMazesGenerated INTEGER,                          WilsonsMazesGenerated INTEGER                      )";      cmd.ExecuteNonQuery();        cmd.CommandText = @"CREATE TRIGGER IF NOT EXISTS CreateStatsRecord                          AFTER INSERT ON User                          BEGIN                              INSERT INTO UserStats                              VALUES (NEW.UID,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      '',                                      -1,                                      0,                                      0,                                      0);                          END;";      cmd.ExecuteNonQuery();        cmd.CommandText = "SELECT COUNT(\*) FROM GlobalStats";      int rowCount = Convert.ToInt32(cmd.ExecuteScalar());      if (rowCount == 0) {          cmd.CommandText = @"INSERT INTO GlobalStats                              VALUES('',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     '',                                     -1,                                     '',                                     0,                                     0,                                     0)";          cmd.ExecuteNonQuery();      }  }    conn.Close();    var builder = WebApplication.CreateBuilder(args);    // Add services to the container.  builder.Services.AddGrpc();    var app = builder.Build();    // Configure the HTTP request pipeline.  app.MapGrpcService<GreeterService>();  app.MapGrpcService<MazeBuilderService>();  app.MapGrpcService<MazeSolverService>();  app.MapGrpcService<RegisterService>();  app.MapGrpcService<CheckerIfUserExistsService>();  app.MapGrpcService<LoginService>();  app.MapGrpcService<SaveMazeService>();  app.MapGrpcService<GetMazesService>();  app.MapGrpcService<LoadMazeService>();  app.MapGrpcService<DeleteMazeService>();  app.MapGrpcService<GlobalStatHandlerService>();  app.MapGrpcService<HandleUserStatsService>();  app.MapGrpcService<GetStatsService>();    app.Run(); |

### Recursive Backtrack Generation.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Server  {      internal class RecursiveBacktrackGeneration : Maze      {          [JsonConstructor]          public RecursiveBacktrackGeneration() {            }          public RecursiveBacktrackGeneration(int cellWidth, int cellHeight)          {              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();          }            public override void InitMaze()          {              MazeActualHeight = 2 \* MazeCellHeight + 1;              MazeActualWidth = 2 \* MazeCellWidth + 1;                MazeWalls = new bool[MazeActualHeight, MazeActualWidth];              MazeCoordinates = new Coordinate[MazeActualHeight, MazeActualWidth];                for (int y = 0; y < MazeActualHeight; y++)              {                  for (int x = 0; x < MazeActualWidth; x++)                  {                      if (y % 2 != 0 && x % 2 != 0)                          MazeWalls[y, x] = false;                      else                          MazeWalls[y, x] = true;                        MazeCoordinates[y, x] = new Coordinate(x, y);                  }              }          }            public override void BuildMaze(Coordinate cell)          {              cell.Visited = true;                List<Coordinate> neighbourCells = GetUnvisitedNeighbours(cell);                while (neighbourCells.Count > 0)              {                  Coordinate targetCell = neighbourCells[rgen.Next(0, neighbourCells.Count)];                  if (targetCell.Visited) return;                  DestroyWall(cell, targetCell);                  neighbourCells.Remove(targetCell);                  BuildMaze(targetCell);              }          }            public override void RemoveWalls(int wallsToRemove)          {              int wallsRemoved = 0;                while (wallsRemoved < wallsToRemove)              {                  int xPos = rgen.Next(1, MazeActualWidth - 1);                  int yPos = rgen.Next(1, MazeActualHeight - 1);                  Coordinate cellToRemove = new(xPos, yPos);                    if (IsWall(cellToRemove))                  {                      MazeWalls[yPos, xPos] = false;                      wallsRemoved++;                  }              }          }            private bool IsWall(Coordinate cell)          {              if (MazeWalls[cell.Ypos + 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == true                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == true)              {                  return true;              }              else if (MazeWalls[cell.Ypos + 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == false                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == false)              {                  return true;              }              