Technical Guide - Prospector

- **Project Title:** Prospector
- **Student 1:** James Hackett - 20308896
- **Student 2:** Alexandru Dorofte - 20414772
- **Supervisor:** Dr Stephen Blott
- **Date Completed:** 2024-04-21

Introduction

Overview

Prospector is a user management and infrastructure-as-a-service tool that allows users to deploy projects through a simple web interface. Consisting of an Angular-based frontend and a Golang-gin based REST API, Prospector interacts with a personal Nomad server to create and manage containers and virtual machines, making project deployments easy.

The platform features simple networking, security and scalability, allowing users to focus on developing and using their projects, instead of deploying them. Prospector is built using powerful and tested frameworks on modern infrastructure solutions, optimising performance and reliability.

Administrators who wish to deploy Prospector can focus on solving issues and improving performance of their hardware, rather than spend time running and managing containers and virtual machines for their users, giving more room for improved security and reliability.

Motivation

The name Prospector came from the occupation of the same name, "a person who searches for mineral deposits", or extracting the most value out of a patch of land. Our project extracts the most value out of bare metal compute, providing a simple platform for users to deploy and manage their containers and virtual machines.

The idea for this project came from real problems encountered by James, who was a Systems Administrator in DCU's Computing Society, Redbrick. Redbrick members would often request websites, projects, game servers and more on Redbrick, which the administrators were happy to oblige, but this required manual effort. User virtual machines have also been keenly sought after, but no good solution was ever proposed. This project reduces the workload on administrators and provides a new service for the society.

This was motivation enough for us to decide to tackle this task. We chose Nomad as our orchestrator because of it's horizontal scalability and relatively small footprint and extensive documentation for future admins who may need to solve problems not covered by our project.

Glossary

Term	Definition
Allocation	A Nomad term, the smallest unit of work that is placed on a single node in a Nomad datacentre.
Go (Golang)	A programming language developed by Google that is known for its simplicity, efficiency and strong support for concurrent programming.
Gin framework	A HTTP web framework for Go. It features blazing fast performance, routing, middleware and request-handling features.
Nomad	A flexible workload orchestrator that enables an organization to automate the deployment of any application on any infrastructure at any scale
Traefik	A modern HTTP reverse proxy and load balancer that makes deploying microservices easy by dynamically managing routing, load balancing, and SSL/TLS termination.

Research

Ansible Server Setup

In order to make it as simple as possible to deploy our application, we had to make it as simple as possible to get the configuration out to all the nodes that would be clustered together and managed by Prospector. There are many ways to achieve this configuration management, everything from Terraform to Puppet or Chef, but we chose Ansible. Ansible is an automation platform built in Python to automate repetitive tasks that must be performed on lots of servers, like copying configuration and running various shell commands. As ansible is used commonly in the industry in DevOps/SRE type roles, and also being a very versatile tool, it seemed a perfect fit for our needs and well worth the time investment.

Website Frontend and Design

Our application while requiring a visually appealing user interface, had to be highly functional to complement our sophisticated backend powering Prospector.

We needed a framework to put up with these evolving requirements, and there are a vast number of frameworks to choose from. However we agreed that we should use this project time to learn something new and to sharpen our software engineering skills. We chose to go with an ambitious framework for our Project which was Angular. In hindsight we should have chosen some of the other frameworks such as React and SvelteKit both known for not yielding such a steep learning curve. However we were determined in familirising ourselves with an industry standard framework. We saw this as a way to align our project with enterprise-grade software practices.

So why Angular after all?

Google, Paypal, Netflix and Tesla.

What do each of these big name companies have in common… Well each of these companies have or are using Angular in their services today. Its a simple fact that Angular is favoured for enterprise level applications. It is no surprise that by being developed by Google, as a platform it comes feature packed with well integrated libraries that cover routing, forms management and much more.

In order to gain some more value in the design department we chose to pair angular with bootstrap, angular material and chart.js.

Bootstrap was something we have used before and we chose it because it is one of the most popular css frameworks. With this it helped in aligning items the way we wanted and utilising extra graphical elements to spice up our project.

Angular material was something that made sense to pair up with Angular since they were made to work together. Though we chose angular material for its extensive library of components that we knew we would benifet in utilising in the making of each element of our project.

Chart.js keeping in the theme of gaining more value in ui we required to display some data in a way that was pleasing to the eye. This library allowed us to create ready made graphs where we would just need to plug in the data and voila.

REST API & Command Line Interface

Our API needed to communicate with Nomad to dispatch payloads to create and manage jobs. It also needed to authenticate users for the frontend and interact with an LDAP server for user management purposes. Much like the frontend, there were many design and implementation choices to pick from. Among the top choices for a backend were Golang, Node.js and Python.

All of the options would have been perfectly capable of carrying out the role of a REST API server, however we also needed a CLI interface to allow for scripting, and because Golang is a compiled language that is very quick and lightweight, as well as offering extensive libraries for command line interactions, we settled on that. Nomad is also written in Golang, so it would be easy to import the types that Nomad uses into the API and ensure that parsed responses are correct etc. Couple this with the fact that neither of us knew Golang before starting this project, it would be a perfect time to learn a new language.

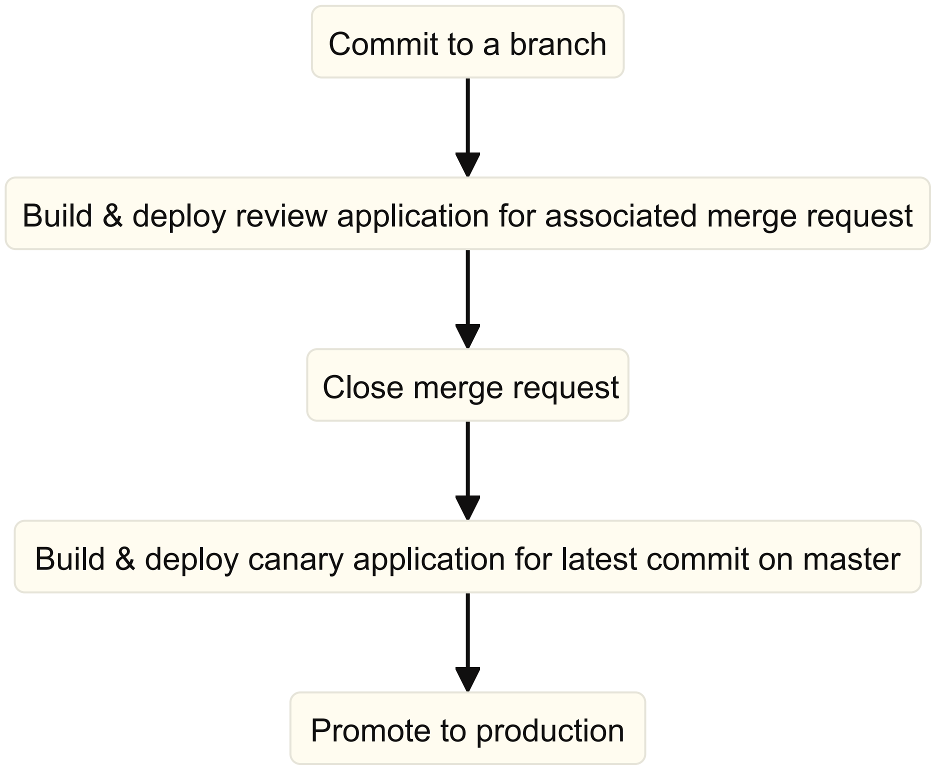
To make full use of Go's speed, we looked into various frameworks for API development and deployment, ultimately settling on Gin for it's simplicity, speed, middleware support and the learning opportunities it provided.

CI/CD

We knew that we would need CI to make sure we followed our own rules of testing every commit, but we weren't sure how to go about deploying that code once it was built. As we are using Gitlab as our primary code storage platform, it seemed sensible to utilise Gitlab's "pipelines". After some research into deployment strategies (for the CD part), we discovered that deploying our own runner would make things a lot easier.

Alongside some constraints like not having a container image registry available on Gitlab, James deployed a runner on his own servers and configured the CI/CD jobs to build, test and push each artefact to a remote container image repository, then deploy review applications for merge requests. This would allow one of us to see the other person's changes live on the website without needing to build the project locally. Deploying a review application on every commit to a branch meant we could also verify that our code did in fact work.

Once a merge request was closed, we could stop the review application and deploy the code to a canary deployment in the production environment. This meant that if there was an unknown difference between testing and production, we could avoid corrupting production traffic. Once we were happy with our canary application, we could promote it to full production and that would complete our CD cycle.


Commit to a branch
Build & deploy review application for associated merge request
Close merge request
Build & deploy canary application for latest commit on master
Promote to production

HTTPS Access & Proxy

Both our frontend and our backend are packaged into a container to allow for portability and ease of deployment. We investigated many ways of routing traffic effectively. Reverse proxying and binding directly to a port and IP address came out as potential options, though because of the potential of many containers running web application (and thus many containers requiring ports 80 and 443), we elected for using a reverse proxy. From here we looked into many cloud-native options (ones that were not statically configured). This mostly ruled out NGINX because it does not have support for dynamic configuration changes.

