Panoptes Technical Specification

Student 1: Malachy Byrne - 19463716
Student 2: Gary Murphy - 19310381
Project Supervisor: Stephen Blott

Abstract

When managing large systems, it is common for there to be many services, each with their own management interface. This can cause some amount of confusion for the people in charge of these systems, which is why there are many services that aggregate these interfaces into one place. However, these services are not ideal, as users of those services will often run into the issue that they are maintaining something that is not supported by the service.

Panoptes is a web interface designed in a similar fashion, but rather than natively supporting a limited number of services, it natively supports no services and can be extended through the use of “modules”; interfaces designed to work with Panoptes. These modules can use a combination of text, forms, graphs, and tables to design an interface for use with Panoptes, which will render the interface as a web page. As a security measure, Panoptes uses Docker to isolate the modules, preventing them from accessing sensitive information unless granted access by the user. The containers the modules run on are placed into their own network along with the backend for Panoptes, allowing the module and application to have a direct line of communication to each other. The network by default is unable to access anything external to the network, preventing them from communicating with each other. With these features, Panoptes is able to greatly assist administrators and keep security risks to a minimum.

1. Introduction

1.1 Overview

Panoptes is an extensible, modular web interface designed to help administrators with management of services. It is built using a .NET Blazor frontend and a Go backend. It uses modules written by users to manage services, rather than supporting them itself. This allows the application to be much more lightweight and deployed very easily, as it is simply two small containers rather than a monolith deployment. These modules are isolated and must be explicitly granted access to any resources, allowing for a high level of security, as it is easy to ensure they do not have access to sensitive information. Modules can render a page from a list of components including forms, graphs, tables, and text; allowing a user to view statistics at a glance and send requests back to the module.

1.2 Motivation

The motivation for this project came from our own experience managing servers, where we were searching for a web based admin panel that could be used to manage some more obscure services we were running, but were unable to find any. Additionally, we both use a program known as Foundry, which offers as a main benefit the ability to create and install modules to change the behaviour of the system, which gave us the inspiration to apply a similar model to an admin panel.

1.3 Business Context

There are several scenarios we identified where businesses could take advantage of Panoptes. The first and most obvious one is when a business has several custom or obscure services in their stack where a more traditional admin panel may not support all of their uses.

1.4 Glossary

Term	**Description**
.NET	A development platform built by Microsoft consisting of languages such as C#, F#, and VB.NET
Blazor	A framework for front end design in .NET
C#	An object oriented programming language designed for .NET
Container	A process that is isolated from other processes on the host machine
Docker	A popular tool for management of containers
Echo	A library for building HTTP servers in Go
Go	A programming language built by Google
JSON	JavaScript Object Notation - a data format used for data serialisation based on JavaScript notation
Protobuf	Protocol Buffers - A data format used for data serialisation which uses schemas to ensure programs on both ends know the structure of the data

2. Research

2.1 Docker

The core part of our project is the way we use Docker. To build this project, we had to do research into Docker container and network management, as the way we were using it was quite different from the ways in which we had used Docker previously. A significant part of our development time went into how best to utilise Docker for our purposes.

The core part of our research focused on dynamically changing a docker container, ideally without restarting it, which we discovered was possible for our purposes. We then had to research how we could install modules, and decided that the best way would be to have each module as its own image on a registry which would be pulled as part of the installation process.

2.2 Blazor

The next part of our research focused on Blazor, and more specifically how we could dynamically render our tables and charts. Neither of us had a lot of experience using Blazor, so we had to search for ways to do some of the more complicated aspects of our project. Fortunately, we discovered two libraries called Radzen and Apex Charts which had components that perfectly suited our use case.

Each Table and Chart is fitted with its own individual timer to update it with new information from the module without having to re-render the entire page.