else return false;          }            private List<Coordinate> GetUnvisitedNeighbours(Coordinate cell)          {                List<Coordinate> cells = new();                if (cell.Ypos - 2 >= 0 && !MazeCoordinates[cell.Ypos - 2, cell.Xpos].Visited)                  cells.Add(MazeCoordinates[cell.Ypos - 2, cell.Xpos]);                if (cell.Xpos + 2 < MazeActualWidth && !MazeCoordinates[cell.Ypos, cell.Xpos + 2].Visited)                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos + 2]);                if (cell.Ypos + 2 < MazeActualHeight && !MazeCoordinates[cell.Ypos + 2, cell.Xpos].Visited)                  cells.Add(MazeCoordinates[cell.Ypos + 2, cell.Xpos]);                if (cell.Xpos - 2 >= 0 && !MazeCoordinates[cell.Ypos, cell.Xpos - 2].Visited)                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos - 2]);                return cells;          }            private void DestroyWall(Coordinate cell1, Coordinate cell2)          {              int midX = Math.Min(cell1.Xpos, cell2.Xpos) + Math.Abs(cell1.Xpos - cell2.Xpos) / 2;              int midY = Math.Min(cell1.Ypos, cell2.Ypos) + Math.Abs(cell1.Ypos - cell2.Ypos) / 2;              MazeWalls[midY, midX] = false;          }            public override void CreateEntranceExit(bool atBorder)          {              MazeWalls[1, 0] = false; //entrance              MazeEntranceCoordinate = new Coordinate(0, 1);                if (atBorder) //border exit              {                  Maze​Walls[MazeActualHeight - 2, MazeActualWidth - 1] = false; //exit                  MazeExitCoordinate = new Coordinate(MazeActualWidth - 1, MazeActualHeight - 2);              }              else //central exit              {                  int centerX, centerY;                  centerX = MazeActualWidth / 2;                  centerY = MazeActualHeight / 2;                  MazeWalls[centerY, centerX] = false;                  MazeExitCoordinate = new Coordinate(centerX, centerY);              }                  ResetVisited();          }      }  } |

### Solving.cs

|  |
| --- |
| namespace Server  {      internal abstract class SolvingAlgorithm      {          public abstract List<Coordinate> SolveMaze(Maze maze);      }  } |

### Wilsons Generation.cs

|  |
| --- |
| using Newtonsoft.Json;    namespace Server  {      internal class WilsonsGeneration : Maze      {          public List<Coordinate> cellsInMaze = new();          public bool exitAtBorder;            [JsonConstructor]          public WilsonsGeneration() {            }          public WilsonsGeneration(int cellWidth, int cellHeight, bool exitAtBorder) {              MazeCellWidth = cellWidth;              MazeCellHeight = cellHeight;              rgen = new();              this.exitAtBorder = exitAtBorder;          }            public override void BuildMaze(Coordinate startCell) {                /\*startCoordinate and endCoordinate should be where the algorithm starts and ends, but since these are in the walls, they are              inaccessible to the algorithm, so we must pick the neighbouring cells.\*/                InitialLoopErasedWalk(MazeCoordinates[1, 1],                  exitAtBorder ? MazeCoordinates[MazeExitCoordinate.Ypos, MazeExitCoordinate.Xpos - 1] : cellsInMaze[(int)(cellsInMaze.Count \* 0.5f)]);              //if the exit is in the centre, we can find the cell by finding the middle element of the cellsInMaze list              while (cellsInMaze.Count > 0) {                  LoopErasedWalk(cellsInMaze[rgen.Next(cellsInMaze.Count)]);              }              ResetVisited();          }            private void InitialLoopErasedWalk(Coordinate startCoordinate, Coordinate endCoordinate) {              List<Coordinate> path = new List<Coordinate>();              Coordinate ctor = new Coordinate(startCoordinate.Xpos, startCoordinate.Ypos);              path.Add(MazeCoordinates[ctor.Ypos, ctor.Xpos]);              MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = true;                while (!ctor.Equals(endCoordinate)) {                  List<Coordinate> unvisitedCells = GetUnvisitedNeighbours(ctor);                  Coordinate targetCell;                  do {                      int index = rgen.Next(unvisitedCells.Count);                      targetCell = unvisitedCells[index];                      if (path.Contains(targetCell)) unvisitedCells.RemoveAt(index);                      if (unvisitedCells.Count == 0) break;                  } while (path.Contains(targetCell));                      if (!path.Contains(targetCell)) {                      // keep walking, we will destroy all walls after the walk.                      // We cannot do this dynamically as we have to backtrack sometimes                      ctor = new Coordinate(targetCell.Xpos, targetCell.Ypos);                      path.Add(MazeCoordinates[ctor.Ypos, ctor.Xpos]);                      MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = true;                  }                  else { //we must have looped, so backtrack until ctor = targetCell and try again                      while (!ctor.Equals(targetCell)) {                          MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = false;                            path.RemoveAt(path.Count - 1);                          ctor = new Coordinate(path.Last().Xpos, path.Last().Ypos);                      }                  }              }                DestroyAllWallsInPath(path);                foreach (Coordinate coord in path) {                  cellsInMaze.Remove(coord);              }                DestroyExitWalls();          }            private void DestroyExitWalls() {              if (exitAtBorder)                  Maze​Walls[MazeActualHeight - 2, MazeActualWidth - 1] = false;              else {                  int centerX = MazeActualWidth / 2;                  int centerY = MazeActualHeight / 2;                  MazeWalls[centerY, centerX] = false;              }          }            private void LoopErasedWalk(Coordinate startCoordinate) { //walk until we hit the maze              List<Coordinate> path = new List<Coordinate>();              Coordinate ctor = new Coordinate(startCoordinate.Xpos, startCoordinate.Ypos);              path.Add(MazeCoordinates[ctor.Ypos, ctor.Xpos]);              MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = true;                while (true) {                  List<Coordinate> unvisitedCells = GetNeighbouringCells(ctor);                  Coordinate targetCell;                  do {                      int index = rgen.Next(unvisitedCells.Count);                      targetCell = unvisitedCells[index];                        if (path.Contains(targetCell)) unvisitedCells.RemoveAt(index);                        else if (targetCell.Visited) { //if we find the maze, which is a visited cell that is not in the path                          ctor = new Coordinate(targetCell.Xpos, targetCell.Ypos);                          path.Add(MazeCoordinates[ctor.Ypos, ctor.Xpos]);                          MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = true;                          DestroyAllWallsInPath(path);                            foreach (Coordinate coord in path) {                              cellsInMaze.Remove(coord);                          }                          return;                      }                        if (unvisitedCells.Count == 0) break;                    } while (path.Contains(targetCell));                      if (!path.Contains(targetCell)) {                      // keep walking, we will destroy all walls after the walk.                      // We cannot do this dynamically as we have to backtrack sometimes                      ctor = new Coordinate(targetCell.Xpos, targetCell.Ypos);                      path.Add(MazeCoordinates[ctor.Ypos, ctor.Xpos]);                      MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = true;                  }                  else { //we must have looped, so backtrack until ctor = targetCell and try again                      while (!ctor.Equals(targetCell)) {                          MazeCoordinates[ctor.Ypos, ctor.Xpos].Visited = false;                            path.RemoveAt(path.Count - 1);                          ctor = new Coordinate(path.Last().Xpos, path.Last().Ypos);                      }                  }              }          }            private void DestroyAllWallsInPath(List<Coordinate> pathToDestroy) {              for (int i = 0; i < pathToDestroy.Count - 1; i++) {                  DestroyWall(pathToDestroy[i], pathToDestroy[i + 1]);              }          }            private void DestroyWall(Coordinate cell1, Coordinate cell2) {              int midX = Math.Min(cell1.Xpos, cell2.Xpos) + Math.Abs(cell1.Xpos - cell2.Xpos) / 2;              int midY = Math.Min(cell1.Ypos, cell2.Ypos) + Math.Abs(cell1.Ypos - cell2.Ypos) / 2;              MazeWalls[midY, midX] = false;          }            private List<Coordinate> GetUnvisitedNeighbours(Coordinate cell) {              List<Coordinate> cells = new List<Coordinate>();              //we can handle adding things to the maze with careful management of the 'visited' property.              //This means we will have to reimplement getUnivisitedNeighbours              //to account for not just being able to go if !Visited.              //We will instead redefine unvisited as having all 4 walls and the visited property being true will              //mean being part of the maze.              //The following checks if all adjacent cells have all 4 walls intact.              