We settled on Traefik due to it's configurability, first party integration with Nomad, Consul and Docker, and for it's automatic SSL offering. We use Traefik to access both the Prospector frontend, and all of the containers that users want to expose.

Metrics/Observability

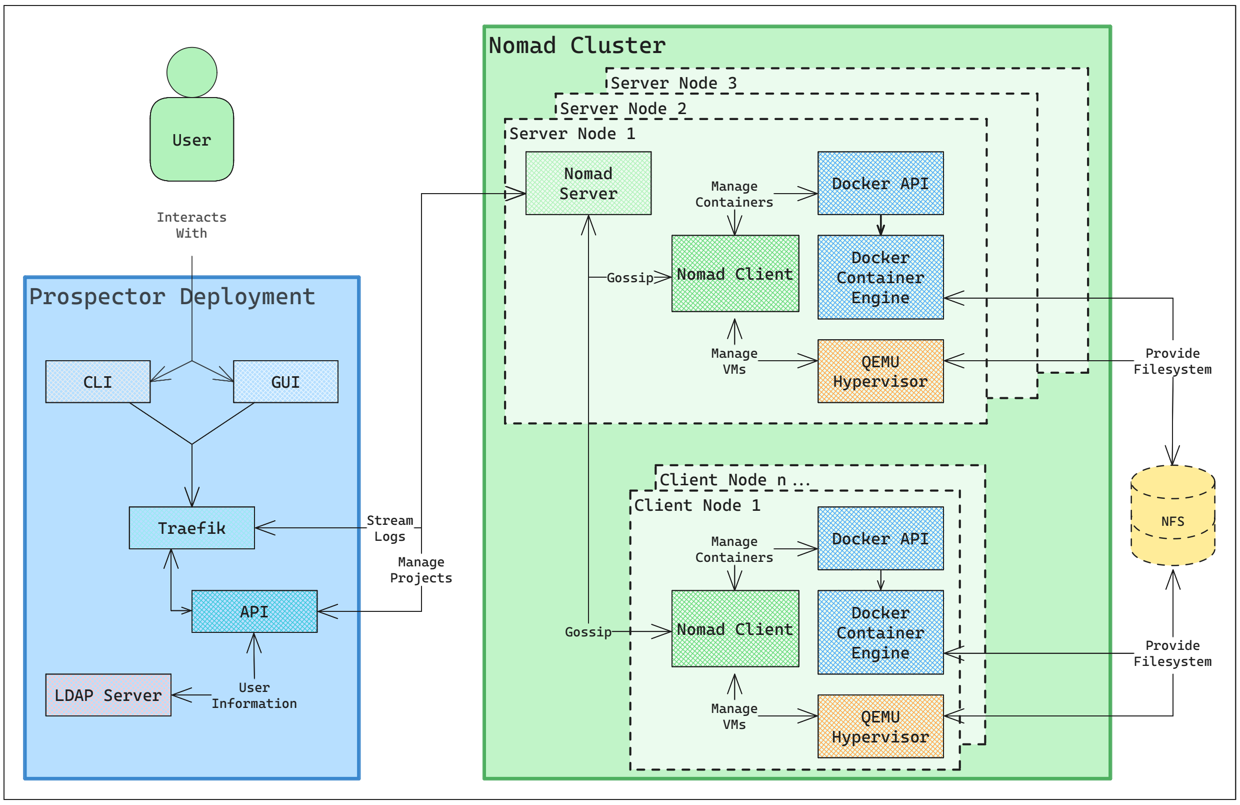
To make the application easier to maintain and deploy, we wanted some metrics on how the project performed, namely the API. In order to collect and display this information, we would need to build a metrics stack. We decided to use Prometheus and Grafana as an industry standard pairing for collecting and displaying collected data from our application. Our API design choices also allowed us to plug middleware into our application with ease, so we added a metrics middleware to provide data for Prometheus to scrape and provide to Grafana to display.

Virtual Machine Hypervisor Management

Extensive research was put in to configuring QEMU on a host. This is largely due to the fact that QEMU is quite an old hypervisor and it's documentation recommends ideas rather than configurations. There were many hours poured into ensuring the networking as we wanted - to appear as if the guest was a physical host on the network, not in a subnet or only routable from the hypervisor host.

Design

System Architecture



Component	Function
Go REST API	Acts as the middleman between Nomad and the CLI/GUI. Health can be checked at https://prospector.ie/api/health.
Angular GUI	Provides the web UI using Angular components, routing and server side rendering. Written in Typescript for type-safe implementation. Accessible at https://prospector.ie/.
Go CLI	Command line interface for scripting and programmable interactions with the application. Download at https://prospector.ie/cli.
Nomad	Deploys and manages containers and virtual machines on a cluster of hardware by communicating with clients, the Docker API and QEMU on each node. Also provides real time logging and monitoring.
Traefik	Reverse proxy for the project. Request routing to the API, GUI, Nomad and any containers that users expose on Prospector. Provides integration with Nomad and Docker to do this.
Docker	Driver for executing Docker type jobs. A docker image specified in a Nomad job will be started by Nomad on a node using Docker.
QEMU	Driver for executing Virtual Machine type jobs. A `cloud-init` enabled operating system is downloaded and started by Nomad on a node using QEMU.
LDAP	Authentication and user storage platform. Used by the API to validate credentials and separate user data from application data.

The above diagram and table illustrates and describes our system architecture. We designed our system with not only our components in mind, but also the eventual accessibility and ease of use for our users. Bearing this in mind, performance was a key metric. Starting at the user, they can interact with our application via the CLI to allow for scripting and programmable interactions, or via the GUI for broader access via various devices.

Both the CLI and the frontend interact with, and are required to be connected to, the API. This is because the API is what performs the heavy lifting when parsing requests both from the user and from Nomad. The GUI is served from an NGINX container and is proxied with Traefik. The API is deployed in a custom Alpine Linux container and is also proxied with Traefik. Requests from the GUI and the CLI are routed to the API and from there the API will perform the necessary actions.

Implementation

Backend

The backend consists of a REST API written in Go using the Gin framework. The backend can be seen as the middleman of the application, allowing communication between our frontend and Nomad by communicating directly with each. Go is a fast and efficient programming language, making it well-suited for building high performance APIs. Additionally, Gin is a powerful and fast web framework that provides an intuitive way to create RESTful APIs.

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

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

Structure

```
.
├── cmd/
├── controllers/
├── docs/
├── helpers/
├── middleware/
├── routes/
└── main.go
```

This is brief overview of the structure of our project. We'll dive into each directory in this section, but for a general overview, we have the entrypoint of the project in `main.go`, commands for the server are implemented in `cmd/`. All of the controllers for the API endpoints are implemented in `controllers/`. Auto generated Swagger documentation is stored in `docs/`. `helpers/` is a collection of helper functions for use across the codebase. `middleware/` implements things like metrics and authentication, and finally all of the routing logic is stored in `routes/`.

I've intentionally left out build related files like the project's `Makefile`, `Dockerfile` and `.gitlab-ci.yml`, they're not really relevant to this section.

Main

```go
package main

import "prospector/cmd"

func main() {
    cmd.Execute()
}
```

Our `main.go` file is quite sparse, because the `prospector` binary (the compiled application) can run both the API server and also the CLI tool. So here we simply call upon the `cmd` module to do it's work.

`prospector` Command

```go
// rootCmd represents the base command when called without any subcommands
var rootCmd = &cobra.Command{
    Use:   "prospector",
    Short: "Infrastructure-as-a-service and user management tool for running containers and
virtual machines",
    Long: `Prospector is a user management and infrastructure-as-a-service tool
enabling easy on-demand deployment of containers and virtual machines.`,
}

func Execute() {
    err := rootCmd.Execute()
    if err != nil {
        os.Exit(1)
    }
}

func init() {
    rootCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

`cmd/root.go` is the base command for `prospector`. It defines some help information and descriptions on the command itself, defines flags, and actually executes the code. `prospector` on its own performs no function, so this command exits with error code 1.

`server` Command

```go
// serverCmd represents the server command
var serverCmd = &cobra.Command{
	Use:   "server",
	Short: "Start the Prospector API server",
	Long:  `Starts the Prospector API server. The server will be bound to the port specified by
the --port flag.`,

	Run: func(cmd *cobra.Command, args []string) {
		r := gin.Default()
		identityKey := cmd.Flag("identity-key").Value.String()

		middleware.CreateStandardMiddlewares(r)
		middleware.CreateAuthMiddlewares(r, identityKey)
		routes.Route(r, identityKey)

		r.Run(":" + cmd.Flag("port").Value.String())
	},
}

func init() {
	rootCmd.AddCommand(serverCmd)
	serverCmd.Flags().StringP("port", "p", "3434", "Port to bind the server to")
	serverCmd.Flags().StringP("identity-key", "i", "id", "Identity key for the JWT middleware")
}
```

The `prospector server` command is what creates a running HTTP server that listens for requests. The `Run` definition creates a Gin engine with a default logger and recovery middleware already attached. This saves some lines of code for setup. We then get the identity key used to sign our JWT tokens passed in via the CLI argument `identity-key`, attach our middlewares, define our routes and finally run the Gin engine on the port specified on the CLI.

Every subcommand of `prospector` follows this file layout, making it easy to add more commands if needed.

Below is a brief overview of the commands implemented

```
$ ./bin/prospector
Prospector is a user management and infrastructure-as-a-service tool
enabling easy on-demand deployment of containers and virtual machines.