2.3 Go

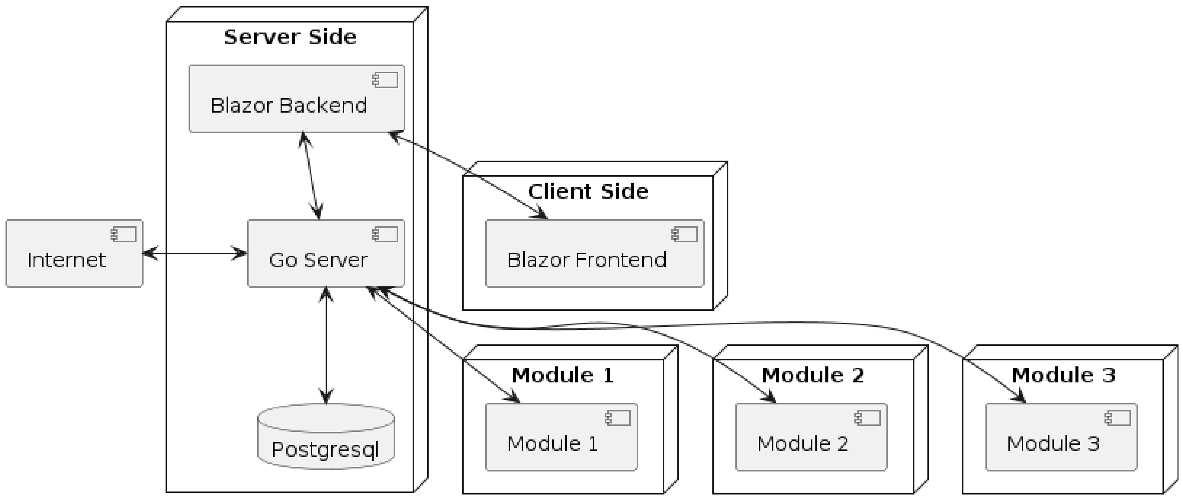
The third part of our research was focused on Go. Neither of us were very experienced in using Go, which meant we had to do some research into the best ways to make our http server. The first solution we found, Gorilla, was unfortunately retired partway through our project, which meant we had to find a new library to build our server in. Fortunately, we discovered another library called Echo which we were able to rebuild with.

2.4 Protobuf

The fourth and final part of our research was into protobufs. We had to figure out how we would format our protobufs to ensure effective communication between the frontend and backend. Ultimately, we decided to switch from protobuf serialisation to JSON as we encountered too many issues with protobufs that while solvable would have taken up too much of our time.

