Technical Guide

- **Project Title**: PlusVasis
- **Student 1:** Jason Henderson, 19309916
- **Student 2:** Conor Joyce, 19425804
- **Supervisor:** Dr Stephen Blott
- **Date Completed:** 05/05/23

Table of Contents

- Technical Guide
- Table of Contents
- Introduction
  - Overview/Motivation
  - Glossary
  - Research
- High Level Design
  - User Login/Register
  - Container Creation
  - Container Update
  - Container Stop
  - Container Start
  - Container Restart
  - Container Deletion
  - Create and Expose a Web Server
  - Import Containers from docker-compose file
  - Start a Live Terminal Session
  - Real-Time Log Streaming
- System Architecture
- Implementation
  - Backend
    - Structure
    - Main
    - Routes
    - Middleware
    - Handlers/Controllers
    - Templates
  - Frontend
    - Structure
    - Routes
    - Components
    - Utils
    - Types
    - Firebase
    - Stores
  - Nomad
    - Topology
    - Server-Client
    - Client
    - Example Nomad Job
  - Traefik
    - Deployment
    - Configuration
  - Webhook
    - Configuration
  - Cloudflare
    - A records
    - CNAME records

Introduction

Overview/Motivation

PlusVasis is a container management and orchestration platform that provides developers with a modern and intuitive interface to manage their containers. The platform consists of a Svelte-based frontend and a backend REST API built using the Golang Echo framework. PlusVasis communicates with a personal Nomad server to create and manage container jobs, making container management a breeze for developers.

The platform features seamless networking, security, and scalability, allowing developers to focus on building their applications instead of worrying about complex backend processes. PlusVasis is built using modern frameworks and infrastructure solutions, ensuring optimal performance and reliability.

Developers can easily manage their containers using PlusVasis's intuitive interface, which streamlines container management and orchestration. PlusVasis enables developers to focus on building their applications, while the platform takes care of the container management and orchestration.

The name "PlusVasis" is derived from the Latin words "plus" meaning "more" and "vasis" meaning "containers". This name reflects the platform's goal of simplifying container management and enabling developers to manage more containers with ease.

The idea for this project came from real problems encountered by Conor, who was a Systems Administrator in DCU's NetSoc - Redbrick. It was often requested by users of Redbrick's infrastructure if they could host their websites, game servers, personal projects etc. on Redbrick, but this required manual effort from the admins. It would have been great to allow users to deploy whatever they like, whenever they like, all by themselves.

Both of us felt motivated by this idea and believed we could create something easy to use that met this goal sufficiently. We decided to use Nomad as our job orchestrator/runner because of its ability to scale horizontally, which is very important for a project where users can potentially use a lot of resources such as memory and CPU time.

Glossary

SvelteKit: A web application framework based on the Svelte JavaScript framework, designed to simplify the process of building web applications.

Vite: A build tool and development server that enables fast and efficient web development, often used in conjunction with SvelteKit.

TypeScript: A programming language that adds static typing to JavaScript, providing enhanced tooling, code organization, and error detection.

Backend: The server-side component of your application, responsible for processing requests, handling data, and communicating with other services.

REST API: An architectural style for designing networked applications, where the API uses standard HTTP methods (such as GET, POST, PUT, DELETE) to perform actions on resources.

API middleware: Software components that sit between an application and an API, providing additional functionality or processing, such as authentication, caching, or logging.

Go (Golang): A programming language developed by Google that is known for its simplicity, efficiency, and strong support for concurrent programming.

Echo framework: A lightweight web framework for Go that simplifies the development of RESTful APIs by providing routing, middleware, and request handling features.

Frontend: The client-side component of your application, responsible for rendering the user interface and handling user interactions.

Testing: The process of verifying the correctness and functionality of your application through various tests, such as unit tests, integration tests, and end-to-end tests.

Vitest: A testing framework or library used for testing SvelteKit applications, providing capabilities for testing components and simulating user interactions.

Error handling: The process of detecting, managing, and recovering from errors or exceptions that occur during the execution of your application.

HTTP status codes: Standard codes used to indicate the status of HTTP requests and responses, providing information about the success or failure of a request.

Cypress: A JavaScript end-to-end testing framework that allows you to write and execute tests in a browser-like environment, simulating user interactions and verifying the behavior of your application.

Docker: A platform for developing, shipping, and running applications in containers, which are lightweight, portable, and self-sufficient. Docker allows you to package an application with all its dependencies and configurations, ensuring consistency across environments.

Docker images: Read-only templates that contain a set of instructions for creating a container that can run on the Docker platform. Images include application code, libraries, tools, dependencies, and other files needed to make an application run. They consist of multiple layers, each one originating from the previous layer but being different from it. Images can be stored in private or public repositories, such as Docker Hub.

Docker containers: Instances of Docker images, which are lightweight, portable, and self-sufficient. Containers represent the execution of a single application, process, or service and consist of the contents of a Docker image, an execution environment, and a standard set of instructions.

Docker volumes: Writable filesystems that containers can use, allowing programs to access a writable filesystem since images are read-only. Volumes live in the host system and are managed by Docker.

docker-compose: A command-line tool and YAML file format for defining and running multi-container applications. Developers can define a single application based on multiple images with one or more .yml files and deploy the whole multi-container application with a single command (`docker-compose up`).

Nomad: A flexible workload orchestrator that enables an organization to automate the deployment of any application on any infrastructure at any scale.

Nomad jobs: Configuration files or definitions that describe how containers should be run and managed within the Nomad cluster, specifying parameters, resources, dependencies, and other details.

Nomad allocations: Individual instances of tasks that are scheduled by Nomad and run within a Nomad client. Allocations are the smallest unit of work in a Nomad cluster.

Traefik: A modern HTTP reverse proxy and load balancer that makes deploying microservices easy by dynamically managing routing, load balancing, and SSL/TLS termination.

SSL certificates: Digital certificates that provide secure and encrypted communication between a client and a server, ensuring the authenticity of the server and the integrity of the transmitted data.

Cloudflare: A global cloud network platform that offers various services, including DNS resolution, DDoS protection, and web security.

A records: DNS records that map a domain name to an IPv4 address, allowing users to access a website or service using a domain name instead of an IP address.

CNAME records: DNS records that map one domain name to another, effectively creating an alias for the target domain name.

Webhooks: User-defined HTTP callbacks that allow real-time notifications and updates when an event occurs in a web application or service.

Research

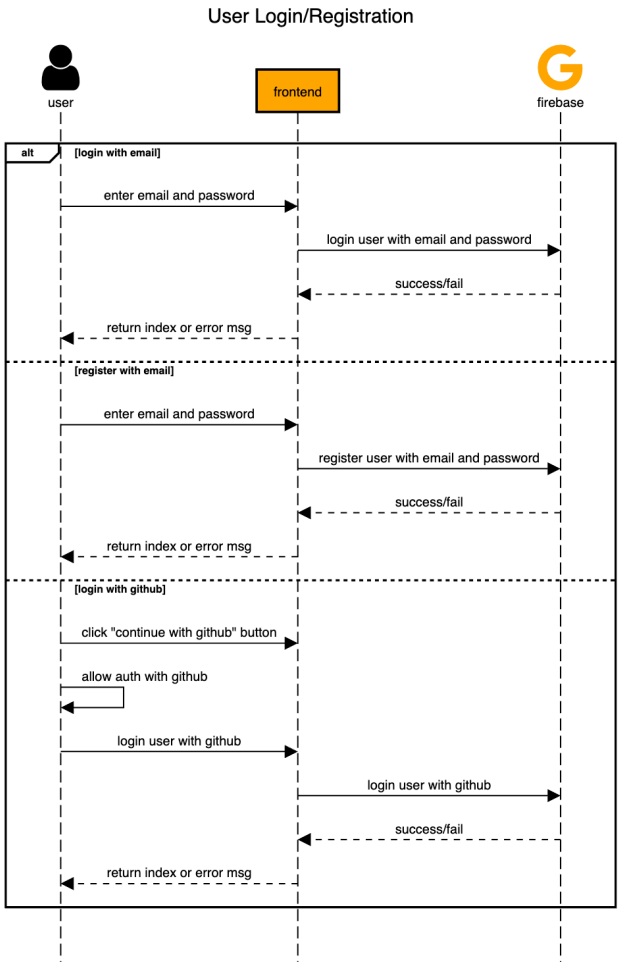
Developing PlusVasis required extensive research to identify the right tools and technologies to use. We began by evaluating different backend frameworks and ultimately chose Echo for its ease of use, flexibility in terms of middleware support, and the new learning opportunities it provided. Further research was required when developing our frontend framework, particularly in regards to server-side rendering in Svelte. Throughout the development process, we encountered challenges such as difficulty finding relevant resources and unexpected roadblocks in implementing certain features.

To ensure the quality of the platform, we also invested time in researching the best testing practices. This included writing unit and integration tests for our REST API, component testing for our frontend, and implementing a series of end-to-end system tests using Cypress. One of the final research efforts was integrating Docker Compose and learning how to create nomad jobs for containers by using Docker Compose.

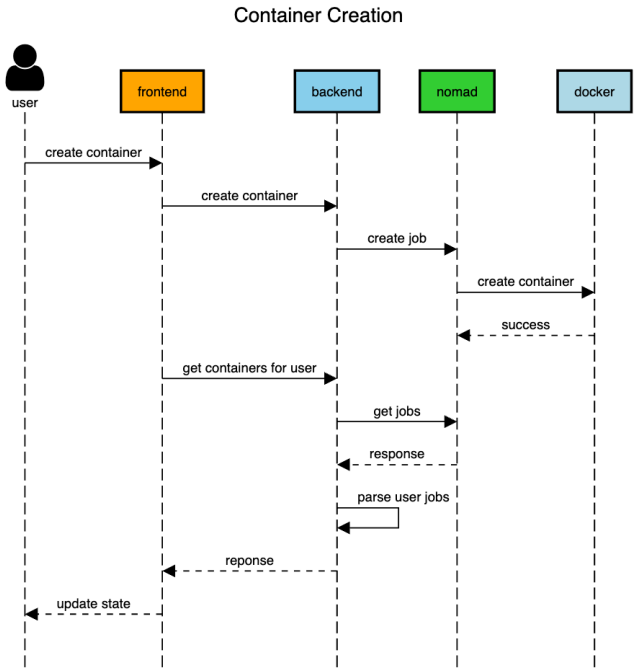
Overall, the research we conducted enabled us to make informed decisions about technology choices and improve the quality of the platform. For example, using Echo was very powerful, yet still provided a good developer experience, while testing best practices helped ensure a high level of reliability and robustness for PlusVasis.

High Level Design

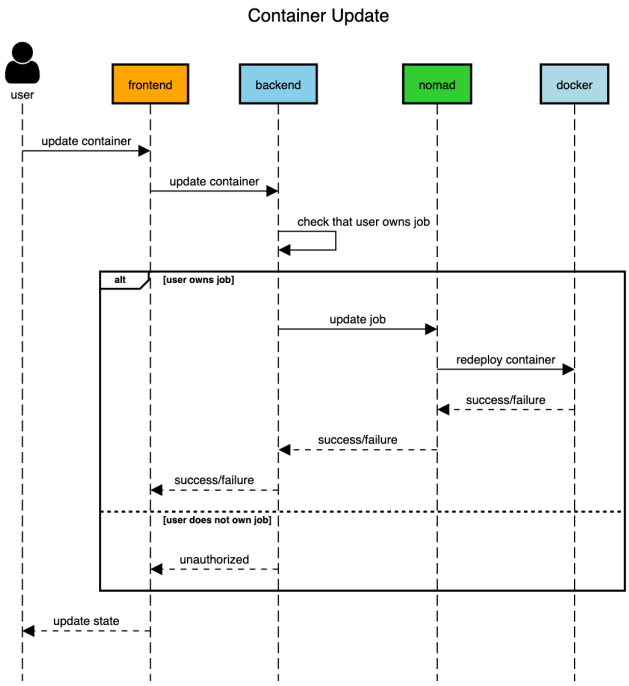
User Login/Register



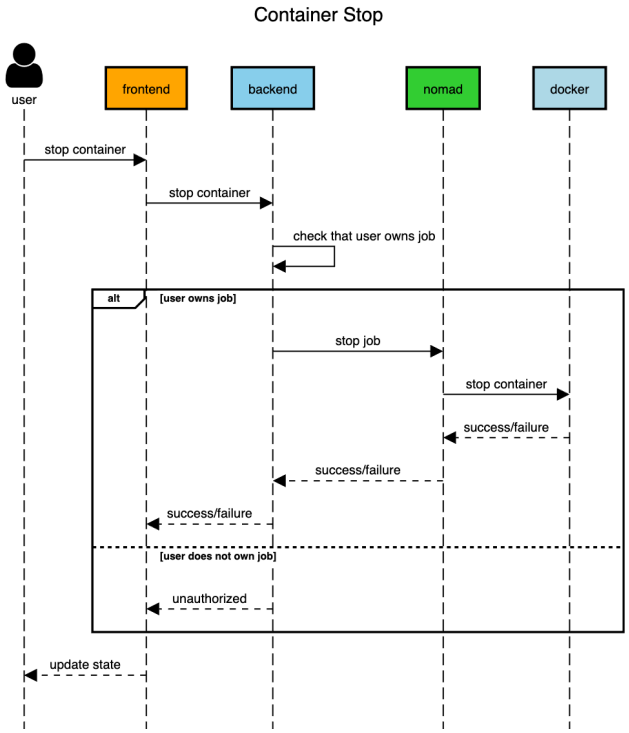
Container Creation



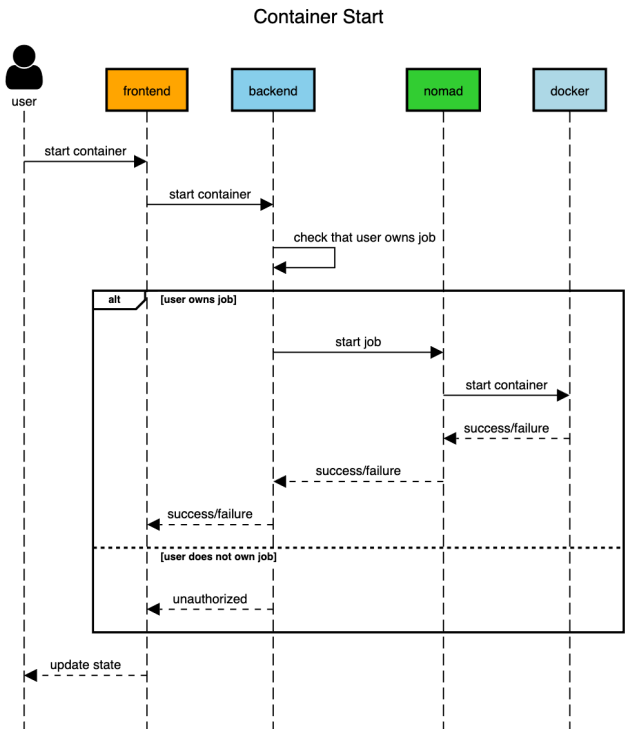
Container Update



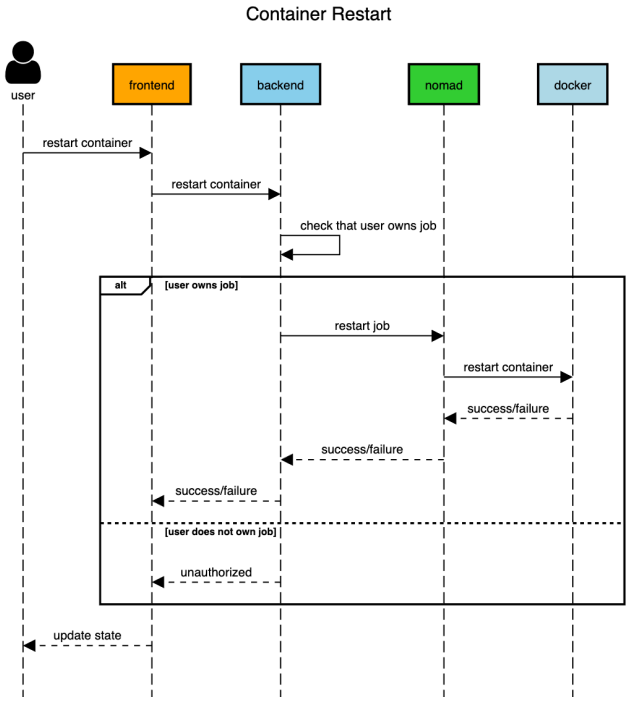
Container Stop



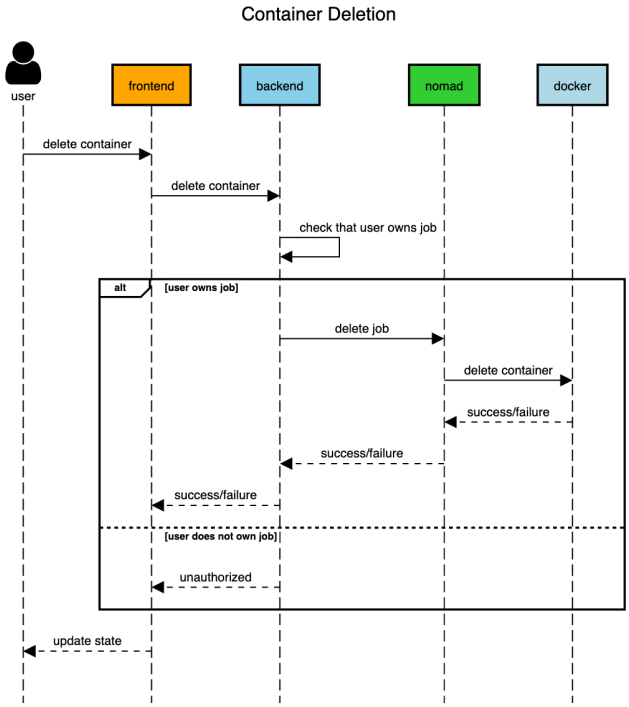
Container Start



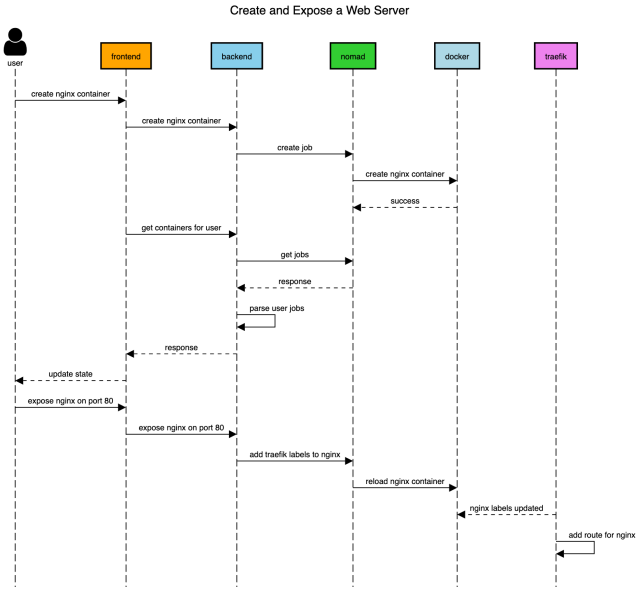
Container Restart



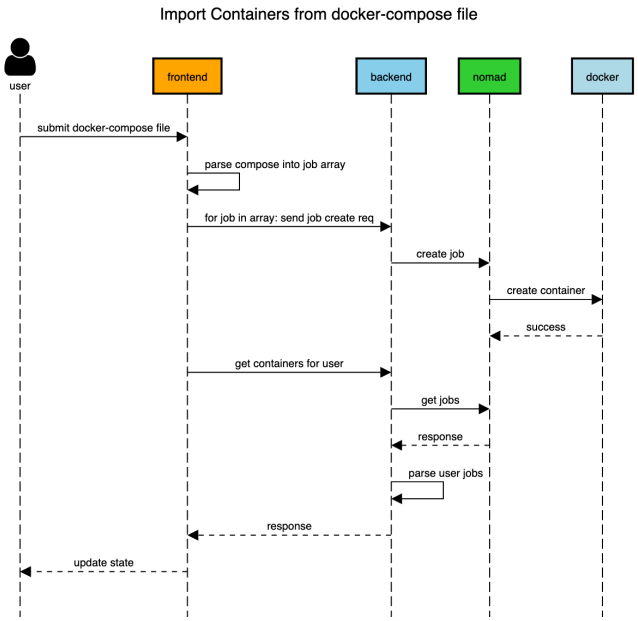
Container Deletion



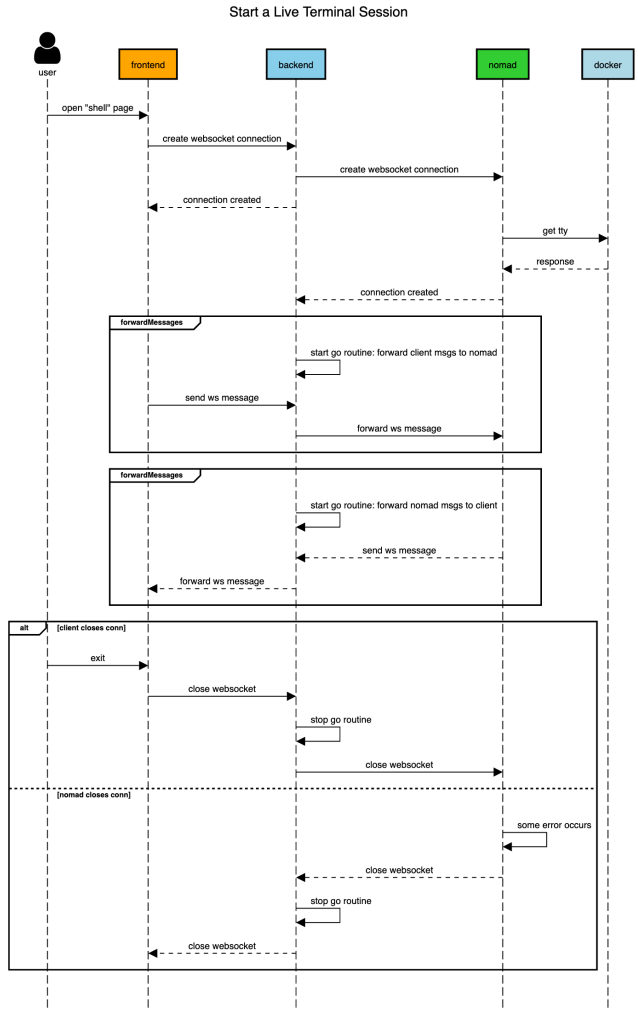
Create and Expose a Web Server



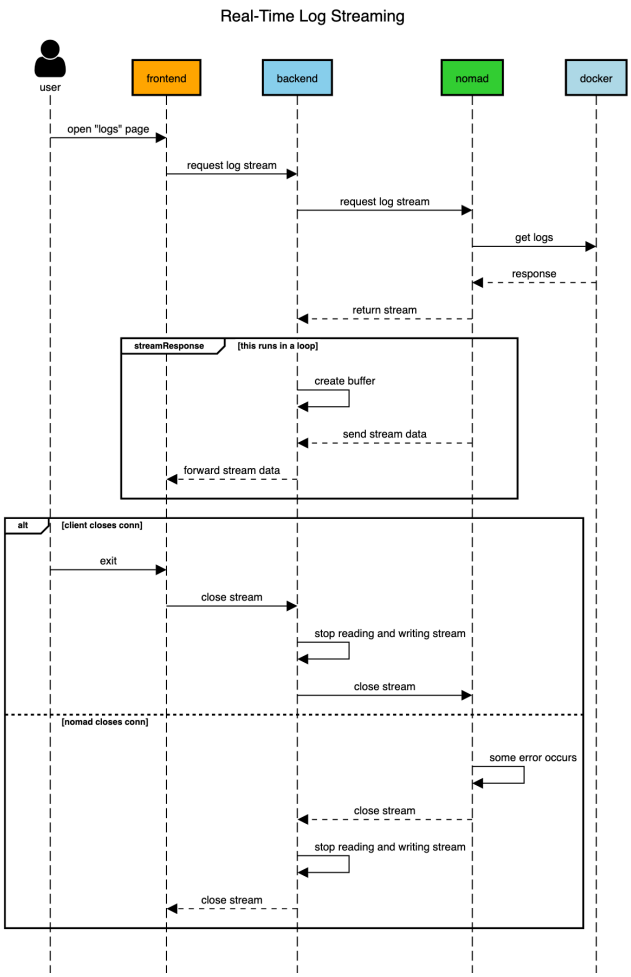
Import Containers from docker-compose file