Usage:
  prospector [command]

Available Commands:
  auth        Authenticate with the server
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  project     A subcommand for managing projects in the Prospector system
  server      Start the Prospector API server

Flags:
  -a, --address string   The address of the Prospector server (default "https://prospector.ie")
  -h, --help             help for prospector
  -t, --toggle           Help message for toggle

Use "prospector [command] --help" for more information about a command.

$ ./bin/prospector project
Error: requires at least 1 arg(s), only received 0
Usage:
  prospector project [flags]
  prospector project [command]

Available Commands:
  create      A subcommand for creating a job in the Prospector system
  list        List all running jobs on the server
  restart     Restart a job
  start       Start a job
  status      Get the status of a job
  stop        Stop a job
  template    A subcommand for creating a project template

Flags:
  -h, --help   help for project

Global Flags:
  -a, --address string   The address of the Prospector server (default "https://prospector.ie")

Use "prospector project [command] --help" for more information about a command.
```

Routes

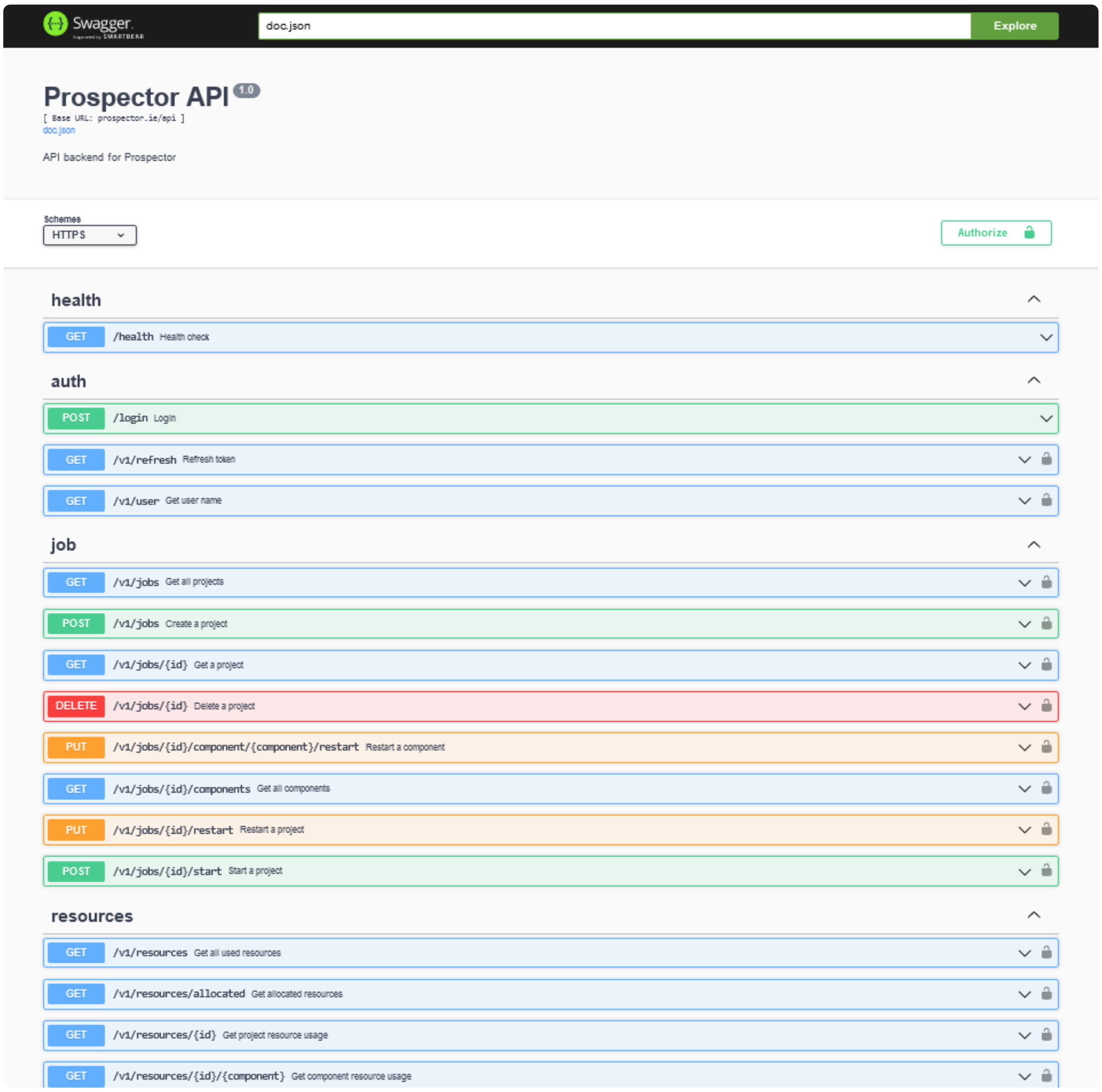
```go
func Route(r *gin.Engine, identityKey string) {
    api := r.Group("/api")
    {
        api.GET("/health", c.Health)
        api.GET("/docs/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
        api.POST("/login", c.Login)

        // serve static files from ./vm-config
        api.GET("/vm-config/*filepath", func(c *gin.Context) {
            filepath := c.Param("filepath")
            c.File("vm-config/" + filepath)
        })
    }

    authenticated := r.Group("/api/v1")
    authenticated.GET("/refresh", c.RefreshToken)
    authenticated.Use(c.JWTMiddleware.MiddlewareFunc())
    {
        authenticated.GET("/user", c.GetUserName)

        jobs := authenticated.Group("/jobs")
        jobs.GET("", c.GetJobs)
        jobs.POST("", c.CreateJob)
        jobs.GET("/:id", c.GetJob)
        jobs.DELETE("/:id", c.DeleteJob)
        jobs.GET("/:id/definition", c.GetJobDefinition)
        jobs.PUT("/:id/definition", c.UpdateJobDefinition)
        jobs.GET("/:id/components", c.GetComponents)
        jobs.GET("/:id/logs", cProxy.StreamLogs)
        jobs.PUT("/:id/restart", c.RestartJob)
        jobs.POST("/:id/start", c.StartJob)
        jobs.PUT("/:id/component/:component/restart", c.RestartAlloc)

        resources := authenticated.Group("/resources")
        resources.GET("", c.GetUserUsedResources)
        resources.GET("/allocated", c.GetUserAllocatedResources)
        resources.GET("/:id", c.GetJobUsedResources)
        resources.GET("/:id/allocated", c.GetJobAllocatedResources)
        resources.GET("/:id/:component", c.GetComponentUsedResources)
        resources.GET("/:id/:component/allocated", c.GetComponentAllocatedResources)
    }
}
```

The `Route` function is where we define what functions shall handle what endpoints. These are broadly divided into unauthenticated requests (logins, health checks, virtual machine configuration files) and authenticated requests, broken further into job, user and resource type requests.

All of these routes are defined further with examples via the Swagger documentation available at the Prospector API docs. This was invaluable in helping to develop the frontend and test the backend during development.

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 geta response that is defined in our schema and is therefore an expected response.



Swagger

doc.json Explore

Prospector API 1.0

[Base URL: prospector.ie/api]

doc.json

API backend for Prospector

Schemes HTTPS

Authorize

health

GET /health Health check

auth

POST /login Login

GET /v1/refresh Refresh token

GET /v1/user Get user name

job

GET /v1/jobs Get all projects

POST /v1/jobs Create a project

GET /v1/jobs/{id} Get a project

DELETE /v1/jobs/{id} Delete a project

PUT /v1/jobs/{id}/component/{component}/restart Restart a component

GET /v1/jobs/{id}/components Get all components

PUT /v1/jobs/{id}/restart Restart a project

POST /v1/jobs/{id}/start Start a project

resources

GET /v1/resources Get all used resources

GET /v1/resources/allocated Get allocated resources

GET /v1/resources/{id} Get project resource usage

GET /v1/resources/{id}/{component} Get component resource usage

Middleware