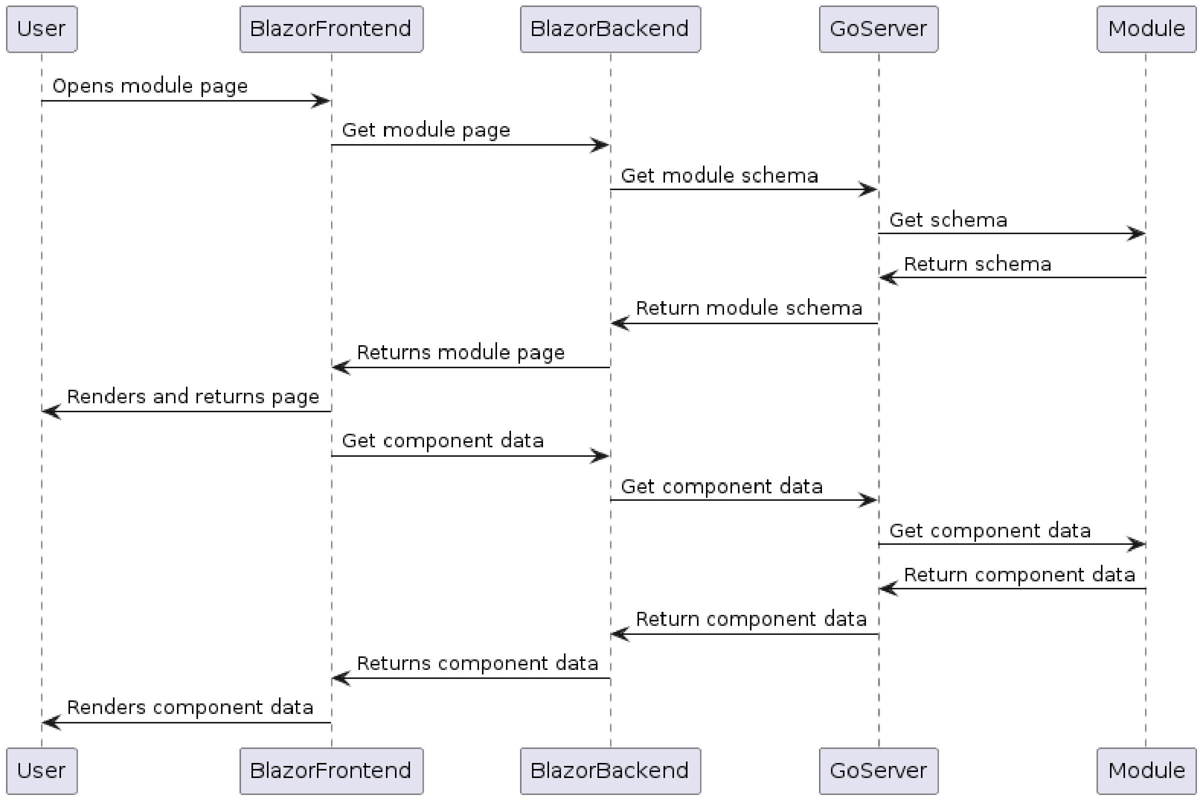
3. Design

3.1 System Architecture



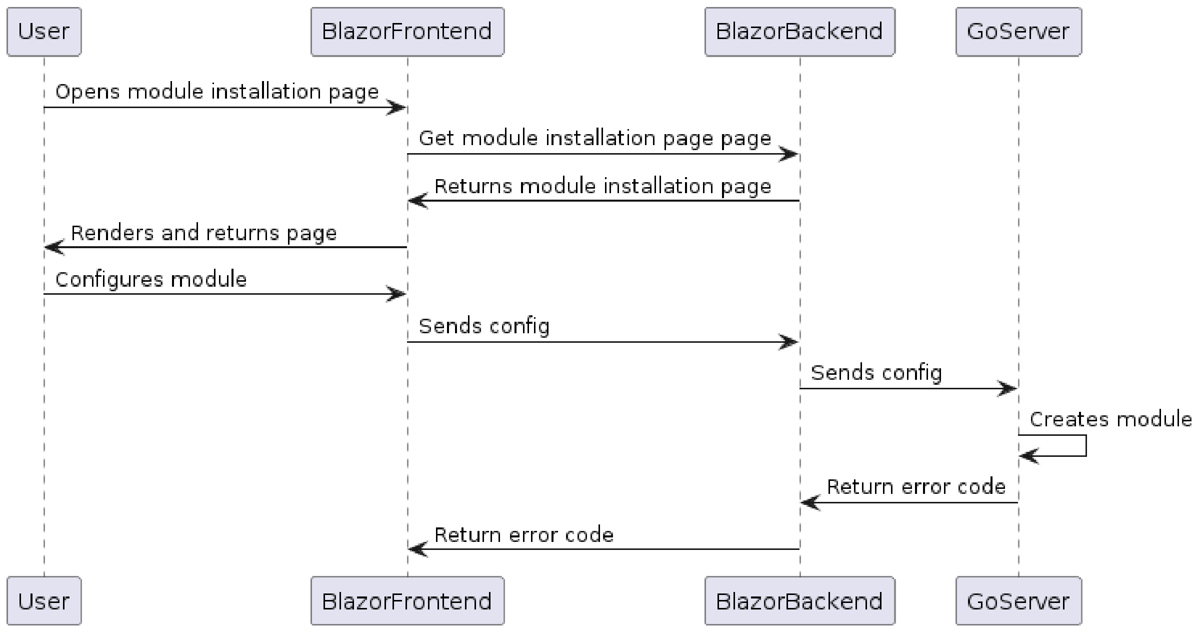
The above diagram shows the design of Panoptes at a high level. Users interact on the client side through the Blazor Frontend, which goes to the server side through the Blazor Backend. The Blazor Backend then communicates with the Go Server, which sends data to be stored in the postgresql database and sends requests to modules. The modules then process these requests before sending them back to the Go Server, which sends them to the Blazor Backend, which sends them to the Blazor Frontend. The modules are isolated from each other and from everything else, meaning they can only access things through the Go Server, which takes requests from them and forwards them on to what they requested.