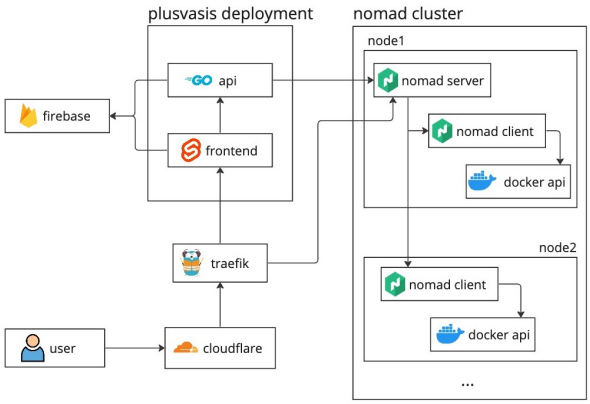
Start a Live Terminal Session



Real-Time Log Streaming



System Architecture



PlusVasis consists of the following components:

- **Go REST API**: Acts as a middleman between Nomad and the frontend. The aliveness of this can be checked at https://api.plusvasis.xyz/health/ (https://api.plusvasis.xyz/health/)
- **SvelteKit frontend**: Provides the web UI using Svelte components, SvelteKit for routing and server-side rendering, and Typescript for a type-safe implementation. Accessible at https://app.plusvasis.xyz/ (https://app.plusvasis.xyz/)
- **Nomad**: Orchestrates jobs on the cluster by communicating with clients and the Docker API for each host/node. Also provides functionality for job management and real-time monitoring.

- **Traefik**: Reverse proxy for the whole system. Routes requests to the api, frontend, nomad and any containers that users expose on PlusVasis. Provides integration with both Nomad and Docker to do all of this.
- **Docker**: Driver for executing Nomad jobs. A docker image specified in a Nomad job will then be started on one of the hosts on the Nomad cluster using Docker.
- **Cloudflare**: DNS resolution for the whole system. Provides SSL certificates for all the web facing components and user containers. Also acts as a proxy to minimize threats from bad actors, e.g. DDoS protection.
- **Firebase**: Authentication layer used by both the frontend and backend to allow separation between user resources and security.

Implementation

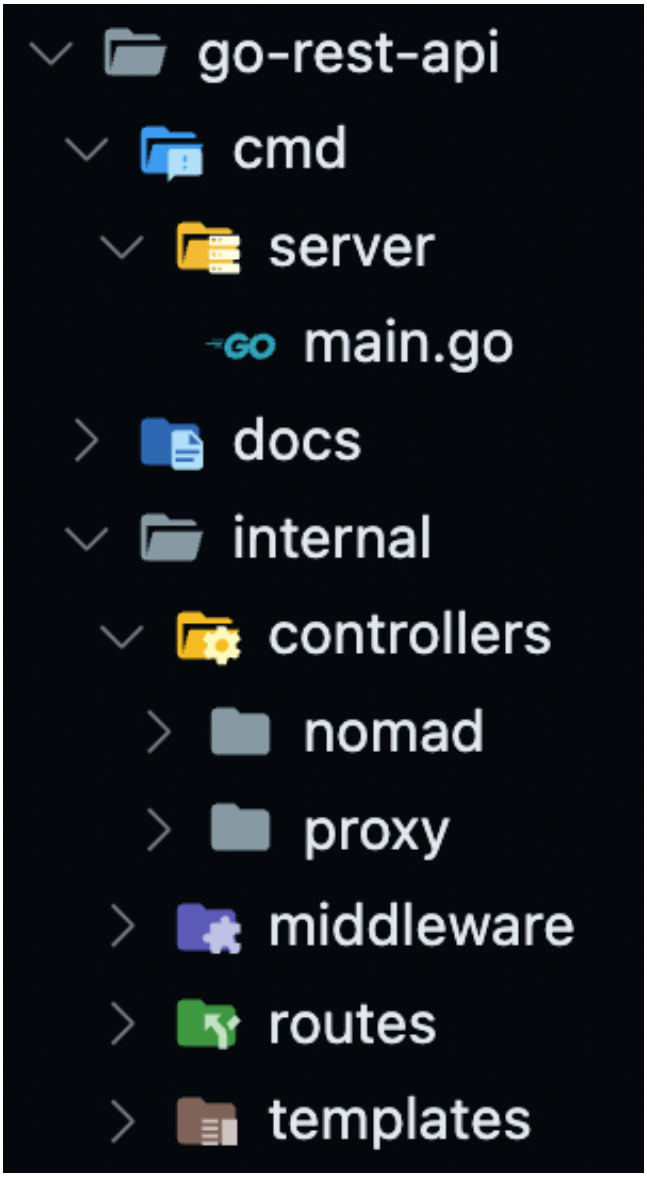
Backend

The backend consists of a REST API written in Go using the Echo framework. The backend can be seen as the middleman of our application. It allows communication between our frontend and Nomad by communicating with them both directly. Go is a fast and efficient programming language, making it well-suited for building high-performance APIs. Additionally, Echo is a lightweight and easy-to-use web framework that provides a simple and intuitive way to create RESTful APIs.

It includes features such as routing, middleware, and request handling that make it easy to build complex APIs with minimal boilerplate code. Echo also has excellent documentation and a large community of developers, making it easy to find help and support when needed. Finally, Echo is a highly scalable framework, which means it can handle large amounts of traffic and is suitable for building enterprise-grade applications.

Overall, the combination of Go and Echo makes for a powerful and flexible API development platform that can meet the needs of a wide range of use cases.

Structure



Main

```go
func main() {
    e := echo.New()

    setupMiddlewares(e)
    setupRoutes(e)

    httpPort := os.Getenv("HTTP_PORT")
    if httpPort == "" {
        httpPort = "8080"
    }

    e.Logger.Fatal(e.Start(":" + httpPort))
}
```

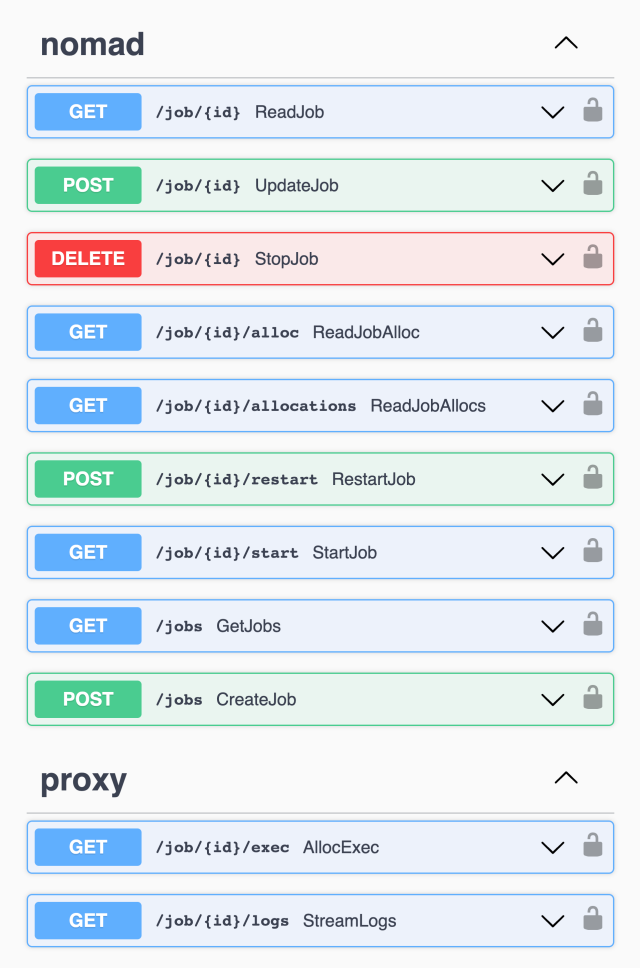
Here we setup the both the middlewares and routes used in our API.

Routes

```go
func setupRoutes(e *echo.Echo) {
    routes.HealthRoutes(e)
    routes.NomadJobs(e)
    routes.NomadProxy(e)

    e.GET("/swagger/*", echoSwagger.WrapHandler)
}
```

The setupRoutes function in our main.go file is where we provide our different route setup functions, organised by purpose - health, nomad jobs and nomad proxy.
Also here we add a route for our swagger api documentaion - which can be accessed at the following address: https://api.plusvasis.xyz/swagger/index.html
(https://api.plusvasis.xyz/swagger/index.html)

Swagger



Swagger allows us to specify schemas and documentation for all of our API routes that can be viewed in a nice web UI. This allows us to manually test our endpoints and ensure that for different inputs, we get a response that is defined in our schema and is therefore an expected response.

HealthRoutes

```go
func Health(c echo.Context) error {
    return c.JSON(http.StatusOK, struct{ Status string }{Status: "OK"})
}

func HealthRoutes(e *echo.Echo) {
    e.GET("/health", Health)
}
```

Simply a basic route that we can use to prove aliveness of the API easily.

NomadJobs

```go
func NomadJobs(e *echo.Echo) {
    controller := nomad.NomadController{Client: &nomad.DefaultNomadClient{}}

    e.GET("/jobs", controller.GetJobs)
    e.POST("/jobs", controller.CreateJob)
    e.GET("/job/:id", controller.ReadJob)
    e.DELETE("job/:id", controller.StopJob)
    e.POST("/job/:id", controller.UpdateJob)
    e.GET("/job/:id/allocations", controller.ReadJobAllocs)
    e.GET("/job/:id/alloc", controller.ReadJobAlloc)
    e.POST("/job/:id/restart", controller.RestartJob)
    e.GET("/job/:id/start", controller.StartJob)
}
```

Here we define all of our routes relating specifically to Nomad jobs and the handler functions that run whenever an endpoint receives a particular request.

NomadProxy

```go
func NomadProxy(e *echo.Echo) {
    controller := proxy.NomadProxyController{
        Client: &nomad.DefaultNomadClient{},
        Dialer: &proxy.DefaultDialer{},
    }

    e.GET("/job/:id/exec", controller.AllocExec)
    e.GET("/job/:id/logs", controller.StreamLogs)
}
```

Here we define all our two routes that provide a real time proxy layer between Nomad and our frontend. `/job/:id/exec` for example is a middleman for a two-way websocket communication channel between the frontend and Nomad, used for the terminal on our container page. `/job/:id/logs` provides a stream of data consisting of container logs that can be read from Nomad and then written to the client's buffer in real-time.

Middleware

```go
func setupMiddlewares(e *echo.Echo) {
    e.Use(middleware.CORSWithConfig(middleware.CORSConfig{
        AllowOrigins: []string{
            "http://localhost:*",
            "https://*.plusvasis.xyz",
        },
        AllowHeaders: []string{"*"},
    }))
    e.Use(middleware.Recover())
    e.Use(middleware.Secure())
    e.Use(middleware.RateLimiter(middleware.NewRateLimiterMemoryStore(20)))

    e.Use(customMw.Logger())
    e.Use(customMw.Firebase())
}
```

The setupRoutes function in our main.go file is where we setup all of our middleware. Some are native to Echo and others we wrote ourselves.

Echo Middleware

- CORS: allows us to define CORS rules such as allowed request origins and allowed headers. This helps us improve security and reduce the likelihood of bad actors trying to access our API.
- Recover: recovers from panics anywhere in the chain and prints a stack trace, which is both useful during development and allows higher availability of the API.
- Secure: provides protection against cross-site scripting (XSS) attacks, content type sniffing, clickjacking, insecure connections and other code injection attacks.
- RateLimiter: middleware for limiting the amount of requests to the API from a particular IP address within a time period. Used to increase security and protect against DDoS attacks.

Custom Middleware

Logger

```go
type Formatter struct{}

func (f *Formatter) Format(entry *logrus.Entry) ([]byte, error) {
    timestamp := entry.Time.Format(time.RFC3339)
    return []byte(fmt.Sprintf("[%s] method=%s, path=%s, ip=%s, status=%d, user=%v, error=%v\n",
        timestamp,
        entry.Data["method"],
        entry.Data["path"],
        entry.Data["ip"],
        entry.Data["status"],
        entry.Data["user"],
        entry.Data["error"],
    )), nil
}

var (
    log                       = logrus.New()
    DefaultRequestLoggerConfig = middleware.RequestLoggerConfig{
        LogMethod:   true,
        LogURIPath:  true,
        LogRemoteIP: true,
        LogStatus:   true,
        LogError:    true,
        LogValuesFunc: func(c echo.Context, values middleware.RequestLoggerValues) error {
            log.WithFields(logrus.Fields{
                "method": values.Method,
                "path":   values.URIPath,
                "ip":     values.RemoteIP,
                "status": values.Status,
                "user":   c.Get("uid"),
                "error":  values.Error,
            }).Info()
            return nil
        },
    }
)

func Logger() echo.MiddlewareFunc {
    log.SetFormatter(&Formatter{})
    c := DefaultRequestLoggerConfig
    return LoggerWithConfig(c)
}

func LoggerWithConfig(config middleware.RequestLoggerConfig) echo.MiddlewareFunc {
    return middleware.RequestLoggerWithConfig(DefaultRequestLoggerConfig)
}
```

An extension to the default echo logger using logrus (https://github.com/sirupsen/logrus) and custom output. This was done to improve the readability and flexibility of our API logs. A notable addition is the user id attached to each request, which can potentially allow incident reporting and limiting for users who abuse PlusVasis. This feature is also a nice-to-have for testing and debug purposes.

Firebase Authentication

```go
type Config struct {
    CredentialsFile string
    Skipper         middleware.Skipper
}

// disable auth for health check
func DefaultSkipper(c echo.Context) bool {
    return c.Path() == "/health" || strings.HasPrefix(c.Path(), "/swagger")
}

var DefaultConfig = Config{
    CredentialsFile: "firebase.json",
    Skipper:         DefaultSkipper,
}

func Firebase() echo.MiddlewareFunc {
    c := DefaultConfig
    return FirebaseWithConfig(c)
}

func FirebaseWithConfig(config Config) echo.MiddlewareFunc {
    opt := option.WithCredentialsFile(config.CredentialsFile)
    app, err := firebase.NewApp(context.Background(), nil, opt)
    if err != nil {
        panic(errors.Wrap(err, "error initializing app"))
    }
    client, err := app.Auth(context.Background())
    if err != nil {
        panic(errors.Wrap(err, "error getting auth client"))
    }

    return func(next echo.HandlerFunc) echo.HandlerFunc {
        return func(c echo.Context) error {
            if config.Skipper(c) {
                return next(c)
            }

            token := ""
            if c.Request().Header.Get("Authorization") != "" {
                header := c.Request().Header.Get("Authorization")
                token = strings.Replace(header, "Bearer ", "", 1)
            } else if c.QueryParam("access_token") != "" {
                token = c.QueryParam("access_token")
            } else {
                return echo.NewHTTPError(http.StatusUnauthorized, "missing auth token")
            }

            user, err := client.VerifyIDToken(context.Background(), token)
            if err != nil {
                return echo.NewHTTPError(http.StatusUnauthorized, err.Error())
            }

            c.Set("uid", user.UID)
            return next(c)
        }
    }
}

```

A custom firebase authentication middleware allows us to easily protect every route in our API with separation of concerns from our route handlers. JWT tokens are provided in each request either within an `Authorization` header or as a query parameter called `access_token` - this was required because additional headers can not be added to a WebSocket request.

The Skipper function allows us to define endpoints that can bypass Authentication. In our case we skip authentication for the `/health` endpoint and any endpoints ending with `/swagger`.

This middleware also sets the current user's id in the context, so that we can later check what user made the request and return appropriate responses based on that.

Handlers/Controllers

NomadClient

```go
const NOMAD_URL = "https://nomad.local.cawnj.dev/v1"

type NomadClient interface {
    Get(endpoint string) ([]byte, error)
    Post(endpoint string, reqBody *bytes.Buffer) ([]byte, error)
    Delete(endpoint string) ([]byte, error)
    ForwardRequest(c echo.Context, url string) (*http.Response, error)
}
```

Provides an interface for communicating with Nomad's API. We use an interface for the communication for separation of concerns and for mocking requests to Nomad in our unit tests - testing of our API should not depend on Nomad.

NomadController

```go
type NomadController struct {
    Client NomadClient
}
```

Implements a NomadClient and contains the methods for each API route handler.

```go
// GetJobs godoc
//
//  @Summary        GetJobs
//  @Description    Get details of all Nomad jobs
//  @Tags           nomad
//  @Produce        json
//  @Success        200 {object}    []nomad.JobListStub
//  @Failure        401 {object}    echo.HTTPError
//  @Failure        500
//  @Security       BearerAuth
//  @Router         /jobs [get]
func (n *NomadController) GetJobs(c echo.Context) error {
    data, err := n.Client.Get("/jobs?meta=true")
    if err != nil {
        return err
    }

    var jobs []nomad.JobListStub
    err = json.Unmarshal(data, &jobs)
    if err != nil {
        return echo.ErrInternalServerError
    }

    var filteredJobs []nomad.JobListStub
    uid := c.Get("uid").(string)
    for _, job := range jobs {
        if job.Meta["user"] == uid {
            filteredJobs = append(filteredJobs, job)
        }
    }

    return c.JSON(http.StatusOK, filteredJobs)
}
```

The GetJobs handler requests all jobs from Nomad and returns only the jobs that belong the user that made the request.

Annotated on the method we see our swagger specification for this API route. We specify the type of request (get, post, delete, etc.), and all the possible response codes alongside the types for the structs that then get marshaled before being returned as JSON - for example, on a successful request, this route returns HTTP code 200 and a JSON representation of an array of nomad.JobListStub structs.

Note: the nomad.JobListStub struct comes straight from Nomad's source, and is the same type used by Nomad when generating the response in the first place. Nomad is open-source and its API is written in Go, therefore we used this to our advantage to provide fully type-safe marshaling/unmarshaling of all our requests to and from Nomad.

```go
// CreateJob godoc
//
//  @Summary        CreateJob
//  @Description    Create a new Nomad job
//  @Tags           nomad
//  @Accept         json
//  @Produce        json
//  @Param          job body        templates.NomadJob  true    "Nomad Job"
//  @Success        200 {object}    nomad.JobRegisterResponse
//  @Failure        400 {object}    echo.HTTPError
//  @Failure        401 {object}    echo.HTTPError
//  @Failure        500
//  @Security       BearerAuth
//  @Router         /jobs [post]
func (n *NomadController) CreateJob(c echo.Context) error {
    var job templates.NomadJob
    err := decodeJobJson(&job, c.Request().Body)
    if err != nil {
        return err
    }

    body, err := templates.CreateJobJson(job)
    if err != nil {
        return errors.Wrap(echo.ErrBadRequest, err.Error())
    }

    data, err := n.Client.Post("/jobs", body)
    if err != nil {
        return err
    }

    var resp nomad.JobRegisterResponse
    err = json.Unmarshal(data, &resp)
    if err != nil {
        return echo.ErrInternalServerError
    }

    return c.JSON(http.StatusOK, resp)
}
```

The CreateJob handler takes json as input, unmarshals to a NomadJob object (this is one of our own types, defined in templates), generates a Job json object that Nomad understands and sends that to Nomad in a POST request. We then return the response from Nomad back to the user.

Notable in the swagger spec is the definition of the NomadJob parameter in the POST body.

```go
// UpdateJob godoc
//
//  @Summary        UpdateJob
//  @Description    Update an existing Nomad job
//  @Tags           nomad
//  @Accept         json
//  @Produce        json
//  @Param          id  path        string              true    "Job ID"
//  @Param          job body        templates.NomadJob  true    "Nomad Job"
//  @Success        200 {object}    nomad.JobRegisterResponse
//  @Failure        400 {object}    echo.HTTPError
//  @Failure        401 {object}    echo.HTTPError
//  @Failure        500
//  @Security       BearerAuth
//  @Router         /job/{id} [post]
func (n *NomadController) UpdateJob(c echo.Context) error {
    var job templates.NomadJob
    err := decodeJobJson(&job, c.Request().Body)
    if err != nil {
        return err
    }

    body, err := templates.CreateJobJson(job)
    if err != nil {
        return errors.Wrap(echo.ErrBadRequest, err.Error())
    }

    uid := c.Get("uid").(string)
    jobId := c.Param("id")
    if err := n.CheckUserAllowed(uid, jobId); err != nil {
        return err
    }

    data, err := n.Client.Post(fmt.Sprintf("/job/%s", jobId), body)
    if err != nil {
        return err
    }

    var resp nomad.JobRegisterResponse
    err = json.Unmarshal(data, &resp)
    if err != nil {
        return echo.ErrInternalServerError
    }

    return c.JSON(http.StatusOK, resp)
}
```

The UpdateJob handler takes the same input as CreateJob but also checks that the user making the request is the owner of the job that is to be updated. If this check passes, we send the updated job to Nomad and returns its response.