```go
func CreateStandardMiddlewares(r *gin.Engine) {
    // Add the metrics middleware
    MetricsMiddleware(r)
}

func CreateAuthMiddlewares(r *gin.Engine, identityKey string) {
    // Setup the authentication middleware
    errInit := AuthMiddleware(identityKey).MiddlewareInit()

    if errInit != nil {
        panic(errInit)
    }
}
```

We attach middleware to our Gin instance in the `server` command, passing the Gin engine into the rest of the application from there. In this file we define helper functions to attach middleware to the Engine.

Metrics

```go
func MetricsMiddleware(r *gin.Engine) {
    m := ginmetrics.GetMonitor()
    m.SetMetricPath("/metrics")
    // set the threshold for a slow request to 1 second
    m.SetSlowTime(1)

    m.Use(r)
}
```

The metrics middleware is used to provide metrics to Prometheus to be displayed in Grafana later. We used a tool called `ginmetrics` to achieve this easily

Authentication

```go
func AuthMiddleware(identityKey string) *jwt.GinJWTMiddleware {
    jwtMiddleware, err := jwt.New(&jwt.GinJWTMiddleware{
        Realm:       "prospector",
        Key:         []byte("secret key"),
        Timeout:     time.Hour * 1,
        MaxRefresh:  time.Hour * 24,
        IdentityKey: identityKey,
        IdentityHandler: func(c *gin.Context) interface{} {
            claims := jwt.ExtractClaims(c)
            return &User{
                Username: claims[identityKey].(string),
            }
        },
        Authenticator: Authenticate,
        Unauthorized: func(c *gin.Context, code int, message string) {
            c.JSON(code, gin.H{
                "code":    code,
                "message": message,
            })
        },
        TimeFunc: time.Now,
    })

    if err != nil {
        fmt.Println("error creating jwt middleware")
    }

    return jwtMiddleware
}
```

Our authentication middleware is a little more complex. Using `gin-jwt` we define the parameters of our JWT authentication. This module was essential to making development smoother, though we still had to implement our own authentication method for LDAP, as you can see below.

```go
func AuthenticateLdap(username string, password string, ldapBindUser string, ldapBindPassword
string) (interface{}, error) {
    l, err := connectLdap("ldap://ldap.forumsys.com:389")
    if err != nil {
        return nil, jwt.ErrFailedAuthentication
    }

    defer l.Close()

    err = bindLdap(l, "cn="+ldapBindUser+",dc=example,dc=com", ldapBindPassword)
    if err != nil {
        return nil, jwt.ErrFailedAuthentication
    }

    searchRequest := ldap.NewSearchRequest(ldapSearchInformation)

    sr, err := l.Search(searchRequest)
    if err != nil {
        fmt.Println("error searching ldap server:", err)
        return nil, jwt.ErrFailedAuthentication
    }

    if len(sr.Entries) != 1 {
        return nil, jwt.ErrFailedAuthentication
    }

    userdn := sr.Entries[0].DN
    err = l.Bind(userdn, password)
    if err != nil {
        return nil, jwt.ErrFailedAuthentication
    } else {
        return &User{
            Username: username,
        }, nil
    }
}
```

The above code connects to an LDAP server, binds as a user, then searches the tree for that user, filling in the `User` struct (which is passed onto each request if valid) with the information it gets. This function only fires when a user calls the `/api/login` endpoint, after that a user is authenticated by decoding the JWT, a much faster operation.

Handlers/Controllers

NomadClient

```go
type NomadClient interface {
    Get(endpoint string) ([]byte, error)
    Post(endpoint string, reqBody *bytes.Buffer) ([]byte, error)
    Delete(endpoint string) ([]byte, error)
    Forward(ctx *gin.Context, endpoint 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 as testing of the API should not have a requirement of a running Nomad instance. This is especially useful for validating each request before it reaches Nomad.

Controller

```go
type Controller struct {
    Client        NomadClient
    IdentityKey   string
    JWTMiddleware *jwt.GinJWTMiddleware
}
```

The main controller of the application, implements a `NomadClient`, contains the `IdentityKey` and contains the JWT middleware for authentication. The controller contains the methods and for each API route handler.

GetJobs

```go
// GetJobs gets all the jobs from nomad that have the word "prospector" in their name
//
//  @Summary        Get all projects
//  @Description    Get all projects from nomad
//  @Tags           job
//  @Accept         json
//  @Produce        json
//  @Security       BearerAuth
//  @Router         /v1/jobs [get]
//  @Param          long    query   boolean false   "Get long project details"
//  @Param          running query   boolean false   "Get running projects"
//  @Code           204 "No projects found"
//  @Success        200 {object}    []ShortJob
func (c *Controller) GetJobs(ctx *gin.Context) {
    claims := jwt.ExtractClaims(ctx)
    ctx.Set(c.IdentityKey, claims[c.IdentityKey])

    data, err := c.Client.Get("/jobs?meta=true")
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

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

    var filteredJobs []nomad.JobListStub = []nomad.JobListStub{}
    for _, job := range jobs {
        if strings.Contains(job.Name, "-prospector") && strings.Contains(job.Name,
claims[c.IdentityKey].(string)) {
            filteredJobs = append(filteredJobs, job)
        }
    }

    var jobSummaries []ShortJob
    for _, job := range filteredJobs {
        jobSummaries = append(jobSummaries, ShortJob{
            ID:      job.ID,
            Status:  job.Status,
            Type:    job.Meta["job-type"],
            Created: job.SubmitTime,
        })
    }

    var runningJobs []ShortJob = []ShortJob{}
    for _, job := range filteredJobs {
        if job.Status == "running" {
            runningJobs = append(runningJobs, ShortJob{
                ID:      job.ID,
                Status:  job.Status,
                Type:    job.Meta["job-type"],
                Created: job.SubmitTime,
            })
        }
    }

    if len(filteredJobs) == 0 {
        ctx.JSON(http.StatusNoContent, gin.H{"message": "No jobs found"})
        return
    }

    switch {
    case ctx.Query("long") == "true":
        ctx.JSON(http.StatusOK, filteredJobs)
        return
    case ctx.Query("running") == "true":
        ctx.JSON(http.StatusOK, runningJobs)
        return
    default:
        ctx.JSON(http.StatusOK, jobSummaries)
        return
    }
}
```

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

Annotated on the method is our Swagger specification for this API route. We specify the type of request (get, post, delete, put, etc.), and all the possible response codes alongside the types for the structs that then get marshalled 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 `ShortJob` - a custom struct made for easy parsing.

Note: some of the data types used in this function come directly 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 marshalling/unmarshalling of all our requests to and from Nomad.

GetJob

```go
// GetJob gets a job from nomad
//
//  @Summary        Get a project
//  @Description    Get a job from nomad
//  @Tags           job
//  @Accept         json
//  @Produce        json
//  @Security       BearerAuth
//  @Router         /v1/jobs/{id} [get]
//  @Param          id  path    string  true    "Project ID"
func (c *Controller) GetJob(ctx *gin.Context) {
    id := ctx.Param("id")

    data, err := c.Client.Get("/job/" + id)
    if err ≠ nil {
        ctx.Error(err)
    }

    var job nomad.Job
    err = json.Unmarshal(data, &job)
    if err ≠ nil {
        ctx.Error(err)
    }

    ctx.JSON(http.StatusOK, job)
}
```

`GetJob` gets a single job from Nomad and sends Nomad's entire response to the user.

CreateJob

```go
// CreateJob creates a container or VM
//
//  @Summary        Create a project
//  @Description    Create and submit a job for nomad to deploy
//  @Tags           job
//  @Accept         json
//  @Produce        json
//  @Security       BearerAuth
//  @Param          job body        Project true    "Project"
//  @Success        200 {object}    Message
//  @Router         /v1/jobs [post]
func (c *Controller) CreateJob(ctx *gin.Context) {
    var job Project
    if err := ctx.BindJSON(&job); err != nil {
        ctx.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }

    job.User = jwt.ExtractClaims(ctx)[c.IdentityKey].(string)

    for i := 0; i < len(job.Components); i++ {
        job.Components[i].UserConfig.User = job.User
        // generate random mac address
        if job.Type == "vm" {
            job.Components[i].Network.Mac = fmt.Sprintf("52:54:00:%02x:%02x:%02x",
byte(rand.Intn(255)), byte(rand.Intn(255)), byte(rand.Intn(255)))
        }
    }

    switch {
    case job.Type == "docker":
        _, err := CreateJobFromTemplate(job, DockerSourceJson)
        if err != nil {
            ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }
    case job.Type == "vm":
        err := WriteTextFilesForVM(job)
        if err != nil {
            ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }

        _, err = CreateJobFromTemplate(job, VMSourceJson)
        if err != nil {
            ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }
    default:
        ctx.JSON(http.StatusBadRequest, gin.H{"error": "Invalid job type"})
        return
    }

    ctx.JSON(http.StatusOK, gin.H{"status": "ok", "message": "Job submitted successfully"})
}
```

The `CreateJob` handler takes JSON as an input, unmarshals it to a custom `Project` object, generates a Job JSON object that Nomad understands and sends the generated JSON object to Nomad via the `CreateJobFromTemplate` function. We return the response from Nomad to the user.