3.2 Open Module Page



The above sequence diagram shows the flow of data when a user opens a module page. They send their request to the frontend of the Blazor application, which asks the backend for the module data to render. The backend then asks the Go server for the module schema. The Go server asks the module for its schema, which returns the schema. The Go server then returns the schema to the backend, which returns the components to the frontend, which then renders the page for the user. The Blazor frontend then asks the backend for the component data for each component on the page, which forwards the requests to the Go server, which asks the module. The data is then returned along the chain to the Blazor frontend, which renders the component data and displays it to the user.

3.3 Install Module

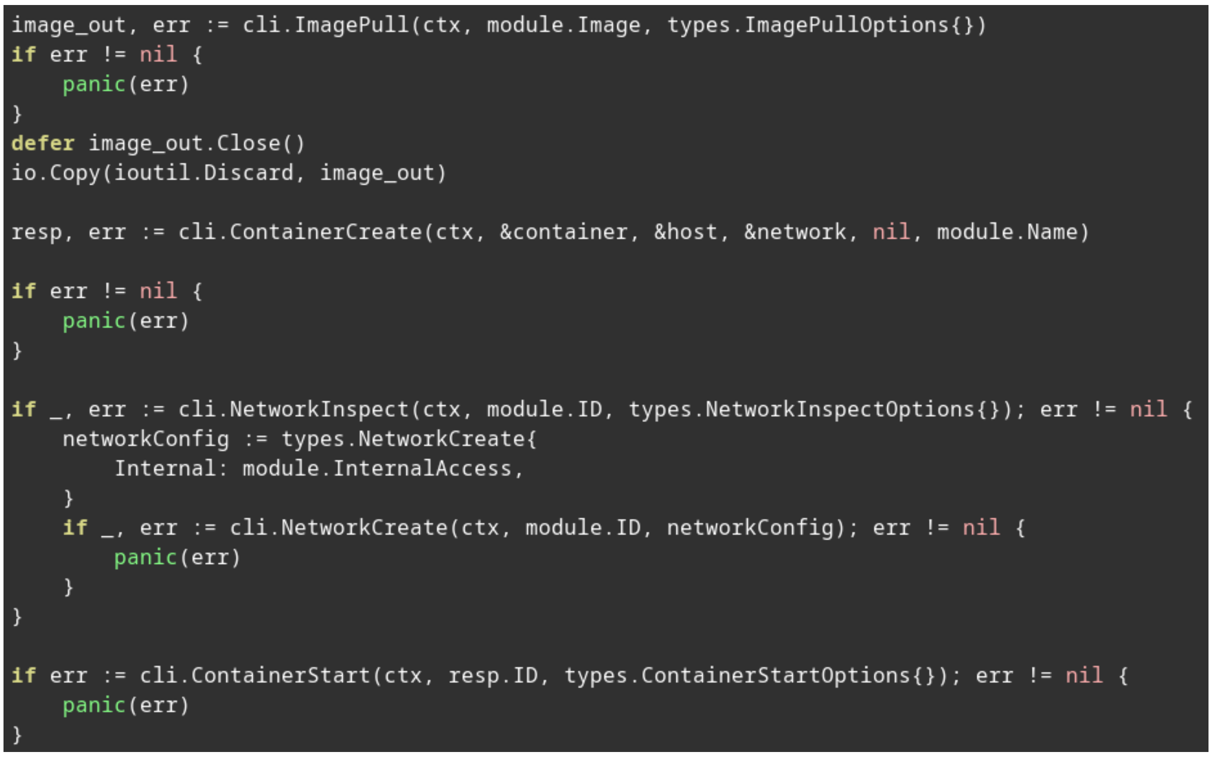


The above sequence diagram shows the flow of data when a user installs a module. The user opens the installation page on the frontend, which requests the page data from the backend. The backend sends the page data to the frontend, which renders it for the user. The user then configures the module and sends that to the frontend, which sends the configuration to the backend. The backend sends the configuration to the Go server, which creates the module as a docker container. The go server then returns an error code to the Blazor backend which forwards it to the frontend.