Notable in the swagger spec here is the job ID string as a path parameter - `/job/:id/`

From here forward we will omit the swagger annotations as there is no more notable attributes that differ greatly from the examples already given.

```
func (n *NomadController) ReadJob(c echo.Context) error {
    uid := c.Get("uid").(string)
    jobId := c.Param("id")

    data, err := n.Client.Get(fmt.Sprintf("/job/%s", jobId))
    if err != nil {
        return err
    }

    var job nomad.Job
    err = json.Unmarshal(data, &job)
    if err != nil {
        return echo.ErrInternalServerError
    }

    // Doing this check here because if we use n.CheckUserAllowed, we will duplicate requests
    if job.Meta["user"] != uid {
        return echo.ErrUnauthorized
    }

    return c.JSON(http.StatusOK, job)
}

```

The ReadJob handler takes a jobId as a path parameter and gets that job from Nomad and returns it to the user if they are the owner.

```
func (n *NomadController) StopJob(c echo.Context) error {
    uid := c.Get("uid").(string)
    jobId := c.Param("id")
    purge := c.QueryParam("purge")

    if err := n.CheckUserAllowed(uid, jobId); err != nil {
        return err
    }

    url := fmt.Sprintf("/job/%s", jobId)
    if purge == "true" {
        url += "?purge=true"
    }
    data, err := n.Client.Delete(url)
    if err != nil {
        return err
    }

    var resp nomad.JobDeregisterResponse
    err = json.Unmarshal(data, &resp)
    if err != nil {
        return echo.ErrInternalServerError
    }

    return c.JSON(http.StatusOK, resp)
}

```

StopJob takes a jobId as a path parameter and a query parameter boolean called purge as inputs and will essentially forwards this request to Nomad, which has the same inputs for its API endpoint.

Instead of forwarding the request outright, we only check if the `purge` query param is set to true and add that to our request to Nomad. The reason for this is to disallow poisoning of the request to Nomad with unwanted query parameters. For the functionality of PlusVasis, we only care about the purge parameter and therefore it is the only one we send to Nomad. This is done purely for security and exploit mitigation reasons.

We also perform the same check as above that the requesting user actually owns the job relating to the provided jobId. This check is done in every API handler that performs a read/write/edit on Nomad jobs and will not be mentioned in future to avoid repetition in this document.

```go
func (n *NomadController) RestartJob(c echo.Context) error {
    uid := c.Get("uid").(string)
    jobId := c.Param("id")

    if err := n.CheckUserAllowed(uid, jobId); err != nil {
        return err
    }

    alloc, err := n.ParseRunningAlloc(jobId)
    if err != nil {
        return err
    }

    body := bytes.NewBuffer([]byte{})
    data, err := n.Client.Post(fmt.Sprintf("/client/allocation/%s/restart", alloc.ID), body)
    if err != nil {
        return err
    }

    var resp nomad.GenericResponse
    err = json.Unmarshal(data, &resp)
    if err != nil {
        return echo.ErrInternalServerError
    }

    return c.JSON(http.StatusOK, resp)
}

```

The RestartJob handler takes a jobId as input and sends a request to Nomad to Restart the currently running allocation (basically a deployment) for the relating job. We send a POST request with an empty body as Nomad handles this by restarting all “tasks” for the given allocation - which for the purposes of PlusVasis and not going into too much detail on Nomad’s API, is exactly what we want to do here. We then return Nomad’s response back to the user.

```go
func (n *NomadController) StartJob(c echo.Context) error {
    uid := c.Get("uid").(string)
    jobId := c.Param("id")

    data, err := n.Client.Get(fmt.Sprintf("/job/%s", jobId))
    if err != nil {
        return err
    }

    var job nomad.Job
    err = json.Unmarshal(data, &job)
    if err != nil {
        return err
    }

    if job.Meta["user"] != uid {
        return echo.ErrUnauthorized
    }

    // Nomad doesn't have a start job endpoint, and this
    // is exactly how they do it in their Web UI
    // It's a bit hacky, but it works
    job.Stop = false
    var jobRequest nomad.JobRegisterRequest
    jobRequest.Job = &job

    body, err := json.Marshal(jobRequest)
    if err != nil {
        return echo.ErrInternalServerError
    }

    data, err = n.Client.Post(fmt.Sprintf("/job/%s", jobId), bytes.NewBuffer(body))
    if err != nil {
        return err
    }

    var resp nomad.JobRegisterResponse
    err = json.Unmarshal(data, &resp)
    if err != nil {
        return echo.ErrInternalServerError
    }

    return c.JSON(http.StatusOK, resp)
}

```

The StartJob handler takes a jobId as input and start the relating job. Nomad doesn’t actually provide a Start Job endpoint, therefore we read the job the exact same way as in the ReadJob handler, set the attribute `job.Stop` to false and then update the job - similarly to the UpdateJob handler - before returning Nomad’s response.

In Nomad’s own web UI, they provide a Start Job button, yet through research via the network tab in Chrome, we discovered how Nomad actually does this themselves even though they do not provide an endpoint for it!

```go
func (n *NomadController) CheckUserAllowed(uid, jobId string) error {
    data, err := n.Client.Get(fmt.Sprintf("/job/%s", jobId))
    if err != nil {
        return err
    }

    var job nomad.Job
    err = json.Unmarshal(data, &job)
    if err != nil {
        return echo.ErrInternalServerError
    }

    if job.Meta["user"] != uid {
        return echo.ErrUnauthorized
    }

    return nil
}

```

This method takes a uid and jobId as input and checks that the given uid matches that of the owner of the job relating to the given jobId.

It will return either the error generated in NomadClient for the request to Nomad, a code 500 internal server error if we fail to unmarshal the response from Nomad, a code 401 unauthorized error if the user does not own this job, or nil (no error), meaning the user does own and therefore is allowed to access this job.

```go
func (n *NomadController) ParseRunningAlloc(jobId string) (*nomad.AllocListStub, error) {
    data, err := n.Client.Get(fmt.Sprintf("/job/%s/allocations", jobId))
    if err != nil {
        return nil, err
    }

    var allocs []nomad.AllocListStub
    err = json.Unmarshal(data, &allocs)
    if err != nil {
        return nil, echo.ErrInternalServerError
    }
    for _, alloc := range allocs {
        if alloc.ClientStatus == "running" || alloc.ClientStatus == "pending" {
            return &alloc, nil
        }
    }

    return nil, echo.ErrNotFound
}

```

This method takes a jobId as input and requests the allocations for the relating job from Nomad. We then check if any of the allocations are running or pending (still in the processing of starting) and return that allocation or a code 404 not found error if one does not exist.

NomadProxyController

```go
type NomadProxyController struct {
    Client nomadController.NomadClient
    Dialer DialerInterface
}

type DialerInterface interface {
    Dial(urlStr string, requestHeader http.Header) (WsConnInterface, *http.Response, error)
}

type DefaultDialer struct{}

func (d *DefaultDialer) Dial(urlStr string, requestHeader http.Header) (WsConnInterface, *http.Response, error) {
    return websocket.DefaultDialer.Dial(urlStr, requestHeader)
}

type WsConnInterface interface {
    ReadMessage() (messageType int, p []byte, err error)
    WriteMessage(messageType int, data []byte) error
    Close() error
    SetReadDeadline(t time.Time) error
}

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true
    },
}

const idleTimeout = 30 * time.Second

```

The NomadProxyController deals with any handlers/routes that act as a **real-time** middleman or proxy between the user and Nomad.

Here we provide interfaces for WebSockets - so that we can mock them during unit testing - and other configuration options relating to WebSocket connections.

```go
func (n *NomadProxyController) AllocExec(c echo.Context) error {
    jobId := c.Param("id")
    command := c.QueryParam("command")
    if command == "" {
        return fmt.Errorf("missing query parameters")
    }

    uid := c.Get("uid").(string)
    if err := n.checkUserAllowed(uid, jobId); err != nil {
        return err
    }

    alloc, err := n.parseRunningAlloc(jobId)
    if err != nil {
        return err
    }

    baseURL := "wss://nomad.local.cawnj.dev/"
    path := "v1/client/allocation/" + alloc.ID + "/exec"
    queryParams := url.Values{}
    queryParams.Add("command", command)
    queryParams.Add("task", alloc.TaskGroup)
    queryParams.Add("tty", "true")
    queryParams.Add("ws_handshake", "true")

    url := baseURL + path + "?" + queryParams.Encode()

    nomadConn, _, err := n.Dialer.Dial(url, nil)
    if err != nil {
        return echo.ErrBadGateway
    }
    defer nomadConn.Close()

    clientConn, err := upgrader.Upgrade(c.Response(), c.Request(), nil)
    if err != nil {
        return errors.Wrap(err, "failed to upgrade connection")
    }
    defer clientConn.Close()

    err = nomadConn.SetReadDeadline(time.Now().Add(idleTimeout))
    if err != nil {
        return echo.ErrInternalServerError
    }
    err = clientConn.SetReadDeadline(time.Now().Add(idleTimeout))
    if err != nil {
        return echo.ErrInternalServerError
    }

    fmt.Printf("Started terminal session for job %s\n", jobId)
    var wg sync.WaitGroup
    wg.Add(2)
    go func() {
        n.forwardMessages(clientConn, nomadConn)
        wg.Done()
    }()
    go func() {
        n.forwardMessages(nomadConn, clientConn)
        wg.Done()
    }()
    wg.Wait()

    fmt.Printf("Stopped terminal session for job %s\n", jobId)
    return nil
}

```

The AllocExec handler takes a jobId and command - `/bin/sh`, `/bin/bash`, etc. - and builds the appropriate request to send to Nomad to initiate a WebSocket connection to the running alloc for the relating job. This is used for the terminal component in the frontend.

This is the most complex handler in the API so therefore we will explain it more in depth, step-by-step:

- Get jobId and command from path and query parameters respectively
  - If command was not provided or is empty, return an error
- Check that the requesting user is allowed to access the current job
  - If unauthorized, return an error
- Get the currently running allocation for the current job
  - If there is not one running, return an error
- Build the correct request to send to Nomad to initiate the WebSocket connection
  - path: allocId
  - query: command, task, tty, ws_handshake
- Initiate WebSocket connection with Nomad
  - If the connection fails, return an error
  - Defer closing of this connection (in case of an error later)

- Upgrade client's request from HTTP to WebSocket, initiating WebSocket connection with the client
  - If the connection fails, return an error
  - Defer closing of this connection (in case of an error later)
- Set deadlines to close the connection if no messages are received within the idleTimeout period (client should send empty heartbeat messages to keep connection alive)
  - If this fails for whatever reason, return an error
- Start a go routine to forward messages received from the client to Nomad
- Start a go routine to forward messages received from Nomad to the client
- Wait until both of these go routines have finished executing
- Exit

```go
func (n *NomadProxyController) forwardMessages(srcConn, dstConn WsConnInterface) {
    for {
        msgType, msg, err := srcConn.ReadMessage()
        if err != nil {
            break
        }
        err = dstConn.WriteMessage(msgType, msg)
        if err != nil {
            break
        }
        err = srcConn.SetReadDeadline(time.Now().Add(idleTimeout))
        if err != nil {
            break
        }
    }
}

```

This method is ran in a go routine and is responsible for forwarding messages from one WebSocket connection to another, simply by reading a message from the source connection and writing it to the destination connection.

This method also updates the read deadline (timeout) whenever a message is successfully forwarded.

```go
func (n *NomadProxyController) StreamLogs(c echo.Context) error {
    jobId := c.Param("id")
    logType := c.QueryParam("type")
    if logType == "" {
        return echo.ErrBadRequest
    }

    uid := c.Get("uid").(string)
    if err := n.checkUserAllowed(uid, jobId); err != nil {
        return err
    }

    alloc, err := n.parseRunningAlloc(jobId)
    if err != nil {
        return err
    }

    baseURL := "https://nomad.local.cawnj.dev/"
    path := "v1/client/fs/logs/" + alloc.ID
    queryParams := url.Values{}
    queryParams.Add("task", alloc.TaskGroup)
    queryParams.Add("type", logType)
    queryParams.Add("follow", "true")
    queryParams.Add("offset", "50000")
    queryParams.Add("origin", "end")

    url := baseURL + path + "?" + queryParams.Encode()
    resp, err := n.Client.ForwardRequest(c, url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()

    fmt.Printf("Started streaming %s for job %s\n", logType, jobId)

    err = streamResponse(c, resp)
    if err != nil {
        return err
    }

    fmt.Printf("Stopped streaming %s for job %s\n", logType, jobId)
    return nil
}

```

The StreamLogs handler takes a jobId as a path parameter and a logType as a query parameter. The logType will be either `stdout` or `stderr`. This handler is used to stream logs in real-time from Nomad to the client.

It functions similarly to the AllocExec handler in how it builds the appropriate request to Nomad and therefore will not be explained in depth again here. Where it differs is in the streamResponse method, explained below.

```go
func streamResponse(c echo.Context, resp *http.Response) error {
    if resp.Header.Get("Content-Encoding") == "gzip" {
        gzipReader, err := gzip.NewReader(resp.Body)
        if err != nil {
            return echo.ErrInternalServerError
        }
        defer gzipReader.Close()
        resp.Body = gzipReader
    }

    buf := make([]byte, 8192)
    for {
        bytesRead, err := resp.Body.Read(buf)
        if err != nil {
            if err == io.EOF {
                break
            }
            return echo.ErrInternalServerError
        }

        select {
        case <-c.Request().Context().Done():
            return nil
        default:
        }

        _, err = c.Response().Writer.Write(buf[:bytesRead])
        if err != nil {
            return echo.ErrInternalServerError
        }

        if flusher, ok := c.Response().Writer.(http.Flusher); ok {
            flusher.Flush()
        }
    }

    return nil
}
```

The streamResponse method takes the current context and the response from Nomad as inputs. It will first check if the request from Nomad is gzip encoded and if so, provides a reader for this. It then creates a buffer from which it will read the response body from Nomad (that contains the current state of the logs as a stream) and then writes this buffer to the response that is sent back to the client.

There are checks for if the data stream has ended, failed, or otherwise does not exist anymore, and appropriate returns and errors for each of these cases.

Additionally the response to the client is flushed on every loop because writers in Go (such as the http ResponseWriter used here) are buffered until the handler returns, which is not what we want to happen here in a real-time log viewer. Flushing the writer periodically solves this issue, and we decided to flush on every loop to ensure the logs can be viewed by the client in as real-time as possible.

Templates

```
type NomadJob struct {
    Name      string     `json:"containerName" validate:"required"`
    Image     string     `json:"dockerImage" validate:"required"`
    User      string     `json:"user" validate:"required"`
    Shell     string     `json:"shell" validate:"required"`
    Volumes   [][]string `json:"volumes" validate:"kvPairs"`
    Env       [][]string `json:"env" validate:"kvPairs"`
    EnvString string     `json:"envString"`
    Port      int        `json:"port" validate:"min=0,max=65535"`
    Expose    bool       `json:"expose"`
    Cpu       int        `json:"cpu" validate:"min=0,max=1000"`
    Memory    int        `json:"memory" validate:"min=0,max=2000"`
}
```

Here is the definition of a NomadJob struct type. Each field is validated using validator (https://github.com/go-playground/validator/) for the correct type and content.

```
func CreateJobJson(job NomadJob) (*bytes.Buffer, error) {
    // create template
    t, err := template.New("").Funcs(template.FuncMap{
        "last": last,
    }).Parse(JOB_TMPL)
    if err != nil {
        return nil, err
    }

    // ensure job is valid
    err = Validate(job)
    if err != nil {
        return nil, err
    }

    // parse env vars for templating
    if len(job.Env) != 0 {
        trimEnv(&job)
        err = parseEnv(&job)
        if err != nil {
            return nil, err
        }
    }

    // execute template
    buf := &bytes.Buffer{}
    err = t.Execute(buf, job)
    if err != nil {
        return nil, err
    }

    // output for debugging
    f, err := os.Create("latest-job.json")
    if err != nil {
        return nil, err
    }
    defer f.Close()
    _, err = f.Write(buf.Bytes())
    if err != nil {
        return nil, err
    }

    return buf, err
}
```

The CreateJobJson method takes a NomadJob struct as input and returns the correct JSON representation of this job that can then be sent to Nomad.

The steps taken by this method to achieve this are commented clearly and of which's implementation will be explained below.

```go
func trimEnv(job *NomadJob) {
    for i, env := range job.Env {
        // trim spaces from key and value
        job.Env[i][0] = strings.TrimSpace(env[0])
        job.Env[i][1] = strings.TrimSpace(env[1])

        // trim either single or double quotes from value
        if strings.HasPrefix(env[1], "'") && strings.HasSuffix(env[1], "'") {
            job.Env[i][1] = strings.Trim(env[1], "'")
        } else if strings.HasPrefix(env[1], "\"") && strings.HasSuffix(env[1], "\"") {
            job.Env[i][1] = strings.Trim(env[1], "\"")
        }
    }
}

```

The trimEnv method takes a pointer to a NomadJob struct and manipulates the Env field to be in a format that we can use in templating later and will be accepted by Nomad. This involves trimming spaces and triming quotes if the environment variable value is surrounded by them.

```go
func parseEnv(job *NomadJob) error {
    for _, env := range job.Env {
        key := env[0]
        value := env[1]

        fmt.Printf("key: %s, value: %s\n", key, value)

        fieldRegex := regexp.MustCompile(`{{\s*(.*?)\s*}}`)
        fields := fieldRegex.FindAllStringSubmatch(value, -1)

        fmt.Printf("fields: %v\n", fields)
        if len(fields) > 0 {
            // ensure fields[i][1] are all equal
            fieldVal := fields[0][1]
            for _, field := range fields {
                if field[1] != fieldVal {
                    return fmt.Errorf("only one other job can be referenced in a templated env var")
                }
            }

            job.EnvString += generateTemplatedEnv(key, value, fieldVal)
        }
    }
    return nil
}

```

The parseEnv method is where we deal with referencing other jobs in environment variables for inter-container communication. This is done with the `{{jobName}}` syntax and allows a sidecar database container to be accessed by a web server, for example.

This is done via regex matching the environment variable values and checking if a jobName with the `{{jobName}}` syntax exists within. For each instance of this (called a field) we check that only one job is specified per value (if specified at all).

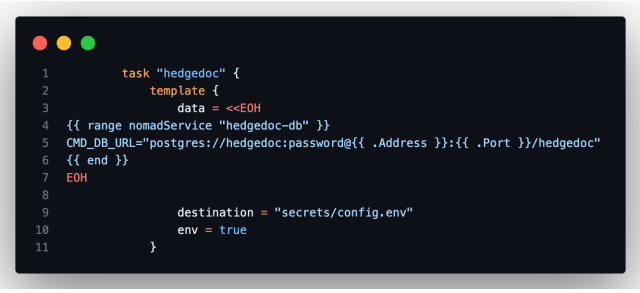
For example `HOT_TAKE="{{nginx}} is better than {{apache}}"` is not allowed, but `FACT="{{nginx}} == {{nginx}}"` is allowed. This is because Nomad does not easily deal with the former case, as it adds a lot of complexity to make work using Go templates - Nomad also uses Go templates for this functionality!

We ensure this isn't the case by storing the first field value, looping through each field, and returning an error if any of the fields are different.

We then update the templated EnvString field for the NomadJob accordingly, which is explained below.

```go
func generateTemplatedEnv(key, value, otherJob string) string {
    // replace every instance of otherJob
    newValue := strings.ReplaceAll(value, fmt.Sprintf("{{%s}}", otherJob), "{{ .Address }}:{{ .Port }}")
    templatedEnv := fmt.Sprintf(ENV_TMPL, otherJob, key, newValue)

    // escape newline characters and double quotes
    // so that the template can be embedded in a JSON string
    templatedEnv = strings.ReplaceAll(templatedEnv, "\n", "\\n")
    templatedEnv = strings.ReplaceAll(templatedEnv, "\"", "\\\"")

    return templatedEnv
}

const ENV_TMPL = `{{ range nomadService "%s" }}
%s="%s"
{{ end }}
`
```

To add some context here, this is one way Nomad deals with inter-job communication, and is the method we use when creating our Nomad jobs:



```
        task "hedgedoc" {
            template {
                data = <<EOH
{{ range nomadService "hedgedoc-db" }}
CMD_DB_URL="postgres://hedgedoc:password@{{ .Address }}:{{ .Port }}/hedgedoc"
{{ end }}
EOH

                destination = "secrets/config.env"
                env = true
            }
```

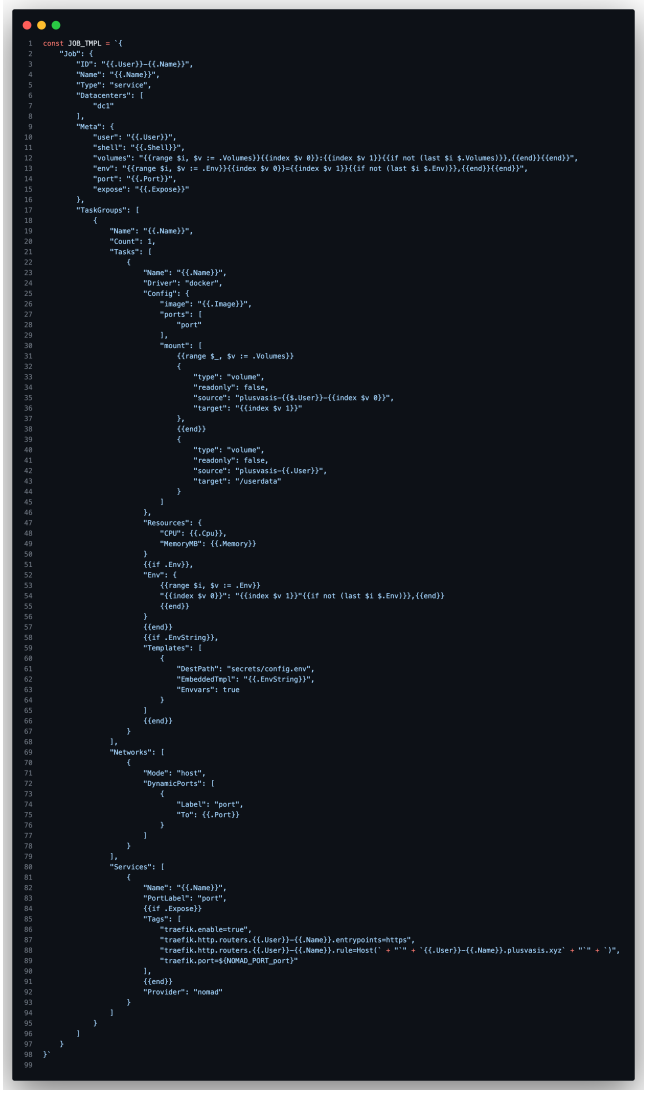
Nomad uses “service discovery” for find the job matching the name given, of which we can access its `Address` and `Port` fields - this will look familiar or will make more sense later, as Nomad uses Go structs to represent its jobs and `Address` and `Port` are fields on that Nomad Job struct.