Notable in the Swagger specification is the definition of the Project parameter in the POST body.

DeleteJob

```go
// DeleteJob deletes a project
//
//  @Summary        Delete a project
//  @Description    Delete a job from nomad
//  @Tags           job
//  @Accept         json
//  @Produce        json
//  @Security       BearerAuth
//  @Success        200 {object}    Message
//  @Router         /v1/jobs/{id} [delete]
//  @Param          id      path    string  true    "Project ID"
//  @Param          purge   query   bool    false   "Purge project"
func (c *Controller) DeleteJob(ctx *gin.Context) {
    id := ctx.Param("id")
    purge := ctx.Query("purge")

    if !helpers.CheckJobHasValidName(id) {
        ctx.JSON(http.StatusForbidden, gin.H{"error": "Invalid job ID"})
        return
    }

    url := fmt.Sprintf("/job/%s", id)
    if purge == "true" {
        url = fmt.Sprintf("%s?purge=true", url)
    }

    data, err := c.Client.Delete(url)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    var message Message
    err = json.Unmarshal(data, &message)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    ctx.JSON(http.StatusOK, message)
}
```

The `DeleteJob` handler takes the same input as the `GetJob` handler from earlier, but also validates that the requested job has a valid name. If this check passes, then the job is either stopped or deleted from Nomad based on the query parameter passed via `?purge={boolean}` in the request, then we return a status 200 with the response from Nomad.

We construct the URL in two stages here in order to keep our request to Nomad clean and not add unnecessary parameters to our call. By default, Nomad will only purge a job if `purge=true` is present in the request URL, so we only append that when we are going to delete a job, instead of just stopping it.

Notable in the Swagger specification here is the project ID string as a path parameter - `/v1/jobs/:id`.

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

RestartJob

```go
func (c *Controller) RestartJob(ctx *gin.Context) {
    id := ctx.Param("id")

    if !helpers.CheckJobHasValidName(id) {
        ctx.JSON(http.StatusForbidden, gin.H{"error": "Invalid job ID"})
        return
    }

    alloc, err := c.parseRunningAllocs(id)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    for _, alloc := range alloc {
        body := bytes.NewBuffer([]byte{})

        data, err := c.Client.Post("/client/allocation/"+alloc.ID+"/restart", body)
        if err != nil {
            ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }

        var response nomad.GenericResponse
        err = json.Unmarshal(data, &response)
        if err != nil {
            ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
            return
        }
    }

    ctx.JSON(http.StatusOK, gin.H{"message": "Project restarted successfully"})
}
```

`RestartJob` takes a project ID as a path parameter and restarts all of the allocations (an allocation is an instance of a project). We sent a post request with an empty body as Nomad parses this as restarting all of "tasks" inside the allocation - without going into too much detail, this performs exactly what we, and the user, want and expect it to do. We return a simple JSON object that the project was restarted successfully.

StartJob

```go
func (c *Controller) StartJob(ctx *gin.Context) {
	id := ctx.Param("id")

	if !helpers.CheckJobHasValidName(id) {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": "Invalid job ID"})
		return
	}

	// read job from nomad
	var job nomad.Job
	data, err := c.Client.Get("/job/" + id)
	if err != nil {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
		return
	}

	err = json.Unmarshal(data, &job)
	if err != nil {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
		return
	}

	if job.Status == "running" {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": "Job is already running"})
		return
	}

	job.Stop = false
	var jobRequest nomad.JobRegisterRequest
	jobRequest.Job = &job

	body, err := json.Marshal(jobRequest)
	if err != nil {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
		return
	}

	// run job against nomad
	data, err = c.Client.Post("/job/"+id, bytes.NewBuffer(body))
	if err != nil {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
		return
	}

	var response nomad.JobRegisterResponse
	err = json.Unmarshal(data, &response)
	if err != nil {
		ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
		return
	}

	ctx.JSON(http.StatusOK, response)
}
```

`StartJob` handles starting a job once it has been stopped by a user. Nomad doesn't actually provide a "start job" endpoint, but it does provide a "read job" endpoint which returns the JSON definition of the job. We can take that JSON definition, marshal it into a `nomad.Job` struct and modify it as we need. In this case, we set `Job.stop` to false. This is actually what the Nomad UI does behind the scenes to start a job after it has been stopped - some reverse engineering was required for that one!

Once we have submitted the updated job, we return the result to the client.

GetJobDefinition

```go
func (c *Controller) GetJobDefinition(ctx *gin.Context) {
    id := ctx.Param("id")

    if !helpers.CheckJobHasValidName(id) {
        ctx.JSON(http.StatusForbidden, gin.H{"error": "Invalid job ID"})
        return
    }

    data, err := c.Client.Get("/job/" + id)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    var job nomad.Job
    err = json.Unmarshal(data, &job)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    project, err := ConvertJobToProject(job)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    ctx.JSON(http.StatusOK, project)
}
```

`GetJobDefinition` fetches a Nomad job and marshals it into our custom `Project` struct. This allows us to make modifications to an existing job (like toggling whether the container is exposed or not). We can then take the updated `Project` object and update the definition in Nomad.

parseRunningAllocs

```go
func (c *Controller) parseRunningAllocs(jobId string) ([]nomad.AllocListStub, error) {
    data, err := c.Client.Get("/job/" + jobId + "/allocations")
    if err != nil {
        return nil, err
    }

    var allocations []nomad.AllocListStub
    err = json.Unmarshal(data, &allocations)
    if err != nil {
        return nil, err
    }

    var runningAllocs []nomad.AllocListStub
    for _, alloc := range allocations {
        if alloc.ClientStatus == "running" || alloc.ClientStatus == "pending" {
            runningAllocs = append(runningAllocs, alloc)
        }
    }

    if len(runningAllocs) > 0 {
        return runningAllocs, nil
    }

    return nil, gin.Error{Err: fmt.Errorf("no running allocations found"), Meta: 404}
}
```

`parseRunningAllocs` takes a project ID as input and requests the allocations for the related job from Nomad. We then check if any of the allocations are running or pending (still in the process of starting) and return that allocation or a code 404 not found error if one does not exist.

StreamLogs

```go
func (c *NomadProxyController) StreamLogs(ctx *gin.Context) {
    jobId := ctx.Param("id")
    logType := ctx.Query("type")
    if logType != "stdout" && logType != "stderr" {
        ctx.JSON(http.StatusBadRequest, gin.H{"error": "Invalid log type"})
        return
    }

    if !helpers.CheckJobHasValidName(jobId) {
        ctx.JSON(http.StatusForbidden, gin.H{"error": "Invalid job ID"})
        return
    }

    allocs, err := c.parseRunningAlloc(jobId)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    if allocs == nil {
        ctx.JSON(http.StatusNotFound, gin.H{"error": "No running allocations found"})
        return
    }

    path := "/client/fs/logs/" + allocs.ID
    queryParams := url.Values{}
    queryParams.Add("task", ctx.Query("task"))
    queryParams.Add("type", logType)
    queryParams.Add("follow", "true")
    queryParams.Add("offset", "5000")
    queryParams.Add("plain", "true")
    queryParams.Add("origin", "end")

    url := path + "?" + queryParams.Encode()
    resp, err := c.Client.Forward(ctx, url)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    defer resp.Body.Close()

    fmt.Println("Started streaming logs")

    err = streamResponse(ctx, resp)
    if err != nil {
        ctx.JSON(http.StatusInternalServerError, gin.H{"error": err.Error()})
        return
    }

    fmt.Println("Finished streaming logs")
}
```

This is one of the more complex functions of the whole API. It is used to stream the logs from Nomad to the frontend. For this, it upgrades a normal HTTP connection to a websocket and writes to it every time there is an update. It uses the `streamResponse` function to do this, which is below.

streamResponse

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

    buf := make([]byte, 1024)
    for {
        bytesRead, err := resp.Body.Read(buf)
        if err != nil {
            if err == io.EOF {
                break
            }
            return err
        }
        select {
        case <-ctx.Request.Context().Done():
            return nil
        default:
        }

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

        if f, ok := ctx.Writer.(http.Flusher); ok {
            f.Flush()
        }
    }

    return nil
}
```

`streamResponse` 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

```go
type Project struct {
    Name       string      `json:"name" validate:"required"`
    Type       string      `json:"type" validate:"required"`
    Components []Component `json:"components" validate:"required"`
    User       string      `json:"-"`
}

type Component struct {
    Name       string   `json:"name" validate:"required"`
    Image      string   `json:"image"`
    Volumes    []string `json:"volumes"`
    Resources  `json:"resources"`
    Network    `json:"network"`
    UserConfig `json:"user_config"`
}

type Network struct {
    Port   int    `json:"port" validate:"min=0,max=65535"`
    Expose bool   `json:"expose" validate:"optional" default:"false"`
    Mac    string `json:"-"`
}
type Resources struct {
    Cpu    int `json:"cpu" validate:"min=0,max=2000"`
    Memory int `json:"memory" validate:"min=0,max=2000"`
}

type UserConfig struct {
    User   string `json:"user" validate:"optional"`
    SSHKey string `json:"ssh_key" validate:"optional"`
}
```

Above is the full definition of a Prospector `Project`. When we marshal JSON into it, we can ensure that each field is correctly populated using validator for both type and content.