4. Implementation

4.1 Module Installation

The first thing a user can do when they install this application is install a module. When the user opens this page, they are given a prompt for the configuration of the module. They enter the details into this form and it is given to the backend to be processed. The backend takes this information and creates a configuration, pulling the image if needed and handling the network and container configuration.



```go
image_out, err := cli.ImagePull(ctx, module.Image, types.ImagePullOptions{})
if err != nil {
    panic(err)
}
defer image_out.Close()
io.Copy(ioutil.Discard, image_out)

resp, err := cli.ContainerCreate(ctx, &container, &host, &network, nil, module.Name)

if err != nil {
    panic(err)
}

if _, err := cli.NetworkInspect(ctx, module.ID, types.NetworkInspectOptions{}); err != nil {
    networkConfig := types.NetworkCreate{
        Internal: module.InternalAccess,
    }
    if _, err := cli.NetworkCreate(ctx, module.ID, networkConfig); err != nil {
        panic(err)
    }
}

if err := cli.ContainerStart(ctx, resp.ID, types.ContainerStartOptions{}); err != nil {
    panic(err)
}
```

When the user attempts to create a module, the go server will attempt to pull the image, create the container, create the network if needed, and start the container.

4.2 Get Module Schema

When a user attempts to load data from a module, they send a request through the Blazor backend and Go server to the module for its schema. The module then returns its schema, sending it back through the Go server and Blazor backend.

```go
func getSchema(c echo.Context) error {
    key := c.Param("container")


    requestURL := fmt.Sprintf("http://%s/schema", key)
    resp, err := http.Get(requestURL)
    if err != nil {
        log.Fatalf("Request to container failed: %v", err)
    }

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        log.Fatalf("Failed to print Body: %v", err)
    }

    return c.JSONBlob(resp.StatusCode, body)
}
```

The Go server sends a request to the module for its schema, then copies its data into a response to the Blazor backend's request.

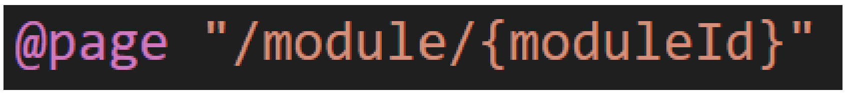
4.3 Get Component Data

When a user attempts to render a module, they need to grab the component data. The component data is retrieved similarly to the schema, sending it through the Blazor backend and the Go server. When the frontend gets this data, it will reload the charts every 5 seconds with new data to ensure data displayed is up to date.

```csharp
protected override void OnAfterRender(bool firstRender)
{
    if (firstRender && !timerInitialized)
    {
        timerInitialized = true;
        timer = new Timer(5000);
        timer.Elapsed += async delegate { await UpdateChartSeries(); };
        timer.Enabled = true;
    }
}

private async Task UpdateChartSeries()
{
    string module = "localhost:8080";
    var response = await httpService.GetAsync<GraphComponent>($"http://localhost:10000/{module}/{ChartId}");
    StatsList = response.Stats;
    await chart.UpdateSeriesAsync(true);
    await InvokeAsync(() => StateHasChanged());
}
```