In this above example we pass a postgres database URL as an environment variable to a hedgedoc instance.

So in the generateTemplatedEnv method, we take in the key and value for an environment variable, and the name of the job referenced with the `{{jobName}}` syntax within the environment variable’s value.

We then replace any instance of `{{jobName}}` with `{{ .Address }}:{{ .Port }}` and use this in a new string called templatedEnv that reflects the example given above, and also escape any problematic characters.

This templatedEnvString will then get appended to the job.EnvString field in the parseEnv method.



```
const JOB_TMPL = `{
    "Job": {
        "ID": "{{.User}}-{{.Name}}",
        "Name": "{{.Name}}",
        "Type": "service",
        "Datacenters": [
            "dc1"
        ],
        "Meta": {
            "user": "{{.User}}",
            "shell": "{{.Shell}}",
            "volumes": "{{range $i, $v := .Volumes}}{{index $v 0}}:{{index $v 1}}{{if not (last $i $.Volumes)}},{{end}}{{end}}",
            "env": "{{range $i, $v := .Env}}{{index $v 0}}={{index $v 1}}{{if not (last $i $.Env)}},{{end}}{{end}}",
            "port": "{{.Port}}",
            "expose": "{{.Expose}}"
        },
        "TaskGroups": [
            {
                "Name": "{{.Name}}",
                "Count": 1,
                "Tasks": [
                    {
                        "Name": "{{.Name}}",
                        "Driver": "docker",
                        "Config": {
                            "image": "{{.Image}}",
                            "ports": [
                                "port"
                            ],
                            "mount": [
                                {{range $_, $v := .Volumes}}
                                {
                                    "type": "volume",
                                    "readonly": false,
                                    "source": "plusvasis-{{$.User}}-{{index $v 0}}",
                                    "target": "{{index $v 1}}"
                                },
                                {{end}}
                                {
                                    "type": "volume",
                                    "readonly": false,
                                    "source": "plusvasis-{{.User}}",
                                    "target": "/userdata"
                                }
                            ]
                        },
                        "Resources": {
                            "CPU": {{.Cpu}},
                            "MemoryMB": {{.Memory}}
                        }
                        {{if .Env}},
                        "Env": {
                            {{range $i, $v := .Env}}
                            "{{index $v 0}}": "{{index $v 1}}"{{if not (last $i $.Env)}},{{end}}
                            {{end}}
                        }
                        {{end}}
                        {{if .EnvString}},
                        "Templates": [
                            {
                                "DestPath": "secrets/config.env",
                                "EmbeddedTmpl": "{{.EnvString}}",
                                "Envvars": true
                            }
                        ]
                        {{end}}
                    }
                ],
                "Networks": [
                    {
                        "Mode": "host",
                        "DynamicPorts": [
                            {
                                "Label": "port",
                                "To": {{.Port}}
                            }
                        ]
                    }
                ],
                "Services": [
                    {
                        "Name": "{{.Name}}",
                        "PortLabel": "port",
                        {{if .Expose}}
                        "Tags": [
                            "traefik.enable=true",
                            "traefik.http.routers.{{.User}}-{{.Name}}.entrypoints=https",
                            "traefik.http.routers.{{.User}}-{{.Name}}.rule=Host(` + "`" + `{{.User}}-{{.Name}}.plusvasis.xyz` + "`" + `)",
                            "traefik.port=${NOMAD_PORT_port}"
                        ],
                        {{end}}
                        "Provider": "nomad"
                    }
                ]
            }
        ]
    }
}`

```

Here is the Go template for a json representation of a Nomad job, that Nomad can understand and accept. Each field of a NomadJob object can be applied to this template with the `{{.FieldName}}` syntax.

We also store extra information on the current job in the `Meta` section, such as the user that owns the job, the shell they use and all other required metadata pertaining to a given job - the reason it is stored here in such a convoluted way is so that PlusVasis does not require a database to store and manage all of this metadata!

Other sections of note here:

- `Job.TaskGroups.Tasks[0].Config.mount` : here we mount a per-user volume to the /userdata path on every container belonging to a user, this is to make it easy to transfer files between containers.
- `Job.TaskGroups.Tasks[0].Templates[0].EmbeddedTmpl` : here is where we input the templated job.EnvString field, explained in depth previously.
- `Job.TaskGroups.Services[0].Tags` : here is where we set the container labels that allow our reverse proxy Traefik to discover the container and expose it publicly on the internet.
    - `${NOMAD_PORT_port}` is a variable set by Nomad that contains the port number of the current service

Frontend

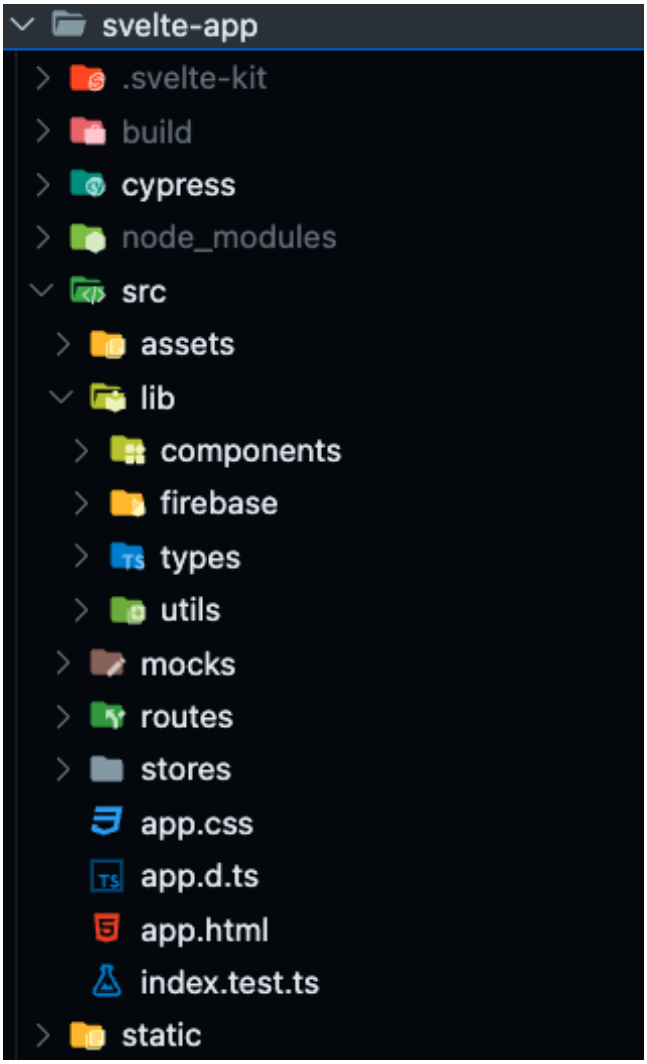
The frontend of our application is a web app created using SvelteKit, a routing framework build on top of the Svelte JavaScript framework. SvelteKit simplifies the process of building web applications by providing a streamlined development experience and built-in features such as

server-side rendering and automatic code splitting.

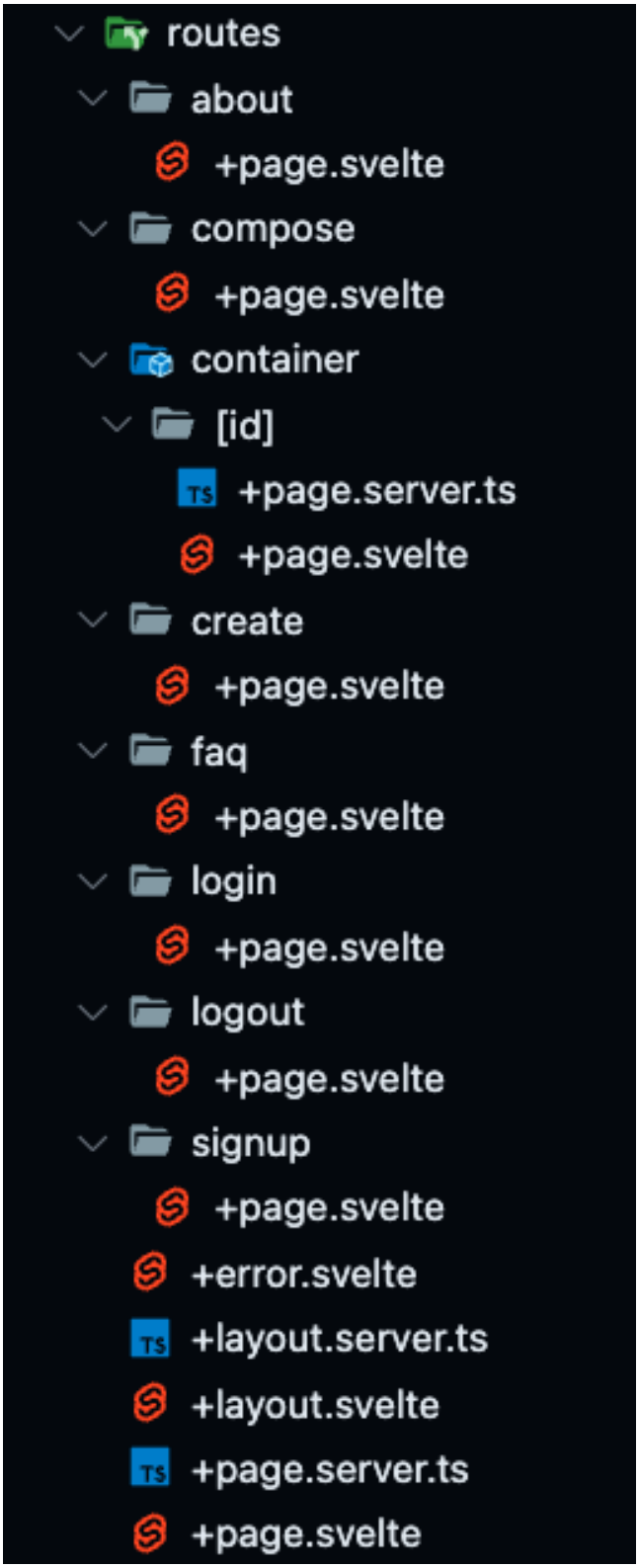
Our front end project also utilizes Vite, a fast and modern build tool for web applications, which helps to improve the development and build process.

TypeScript is used throughout the frontend project, providing a statically typed and robust codebase that is easier to maintain and debug. Overall, our front end is designed to provide a fast, intuitive, and user-friendly experience for managing containers and orchestrating applications.

Structure



Routes



From the get-go of the SvelteKit project initialisation, we already had a boiler plated format from which we could build upon. SvelteKit utilises the routes directory for all of the pages that a user will physically see. Each of these pages are “+page.svelte” files. For example the “+page.svelte” file at the top level of the routes directory correlates to the index page of the web app where as the “+page.svelte” files under the login and logout sub folders represent the /login and /logout pages of the web app respectively.

To explain some basics before describing the implementation further, all Svelte files (.svelte) allow for using a script to specify TypeScript or JavaScript, HTML for what the user will see on that given page and CSS. For example take a look at our “+page.svelte” file under the login sub-

directory.

```svelte
<script>
    import Login from '$lib/components/LoginForm.svelte';
</script>

<div class="grid place-items-center px-4 py-4 md:px-16 md:py-16">
    <Login title="Sign Up" />
</div>

```

We also made use of Sveltes layout functionalities. We created a “+layout.svelte” file to specify our CSS across all the routes.

This worked by using a script to import our global CSS which was configured with Tailwind and uses a slot, which is where the current page contents are substituted in.

As well as having layout functionalities, Svelte also has it’s own error capabilities, so we made a “+error.svelte” file to create an error message to be displayed to users if they stumbled across an invalid URL or wrong page.

```svelte
<script>
    import Nav from '$lib/components/NavBar.svelte';
</script>

<Nav />
<div class="px-4 pb-4 text-white md:px-16">
    <h1 class="text-3xl font-bold md:text-5xl">Whoops!</h1>
    <p class="text-xl md:text-2xl">You shouldn't be here...</p>
</div>

```

We wanted to limit the amount of logic stored in these routing files, which led to use of creating components for functionality, so this left our “+page.svelte” files looking tidy and maximising readability.

For example here’s our index page file:

```svelte
<script lang="ts">
    import Index from '$lib/components/IndexPage.svelte';
    import Nav from '$lib/components/NavBar.svelte';

    import type { PageData } from './$types';

    export let data: PageData;
</script>

<Nav />
<Index {data} />

```

And here's our file under the container/[id] sub directory:

```svelte
<script lang="ts">
    import Container from '$lib/components/ContainerPage.svelte';
    import Nav from '$lib/components/NavBar.svelte';
</script>

<Nav />
<Container />
```

Another cool feature of the routing was for the example above with the container page routing, we could use "[id]" as a naming placeholder for dynamic routes that would be specific for different container names and what not.

One last functionality of SvelteKit we used in the routing process, was its server side load abilities which meant we could render things server side before a user even hits a page. This eliminated any load times that a user could experience making navigation between pages seemless.

This server side rendering means PlusVasis does not have any loading spinners when fetching data from our API, which greatly improves the user experience.

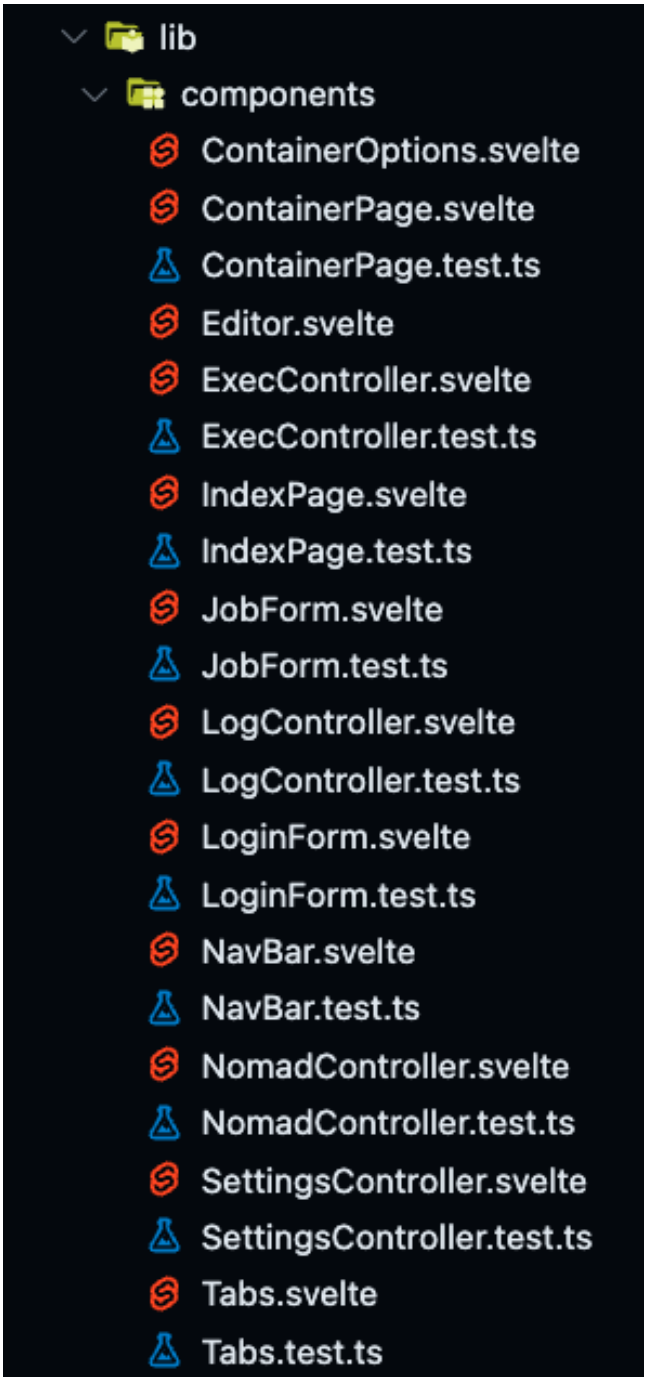
In our testing we noticed improvements of load times up to 3x!

We used a server side layout file "+layout.server.ts" to check the auth of a current user by searching for a valid token in the cookies. If there was no token present the user would be redirected to the login screen with a code 307 temporary redirect.

This is the "+page.server.ts" file for our index page, which will fetch a users current containers and have them displayed on the dashboard instantaneously when a user logs in or navigates to the home page.

```ts
import { hostname } from '../stores/environmentStore';
import type { PageServerLoadEvent } from './$types';

export async function load({ fetch, cookies }: PageServerLoadEvent) {
    const token = cookies.get('token');
    if (!token) return;

    const fetchJobs = async () => {
        const res = await fetch(`${hostname}/jobs`, {
            headers: {
                Authorization: `Bearer ${cookies.get('token')}`
            }
        });
        if (res.ok) {
            return await res.json();
        } else {
            return [];
        }
    };
    try {
        const jobs = await fetchJobs();
        return { jobs, error: null };
    } catch (e: unknown) {
        if (e instanceof Error) {
            console.log(e);
            return { jobs: [], error: e.message };
        }
    }
}
```

Components



Here we stored all the direct page logic in the lib/components directory and our component tests using Vitest.

To start off we made use of a LoginForm component.

```svelte
<script lang="ts">
    import { faArrowLeft, faExclamationTriangle } from '@fortawesome/free-solid-svg-icons';
    import { FirebaseError } from 'firebase/app';
    import { A, Alert, Button, Card, Hr, Input, Label } from 'flowbite-svelte';
    import { LoginButton } from 'svelte-auth-ui';
    import Fa from 'svelte-fa';

    import { goto } from '$app/navigation';

    import logo from '../../assets/logo.png';
    import { createUserWithEmail, signInWithEmail, signInWithGithub } from '../../stores/auth';

    export let title: string;
    let errorCode: string | null = null;
    let loading = false;

    async function handleFormSubmit(event: Event) {
        let email: string;
        let password: string;
        if (event.target instanceof HTMLFormElement) {
            const formData = new FormData(event.target);
            email = formData.get('email') as string;
            password = formData.get('password') as string;
        } else {
            errorCode = 'auth/invalid-form-input';
            return;
        }

        try {
            if (title === 'Login') {
                await signInWithEmail(email, password);
            } else {
                await createUserWithEmail(email, password);
            }
            goto('/');
        } catch (error: unknown) {
            if (error instanceof FirebaseError) {
                errorCode = error.code;
            }
        }
    }

    async function handleGithubLogin() {
        loading = true;
        try {
            await signInWithGithub();
            goto('/');
        } catch (error: unknown) {
            if (error instanceof FirebaseError) {
                errorCode = error.code;
            }
        } finally {
            loading = false;
        }
    }
</script>

<Card>
    <form class="flex flex-col space-y-6" on:submit|preventDefault={handleFormSubmit}>
        <div class="flex items-center">
            {#if title === 'Login'}
                <img alt="The project logo" src={logo} class="mr-3 h-8 w-8" />
            {:else}
                <Button pill={true} class="mr-3 h-8 w-8 !p-2" on:click={() => history.back()}>
                    <Fa icon={faArrowLeft} size="lg" />
                </Button>
            {/if}
            <h5 class="text-xl font-medium text-gray-900 dark:text-white">PlusVasis {title}</h5>
        </div>
        <Label class="space-y-2">
            <span>Email</span>
            <Input type="email" name="email" placeholder="name@example.com" required />
        </Label>
        <Label class="space-y-2">
            <span>Your password</span>
            <Input type="password" name="password" placeholder="••••••••••••" required />
        </Label>
        <div class="flex flex-col space-y-6 py-2">
            {#if errorCode}
                <Alert color="none" class="border-red-800 bg-red-100 !py-3 text-red-600">
                    <span slot="icon">
                        <Fa icon={faExclamationTriangle} class="mr-2" />
                    </span>
                    <span>
                        {errorCode}
                    </span>
                </Alert>
            {/if}
            {#if title === 'Login'}
                <Button type="submit" class="w-full"><span class="text-base">Login with email</span></Button
                >
                <Hr class="my-2" width="w-64">or</Hr>
                <div class="space-y-4">
                    <LoginButton
                        provider="github"
                        {loading}
                        withLoader
                        on:click={handleGithubLogin}
                        class="h-full w-full"
                    />
                </div>
            {:else}
                <Button type="submit" class="w-full"
                    ><span class="text-base">Sign up with email</span></Button
                >
            {/if}
        </div>
        <div class="flex flex-col space-y-2">
            {#if title === 'Login'}
                <div class="text-sm font-medium text-gray-500 dark:text-gray-300">
                    Not registered? <A href="/signup" class="text-blue-700 hover:underline dark:text-blue-500"
                        >Create an account</A
                    >
                </div>
            {/if}
            <div class="text-sm font-medium text-gray-500 dark:text-gray-300">
                We'll never share your details with anyone else.
            </div>
        </div>
    </form>
</Card>

```

This was the component used for the login and signup screens and granted users the options to sign in or create an account with an email or via GitHub. This was done through setting up a Firebase app which will be explained later.

Here’s our NavBar component which we used across all the routing.