```go
func CreateJobFromTemplate(project Project, jobSource string) (int, error) {
    t, err := template.New("").Funcs(template.FuncMap{
        "last":         last,
        "json":         project.ToJson,
        "escapeQuotes": escapeQuotes,
    }).Parse(jobSource)
    if err != nil {
        return 500, err
    }

    body := &bytes.Buffer{}
    err = t.Execute(body, project)
    if err != nil {
        return 500, err
    }

    println(body.String())

    // run job against nomad
    data, err := http.Post("http://zeus.internal:4646/v1/jobs", "application/json", body)
    if err != nil {
        return 500, err
    }

    body = &bytes.Buffer{}
    _, err = io.Copy(body, data.Body)
    if err != nil {
        return 500, err
    }

    var resp nomad.JobRegisterResponse
    err = json.Unmarshal(body.Bytes(), &resp)
    if err != nil {
        return 500, err
    }

    return http.StatusOK, nil
}
```

The `CreateJobFromTemplate` function takes our simple `Project` struct and constructs a fully fledged Nomad job. It "canonicalises" a sparse JSON job by filling in default values.

Template Sources

DockerSource

```
var DockerSourceJson = `{
    "Job": {
        "ID": "{{ .User }}-{{ .Name }}-prospector",
        "Name": "{{ .User }}-{{ .Name }}-prospector",
        "Type": "service",
        "Datacenters": [
            "dc1"
        ],
        "TaskGroups": [
            {{ range $i, $_ := .Components }}{
                {{ $component := . }}
                "Name": "{{ .Name }}",
                "Count": 1,
                "Tasks": [
                    {
                        "Name": "{{ .Name }}",
                        "Driver": "docker",
                        "Config": {
                            "image": "{{ .Image }}",
                            "ports": [
                                "{{ .Name }}"
                            ],
                            "volumes": [
                                "/data/prospector/{{ .UserConfig.User }}:/mnt/user-storage"{{ if
.Volumes }},{{ end }}
                                {{ if .Volumes }}{{ range $i, $v := .Volumes }}
                                "/data/prospector/{{ $component.UserConfig.User }}/{{
$component.Name }}/{{ $v }}:/mnt/component-storage/{{ $v }}"{{ if not (last $i
$component.Volumes) }},{{ end }}
                                {{ end }}{{ end }}
                            ]
                        },
                        "Services": [
                            {
                                "Name": "{{ .Name }}",
                                {{ if .Network.Expose }}"Tags": [
                                    "traefik.enable=true",
                                    "traefik.http.routers.{{ .UserConfig.User }}-{{ $.Name }}-{{
.Name }}-prospector.rule=Host(` + "`" + `{{ .UserConfig.User }}.{{ $.Name }}-{{ .Name }}-
prospector.ie` + "`" + `)",
                                    "traefik.http.routers.{{ .UserConfig.User }}-{{ $.Name }}-{{
.Name }}-prospector.entrypoints=websecure",
                                    "traefik.http.routers.{{ .UserConfig.User }}-{{ $.Name }}-{{
.Name }}-prospector.tls=true",
                                    "traefik.http.routers.{{ .UserConfig.User }}-{{ $.Name }}-{{
.Name }}-prospector.tls.certresolver=lets-encrypt",
                                    "prometheus.io/scrape=false"
                                ],{{ else }}"Tags": [
                                    "prometheus.io/scrape=false"
                                ],{{ end }}
                                "PortLabel": "{{ .Name }}"
                            }
                        ],
                        "Resources": {
                            "CPU": {{ .Resources.Cpu }},
                            "MemoryMB": {{ .Resources.Memory }}
                        }
                    }
                ],
                "Networks": [
                    {
                        "DynamicPorts": [
                            {
                                "Label": "{{ .Name }}",
                                "Value": 0,
                                "To": {{ .Network.Port }}
                            }
                        ]
                    }
                ]
            }{{ if not (last $i $.Components) }},{{ end }}{{ end }}
        ],
        "Meta": {
            "job-type": "docker",
            "job-definition": "{{ json | escapeQuotes }}"
        }
    }
}`
```

Above is our container definition that we pass to Nomad to parse and parse in the rest of the fields. We do this so that we can avoid setting values for parameters we do not care about, ensuring that our project can stay more up to date with whatever Nomad changes in the future.

Of note here is the use of `Range` for looping over our components. We elected to simplify the data structure of a Nomad job for our users by removing a layer of abstraction. Nomad uses 3 layers to define different parts of a job, we removed the middle one - `group`. The implications of this were weighed heavily when implementing this and will be discussed more later in this document.

VMSource

```
var VMSourceJson = `{
    "Job": {
        "ID": "{{ .User }}-{{ .Name }}-prospector",
        "Name": "{{ .User }}-{{ .Name }}-prospector",
        "Type": "service",
        "Datacenters": [
            "dc1"
        ],
        "TaskGroups": [
            {{ range $i, $_ := .Components }}{
                "Name": "{{ .Name }}",
                "Count": 1,
                "Tasks": [
                    {
                        "Name": "{{ .Name }}",
                        "Driver": "qemu",
                        "Config": {
                            "accelerator": "kvm",
                            "args": [
                                "-netdev",
                                "bridge,id=hn0",
                                "-device",
                                "virtio-net-pci,netdev=hn0,id=nic1,mac={{ .Mac }}",
                                "-smbios",
                                "type=1,serial=ds=nocloud-net;s=https://prospector.ie/api/vm-
config/{{ .Name }}-vm/"
                            ],
                            "drive_interface": "virtio",
                            "image_path": "local/{{ .Name }}-vm.qcow2"
                        },
                        "Constraints": [
                            {
                                "LTarget": "${attr.unique.hostname}",
                                "RTarget": "hermes",
                                "Operand": "="
                            }
                        ],
                        "Resources": {
                            "CPU": {{ .Resources.Cpu }},
                            "MemoryMB": {{ .Resources.Memory }}
                        },
                        "Artifacts": [
                            {
                                "GetterSource":
"https://cloud.debian.org/images/cloud/bookworm/latest/debian-12-genericcloud-amd64.qcow2",
                                "GetterMode": "file",
                                "RelativeDest": "local/{{ .Name }}-vm.qcow2"
                            }
                        ]
                    }
                ],
                "Services": [
                    {
                        "Name": "{{ .Name }}-vm"
                    }
                ]
            }{{ if not (last $i $.Components) }},{{ end }}{{ end }}
        ],
        "Meta": {
            "job-type": "vm"
        }
    }
}`
```

This is our definition for a virtual machine. It also uses `Range` to loop over components and create multiple in the same project. Of note here are the QEMU arguments that drive our VMs on a hypervisor. QEMU, being an old but tried and tested hypervisor, is not very well documented for use with modern day operating systems. There was considerable effort put into ensuring a virtual machine could connect to a network.

Frontend

The frontend is built with Angular, a development platform built on Typescript.

As mentioned before this platform allowed us to work with a component based framework, that is widely used in the industry to build scalable web apps. What made angular an attractive option was the included collection of libraries that covered useful features such as routing, forms management and client server communication, right out of the box.

We also paired up our angular frontend with Bootstrap, Angular Material and chart.js, which all aided in the design of our GUI.

Structure of the frontend.