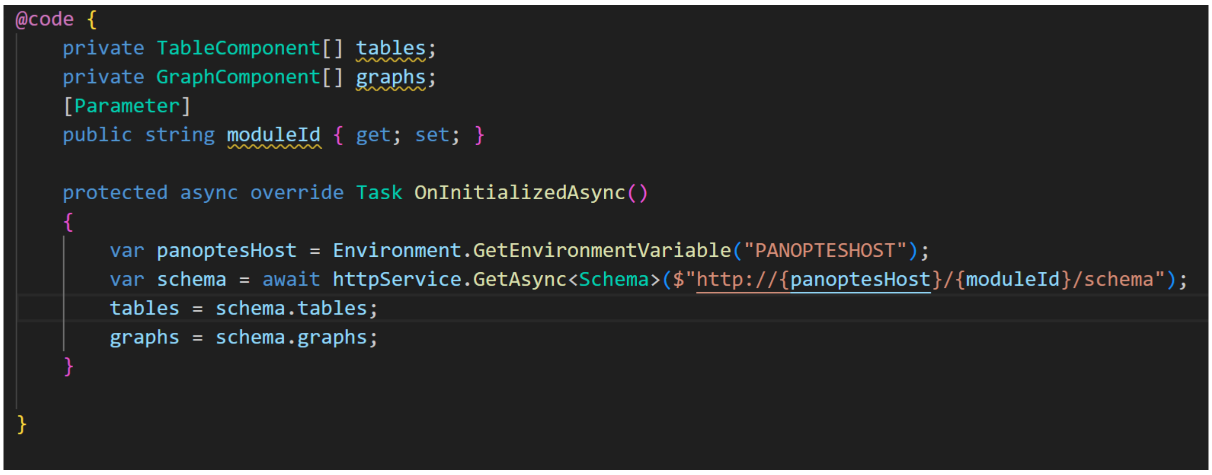
4.4 Display Module Components

Each Module can be dynamically displayed using the same Blazor Component. To Achieve this we first had to parameterize the route path



```
@page "/module/{moduleId}"
```

This parameter is received from our database and the page can be accessed from our navigation panel.



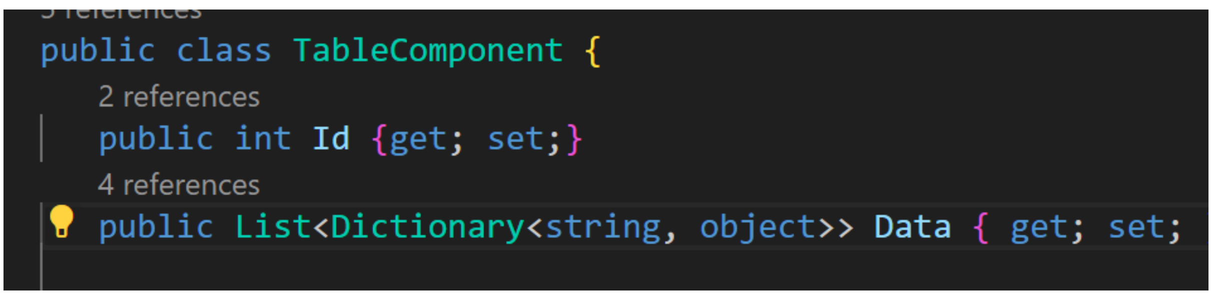
```
@code {
    private TableComponent[] tables;
    private GraphComponent[] graphs;
    [Parameter]
    public string moduleId { get; set; }

    protected async override Task OnInitializedAsync()
    {
        var panoptesHost = Environment.GetEnvironmentVariable("PANOPTESHOST");
        var schema = await httpService.GetAsync<Schema>($"http://{panoptesHost}/{moduleId}/schema");
        tables = schema.tables;
        graphs = schema.graphs;
    }

}
```

The Schema is then retrieved using an synchronous Get request to a dynamic url made up of the ‘panoptesHost’ environment variable, and the moduleId parameter we receive from the route.

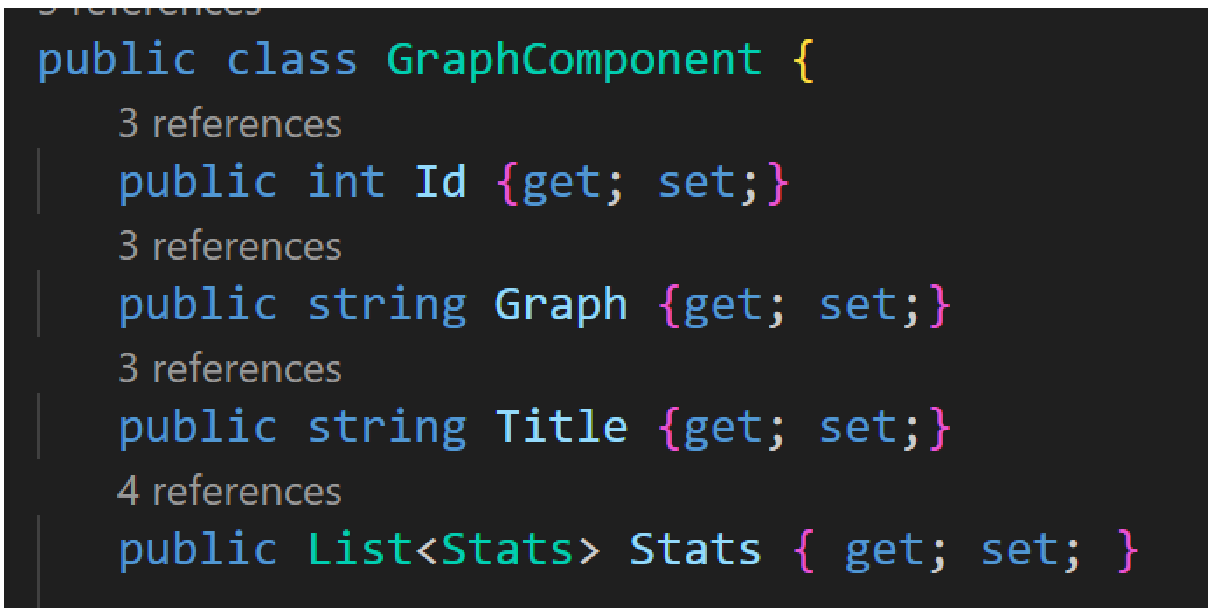
From there we receive Json detailing the types of charts we want and their associated data, and the data needed to populate tables and we generate the UI.