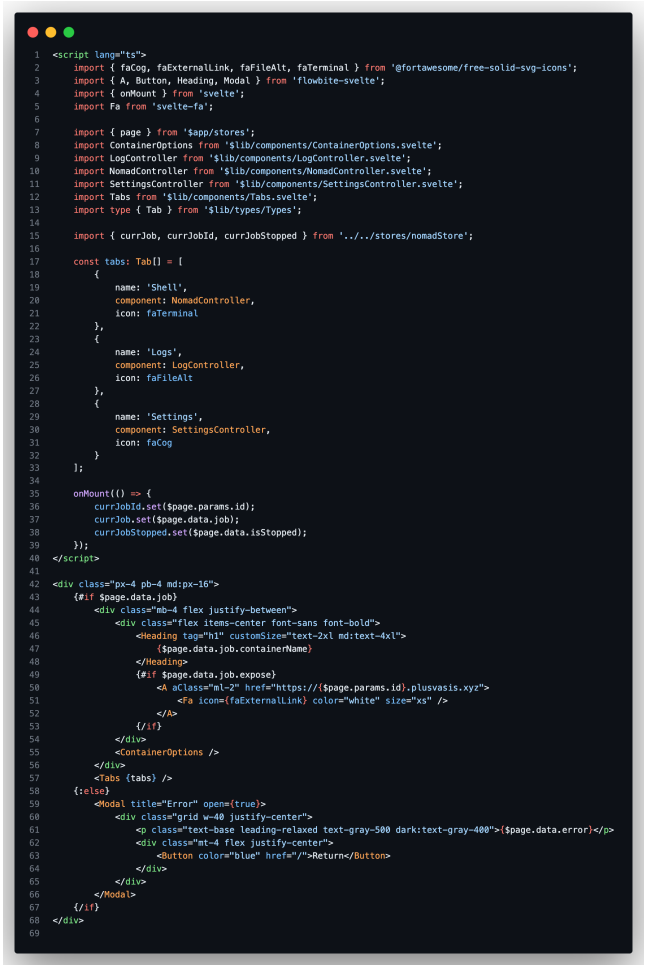
```
<script>
    import { Navbar, NavBrand, NavHamburger, NavLi, NavUl } from 'flowbite-svelte';

    import logo from '../../assets/logo.png';
</script>

<div class="mx-1 my-2">
    <Navbar let:hidden let:toggle>
        <NavBrand href="/">
            <img src={logo} class="mr-3 h-6 sm:h-9" alt="PlusVasis Logo" />
            <span
                class="self-center whitespace-nowrap text-xl font-bold hover:text-blue-400 dark:text-white md:text-2xl"
            >
                PlusVasis
            </span>
        </NavBrand>
        <NavHamburger on:click={toggle} />
        <NavUl data-testid="navbar-ul" {hidden}>
            <NavLi href="/" nonActiveClass="hover:text-blue-400 text-xl">Home</NavLi>
            <NavLi href="/about" nonActiveClass="hover:text-blue-400 text-xl">About</NavLi>
            <NavLi href="/faq" nonActiveClass="hover:text-blue-400 text-xl">FAQ</NavLi>
            <NavLi href="/logout" nonActiveClass="hover:text-blue-400 text-xl">Sign Out</NavLi>
        </NavUl>
    </Navbar>
</div>

```

Next we have the IndexPage component for our home page.

```
<script lang="ts">
    import { faInfoCircle, faTerminal } from '@fortawesome/free-solid-svg-icons';
    import { Alert, Button, Card, Modal } from 'flowbite-svelte';
    import Fa from 'svelte-fa';

    import type { PageData } from '../../routes/$types';

    export let data: PageData;
</script>

<div class="px-4 pb-4 md:px-16">
    <Button color="blue" href="/create" class="mr-2">Create Container</Button>
    {#if !data.jobs}
        <Alert class="my-4 max-w-md">
            <span slot="icon"><Fa icon={faInfoCircle} /> </span>
            <span class="font-medium">It seems like you haven't created any containers yet...</span>
        </Alert>
    {:else}
        <div data-testid="job-list">
            {#each data.jobs as job}
                <Card class="my-3" href={`/container/${job.ID}`}>
                    <div class="my-1 flex items-center">
                        <Fa icon={faTerminal} color="white" class="pr-6" />
                        <span class="text-xl text-white">{job.Name}</span>
                    </div>
                </Card>
            {/each}
        </div>
    {/if}
    {#if data.error}
        <Modal title="Error" open={true}>
            <div class="grid w-40 justify-center">
                <p class="text-base leading-relaxed text-gray-500 dark:text-gray-400">
                    {data.error}
                </p>
                <div class="mt-4 flex justify-center">
                    <Button color="blue" href="javascript:window.location.href=window.location.href"
                        >Retry</Button
                    >
                </div>
            </div>
        </Modal>
    {/if}
</div>

```

This component exports a prop called data, which is then passed in from the server side loading in the “+page.server.ts” file, which allowed us to list the fetched jobs to a user instantly. If no containers were present an alert would be displayed and if there was an error encountered during the server side fetch we would display the error along with a retry button.

Moving onto the ContainerPage component.



```svelte
<script lang="ts">
    import { faCog, faExternalLink, faFileAlt, faTerminal } from '@fortawesome/free-solid-svg-icons';
    import { A, Button, Heading, Modal } from 'flowbite-svelte';
    import { onMount } from 'svelte';
    import Fa from 'svelte-fa';

    import { page } from '$app/stores';
    import ContainerOptions from '$lib/components/ContainerOptions.svelte';
    import LogController from '$lib/components/LogController.svelte';
    import NomadController from '$lib/components/NomadController.svelte';
    import SettingsController from '$lib/components/SettingsController.svelte';
    import Tabs from '$lib/components/Tabs.svelte';
    import type { Tab } from '$lib/types/Types';

    import { currJob, currJobId, currJobStopped } from '../../stores/nomadStore';

    const tabs: Tab[] = [
        {
            name: 'Shell',
            component: NomadController,
            icon: faTerminal
        },
        {
            name: 'Logs',
            component: LogController,
            icon: faFileAlt
        },
        {
            name: 'Settings',
            component: SettingsController,
            icon: faCog
        }
    ];

    onMount(() => {
        currJobId.set($page.params.id);
        currJob.set($page.data.job);
        currJobStopped.set($page.data.isStopped);
    });
</script>

<div class="px-4 pb-4 md:px-16">
    {#if $page.data.job}
        <div class="mb-4 flex justify-between">
            <div class="flex items-center font-sans font-bold">
                <Heading tag="h1" customSize="text-2xl md:text-4xl">
                    {$page.data.job.containerName}
                </Heading>
                {#if $page.data.job.expose}
                    <A aClass="ml-2" href="https://{$page.params.id}.plusvasis.xyz">
                        <Fa icon={faExternalLink} color="white" size="xs" />
                    </A>
                {/if}
            </div>
            <ContainerOptions />
        </div>
        <Tabs {tabs} />
    {:else}
        <Modal title="Error" open={true}>
            <div class="grid w-40 justify-center">
                <p class="text-base leading-relaxed text-gray-500 dark:text-gray-400">{$page.data.error}</p>
                <div class="mt-4 flex justify-center">
                    <Button color="blue" href="/">Return</Button>
                </div>
            </div>
        </Modal>
    {/if}
</div>

```

This component is used for the individual container pages and in itself utilises 5 other component. It uses the Tabs component to flick between three different components while staying on the same route.

Here's a look at the Tabs component.

```svelte
<script lang="ts">
    import { TabItem, Tabs } from 'flowbite-svelte';
    import Fa from 'svelte-fa';

    import type { Tab } from '$lib/types/Types';

    export let tabs: Tab[];
</script>

<Tabs style="underline">
    {#if !tabs || tabs.length === 0}
        <TabItem open>
            <div slot="title">No tabs</div>
        </TabItem>
    {:else}
        {#each tabs as tab, index}
            <TabItem open={index === 0}>
                <div slot="title" class="flex items-center gap-2 text-xs md:text-sm">
                    <Fa icon={tab.icon} color="white" />
                    {tab.name}
                </div>
                <svelte:component this={tab.component} />
            </TabItem>
        {/each}
    {/if}
</Tabs>
```

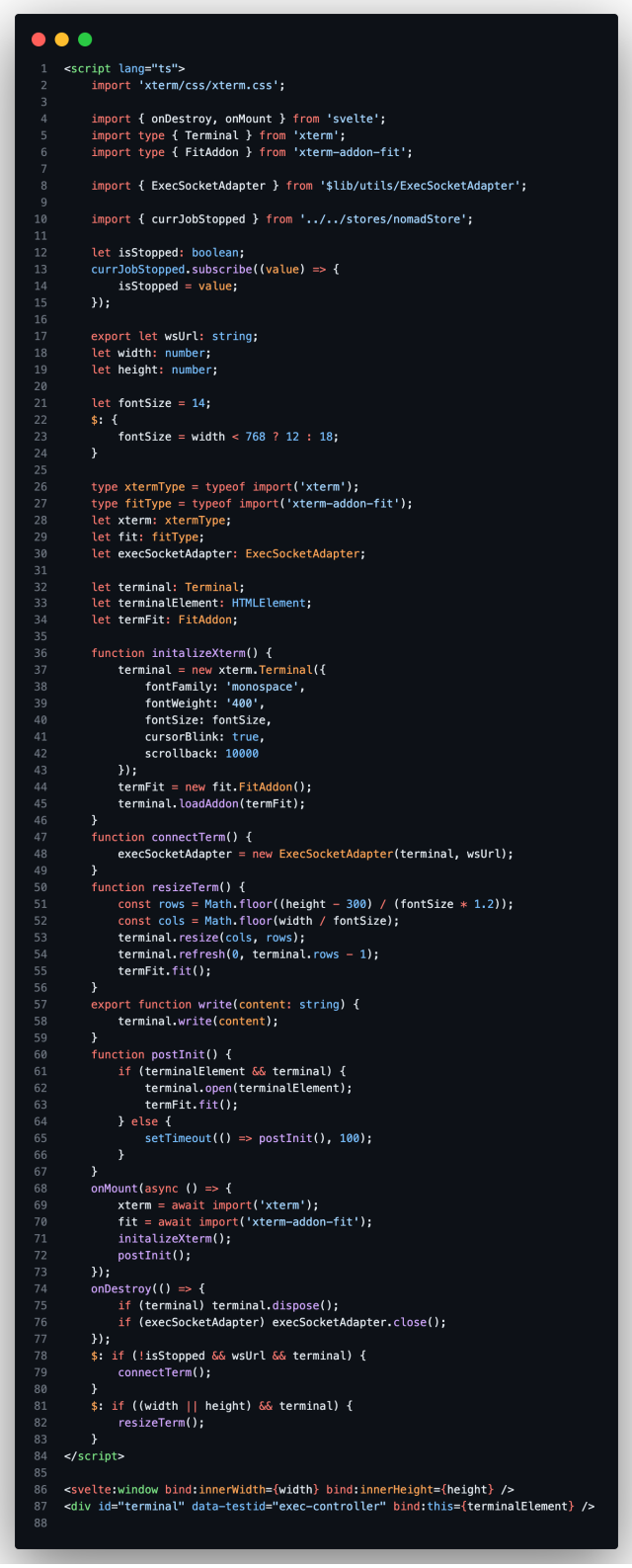
It simply creates a Tab bar with a list of TabItems which in our case relate to components. The components we loop through are the NomadController for the container shell, the LogController for the container logs and the SettingsController for the container settings.

Here we have the NomadController.

```svelte
<script lang="ts">
    import ExecController from '$lib/components/ExecController.svelte';
    import type { Job } from '$lib/types/Types';

    import { token } from '../../stores/auth';
    import { hostname } from '../../stores/environmentStore';
    import { currJob, currJobId, currJobStopped } from '../../stores/nomadStore';

    let execControllerComponent: ExecController;
    let wsUrl: string;

    let jobId: string;
    let job: Job;
    let isStopped: boolean;
    let authToken: string | undefined;
    currJobId.subscribe((value) => {
        jobId = value;
    });
    currJob.subscribe((value) => {
        job = value;
    });
    currJobStopped.subscribe((value) => {
        isStopped = value;
    });
    token.subscribe((value) => {
        authToken = value;
    });

    function setExecUrl() {
        const url = new URL(`${hostname}/job/${jobId}/exec`);
        url.protocol = url.protocol.replace('http', 'ws');
        url.searchParams.append('command', `["${job.shell}"]`);

        if (authToken) url.searchParams.append('access_token', authToken);

        wsUrl = url.toString();
    }

    $: if (!isStopped && jobId && job && authToken) {
        setExecUrl();
    }
</script>

<ExecController bind:this={execControllerComponent} {wsUrl} />
```

The NomadController sets current job attributes imported from our stores and also sets the websocket URL with the setExecUrl function.

We then use the ExecController Component to display the terminal to a user.



```
<script lang="ts">
    import 'xterm/css/xterm.css';

    import { onDestroy, onMount } from 'svelte';
    import type { Terminal } from 'xterm';
    import type { FitAddon } from 'xterm-addon-fit';

    import { ExecSocketAdapter } from '$lib/utils/ExecSocketAdapter';

    import { currJobStopped } from '../../stores/nomadStore';

    let isStopped: boolean;
    currJobStopped.subscribe((value) => {
        isStopped = value;
    });

    export let wsUrl: string;
    let width: number;
    let height: number;

    let fontSize = 14;
    $: {
        fontSize = width < 768 ? 12 : 18;
    }

    type xtermType = typeof import('xterm');
    type fitType = typeof import('xterm-addon-fit');
    let xterm: xtermType;
    let fit: fitType;
    let execSocketAdapter: ExecSocketAdapter;

    let terminal: Terminal;
    let terminalElement: HTMLElement;
    let termFit: FitAddon;

    function initalizeXterm() {
        terminal = new xterm.Terminal({
            fontFamily: 'monospace',
            fontWeight: '400',
            fontSize: fontSize,
            cursorBlink: true,
            scrollback: 10000
        });
        termFit = new fit.FitAddon();
        terminal.loadAddon(termFit);
    }
    function connectTerm() {
        execSocketAdapter = new ExecSocketAdapter(terminal, wsUrl);
    }
    function resizeTerm() {
        const rows = Math.floor((height - 300) / (fontSize * 1.2));
        const cols = Math.floor(width / fontSize);
        terminal.resize(cols, rows);
        terminal.refresh(0, terminal.rows - 1);
        termFit.fit();
    }
    export function write(content: string) {
        terminal.write(content);
    }
    function postInit() {
        if (terminalElement && terminal) {
            terminal.open(terminalElement);
            termFit.fit();
        } else {
            setTimeout(() => postInit(), 100);
        }
    }
    onMount(async () => {
        xterm = await import('xterm');
        fit = await import('xterm-addon-fit');
        initalizeXterm();
        postInit();
    });
    onDestroy(() => {
        if (terminal) terminal.dispose();
        if (execSocketAdapter) execSocketAdapter.close();
    });
    $: if (!isStopped && wsUrl && terminal) {
        connectTerm();
    }
    $: if ((width || height) && terminal) {
        resizeTerm();
    }
</script>

<svelte:window bind:innerWidth={width} bind:innerHeight={height} />
<div id="terminal" data-testid="exec-controller" bind:this={terminalElement} />

```

This ExecController component is simply used for managing the terminal displayed to a user on the Shell tab of the Container page.

For viewing the container logs we have the LogController component.

```svelte
<script lang="ts">
    import { Button, Chevron, Dropdown, Radio } from 'flowbite-svelte';
    import { onDestroy, onMount } from 'svelte';

    import { decode, getStream } from '$lib/utils/StreamLogs';

    import { currJobStopped } from '../../stores/nomadStore';

    let isStopped: boolean;
    currJobStopped.subscribe((value) => {
        isStopped = value;
    });

    let type = 'stdout';
    let logs = '';
    let abortController: AbortController;

    let preEl: HTMLPreElement;
    let stream: ReadableStream;

    // https://github.com/hashicorp/nomad/blob/main/ui/app/utils/classes/stream-logger.js
    async function readLogs() {
        logs = '';
        let streamClosed = false;
        let buffer = '';
        const decoder = new TextDecoder();
        const reader = stream.getReader();

        while (!streamClosed) {
            try {
                const { done, value } = await reader.read();
                streamClosed = done;
                buffer += decoder.decode(value, { stream: true });
                if (buffer.indexOf('}') !== -1) {
                    const match = buffer.match(/(.*\})(.*)$/);
                    if (match) {
                        const [, chunk, newBuffer] = match;
                        buffer = newBuffer;
                        const result = decode(chunk);
                        if (result) {
                            logs += result.message;
                        }
                    }
                }
            } catch (error) {
                if (!(error instanceof DOMException && error.name === 'AbortError')) {
                    console.error('Error reading logs:', error);
                }
                break;
            }
        }
    }

    async function changeType() {
        if (stream) abortController.abort();
        if (!isStopped) {
            abortController = new AbortController();
            stream = await getStream(type, abortController);
            readLogs();
        }
    }
    function scrollToBottom() {
        if (preEl) {
            preEl.scrollTop = preEl.scrollHeight;
        }
    }

    onMount(async () => {
        if (!isStopped) {
            abortController = new AbortController();
            stream = await getStream(type, abortController);
            readLogs();
        }
    });
    onDestroy(() => {
        if (stream) abortController.abort();
    });

    $: {
        if (logs)
            requestAnimationFrame(() => {
                setTimeout(scrollToBottom, 0);
            });
    }
</script>

<div class="pb-4">
    <Button><Chevron>{type}</Chevron></Button>
    <Dropdown class="w-32 space-y-3 p-3 text-sm">
        <li>
            <Radio name="type" bind:group={type} value={'stdout'} on:change={changeType}>stdout</Radio>
        </li>
        <li>
            <Radio name="type" bind:group={type} value={'stderr'} on:change={changeType}>stderr</Radio>
        </li>
    </Dropdown>
</div>

<pre class="pre-logs" bind:this={preEl}>{logs}
</pre>

```

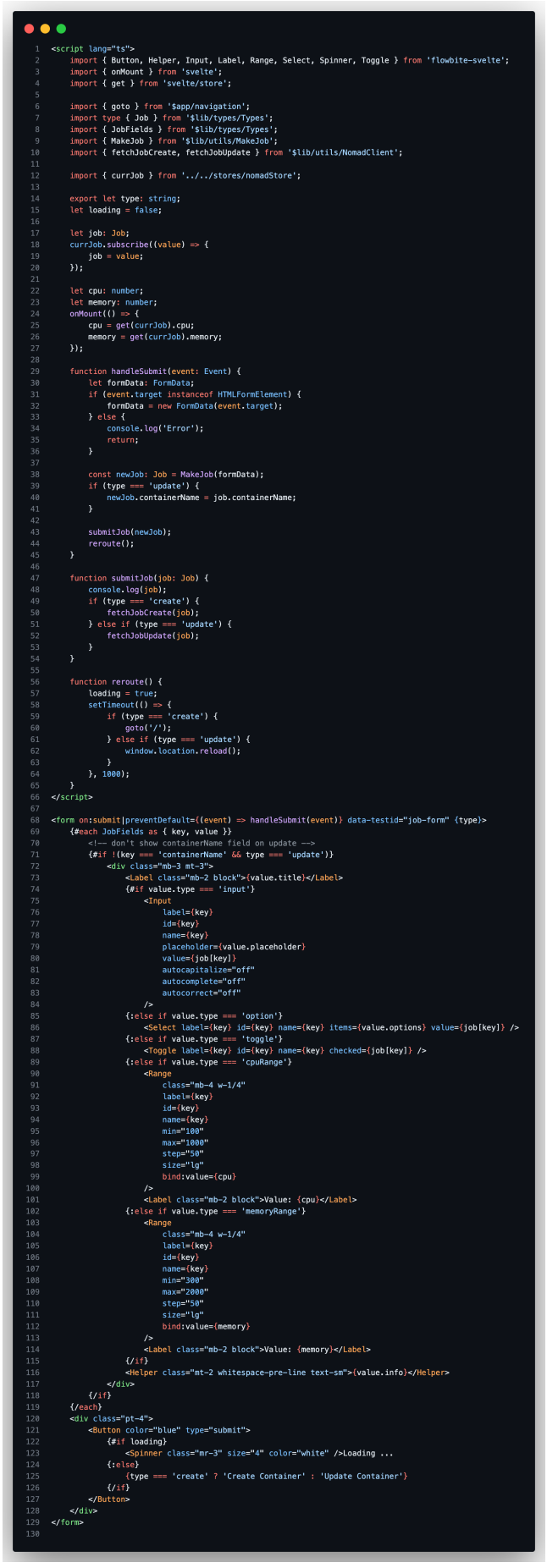
Here we use a stream to read either the standard output or standard error logs of that given container.

For the containers settings we have the SettingsController component.

```svelte
<script lang="ts">
    import type { Job } from '$lib/types/Types';

    import { currJob, currJobId } from '../../stores/nomadStore';
    import JobForm from './JobForm.svelte';

    let jobId: string;
    let job: Job;
    currJobId.subscribe((value) => {
        jobId = value;
    });
    currJob.subscribe((value) => {
        job = value;
    });
</script>

<JobForm type="update" />

```

We set the current job attributes by using our stores like before on the NomadController and use the JobForm component with type “update” to update any of the given current containers settings.

Here's the JobForm component.



```svelte
<script lang="ts">
    import { Button, Helper, Input, Label, Range, Select, Spinner, Toggle } from 'flowbite-svelte';
    import { onMount } from 'svelte';
    import { get } from 'svelte/store';

    import { goto } from '$app/navigation';
    import type { Job } from '$lib/types/Types';
    import { JobFields } from '$lib/types/Types';
    import { MakeJob } from '$lib/utils/MakeJob';
    import { fetchJobCreate, fetchJobUpdate } from '$lib/utils/NomadClient';

    import { currJob } from '../../stores/nomadStore';

    export let type: string;
    let loading = false;

    let job: Job;
    currJob.subscribe((value) => {
        job = value;
    });

    let cpu: number;
    let memory: number;
    onMount(() => {
        cpu = get(currJob).cpu;
        memory = get(currJob).memory;
    });

    function handleSubmit(event: Event) {
        let formData: FormData;
        if (event.target instanceof HTMLFormElement) {
            formData = new FormData(event.target);
        } else {
            console.log('Error');
            return;
        }

        const newJob: Job = MakeJob(formData);
        if (type === 'update') {
            newJob.containerName = job.containerName;
        }

        submitJob(newJob);
        reroute();
    }

    function submitJob(job: Job) {
        console.log(job);
        if (type === 'create') {
            fetchJobCreate(job);
        } else if (type === 'update') {
            fetchJobUpdate(job);
        }
    }

    function reroute() {
        loading = true;
        setTimeout(() => {
            if (type === 'create') {
                goto('/');
            } else if (type === 'update') {
                window.location.reload();
            }
        }, 1000);
    }
</script>

<form on:submit|preventDefault={(event) => handleSubmit(event)} data-testid="job-form" {type}>
    {#each JobFields as { key, value }}
        <!-- don't show containerName field on update -->
        {#if !(key === 'containerName' && type === 'update')}
            <div class="mb-3 mt-3">
                <Label class="mb-2 block">{value.title}</Label>
                {#if value.type === 'input'}
                    <Input
                        label={key}
                        id={key}
                        name={key}
                        placeholder={value.placeholder}
                        value={job[key]}
                        autocapitalize="off"
                        autocomplete="off"
                        autocorrect="off"
                    />
                {:else if value.type === 'option'}
                    <Select label={key} id={key} name={key} items={value.options} value={job[key]} />
                {:else if value.type === 'toggle'}
                    <Toggle label={key} id={key} name={key} checked={job[key]} />
                {:else if value.type === 'cpuRange'}
                    <Range
                        class="mb-4 w-1/4"
                        label={key}
                        id={key}
                        name={key}
                        min="100"
                        max="1000"
                        step="50"
                        size="lg"
                        bind:value={cpu}
                    />
                    <Label class="mb-2 block">Value: {cpu}</Label>
                {:else if value.type === 'memoryRange'}
                    <Range
                        class="mb-4 w-1/4"
                        label={key}
                        id={key}
                        name={key}
                        min="300"
                        max="2000"
                        step="50"
                        size="lg"
                        bind:value={memory}
                    />
                    <Label class="mb-2 block">Value: {memory}</Label>
                {/if}
                <Helper class="mt-2 whitespace-pre-line text-sm">{value.info}</Helper>
            </div>
        {/if}
    {/each}
    <div class="pt-4">
        <Button color="blue" type="submit">
            {#if loading}
                <Spinner class="mr-3" size="4" color="white" />Loading ...
            {:else}
                {type === 'create' ? 'Create Container' : 'Update Container'}
            {/if}
        </Button>
    </div>
</form>

```

This component is used for creating new jobs or updating jobs and uses a submitable form to do so that either calls our fetchJobCreate or fetchJobUpdate methods on the job and then re routes the user to either the homepage or the shell tab of the container page.