```
.
├── src
│   ├── app
│   │   ├── dialog-content
│   │   │   ├── dialog-content.component.css
│   │   │   ├── dialog-content.component.html
│   │   │   └── dialog-content.component.ts
│   │   ├── footer
│   │   │   ├── footer.component.css
│   │   │   ├── footer.component.html
│   │   │   └── footer.component.ts
│   │   ├── header
│   │   │   ├── header.component.css
│   │   │   ├── header.component.html
│   │   │   └── header.component.ts
│   │   ├── home
│   │   │   ├── home.component.css
│   │   │   ├── home.component.html
│   │   │   └── home.component.ts
│   │   ├── login
│   │   │   ├── login.component.css
│   │   │   ├── login.component.html
│   │   │   └── login.component.ts
│   │   ├── user-create-job
│   │   │   ├── user-create-job.component.css
│   │   │   ├── user-create-job.component.html
│   │   │   └── user-create-job.component.ts
│   │   ├── user-dashboard
│   │   │   ├── user-dashboard.component.css
│   │   │   ├── user-dashboard.component.html
│   │   │   └── user-dashboard.component.ts
│   │   ├── user-header
│   │   │   ├── user-header.component.css
│   │   │   ├── user-header.component.html
│   │   │   └── user-header.component.ts
│   │   ├── user-sidebar
│   │   │   ├── user-sidebar.component.css
│   │   │   ├── user-sidebar.component.html
│   │   │   └── user-sidebar.component.ts
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.ts
│   │   ├── app.config.server.ts
│   │   ├── app.config.ts
│   │   ├── app.routes.ts
│   │   ├── auth.guard.ts
│   │   ├── auth.service.ts
│   │   ├── info.service.ts
│   │   ├── state-management.service.ts
│   │   └── usersidebar.service.ts
│   ├── assets
│   │   └── favicon.ico
│   ├── environments
│   │   ├── environment.development.ts
│   │   └── environment.ts
│   ├── index.html
│   ├── main.server.ts
│   ├── main.ts
│   ├── script.js
│   └── styles.css
├── angular.json
├── package.json
├── server.ts
├── tsconfig.app.json
└── tsconfig.json
```

To begin explaining our GUI, let's follow the path a user takes through the application.

LoginPage

`auth.service.ts`

```
  // logging in implementation
  login(username: string, password: string): void {
    axios
      .post(
        this.apiUrl+`/login`,
        { username: username, password: password },
        { headers: this.headers }
      )
      .then((response) => {
        this.cookieService.set("sessionToken", response.data.token);
      });
  }

  // logging out implementation
  logOut(): void {
    this.cookieService.delete("sessionToken");
    if (this.isLoggedIn() === false) {
      this.router.navigate(["/login"]);
    }
  }

  // checking if user is logged in implementation
  isLoggedIn(): boolean {
    if (this.cookieService.get("sessionToken") === "") {
      return false;
    } else {
      return true;
    }
  }
}
```

This service file provides the frontend with the ability to validate a user is logged in, or manage their session if not.

We seperate out our functions from the HTML into `service.ts` files so we could maintain a modular code base, increasing readability and reducing errors. This is why our API calls are inside service files, and the design code is seperated.

Each page component, like the login page, is comprised of css, html, typescript and a test file:

```
.
├── login
|   ├── login.component.css
|   ├── login.component.html
|   ├── login.component.spec.ts
|   └── login.component.ts
```

We implement the logic in the `*.component.ts` file and display what was required using `*.component.html` and `*.component.css`.

`login.component.ts`

```
  constructor(private router: Router, private application: MatSnackBar, private cookieService:
CookieService, public authService: AuthService) {}

  ngOnInit(): void{
    if (LoginComponent.showPleaseLogin === true) {
      this.application.open('Unauthorised access, Please login', 'Close', {
        horizontalPosition: this.horizontalPosition,
        verticalPosition: this.verticalPosition,
      });
    }
  }

  onSubmit(username: string, password: string, event: Event): void {
    event.preventDefault();
    this.loading = true;
    this.authService.login(username, password);
    setTimeout(() => {
      this.loading = false;
      if (this.authService.isLoggedIn() === true) {
        this.errorMessage = null;
        this.router.navigate(["/user-dashboard"]);
      } else {
        this.errorMessage = "Invalid Username or Password";
      }
    }, 1000);
  }
}
```

The login functionality was quite easy as it would accept a username and password. These are passed to the API which returns a session cookie. This then acts like the key to get into Prospector. It also provides claims information.

In order to stop users from accessing parts of Prospector that they were not permitted to, we check that the user has the correct permissions to access that page. These secure areas are called protected routes.

`auth.guard.ts`

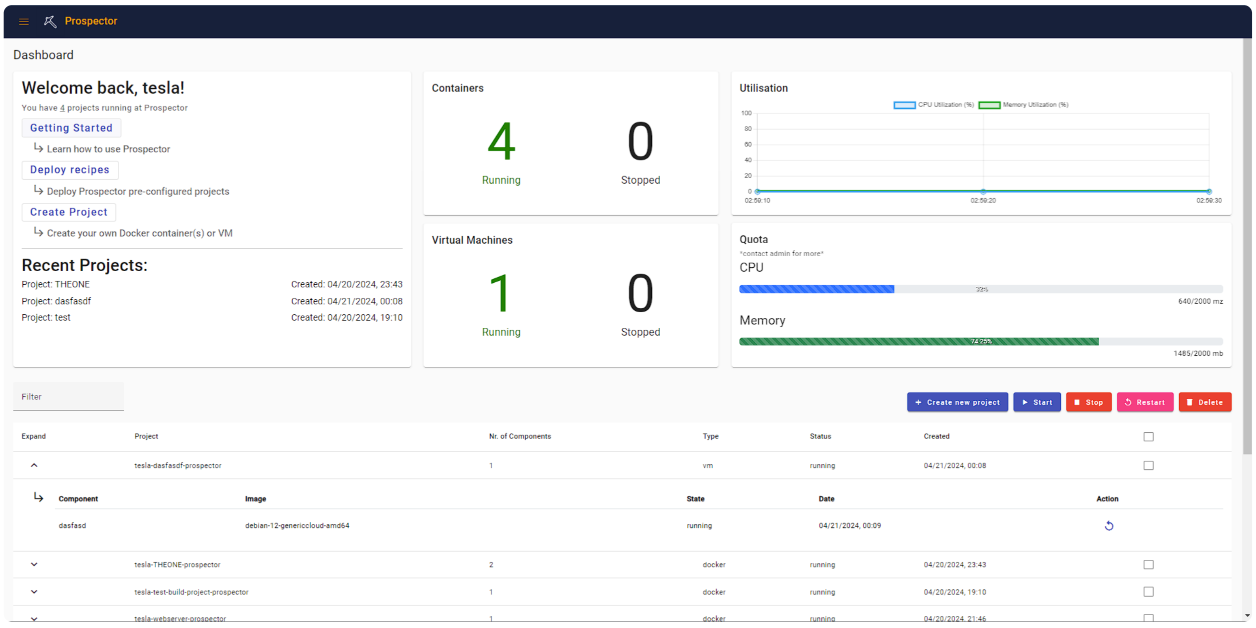
```
import { CanActivateFn, Router, ActivatedRouteSnapshot, RouterStateSnapshot} from '@angular/router';
import { inject } from '@angular/core';
import { AuthService } from './auth.service';

export const authGuard: CanActivateFn = (route: ActivatedRouteSnapshot, state: RouterStateSnapshot) => {
  const router: Router = inject(Router);
  const authService: AuthService = inject(AuthService);
  const protectedRoutes: string[] = ['/user-dashboard'];
  // checks if the user session is unactive and redirects to login page else allows the user to access the page
  if (protectedRoutes.includes(state.url) && authService.isLoggedIn() === false){
    router.navigate(['/login']);
    return false;
  } else {
    return true;
  }
};
```

Great! A user is now logged in!

Dashboard

```
export interface ProjectConstruct {
  project: string;
  component: string;
  type: string;
  status: string;
  created: string;
  isExpanded?: boolean;
}

const ProjectData: ProjectConstruct[] = [
];

export class UserDashboardComponent implements OnInit, OnDestroy {
  constructor(private authService: AuthService, private StateManagementService:
StateManagementService, private elementRef: ElementRef, private InfoService: InfoService, public
dialog: MatDialog, private router: Router) { }

  ngAfterViewInit() {
    this.createUtilChart();
  }

  ngOnInit() {
    this.InfoService.getUser().then(((response: any) => {
      this.userName = response.userName;
    }));

    this.getStats();
    this.getAllocatedResources();
    this.updateRealTimeData();
    this.getRecentProjects();
    this.populateTable();
  }

  ngOnDestroy(): void {
    if (this.updateIntervalId) {
      clearInterval(this.updateIntervalId);
    }
  }


  getStats() {
    this.InfoService.getAllProjects().then((response: any[]) => {
      // gets amount of projects
      this.lengthOfProjects = response.length;

      this.dockerProjects = response.filter(project => project.type === 'docker');
      this.vmProjects = response.filter(project => project.type === 'vm');
      this.runningVM = this.vmProjects.filter(project => project.status === 'running').length;
      this.stoppedVM = this.vmProjects.filter(project => project.status === 'stopped').length;

      for (let i = 0; i < this.dockerProjects.length; i++) {
        const projectId = this.dockerProjects[i].id;
        // gets the components of the project
        this.InfoService.getProjectComponents(projectId).then((response: any[]) => {
          // if the project has more than one component
          if (response.length > 1) {
            response.forEach((component: any) => {
              if (component.state === 'running') {
                this.runningDocker++;
              }
              else if (component.state === 'stopped') {
                this.stoppedDocker++;
              }
            });
          } else {
            if (this.dockerProjects[i].status === 'running') {
              this.runningDocker++;
            }
            else {
              this.stoppedDocker++;
            }
          }
        });
      }
    });
  }

  private updateIntervalId: any;

  updateRealTimeData() {
    this.getCurrentResourceUtilisation();
    this.updateIntervalId = setInterval(() => {
      this.getCurrentResourceUtilisation();