4.5 Dynamic Charts and Tables

We provide access to the DynamicChart and DynamicTable components for modules to use to use.

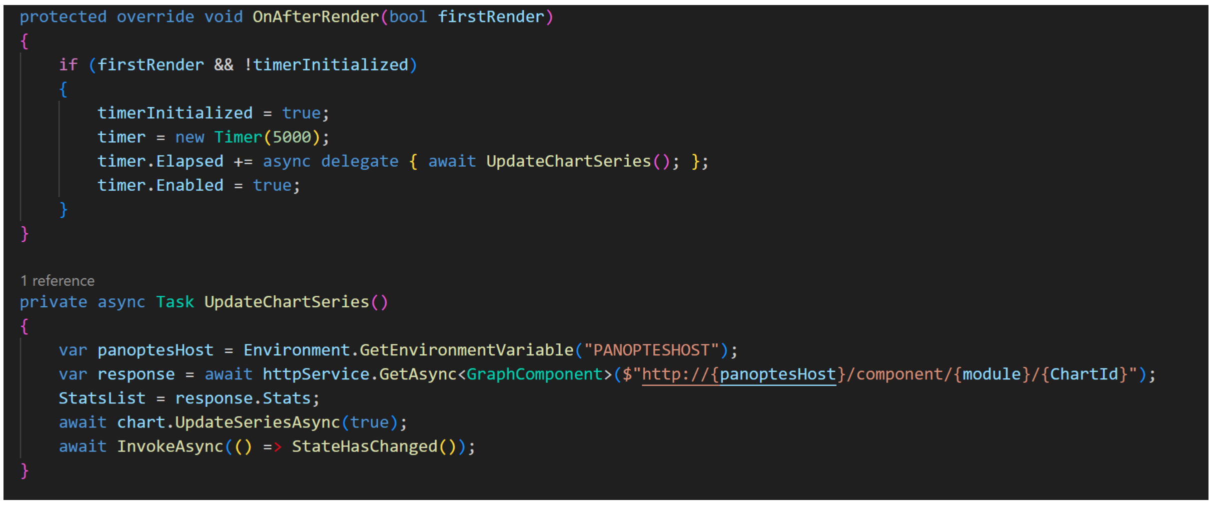


```
public class TableComponent {
    2 references
    public int Id {get; set;}
    4 references
    public List<Dictionary<string, object>> Data { get; set;
```



```
public class GraphComponent {
    3 references
    public int Id {get; set;}
    3 references
    public string Graph {get; set;}
    3 references
    public string Title {get; set;}
    4 references
    public List<Stats> Stats { get; set; }
```