Now we have our ContainerOptions component.

```svelte
<script lang="ts">
    import { faPlay, faRefresh, faStop, faTrash } from '@fortawesome/free-solid-svg-icons';
    import { Button, ButtonGroup } from 'flowbite-svelte';
    import Fa from 'svelte-fa';

    import { goto } from '$app/navigation';
    import {
        fetchJobDelete,
        fetchJobRestart,
        fetchJobStart,
        fetchJobStop
    } from '$lib/utils/NomadClient';

    import { currJobStopped } from '../../stores/nomadStore';

    let width: number;
    let size: 'xs' | 'sm' | 'md' | 'lg' | 'xl' | undefined;

    $: {
        size = width < 768 ? 'xs' : 'md';
    }
</script>

<svelte:window bind:innerWidth={width} />
<ButtonGroup {size}>
    <Button
        {size}
        disabled={!$currJobStopped}
        on:click={() => fetchJobStart().then(() => window.location.reload())}
    >
        <Fa icon={faPlay} color="green" class="md:mr-2" />
        <span class="hidden md:block">Start</span>
    </Button>
    <Button
        {size}
        disabled={$currJobStopped}
        on:click={() => fetchJobRestart().then(() => window.location.reload())}
    >
        <Fa icon={faRefresh} color="orange" class="md:mr-2" />
        <span class="hidden md:block">Restart</span>
    </Button>
    <Button
        {size}
        disabled={$currJobStopped}
        on:click={() => fetchJobStop().then(() => window.location.reload())}
    >
        <Fa icon={faStop} color="red" class="md:mr-2" />
        <span class="hidden md:block">Stop</span>
    </Button>
    <Button {size} on:click={() => fetchJobDelete().then(() => goto('/'))}>
        <Fa icon={faTrash} class="md:mr-2" />
        <span class="hidden md:block">Delete</span>
    </Button>
</ButtonGroup>

```

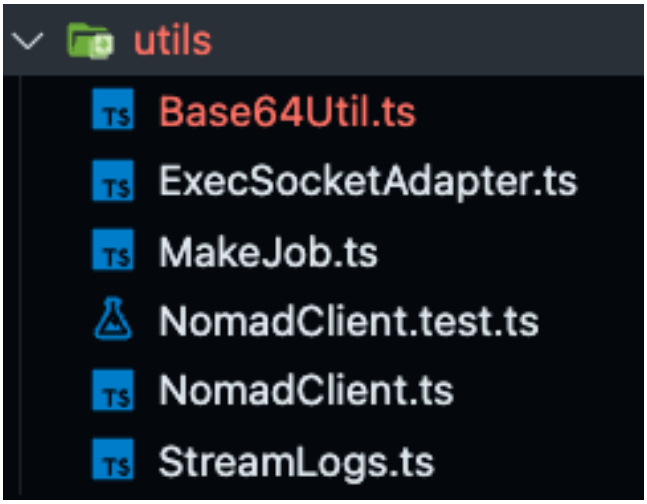
This is used for the four buttons on the right hand side of the page under the NavBar on the container page, which allows users to start, stop, restart or delete the container by calling methods from our utils to do the fetch requests to our API to make these possible.

Finally we have the Editor component.

```svelte
<script lang="ts">
    import { LanguageSupport, StreamLanguage } from '@codemirror/language';
    import * as yamlMode from '@codemirror/legacy-modes/mode/yaml';
    import { oneDark } from '@codemirror/theme-one-dark';
    import { onMount } from 'svelte';
    import CodeMirror from 'svelte-codemirror-editor';

    export let value = '';
    const yaml: LanguageSupport = new LanguageSupport(StreamLanguage.define(yamlMode.yaml));

    const placeholder = `# example docker-compose
# with custom plusvasis labels

version: "3.8"

services:
  nginx:
    container_name: "nginx"
    image: "nginx"
    expose:
      - 80
    environment:
      TEST_VAR: "Hello World!"
    volumes:
      - "data:/usr/share/nginx"
    labels:
      shell: "/bin/bash"
      cpu: "100"
      memory: "300"

volumes:
  data:

`;

    // disable grammarly
    onMount(() => {
        const cmContent = document.getElementsByClassName('cm-content');
        cmContent[0].setAttribute('data-enable-grammarly', 'false');
    });
</script>

<CodeMirror
    bind:value
    lang={yaml}
    theme={oneDark}
    styles={{
        '&': {
            width: '100%',
            height: '70vh'
        }
    }}
    {placeholder}
    class="text-white"
/>

```

This is the component used for an in-browser text editor, where users can paste or write a docker-compose file which we then convert into Nomad jobs.

Utils



Here we stored any TypeScript utils that were used across components or the routes pages.

```
import base64js from 'base64-js';
import { TextDecoderLite, TextEncoderLite } from 'text-encoder-lite';

export function b64encode(string: string) {
    const encoded = new TextEncoderLite('utf-8').encode(string);
    return base64js.fromByteArray(encoded);
}
export function b64decode(string: string) {
    const uint8array = base64js.toByteArray(string);
    return new TextDecoderLite('utf-8').decode(uint8array);
}

```

The Base64Util.ts file is used for message encoding. The reason for this to allow control characters to be json serializable in our live terminal.

```ts
import type * as xterm from 'xterm';

import { b64decode, b64encode } from './Base64Util.js';

export const HEARTBEAT_INTERVAL = 10000;
export const MAX_RETRIES = 10;

export class ExecSocketAdapter {
    terminal: xterm.Terminal;
    socket: WebSocket;
    heartbeatTimer: NodeJS.Timer | undefined;
    retries: number;

    constructor(terminal: xterm.Terminal, url: string) {
        this.terminal = terminal;
        this.socket = new WebSocket(url);
        this.retries = 0;

        if (!this.terminal) {
            throw new Error('Terminal is not defined.');
        }
        this.terminal.onData((data) => {
            this.handleData(data);
        });
        this.connect();
    }

    connect() {
        this.socket.onopen = () => {
            this.sendWsHandshake();
            this.sendTtySize();
            this.startHeartbeat();
            this.terminal.reset();
        };

        this.socket.onmessage = (e) => {
            const json = JSON.parse(e.data);

            if (json.stdout && json.stdout.data) {
                this.retries = 0; // reset retries on successful message
                this.terminal.write(b64decode(json.stdout.data));
            }
        };

        this.socket.onclose = () => {
            this.stopHeartbeat();
            if (this.retries < MAX_RETRIES) {
                this.retries++;
                setTimeout(() => {
                    this.reconnect();
                }, 1000);
            } else {
                this.terminal.reset();
                this.terminal.write('Failed to connect to server');
            }
        };

        this.terminal.onResize(() => {
            if (this.socket.readyState != WebSocket.OPEN) {
                return;
            }
            this.sendTtySize();
        });
    }

    reconnect() {
        console.log('reconnecting...');
        this.socket = new WebSocket(this.socket.url);
        this.connect();
    }

    sendTtySize() {
        this.socket.send(
            JSON.stringify({
                tty_size: { width: this.terminal.cols, height: this.terminal.rows }
            })
        );
    }

    sendWsHandshake() {
        this.socket.send(JSON.stringify({ version: 1, auth_token: '' }));
    }

    startHeartbeat() {
        this.heartbeatTimer = setInterval(() => {
            this.socket.send(JSON.stringify({}));
        }, HEARTBEAT_INTERVAL);
    }

    stopHeartbeat() {
        clearInterval(this.heartbeatTimer);
    }

    handleData(data: string) {
        if (this.socket.readyState != WebSocket.OPEN) {
            return;
        }
        this.socket.send(JSON.stringify({ stdin: { data: b64encode(data) } }));
    }

    close() {
        this.socket.close();
    }
}

```

The ExecSocketAdapter.ts file is used to manage the websocket connections - also for the live terminal.

```ts
import type { DockerCompose, Job } from '$lib/types/Types';

import { user } from '../../stores/auth';

let uid: string | undefined;
user.subscribe((value) => {
    uid = value?.uid;
});
export function MakeJob(formData: FormData) {
    const containerName = formData.get('containerName') as string;
    const dockerImage = formData.get('dockerImage') as string;
    const shell = formData.get('shell') as string;
    const volumeStr = formData.get('volumes') as string;
    const envStr = formData.get('env') as string;
    const port = formData.get('port') as string;
    const exposeStr = formData.get('expose') as string;
    const cpu = formData.get('cpu') as string;
    const memory = formData.get('memory') as string;

    const volumes: [string, string][] = [];
    for (const volume of volumeStr.split(',')) {
        if (volume === '') {
            continue;
        }
        volumes.push(volume.split(':') as [string, string]);
    }

    const envs: [string, string][] = [];
    for (const env of envStr.split(',')) {
        if (env === '') {
            continue;
        }
        envs.push(env.split('=') as [string, string]);
    }

    const expose: boolean = exposeStr != null;

    // When creating a job, we need to make sure we have the user's uid
    if (!uid) throw new Error('uid is undefined');

    const job: Job = {
        user: uid,
        containerName: containerName,
        dockerImage: dockerImage,
        shell: shell,
        volumes: volumes,
        env: envs,
        port: Number(port),
        expose: expose,
        cpu: Number(cpu),
        memory: Number(memory)
    };

    return job;
}

```

–

```ts
export function MakeJobsFromCompose(dockerCompose: DockerCompose): Job[] {
    const jobs: Job[] = [];
    if (!uid) throw new Error('uid is undefined');

    for (const serviceName in dockerCompose.services) {
        const service = dockerCompose.services[serviceName];

        // required
        const containerName = service.container_name;
        const dockerImage = service.image;

        // optional
        const volumes: [string, string][] = [];
        const envs: [string, string][] = [];
        let port = 0;
        let expose = false;

        // defaults
        let shell = '/bin/sh';
        let cpu = 100;
        let memory = 300;

        if (service.labels) {
            let labels = service.labels;

            // if labels string array, convert to record
            if (Array.isArray(labels)) {
                const labelsRecord: Record<string, string> = {};
                for (const label of labels) {
                    const [key, value] = label.split('=');
                    labelsRecord[key] = value;
                }
                labels = labelsRecord;
            }

            shell = labels['shell'] || '/bin/sh';
            cpu = Number(labels['cpu']) || 100;
            memory = Number(labels['memory']) || 300;
        }
        if (service.expose) {
            port = service.expose[0] || 0;
            expose = true;
        }
        if (service.environment) {
            let env = service.environment;

            // if env string array, convert to record
            if (Array.isArray(env)) {
                const envRecord: Record<string, string> = {};
                for (const e of env) {
                    if (e === '') {
                        continue;
                    }
                    const [key, value] = e.split('=');
                    envRecord[key] = value;
                }
                env = envRecord;
            }
            for (const [key, value] of Object.entries(env)) {
                envs.push([key, value]);
            }
        }
        if (service.volumes) {
            for (const vol of service.volumes) {
                if (vol === '') {
                    continue;
                }
                const [host, container] = vol.split(':');
                volumes.push([host, container]);
            }
        }

        const job = {
            user: uid,
            containerName: containerName,
            dockerImage: dockerImage,
            shell: shell,
            volumes: volumes,
            env: envs,
            port: port,
            expose: expose,
            cpu: cpu,
            memory: memory
        };

        jobs.push(job);
    }

    return jobs;
}
```

The MakeJob.ts file is used for creating and updating jobs and setting the new attributes from the JobForm component previously mentioned or creating multiple jobs from an imported docker-compose file.

```ts
import fetch from 'cross-fetch';

import type { Job } from '$lib/types/Types';

import { token } from '../../stores/auth';
import { hostname } from '../../stores/environmentStore';
import { currJobId } from '../../stores/nomadStore';

let jobId: string;
let authToken: string | undefined;
currJobId.subscribe((value) => {
    jobId = value;
});
token.subscribe((value) => {
    authToken = value;
});

export async function fetchJobCreate(job: Job) {
    const url = `${hostname}/jobs`;
    const res = await fetch(url, {
        method: 'POST',
        body: JSON.stringify(job),
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Created');
    } else {
        console.log('Error');
    }
}

export async function fetchJobUpdate(job: Job) {
    const url = `${hostname}/job/${jobId}`;
    const res = await fetch(url, {
        method: 'POST',
        body: JSON.stringify(job),
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Updated');
    } else {
        console.log('Error');
    }
}

```

–

```
export async function fetchJobStop() {
    const url = `${hostname}/job/${jobId}`;
    const res = await fetch(url, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Stopped');
    } else {
        console.log('Error');
    }
}

export async function fetchJobDelete() {
    const url = `${hostname}/job/${jobId}?purge=true`;
    const res = await fetch(url, {
        method: 'DELETE',
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Deleted');
    } else {
        console.log('Error');
    }
}

export async function fetchJobRestart() {
    const url = `${hostname}/job/${jobId}/restart`;
    const res = await fetch(url, {
        method: 'POST',
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Restarted');
    } else {
        console.log('Error');
    }
}

export async function fetchJobStart() {
    const url = `${hostname}/job/${jobId}/start`;
    const res = await fetch(url, {
        method: 'GET',
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });

    if (res.ok) {
        console.log('Container Started');
    } else {
        console.log('Error');
    }
}
```

The NomadClient.ts file is used to store all the async functions that perform the fetch requests to our API, and is used throughout our frontend.

```ts
import type { Job } from '$lib/types/Types';

import { token } from '../../stores/auth';
import { hostname } from '../../stores/environmentStore';
import { currJob, currJobId } from '../../stores/nomadStore';
import { b64decode } from './Base64Util';

let job: Job;
let jobId: string;
let authToken: string | undefined;
currJob.subscribe((value) => {
    job = value;
});
currJobId.subscribe((value) => {
    jobId = value;
});
token.subscribe((value) => {
    authToken = value;
});

// https://github.com/hashicorp/nomad/blob/main/ui/app/utils/stream-frames.js
export function decode(chunk: string): { offset: number; message: string } | null {
    const lines = chunk.replace(/\}\{/g, '}\n{').split('\n').filter(Boolean);
    const frames = lines.map((line) => JSON.parse(line)).filter((frame) => frame.Data);

    if (frames.length) {
        frames.forEach((frame) => (frame.Data = b64decode(frame.Data)));
        return {
            offset: frames[frames.length - 1].Offset,
            message: frames.map((frame) => frame.Data).join('')
        };
    }

    return null;
}

export async function getStream(type: string, abortController: AbortController) {
    const urlBuilder = new URL(`${hostname}/job/${jobId}/logs`);
    urlBuilder.searchParams.append('task', job.containerName);
    urlBuilder.searchParams.append('type', type);
    const url = urlBuilder.toString();

    const response = await fetch(url, {
        signal: abortController.signal,
        headers: {
            Authorization: `Bearer ${authToken}`
        }
    });
    if (!response.ok) {
        throw new Error(response.statusText);
    } else if (!response.body) {
        throw new Error('No response body');
    }

    return response.body as ReadableStream;
}

```

Lastly the StreamLogs.ts file is used for getting the stream for the container logs.

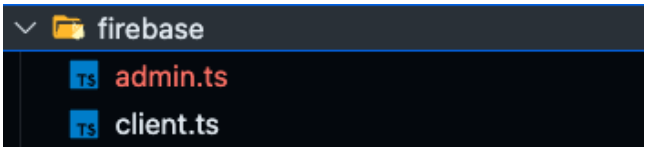
Types

In our types directory we have one file Types.ts.

Here we specify the typings of a Job, the job we wish to create on Nomad for the users container, JobFields, which we use to define the inputs for a user creating a job, Tab for the tabs displayed to the user, Service, which relates to the service section of a docker compose and of a nomad job and finally DockerCompose - there can be multiple services in a DockerCompose.

Firebase

Structure



Admin

```ts
import { cert, getApps, initializeApp } from 'firebase-admin/app';
import { getAuth } from 'firebase-admin/auth';
import { getFirestore } from 'firebase-admin/firestore';

import { FIREBASE_ADMIN_CLIENT_EMAIL, FIREBASE_ADMIN_PRIVATE_KEY } from '$env/static/private';
import { PUBLIC_FIREBASE_PROJECT_ID } from '$env/static/public';

function makeApp() {
    const apps = getApps();
    if (apps.length > 0) {
        return apps[0];
    }

    return initializeApp({
        credential: cert({
            privateKey: FIREBASE_ADMIN_PRIVATE_KEY,
            clientEmail: FIREBASE_ADMIN_CLIENT_EMAIL,
            projectId: PUBLIC_FIREBASE_PROJECT_ID
        }),
        databaseURL: `https://${PUBLIC_FIREBASE_PROJECT_ID}.firebaseio.com`
    });
}

export const firebase = makeApp();
export const auth = getAuth(firebase);
export const firestore = getFirestore();
```

The admin file allows us to get a firebase auth instance using an admin service account. The credentials for this account are stored in a .env file. These are then used to verify users during any server side rendered requests that perform a fetch and require the user's id to be sent to the API - this is all without client interaction.

Client

```ts
import { getApps, initializeApp } from 'firebase/app';
import { getAuth } from 'firebase/auth';

import {
    PUBLIC_FIREBASE_API_KEY,
    PUBLIC_FIREBASE_APP_ID,
    PUBLIC_FIREBASE_AUTH_DOMAIN,
    PUBLIC_FIREBASE_MESSAGE_SENDER_ID,
    PUBLIC_FIREBASE_PROJECT_ID,
    PUBLIC_FIREBASE_STORAGE_BUCKET
} from '$env/static/public';

function makeApp() {
    const apps = getApps();
    if (apps.length > 0) {
        return apps[0];
    }

    return initializeApp({
        apiKey: PUBLIC_FIREBASE_API_KEY,
        authDomain: PUBLIC_FIREBASE_AUTH_DOMAIN,
        projectId: PUBLIC_FIREBASE_PROJECT_ID,
        storageBucket: PUBLIC_FIREBASE_STORAGE_BUCKET,
        messagingSenderId: PUBLIC_FIREBASE_MESSAGE_SENDER_ID,
        appId: PUBLIC_FIREBASE_APP_ID,
        databaseURL: `https://${PUBLIC_FIREBASE_PROJECT_ID}.firebaseio.com`
    });
}

export const firebase = makeApp();
export const auth = getAuth(firebase);
```

The client file is used in any client side code and is what allows users to login or register accounts with firebase. The data for this is also stored in a .env file for convenience sake, but is PUBLIC as this code runs on the client while the above admin code only runs on the server.

Stores

We use stores in Svelte to store variables or constants that can be accessible across the whole project and from any file (except for server side fetch requests!)

```ts
import cookie from 'cookie';
import {
    createUserWithEmailAndPassword,
    GithubAuthProvider,
    signInWithEmailAndPassword,
    signInWithPopup,
    type User
} from 'firebase/auth';
import { writable } from 'svelte/store';

import { browser } from '$app/environment';
import { auth } from '$lib/firebase/client';

export const user = writable<User | null>(null);
export const token = writable<string | undefined>(undefined);

export async function signOut() {
    return auth.signOut();
}

export async function signInWithGithub() {
    await signInWithPopup(auth, new GithubAuthProvider());
}

export async function signInWithEmail(email: string, password: string) {
    await signInWithEmailAndPassword(auth, email, password);
}

export async function createUserWithEmail(email: string, password: string) {
    await createUserWithEmailAndPassword(auth, email, password);
}

if (browser) {
    auth.onIdTokenChanged(async (newUser) => {
        const newToken = newUser ? await newUser?.getIdToken() : undefined;
        document.cookie = cookie.serialize('token', newToken ?? '', {
            path: '/',
            maxAge: token ? undefined : 0
        });
        user.set(newUser);
        token.set(newToken);
    });

    // refresh the ID token every 10 minutes
    setInterval(async () => {
        if (auth.currentUser) {
            await auth.currentUser.getIdToken(true);
        }
    }, 10 * 60 * 1000);
}

```

This is our authStore.ts and it's used to store the user data if they're logged in and it stores the users token. We also keep the functions corresponding to authentication here such as signing in/ signing out or creating an account etc.

```ts
export const hostname = import.meta.env.PROD
    ? 'https://api.plusvasis.xyz'
    : 'http://localhost:8080';

```

This is our environmentStore.ts used for storing the hostname for our API, which will either be localhost during development or the production API endpoint when the frontend is also deployed in production.

```ts
import { writable } from 'svelte/store';

import type { Job } from '$lib/Types';

export const currJobId = writable('');
export const currJob = writable({} as Job);
export const currJobStopped = writable(false);
```

Lastly this is our nomadStore.ts and is used for storing the current job, the current jobs ID and a boolean to determine if the current job is stopped.

Nomad

https://github.com/hashicorp/nomad (https://github.com/hashicorp/nomad)
Nomad is an open-source product by HashiCorp (who make Terraform, Vault and Consul also) and is a simple and flexible scheduler and orchestrator for managing containers at scale.