if (cell.Ypos - 2 >= 0 &&                  MazeWalls[cell.Ypos - 2 - 1, cell.Xpos] &&                  MazeWalls[cell.Ypos - 2, cell.Xpos + 1] &&                  MazeWalls[cell.Ypos - 2 + 1, cell.Xpos] &&                  MazeWalls[cell.Ypos - 2, cell.Xpos - 1]) // N                  cells.Add(MazeCoordinates[cell.Ypos - 2, cell.Xpos]);              if (cell.Xpos + 2 < MazeActualWidth &&                  MazeWalls[cell.Ypos - 1, cell.Xpos + 2] &&                  MazeWalls[cell.Ypos, cell.Xpos + 2 + 1] &&                  MazeWalls[cell.Ypos + 1, cell.Xpos + 2] &&                  MazeWalls[cell.Ypos, cell.Xpos + 2 - 1]) // E                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos + 2]);              if (cell.Ypos + 2 < MazeActualHeight &&                  MazeWalls[cell.Ypos + 2 - 1, cell.Xpos] &&                  MazeWalls[cell.Ypos + 2, cell.Xpos + 1] &&                  MazeWalls[cell.Ypos + 2 + 1, cell.Xpos] &&                  MazeWalls[cell.Ypos + 2, cell.Xpos - 1]) // S                  cells.Add(MazeCoordinates[cell.Ypos + 2, cell.Xpos]);              if (cell.Xpos - 2 >= 0 &&                  MazeWalls[cell.Ypos - 1, cell.Xpos - 2] &&                  MazeWalls[cell.Ypos, cell.Xpos - 2 + 1] &&                  MazeWalls[cell.Ypos + 1, cell.Xpos - 2] &&                  MazeWalls[cell.Ypos, cell.Xpos - 2 - 1]) // W                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos - 2]);                return cells;          }          private List<Coordinate> GetNeighbouringCells(Coordinate cell) {              List<Coordinate> cells = new List<Coordinate>();                if (cell.Ypos - 2 >= 0) // N                  cells.Add(MazeCoordinates[cell.Ypos - 2, cell.Xpos]);              if (cell.Xpos + 2 < MazeActualWidth) // E                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos + 2]);              if (cell.Ypos + 2 < MazeActualHeight) // S                  cells.Add(MazeCoordinates[cell.Ypos + 2, cell.Xpos]);              if (cell.Xpos - 2 >= 0) // W                  cells.Add(MazeCoordinates[cell.Ypos, cell.Xpos - 2]);                return cells;          }              public override void CreateEntranceExit(bool atBorder) {              MazeWalls[1, 0] = false; //entrance              MazeEntranceCoordinate = new Coordinate(0, 1);                if (atBorder) //border exit              {                   //exit                  MazeExitCoordinate = new Coordinate(MazeActualWidth - 1, MazeActualHeight - 2);              }              else //central exit              {                  int centerX, centerY;                  centerX = MazeActualWidth / 2;                  centerY = MazeActualHeight / 2;                    MazeExitCoordinate = new Coordinate(centerX, centerY);              }          }            public override void InitMaze() {              MazeActualHeight = 2 \* MazeCellHeight + 1;              MazeActualWidth = 2 \* MazeCellWidth + 1;                MazeWalls = new bool[MazeActualHeight, MazeActualWidth];              MazeCoordinates = new Coordinate[MazeActualHeight, MazeActualWidth];                for (int y = 0; y < MazeActualHeight; y++) {                  for (int x = 0; x < MazeActualWidth; x++) {                      MazeCoordinates[y, x] = new Coordinate(x, y);                        if (y % 2 != 0 && x % 2 != 0) {                          MazeWalls[y, x] = false;                          cellsInMaze.Add(MazeCoordinates[y, x]);                      }                        else                          MazeWalls[y, x] = true;                  }              }          }            public override void RemoveWalls(int wallsToRemove) {              int wallsRemoved = 0;                while (wallsRemoved < wallsToRemove) {                  int xPos = rgen.Next(1, MazeActualWidth - 1);                  int yPos = rgen.Next(1, MazeActualHeight - 1);                  Coordinate cellToRemove = new(xPos, yPos);                    if (IsWall(cellToRemove)) {                      MazeWalls[yPos, xPos] = false;                      wallsRemoved++;                  }              }          }          private bool IsWall(Coordinate cell) {              if (MazeWalls[cell.Ypos + 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == false                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == true                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == true) {                  return true;              }              else if (MazeWalls[cell.Ypos + 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos - 1, cell.Xpos] == true                  && MazeWalls[cell.Ypos, cell.Xpos + 1] == false                  && MazeWalls[cell.Ypos, cell.Xpos - 1] == false) {                  return true;              }              else return false;          }      }  } |