By sending Json data to our Blazor backend in the above formats we are able to generate tables and charts based on the information received. Each Component generated this way has its own individual timer which starts on initialisation that allows them re-render themselves to receive updates.



```
protected override void OnAfterRender(bool firstRender)
{
    if (firstRender && !timerInitialized)
    {
        timerInitialized = true;
        timer = new Timer(5000);
        timer.Elapsed += async delegate { await UpdateChartSeries(); };
        timer.Enabled = true;
    }
}

1 reference
private async Task UpdateChartSeries()
{
    var panoptesHost = Environment.GetEnvironmentVariable("PANOPTESHOST");
    var response = await httpService.GetAsync<GraphComponent>($"http://{panoptesHost}/component/{module}/{ChartId}");
    StatsList = response.Stats;
    await chart.UpdateSeriesAsync(true);
    await InvokeAsync(() => StateHasChanged());
}
```

Here is a code snippet from our Dynamic Chart Component. Here we see that after the first render a timer is initialised and every 5 seconds the UpdateChartSeries() method is called which makes a request to the module to receive any updated information.

4.6 User Authentication and Authorization

Our Go Server as endpoints for registering, logging in a user and logging out a user.

When the registration endpoint receives data from the frontend it first makes sure this user doesn't already exist in our database. If it is a brand new user then it hashes their password and makes a new entry in our User table.

When a user wants to log in, the go server receives the entered username and password from Blazor. Checks that the user has registered, compares the hashed password with the one stored in our database and then generates a signed jwt token string. Blazor receives this information and stores it in local storage for future authorization checks.

The logout endpoint simply deletes the generated jwt token string and cookie.

5. Problems Solved

5.1 Security

A major issue with the idea of running these modules, which by their nature can often be someone else's unverified code, is that it can introduce a security flaw into the system. To help mitigate that, we decided to isolate each module as a container to prevent it from accessing sensitive data. Instead, the module requires a user to manually grant it access to a piece of data or to an external service. This helps to prevent the module from stealing sensitive information or from destroying systems unintentionally by ensuring that the module can only access what it has been granted explicit permission to access.

5.2 Networking

Another issue we had with our program was the networking architecture. With our plan of isolating everything for security, we would have to ensure that everything was able to communicate with what it needed. To do this, we implemented our network in a way where each module was in its own container on its own network, which by default had no access to anything outside the network. The Go backend server would then attach its container into the same network, ensuring that it was able to communicate with each module and vice versa.

5.3 Dynamic Components

To ensure our components worked with any module they had to be dynamic in their design and typing. We achieved this by allowing users to choose the type of data chart by adding a new field to our GraphComponent class and by allowing our Table to work with any data type.

5.4 Rendering updates

We wanted components to be able to receive updates from modules. This proved difficult as the component had to be re-rendered to show the update. We initially just re-rendered the whole page every 5 seconds however this was not conducive to a good user experience so instead we attached a timer to each individual component and added methods so that they can re-render themselves individually.

6. Future Work

6.1 Roles

In the future we would like to add a role hierarchy to our users so that certain components or modules could only be accessed by someone with a high enough role.

This would make our program more desirable in an enterprise environment. I.e the junior developer cant accidentally delete the container.

6.2 Module UI Configuration

Like the actual module config page we have where you can change the image, volumes, environment variables… etc

We planned to make a similar style page where you could select an installed module and edit the already existing components in the module i.e change chart type or data source. And also be able to add a new component to the Modules UI page.

6.3 Module Communication

We also would like to create a dynamic form component that can be implemented into your module to receive whatever input the user specifies. This could be used to send commands to, and communicate with the module allowing for wider functionality.

We had started to implement this feature but could not commit to finishing it with our time schedules.

6.4 Module Lists

Another feature we would consider adding in the future is grouping modules with shared functionality into lists and saving them, allowing you to install them all with one command.

7. Continuous Integration and Testing

We made use of a multi stage gitlab ci pipeline which ran after each push to our repository.

The first stage simply built our go server and blazor project.

The second stage performed the unit tests we had created for both the front and backend. Our testing was primarily unit testing. We tested our various components to ensure they rendered correctly and accepted input directly. We also tested functions in our backend related to our postgresql database.

To make sure our tests did not interfere with our production environment we created a PanoptesTest project which can render components from our production code and run tests on it. We also spin up a mock database environment and populate it with mock data to run our tests on.

Had we more time we would have liked to have carried out more robust end-to-end testing that was automated by our pipeline, instead we resorted to ad-hoc testing to ensure everything continued to work as expected.