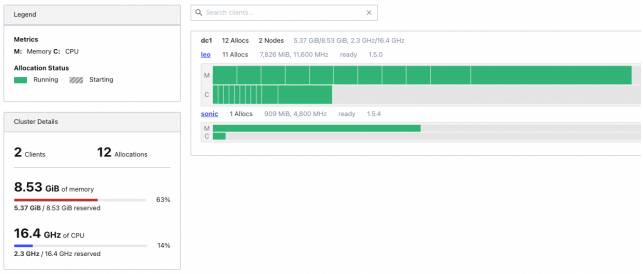
We decided to use Nomad instead of directly communicating with the Docker API ourselves, as it provides scalability and availability by default, and for a project like PlusVasis where users can use a lot of resources, we thought it was important to be scalable from the get-go.

Nomad exposes a HTTP API to provide interaction with jobs, allocations, volumes, etc - and provides great documentation for this, which can be found here: https://developer.hashicorp.com/nomad/api-docs (https://developer.hashicorp.com/nomad/api-docs)

It is this API that our backend Go REST API sends requests to, to perform all the different operations and features supported by PlusVasis.

Topology

Here is a screenshot from Nomad's Web UI showing our topology:



We can see here that we have two nodes - named "leo" and "sonic" - with 11 allocations deployed on "leo" and 1 allocation deployed on "sonic".

Server-Client

Here is the Nomad config for “leo”:

```
data_dir = "/opt/nomad/data"
bind_addr = "0.0.0.0"

name = "leo"

advertise {
  http = "192.168.1.201"
  rpc  = "192.168.1.201"
  serf = "192.168.1.201"
}

server {
  enabled = true
  bootstrap_expect = 1
}

client {
  enabled = true
  servers = ["127.0.0.1"]
}

plugin "docker" {
  config {
    volumes {
      enabled = true
    }
  }
}
```

We see that the server stanza is enabled, so this node will run as a master server, that does all the orchestrating across the Nomad cluster.

The client stanza is enabled also, so this node will allow allocations to deployed on it. We specify the IP address of the master server we connect to, in this case, it is the localhost IP address.

We also enable volume support for the docker plugin - this simply allows any docker containers deployed to mount docker volumes.

Client

Here is the Nomad config for “sonic”:

```
data_dir = "/opt/nomad/data"
bind_addr = "0.0.0.0"

name = "sonic"

advertise {
  http = "192.168.1.202"
  rpc  = "192.168.1.202"
  serf = "192.168.1.202"
}

client {
  enabled = true
  servers = ["192.168.1.201"]
}

plugin "docker" {
  config {
    volumes {
      enabled = true
    }
  }
}
```

The client stanza is enabled here, but differs from the other config, as the server speicified is the IP address of the “leo” node, and not localhost.

Docker volumes are enabled on this node also.

Example Nomad Job

Here is an example Nomad job that runs an nginx web server:

```
job "nginx" {
  datacenters = ["dc1"]
  type = "service"
  group "nginx" {
    count = 1
    network {
      mode = "bridge"
      port "http" {
        to = 80
      }
    }
    service {
      name = "nginx"
      port = "http"
      provider = "nomad"
      tags = [
        "traefik.enable=true",
        "traefik.http.routers.nginx.entrypoints=https",
        "traefik.http.routers.nginx.rule=Host(`nginx.plusvasis.xyz`)",
      ]
    }
    task "server" {
      driver = "docker"
      config {
        image = "nginx"
        ports = ["http"]
      }
    }
  }
}
```

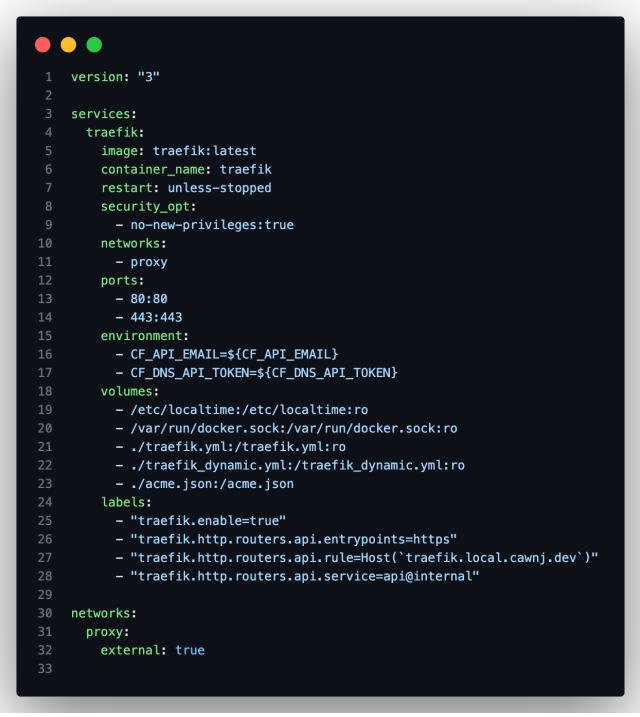
Traefik

https://github.com/traefik/traefik (https://github.com/traefik/traefik)
Traefik is an open-source reverse proxy and load balancer, which we use for providing SSL certificates for HTTPS and exposing of services within PlusVasis.

Traefik provides integration with both Docker and Nomad to discover services deployed on the network and provide routes for accessing these services over HTTP.

Deployment

Here is our `docker-compose.yml` file for Traefik:



```
version: "3"

services:
  traefik:
    image: traefik:latest
    container_name: traefik
    restart: unless-stopped
    security_opt:
      - no-new-privileges:true
    networks:
      - proxy
    ports:
      - 80:80
      - 443:443
    environment:
      - CF_API_EMAIL=${CF_API_EMAIL}
      - CF_DNS_API_TOKEN=${CF_DNS_API_TOKEN}
    volumes:
      - /etc/localtime:/etc/localtime:ro
      - /var/run/docker.sock:/var/run/docker.sock:ro
      - ./traefik.yml:/traefik.yml:ro
      - ./traefik_dynamic.yml:/traefik_dynamic.yml:ro
      - ./acme.json:/acme.json
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.api.entrypoints=https"
      - "traefik.http.routers.api.rule=Host(`traefik.local.cawnj.dev`)"
      - "traefik.http.routers.api.service=api@internal"

networks:
  proxy:
    external: true

```

This is what we use to deploy Traefik, important things to note here are the environment variables `CF_API_EMAIL` and `CF_DNS_API_TOKEN`. These are the credentials for API access to the Cloudflare account that provides the DNS resolution and SSL ceritificates for PlusVasis and its related domains.

Configuration

Here is our main `traefik.yml` config file:

```yaml
api:
  dashboard: true

entryPoints:
  http:
    address: ":80"
    http:
      redirections:
        entryPoint:
          to: "https"
          scheme: "https"
          permanent: true
  https:
    address: ":443"
    http:
      tls:
        certresolver: cloudflare
        domains:
          - main: "cawnj.dev"
            sans: "*.cawnj.dev"
          - main: "local.cawnj.dev"
            sans: "*.local.cawnj.dev"
          - main: "plusvasis.xyz"
            sans: "*.plusvasis.xyz"
          - main: "local.plusvasis.xyz"
            sans: "*.local.plusvasis.xyz"

certificatesResolvers:
  cloudflare:
    acme:
      storage: "acme.json"
      dnsChallenge:
        provider: "cloudflare"
        delayBeforeCheck: 30
        resolvers:
          - "1.1.1.1:53"

providers:
  docker:
    watch: true
    network: "proxy"
    exposedbydefault: false
    endpoint: "unix:///var/run/docker.sock"
  file:
    filename: "traefik_dynamic.yml"
  nomad:
    endpoint:
      address: "http://192.168.1.201:4646"

```

We see here that we have two entrypoints, `http` and `https` respectively. These are not defining url schemes, but simply are the names of the entrypoints.

The `http` entrypoint is mapped to port 80 and simply redirects any requests to the `https` entrypoint.

The `https` entrypoint is mapped to port 443 and also defines what certificate resolver to use for the listed domains - in our case we have a "certresolver" called `cloudflare`.

The `cloudflare` certificate resolver uses the API credentials supplied via environment variables in the `docker-compose.yml` file to grab SSL certificates from Cloudflare as needed. A dns challenge is required to prove ownership of the domain.

The `providers` block defines where Traefik will grab/watch rules and other configuration from, these are the following:

- Docker: from Docker labels for any services within the `proxy` Docker network.
  - example in Traefik's own docker-compose file
- File: from a file named `traefik_dynamic.yml`
  - example to be shown below
- Nomad: from any services deployed via Nomad
  - example in the Nomad nginx job provided above

Here is our `traefik_dynamic.yml` file:

```yaml
http:
  routers:
    nomad:
      rule: "Host(`nomad.local.cawnj.dev`)"
      entryPoints:
        - "https"
      middlewares:
        - "nomadHeaders"
      service: "nomad"
    webhook:
      rule: "Host(`webhook.plusvasis.xyz`)"
      entryPoints:
        - "https"
      service: "webhook"
  services:
    nomad:
      loadBalancer:
        servers:
          - url: "http://192.168.1.201:4646"
    webhook:
      loadBalancer:
        servers:
          - url: "http://192.168.1.201:9000"
  middlewares:
    nomadHeaders:
      headers:
        customRequestHeaders:
          Origin: "https://nomad.local.cawnj.dev"

```

In this file we can manually define host:port mappings for a given address, alongside this we can add custom middlewares.

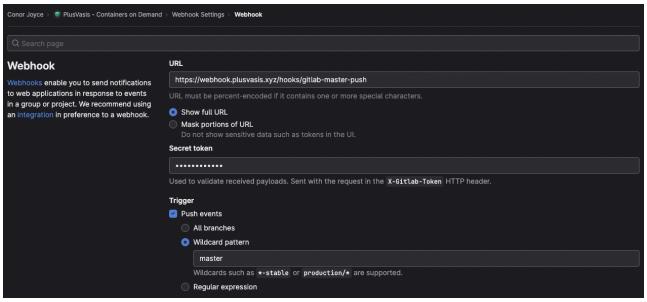
Here we see the rules for Nomad, which is running at `https://192.168.1.201:4646` and also has a middleware that changes the Origin header to be `https://nomad.local.cawnj.dev`. The reason for this is because Nomad's HTTP API does not allow other origins to access it. This rule was added to allow WebSockets requests from domains other than Nomad's own, which is required for our real-time terminal feature.

We also see the rules for a webhook service, this is used for automatic deployment of PlusVasis on pushes to master from GitLab, which we'll explain next.

Webhook

https://github.com/adnanh/webhook (https://github.com/adnanh/webhook)
We use a open-source project called “webhook” to provide a HTTP endpoint that we can use for continuous deployment purposes.

Gitlab allows us to send a HTTP request on changes to our repository. We use this feature to send a request to the webhook service on any push to the master branch of our project - this includes when any merge requests get merged.



Configuration

The webhook service is configured as such:

```
[
  {
    "id": "gitlab-master-push",
    "execute-command": "/etc/plusvasis/src/deploy.sh",
    "command-working-directory": "/etc/plusvasis/src",
    "trigger-rule":
    {
      "and":
      [
        {
          "match":
          {
            "type": "value",
            "value": "xxxxxxxx",
            "parameter":
            {
              "source": "header",
              "name": "X-Gitlab-Token"
            }
          }
        },
        {
          "match":
          {
            "type": "value",
            "value": "push",
            "parameter":
            {
              "source": "payload",
              "name": "object_kind"
            }
          }
        },
        {
          "match":
          {
            "type": "value",
            "value": "refs/heads/master",
            "parameter":
            {
              "source": "payload",
              "name": "ref"
            }
          }
        }
      ]
    }
  }
]
```

The webhook services allows us to provide trigger rules, which are the following:

- The secret token supplied in the `X-Gitlab-Token` request header matches a given value, redacted here
- The `object_kind` key in the request payload matches the value “push”
- The `red` key in the request payload matches the value “refs/heads/master”

These rules ensure that only we can trigger a deployment, through use of the secret token, and that the deployment will only ever run on pushes to the master branch.

Cloudflare

Cloudflare provides both DNS resolution and SSL ceritificates for PlusVasis and its related domains, as mentioned previous. Cloudflare also provides a proxy layer for any sites managed by it, enabling DDoS protection, analytics and other useful features.

Here are the DNS records exported from Cloudflare:

```
;; A Records
*.local.plusvasis.xyz.  1   IN  A   192.168.1.201
local.plusvasis.xyz.    1   IN  A   192.168.1.201
plusvasis.xyz.  1   IN  A   XXX.XXX.X.XXX

;; CNAME Records
*.plusvasis.xyz.    1   IN  CNAME   plusvasis.xyz.

```

A records

- plusvasis.xyz -> the public ip address where PlusVasis is deployed, which is redacted here
- local.plusvasis.xyz -> the local IP address of PlusVasis, for internal use
- *.local.plusvasis.xyz -> a wilcard record that maps any subdomain of local.plusvasis.xyz to the local IP address above

CNAME records

- *.plusvasis.xyz -> a wildcard record that maps any subdomain of plusvasis.xyz to plusvasis.xyz
    - This is used to provide the DNS resolution for any/all services deployed by PlusVasis.

Problems Solved

Framework Choice

When selecting a framework for our backend, we sought a solution that would enable us to build a performant and scalable REST API while minimizing the amount of boilerplate code. We evaluated several options and ultimately decided on Echo, a lightweight and feature-rich web framework for Go.

Echo stood out for its simplicity, ease of use, and excellent documentation. It provides a robust set of features for building RESTful APIs, including routing, middleware, request handling, and more, all while keeping the codebase concise and organized. Echo also has a large and active community, which made it easy to find resources and support when needed.

Moreover, Echo is highly performant and scalable, making it a great choice for building large-scale applications that can handle a high volume of requests. Overall, our choice of Echo as the framework for our backend was driven by its combination of ease of use, feature set, performance, and community support.

Docker Compose

Support for docker-compose files was something we wanted for the project straight away, as users that are familiar with Docker, are most likely familiar with docker-compose also, as it often the simplest way to create idempotent deployments with Docker.

While Docker is used to run our containers/jobs, we are not directly communicating with the Docker API, but with Nomad's API instead, therefore we needed to parse docker-compose and convert them to a format that Nomad will understand, not Docker.

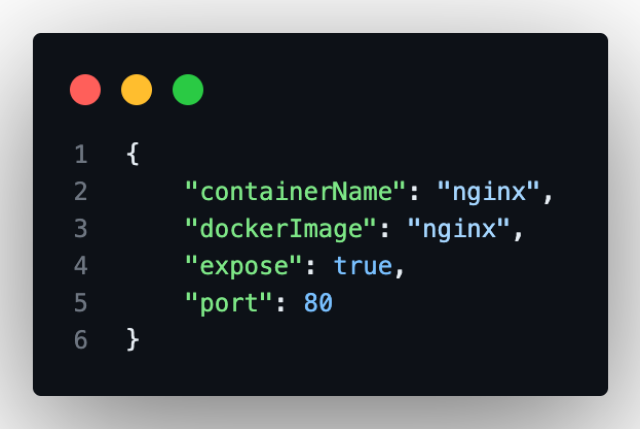
Nomad allows the most important features of Docker, but has replacements or its own implmentation for features that do not inherently work on a distributed environment. Therefore Nomad job specifications and docker-compose files are not directly translatable, even though on the surface it may seem that they perform the exact same task - container deployment.

The way we dealt with this was by instead of focusing on Docker containers, focusing on Nomad jobs instead. Our backend API's CreateJob endpoint accepts given fields, that we then use to create a Nomad job. What we then did is wrote a parser in the frontend that will grab the data that aligns with these fields from a docker-compose file.

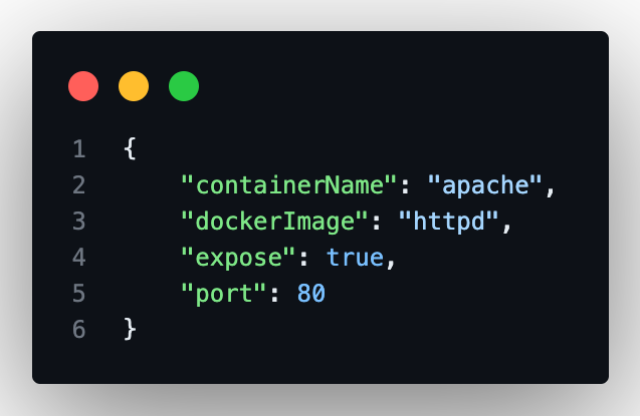
For example, in a docker-compose file that looks like this:

```yaml
version: "3.8"

services:
  nginx:
    container_name: "nginx"
    image: "nginx"
    expose:
      - 80
  apache:
    container_name: "apache"
    image: "httpd"
    expose:
      - 80
```

We can convert this into the following JSON blobs (with type-safety of course on both the frontend and backend, which is not as important to explain here):



and



Our API can then accept both of these and create Nomad jobs based on them.
All the options supported in PlusVasis in the API NomadJob type are therefore supported in docker-compose files also.

Networking

Two features we wanted to provide for users was:

- the ability to publicly exposes their containers
- allow communication between their containers

Enabling both of these features while maintaining the goal of simplicity for users was not going to be an easy task to accomplish, but luckily both Traefik and Nomad provide features that helped us accomplish both of these.

Exposing containers

Traefik does the work for us here is regards to routing, but we had to implement ourselves a way for Traefik to know what to route and where to.

We decided to only allow use of our https entrypoint in Traefik for simplicity and security purposes. Therefore only HTTP services can be publicly exposed. This means that a database cannot be accessed publicly, but a HTTP service can - but this service can and is able to consume data from a database, for example, if it is also deployed on PlusVasis. We felt this is a good compromise without drastically increasing the complexity of both our implementation and user experience.

The way we accomplished this was by allowing the user to enable exposing of their service, and specifying what port to expose.

We used these options in our job template, to allow users to expose their container at a domain like `user123-website.plusvasis.xyz`, for example. Nomad maps the provided port on the container to a random port on the host, and this is then passed to Traefik which can deal with the routing.

Inter-container communication

For this, we took advantage of Nomad's service discovery and templating features. We use the service discovery to get the IP address and dynamic port of a service and use templating to create an environment variable containing this data.

With this we can set environment variables dynamically, for example if a container has an environment variable that should contain the connection address to a database, we can resolve this to the correct IP address and port for the container that then contains that database.

Testing

Testing was a crucial aspect of our project, and we encountered a few challenges while implementing it. Initially, we tested our backend using integration and unit tests, using the Testify library to write assertions.

However, when we turned to testing our frontend, we faced some issues. We started with Jest due to its mocking capabilities, which would have been helpful for our planned component tests. However, we found it challenging to configure Jest with our SvelteKit project built on Vite and TypeScript.

After some research, we switched to Vitest, which offered similar capabilities to Jest while being better suited for our project. Vitest does not have as much community documentation due to being a less popular project than Jest. But with Vitest, we were able to efficiently test all the components of our frontend.

We later implemented end-to-end system tests using Cypress, leveraging our experience with testing our SvelteKit project.

Learning New Things

We encountered a number of new technologies during the development of our project, including SvelteKit, Echo, and Cypress. These were all new to us, and we had to learn how to use them effectively. We were able to overcome this challenge by spending time researching and practicing with these tools.

We started by reading documentation and watching tutorial videos, and then we began experimenting with them in our own code - writing simple proof of concepts to help learn the foundations for these technologies and how we could apply them in our project. We found that by working together and sharing knowledge on this, we were able to learn quickly and effectively.

We also made use of online resources, such as forums posts and discussion pages, to get insight on errors we faced or to see other developers' opinions on certain topics.

Ultimately, we were able to become proficient in these technologies, and we feel that the experience has made us better developers overall.

Error Handling

Error handling was an important consideration in our project, especially for the backend REST API written in Go using the Echo framework.

We followed the Go convention of emphasizing good error handling, and spent time testing various edge cases and implementing flexible error handling statements that could handle each scenario.
For example, we utilized HTTP status codes and Echo's wrappers for these status codes to ensure that the API could properly communicate errors to the frontend and so that these errors could be logged effectively in our backend.

In addition, we also utilized Svelte's error handling capabilities for the frontend, ensuring that errors were caught and displayed to the user in a clear and informative manner.

Overall, we prioritized good error handling practices throughout our project, both in the backend and frontend, to ensure that our application could both handle and communicate errors properly and provide a smooth user experience.

Testing

Integration/ Unit Testing

For testing the backend we wrote a set of integration/ unit tests that test each of the endpoints and controllers functionalities through use of valid mocks and assertions. We used the testify library for the assertions. For the nomad controller we were able to mock the nomad client like so.

```go
type MockNomadClient struct {
    mock.Mock
}

func (m *MockNomadClient) Get(endpoint string) ([]byte, error) {
    args := m.Called(endpoint)
    return args.Get(0).([]byte), args.Error(1)
}

func (m *MockNomadClient) Post(endpoint string, reqBody *bytes.Buffer) ([]byte, error) {
    args := m.Called(endpoint, reqBody)
    return args.Get(0).([]byte), args.Error(1)
}

func (m *MockNomadClient) Delete(endpoint string) ([]byte, error) {
    args := m.Called(endpoint)
    return args.Get(0).([]byte), args.Error(1)
}

func (m *MockNomadClient) ForwardRequest(c echo.Context, url string) (*http.Response, error) {
    return nil, nil
}
```

We could then create a setup method to create a new Echo app with the mocked nomad controller and client for testing our requests.

```go
func setup(method string, url string) (
    *httptest.ResponseRecorder, echo.Context, *MockNomadClient, NomadController,
) {
    e := echo.New()
    req := httptest.NewRequest(method, url, nil)
    rec := httptest.NewRecorder()
    c := e.NewContext(req, rec)

    nomadClient := new(MockNomadClient)
    nomadController := NomadController{nomadClient}

    return rec, c, nomadClient, nomadController
}
```

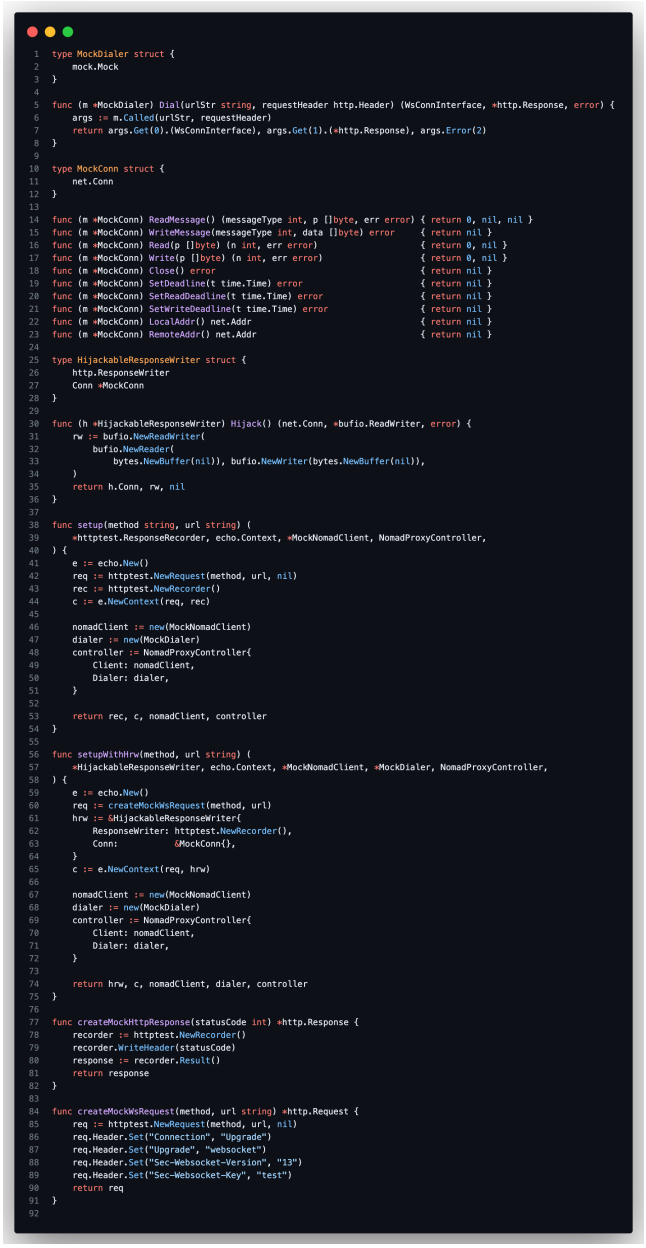
Then for the actual tests it was just a case of specifying the requests to make and mocking them as they would be called on the production environment. Take a look at these two tests for example, testing the "/jobs" endpoint for getting or creating jobs with either a "GET" or "POST"

request.

```go
func TestGetJobs(t *testing.T) {
    // Setup
    rec, c, nomadClient, nomadController := setup(http.MethodGet, "/jobs")
    c.Set("uid", "test")

    // Mocks
    nomadJobs := []nomad.JobListStub{
        {
            ID: "test",
            Meta: map[string]string{
                "user": "test",
            },
        },
        {
            ID: "test2",
            Meta: map[string]string{
                "user": "test2",
            },
        },
    }
    jobsJson, _ := json.Marshal(nomadJobs)
    nomadClient.On("Get", "/jobs?meta=true").Return(jobsJson, nil)

    // Assertions
    expected := []nomad.JobListStub{
        {
            ID: "test",
            Meta: map[string]string{
                "user": "test",
            },
        },
    }
    expectedJson, _ := json.Marshal(expected)
    expectedCode := http.StatusOK
    if assert.NoError(t, nomadController.GetJobs(c)) {
        assert.Equal(t, expectedCode, rec.Code)
        assert.JSONEq(t, string(expectedJson), rec.Body.String())
    }
}

func TestCreateJob(t *testing.T) {
    // Setup
    rec, c, nomadClient, nomadController := setup(http.MethodPost, "/jobs")

    // Mocks
    nomadJob := templates.NomadJob{
        Name:  "test",
        Image: "test",
        User:  "test",
        Shell: "test",
    }
    nomadJobJson, _ := json.Marshal(nomadJob)
    c.Request().Body = io.NopCloser(bytes.NewBuffer(nomadJobJson))

    nomadRegister := nomad.JobRegisterResponse{
        EvalID:          "test",
        EvalCreateIndex: 1,
        JobModifyIndex:  1,
    }
    nomadRegisterJson, _ := json.Marshal(nomadRegister)
    nomadClient.On("Post", "/jobs", mock.Anything).Return(nomadRegisterJson, nil)

    // Assertions
    expectedJson := nomadRegisterJson
    expectedCode := http.StatusOK
    if assert.NoError(t, nomadController.CreateJob(c)) {
        assert.Equal(t, expectedCode, rec.Code)
        assert.JSONEq(t, string(expectedJson), rec.Body.String())
    }
}
```

Then for the proxy controller we were able to do the same mocking for the nomad client but had additional setup and mocking to do, this complex mocking was all related to our use of WebSockets in the proxy controller, and was needed to ensure we were testing these methods

effectively.



```go
type MockDialer struct {
    mock.Mock
}

func (m *MockDialer) Dial(urlStr string, requestHeader http.Header) (WsConnInterface, *http.Response, error) {
    args := m.Called(urlStr, requestHeader)
    return args.Get(0).(WsConnInterface), args.Get(1).(*http.Response), args.Error(2)
}

type MockConn struct {
    net.Conn
}

func (m *MockConn) ReadMessage() (messageType int, p []byte, err error) { return 0, nil, nil }
func (m *MockConn) WriteMessage(messageType int, data []byte) error     { return nil }
func (m *MockConn) Read(p []byte) (n int, err error)                    { return 0, nil }
func (m *MockConn) Write(p []byte) (n int, err error)                   { return 0, nil }
func (m *MockConn) Close() error                                        { return nil }
func (m *MockConn) SetDeadline(t time.Time) error                       { return nil }
func (m *MockConn) SetReadDeadline(t time.Time) error                   { return nil }
func (m *MockConn) SetWriteDeadline(t time.Time) error                  { return nil }
func (m *MockConn) LocalAddr() net.Addr                                 { return nil }
func (m *MockConn) RemoteAddr() net.Addr                                { return nil }

type HijackableResponseWriter struct {
    http.ResponseWriter
    Conn *MockConn
}

func (h *HijackableResponseWriter) Hijack() (net.Conn, *bufio.ReadWriter, error) {
    rw := bufio.NewReadWriter(
        bufio.NewReader(
            bytes.NewBuffer(nil)), bufio.NewWriter(bytes.NewBuffer(nil)),
    )
    return h.Conn, rw, nil
}

func setup(method string, url string) (
    *httptest.ResponseRecorder, echo.Context, *MockNomadClient, NomadProxyController,
) {
    e := echo.New()
    req := httptest.NewRequest(method, url, nil)
    rec := httptest.NewRecorder()
    c := e.NewContext(req, rec)

    nomadClient := new(MockNomadClient)
    dialer := new(MockDialer)
    controller := NomadProxyController{
        Client: nomadClient,
        Dialer: dialer,
    }

    return rec, c, nomadClient, controller
}

func setupWithHrw(method, url string) (
    *HijackableResponseWriter, echo.Context, *MockNomadClient, *MockDialer, NomadProxyController,
) {
    e := echo.New()
    req := createMockWsRequest(method, url)
    hrw := &HijackableResponseWriter{
        ResponseWriter: httptest.NewRecorder(),
        Conn:           &MockConn{},
    }
    c := e.NewContext(req, hrw)

    nomadClient := new(MockNomadClient)
    dialer := new(MockDialer)
    controller := NomadProxyController{
        Client: nomadClient,
        Dialer: dialer,
    }

    return hrw, c, nomadClient, dialer, controller
}

func createMockHttpResponse(statusCode int) *http.Response {
    recorder := httptest.NewRecorder()
    recorder.WriteHeader(statusCode)
    response := recorder.Result()
    return response
}

func createMockWsRequest(method, url string) *http.Request {
    req := httptest.NewRequest(method, url, nil)
    req.Header.Set("Connection", "Upgrade")
    req.Header.Set("Upgrade", "websocket")
    req.Header.Set("Sec-Websocket-Version", "13")
    req.Header.Set("Sec-Websocket-Key", "test")
    return req
}
```

Then to test the proxy endpoints we could use the same strucutre as used before in the previous tests and test the “/exec” and “/logs” endpoints.

```go
func TestAllocExec(t *testing.T) {
    // Setup
    jobName := "test"
    hrw, c, client, dialer, controller := setupWithHrw(http.MethodGet, "/job/"+jobName+"/exec")
    c.SetParamNames("id")
    c.SetParamValues(jobName)
    c.QueryParams().Set("command", "[\"/bin/bash\"]")
    c.Set("uid", "test")

    // Mocks
    nomadJobAlloc := []nomad.AllocListStub{
        {
            ID:           "test",
            ClientStatus: "running",
        },
    }
    allocsJson, _ := json.Marshal(nomadJobAlloc)
    client.On("Get", "/job/"+jobName+"/allocations").Return(allocsJson, nil)

    nomadJobCheckUser := nomad.Job{
        ID: "test",
        Meta: map[string]string{
            "user": "test",
        },
    }
    nomadJobCheckUserJson, _ := json.Marshal(nomadJobCheckUser)
    client.On("Get", "/job/"+jobName).Return(nomadJobCheckUserJson, nil)

    httpResponse := createMockHttpResponse(http.StatusOK)
    dialer.On("Dial", mock.Anything, mock.Anything).Return(hrw.Conn, httpResponse, nil)

    // Assertions
    assert.NoError(t, controller.AllocExec(c))
}

func TestStreamLogs(t *testing.T) {
    // Setup
    jobName := "test"
    rec, c, client, controller := setup(http.MethodGet, "/job/"+jobName+"/logs")
    c.SetParamNames("id")
    c.SetParamValues(jobName)
    c.QueryParams().Set("task", "test")
    c.QueryParams().Set("type", "test")
    c.Set("uid", "test")

    // Mocks
    nomadJobAlloc := []nomad.AllocListStub{
        {
            ID:           "test",
            ClientStatus: "running",
        },
    }
    allocsJson, _ := json.Marshal(nomadJobAlloc)
    client.On("Get", "/job/"+jobName+"/allocations").Return(allocsJson, nil)

    nomadJobCheckUser := nomad.Job{
        ID: "test",
        Meta: map[string]string{
            "user": "test",
        },
    }
    nomadJobCheckUserJson, _ := json.Marshal(nomadJobCheckUser)
    client.On("Get", "/job/"+jobName).Return(nomadJobCheckUserJson, nil)

    httpResponse := createMockHttpResponse(http.StatusOK)
    client.On("ForwardRequest", mock.Anything).Return(httpResponse, nil)

    // Assertions
    expectedCode := http.StatusOK
    if assert.NoError(t, controller.StreamLogs(c)) {
        assert.Equal(t, expectedCode, rec.Code)
    }
}
```

Component Testing

To ensure the reliability and functionality of our frontend SvelteKit web app, we employed component testing. For this purpose, we chose to use Vitest, a powerful testing framework that facilitated efficient and comprehensive component testing.

With Vitest, we were able to systematically test each individual component of our web app, ensuring that they functioned correctly in isolation and in conjunction with other components. This allowed us to identify and address any issues or bugs early on, improving the overall quality and stability of our application.

Through component testing, we were able to simulate user interactions, input various scenarios, and validate the expected behavior of our components. This testing approach provided us with valuable insights into the functionality, responsiveness, and user experience of our web app.

Additionally, Vitest offered a rich set of testing capabilities, such as mocking dependencies and simulating asynchronous operations, enabling us to create robust and reliable tests. It provided us with a comprehensive toolkit for writing assertions, making it easier to verify the correctness of component states, DOM manipulation, and event handling.

By incorporating component testing into our development process, we gained confidence in the stability and correctness of our frontend code. It helped us detect and resolve issues early on, minimizing the chances of encountering unexpected bugs or regressions during the

development lifecycle.

Overall, component testing with Vitest played a vital role in ensuring the quality and reliability of our frontend SvelteKit web app, providing us with a solid foundation for delivering a seamless and user-friendly experience to our users.

For some examples take a look how we tested the IndexPage and ContainerPage components.

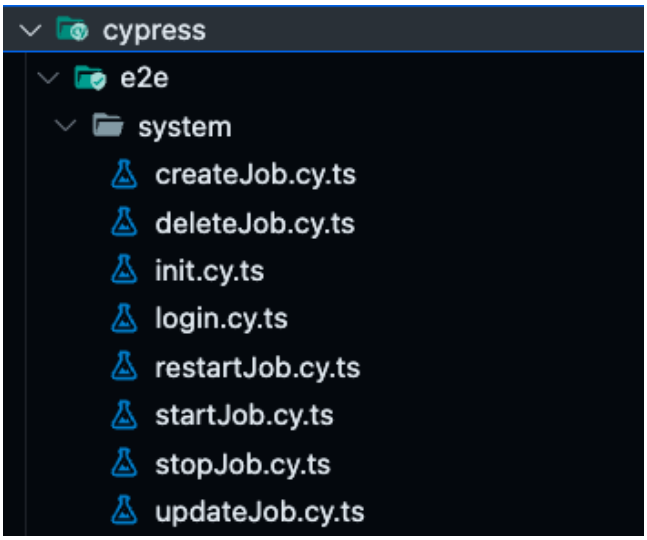
This is the IndexPage component test.

```
import { cleanup, fireEvent, render, waitFor } from '@testing-library/svelte';
import { tick } from 'svelte';
import { describe, expect, it } from 'vitest';

import type { PageData } from '../../routes/$types';
import IndexPage from './IndexPage.svelte';

const defaultData: PageData = {
    uid: 'test',
    token: 'test',
    jobs: null,
    error: null
};

describe('IndexPage', () => {
    afterEach(cleanup);

    it('should render a "Create Container" button', () => {
        const { getByText } = render(IndexPage, { props: { data: defaultData } });
        const button = getByText('Create Container');
        expect(button).toBeInTheDocument();
    });

    it('should navigate to the Create page when the "Create Container" button is clicked', async () => {
        // Render the component
        const { getByText } = render(IndexPage, { props: { data: defaultData } });

        // Click the "Create Container" button
        fireEvent.click(getByText('Create Container'));

        // Wait for the next page to render
        await tick();
    });

    it('should show a message if there are no jobs', async () => {
        const { getByText } = render(IndexPage, { props: { data: defaultData } });

        // Wait for the message to appear
        await waitFor(() => {
            expect(
                getByText("It seems like you haven't created any containers yet...")
            ).toBeInTheDocument();
        });
    });

    it('should show an error modal if the server returns an error', async () => {
        // Mock ssrFetch data
        const data: PageData = {
            uid: 'test',
            token: 'test',
            jobs: [],
            error: 'some failure'
        };

        const { getByText } = render(IndexPage, { props: { data: data } });

        // Wait for the error modal to appear
        await waitFor(() => {
            expect(getByText('Error')).toBeInTheDocument();
        });
    });

    it('should show a list of jobs if there are jobs', async () => {
        // Mock ssrFetch data
        const data: PageData = {
            uid: 'test',
            token: 'test',
            jobs: [
                {
                    ID: 'test',
                    Name: 'test'
                }
            ],
            error: null
        };

        const { getByText } = render(IndexPage, { props: { data: data } });

        // Wait for the jobs to appear
        await waitFor(() => {
            expect(getByText('test')).toBeInTheDocument();
        });
    });
});

```

This is the ContainerPage component test.

```ts
import { render } from '@testing-library/svelte';
import { describe, expect, it } from 'vitest';

import type { Job } from '$lib/types/Types';

import { currJob } from '../../stores/nomadStore';
import ContainerPage from './ContainerPage.svelte';

const mockJob: Job = {
    user: 'test',
    containerName: 'test',
    dockerImage: 'test',
    shell: 'test',
    volumes: [],
    env: [],
    port: 0,
    expose: false,
    cpu: 100,
    memory: 300
};
currJob.set(mockJob);

describe('ContainerPage', () => {
    it('renders without errors', () => {
        const { container } = render(ContainerPage);
        expect(container).toBeDefined();
    });

    it('does not display the link to the container if job.expose is false', async () => {
        const { queryByText } = render(ContainerPage);
        await new Promise((r) => setTimeout(r, 1000)); // wait for fetchAndSetJob to resolve
        expect(queryByText('https://123.plusvasis.xyz')).not.toBeInTheDocument();
    });

    it('displays error modal when fetching job data fails', async () => {
        const { getByText } = render(ContainerPage);
        await new Promise((resolve) => setTimeout(resolve, 1000)); // simulate delay in fetching data
        expect(getByText('Error')).toBeInTheDocument();
    });
});

```

E2E System Testing



To ensure the overall functionality and integration of our application, we conducted end-to-end (E2E) system testing using the Cypress testing framework. Cypress provided a robust and intuitive platform for simulating real user interactions and validating the behavior of our application as a whole.

With Cypress, we were able to write comprehensive test scenarios that covered critical user flows and key functionalities. These tests involved simulating user actions, such as clicking buttons, entering data into forms, and navigating between different pages. Cypress provided a user-friendly interface for visually inspecting the application's state during test execution, making it easier to debug and troubleshoot any issues that arose.

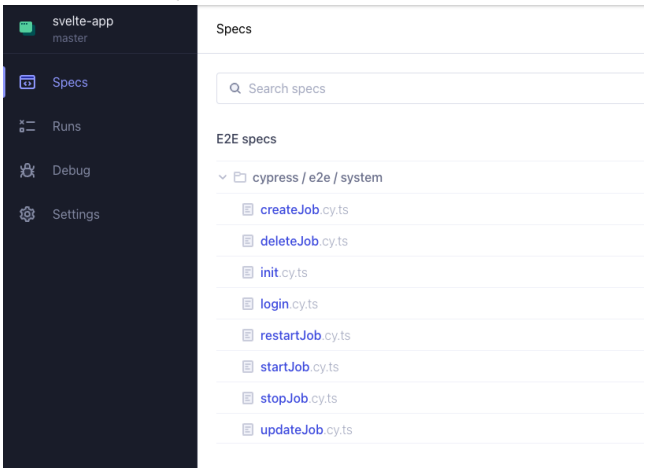
Using Cypress's powerful API, we could interact with the DOM elements, make assertions, and verify expected outcomes. We wrote a series of test scripts that encompassed various user journeys, ensuring that our application performed seamlessly under different scenarios.

By executing these E2E system tests, we were able to identify and address any issues related to data flow, navigation, and overall application behavior. Cypress also provided valuable features like snapshot testing, network stubbing, and time travel debugging, which further enhanced our testing capabilities.

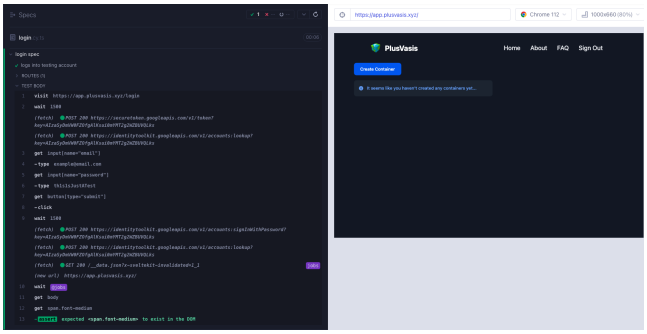
Through E2E system testing, we gained confidence in the reliability and stability of our application across different browsers and devices. It allowed us to validate the end-to-end user experience, ensuring that all components, APIs, and integrations functioned harmoniously.

Overall, E2E system testing with Cypress played a crucial role in validating the integrity and performance of our application. It helped us deliver a robust and user-friendly solution that met the expectations of our users and provided a seamless experience in real-world scenarios.

When running the system tests you'll encounter the nice UI that Cypress offers and be greeted with the test specs.



When you run a spec, Cypress actually allows you to physically view the real time testing on the site. So lets take the login spec for example, this is what it looks like when run.



This is the actual Typescript file then for that login spec.

```typescript
describe('login spec', () => {
    it('logs into testing account', () => {
        cy.intercept('GET', '/__data.json?x-sveltekit-invalidated=1_1').as('jobs');
        cy.visit('https://app.plusvasis.xyz/login');

        cy.wait(1500);

        cy.get('input[name="email"]').type('example@email.com');
        cy.get('input[name="password"]').type('th1s1sJustATest');
        cy.get('button[type="submit"]').click();

        // Wait for a few seconds (e.g., 3 seconds) before proceeding
        cy.wait(1500);

        cy.wait('@jobs');
        cy.get('body').then(($body) => {
            if ($body.find('span.font-medium').length > 0) {
                cy.get('span.font-medium').should('exist');
            } else {
                cy.get('div[data-testid="job-list"]').should('exist');
            }
        });
    });
});
```

We used the same structure and ideology across all the tests and tested the main functionality of the site, such as logging in, creating containers and updating such containers.

Future Work

While the current implementation of our container management and orchestration platform, PlusVasis, provides a solid foundation for developers, there are several areas that can be further improved and expanded upon in future iterations.

One area of focus for future work is enhancing the platform's scalability and performance. As the demand for containerization continues to grow, optimizing PlusVasis to handle larger workloads, scale, and manage resources effectively will be crucial. This can involve fine-tuning the backend infrastructure, and implementing advanced caching and load balancing techniques.

Additionally, incorporating advanced monitoring and logging capabilities into PlusVasis would enable developers to gain deeper insights into their containerized applications. Integrating with tools like Prometheus and Grafana can provide real-time performance metrics, while centralized logging solutions such as ELK (Elasticsearch, Logstash, and Kibana) can offer comprehensive log analysis and troubleshooting capabilities.

Furthermore, expanding PlusVasis's integration capabilities with other popular development tools and services can enhance its usability and versatility. Integrating with popular CI/CD platforms like Jenkins or GitLab CI/CD can streamline the deployment pipeline, while adding optional integration with cloud providers like AWS or Azure can enable seamless deployment and scaling of containers for users who want to take advantage of cloud infrastructure.

Lastly, ongoing improvements in security measures will be essential to ensure the protection of containerized applications. Implementing container image vulnerability scanning, enhancing access control mechanisms, and keeping up with the latest security best practices will help safeguard PlusVasis and the applications it manages.

By addressing these areas of future work, PlusVasis can continue to evolve and meet the evolving needs of developers, providing them with a robust and comprehensive container management and orchestration platform.

References

https://echo.labstack.com/ (https://echo.labstack.com/)
https://developer.hashicorp.com/nomad/docs?product_intent=nomad (https://developer.hashicorp.com/nomad/docs?product_intent=nomad)
https://kit.svelte.dev/docs/introduction (https://kit.svelte.dev/docs/introduction)
https://github.com/vitejs/vite (https://github.com/vitejs/vite)
https://github.com/stretchr/testify (https://github.com/stretchr/testify)
https://github.com/vitest-dev/vitest (https://github.com/vitest-dev/vitest)
https://firebase.google.com/docs (https://firebase.google.com/docs)
https://docs.cypress.io/guides/overview/why-cypress (https://docs.cypress.io/guides/overview/why-cypress)
https://github.com/hashicorp/nomad (https://github.com/hashicorp/nomad)
https://github.com/traefik/traefik (https://github.com/traefik/traefik)
https://github.com/adnanh/webhook (https://github.com/adnanh/webhook)