      const time = new Date().toLocaleTimeString(undefined, { hour12: false });
      if (this.chart.data.labels.length >= 30) {
        this.chart.data.labels.shift();
      }
      this.chart.data.labels.push(time);

      if (this.chart.data.datasets[0].data.length >= 30) {
        this.chart.data.datasets[0].data.shift();
      }
      this.chart.data.datasets[0].data.push(this.cpuUtilPercentage);

      if (this.chart.data.datasets[1].data.length >= 30) {
        this.chart.data.datasets[1].data.shift();
      }
      this.chart.data.datasets[1].data.push(this.memUtilPercentage);

      this.chart.update();
    }, 10000);
  }

  getCurrentResourceUtilisation() {
    this.InfoService.getCurrentUtilisation().then((response: any) => {
      this.cpuUtil = response.cpu;
      this.memUtil = response.memory;
      this.cpuUtilPercentage = (response.cpu / this.cpuAllocated) * 100;
      this.memUtilPercentage = (response.memory / this.memAllocated) * 100;
    });
  }

  getAllocatedResources() {
    this.InfoService.getAllocatedResources().then((response: any) => {
      this.cpuAllocated = response.cpu;
      this.memAllocated = response.memory;
      this.cpuQuotaPercentage = (this.cpuAllocated / this.cpuQuota) * 100;
      this.memQuotaPercentage = (this.memAllocated / this.memQuota) * 100;
    });
  }

  dataSource = new MatTableDataSource<ProjectConstruct>(ProjectData);
  selection = new SelectionModel<ProjectConstruct>(true, []);
  displayedColumns: string[] = ['expand', 'project', 'numberOfComponents', 'type', 'status',
'created', 'select'];
  @ViewChild(MatTable) table!: MatTable<ProjectConstruct>;
  populateTable() {
    this.InfoService.getAllProjects().then((projectsResponse: any) => {
      this.dataSource.data = []; // So we can start fresh - Clear previous data

      projectsResponse.forEach((projectSingle: any) => {
        const submitTime = new Date(projectSingle.SubmitTime / 1000000).toLocaleString('en-US',
{ hour12: false });

        // gets components for each project
        this.InfoService.getProjectComponents(projectSingle.id).then((componentsResponse: any[])
=> {
          this.pushProjectData(
            projectSingle.id,
            componentsResponse,
            projectSingle.type,
            // need to see if this component status or project status are they tied ???
            projectSingle.status,
            projectSingle.created
          );
          this.table.renderRows();
        });
      });
    });
  }

  pushProjectData(project: any, component: any, type: any, status: any, created: any) {
    let info = {
      project: project,
      component: component,
      type: type,
      status: status,
      created: this.convertNanosecondsToDate(created)
    };
    this.dataSource.data.push(info);
  }

  applyFilter(event: Event) {
    const filterValue = (event.target as HTMLInputElement).value;
    this.dataSource.filter = filterValue.trim().toLowerCase();
  }

  /** Whether the number of selected elements matches the total number of rows. */
  isAllSelected() {
    const numSelected = this.selection.selected.length;
    const numRows = this.dataSource.data.length;
    return numSelected === numRows;
  }

  /** Selects all rows if they are not all selected; otherwise clear selection. */
  toggleAllRows() {
    if (this.isAllSelected()) {
      this.selection.clear();
      return;
    }

    this.selection.select(...this.dataSource.filteredData);
  }

  /** The label for the checkbox on the passed row */
  checkboxLabel(row?: ProjectConstruct): string {
    if (!row) {
      return `${this.isAllSelected() ? 'deselect' : 'select'} all`;
    }
    return `${this.selection.isSelected(row) ? 'deselect' : 'select'} row ${row.project}`;
  }
  // hack to have only one row expanded at a time
  expandedRow: ProjectConstruct | null = null;

  toggleRow(row: ProjectConstruct) {
    if (this.expandedRow) {
      this.expandedRow.isExpanded = false;
    }
    if (this.expandedRow === row) {
      this.expandedRow = null;
    } else {
      row.isExpanded = true;
      this.expandedRow = row;
      this.getComponentInfo(row.project);
    }
    this.table.renderRows();
  }

  isExpansionDetailRow = (index: number, row: any) => row.isExpanded;

  getRecentProjects() {
    this.InfoService.getAllProjects().then((response: any[]) => {
      if (response.length > 0) {
        this.recentProjects = response.slice(0, 3).map(project => ({
          id: this.extractProjectName(project.id),
          Created: this.convertNanosecondsToDate(project.created)
        }));
      } else {
        this.recentProjects = [];
      }
    });
  }

  extractProjectName(projectId: string): string {
    const parts = projectId.split('-');
    return parts[1];
  }

  componentStats: any[] = [];

  getComponentInfo(projectId: string) {
    this.InfoService.getProjectComponents(projectId).then((response: any[]) => {
      this.componentStats = response;
    });
  }
```

This is one of the biggest files in the frontend, the dashboard dealt with a couple of moving parts since it was displaying critical and useful information to the user.



Welcome back, tesla!

You have 4 projects running at Prospector

↳ Learn how to use Prospector

↳ Deploy Prospector pre-configured projects

↳ Create your own Docker container(s) or VM

Recent Projects:

Project: THEONE	Created: 04/20/2024, 23:43
Project: dasfasdf	Created: 04/21/2024, 00:08
Project: test	Created: 04/20/2024, 19:10

Containers

4 Running 0 Stopped

Virtual Machines

1 Running 0 Stopped

Quota

contact admin for more

CPU 32% 640/2000 mz

Memory 74.25% 1485/2000 mb

Expand	Project	Nr. of Components	Type	Status	Created	☐
^	tesla-dasfasdf-prospector	1	vm	running	04/21/2024, 00:08	☐
v	tesla-THEONE-prospector	2	docker	running	04/20/2024, 23:43	☐
v	tesla-test-build-project-prospector	1	docker	running	04/20/2024, 19:10	☐
v	tesla-webserver-prospector	1	docker	running	04/20/2024, 21:46	☐

Component	Image	State	Date	Action
dasfasd	debian-12-genericcloud-amd64	running	04/21/2024, 00:09	↻

As the page loads, we perform some API calls to populate some information, including:

- Setting the username in the welcome message
- Get Stats, which performs a few tasks:
  - We filter projects based on their deployment type
  - We get the status of every project's components (running or stopped). Green numbers mean running, red numbers mean stoped.
- Fetch the user's quota, resources allocated to projects, and current utilisation of that current allocation.
- Fetch recent projects
- Fetch the information for the table, displaying all of the user's projects.

Below you will see the info service handler where all of the above mentioned data is parsed. This is another example of us splitting up responsibilities. These function calls would be lost in the file above, here they are much easier maintained.

`info.service.ts`

```
import { Injectable } from '@angular/core';
import axios from 'axios';
import { CookieService } from 'ngx-cookie-service';
import { environment } from '../environments/environment';

@Injectable({
  providedIn: 'root'
})
export class InfoService {

  constructor(private cookieService: CookieService) {}

  apiUrl = environment.apiUrl;

  getUser() {
    return axios.get(this.apiUrl+`/v1/user`, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
    .then(response => {
      // console.log(response.data);
      return response.data;
    });
  }

  // placeholder for now to post JOB
  postJob(data: any) {
    return axios.post(this.apiUrl + `/v1/jobs`, data, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
      .then(response => {

        return response.data;
      });
  }

  // get all projects created
getAllProjects(){
  return axios.get(this.apiUrl+`/v1/jobs`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  })
  .then(response => {
    // console.log(response.data);
    return response.data;
  });
}

// get all long projects (/v1/jobs?long=true)
getAllLongProjects(){
  return axios.get(this.apiUrl+`/v1/jobs?long=true`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  })
  .then(response => {
    // console.log(response.data);
    return response.data;
  });
}

// get project components
getProjectComponents(projectId: string){
  return axios.get(this.apiUrl+`/v1/jobs/${projectId}/components`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  })
  .then(response => {
    // console.log(response.data);
    return response.data;
  });
}


// return project when given id
getProjectById(id: number): Promise<any> {
  return axios.get(this.apiUrl+`/jobs/${id}`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  })
  .then(response => {
    return response.data;
  });
}

//get Utilization
getCurrentUtilisation(){
  return axios.get(this.apiUrl+`/v1/resources`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  }).then(response => {
    // console.log(response.data);
    return response.data;
  });
}

getAllocatedResources(){
  return axios.get(this.apiUrl+`/v1/resources/allocated`, {
    headers: {
      "Content-Type": "application/json",
      'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
    }
  }).then(response => {
    // console.log(response.data);
    // convert to percentage
    return response.data
  });
}

}
```

The dashboard provides the user with the ability to engage or view stats from their running projects in Prospector.

The `UpdateRealtimeData` function updates the user's current resource usage every 10 seconds, displaying it on the dashboard.

```
getCurrentResourceUtilisation() {
  this.InfoService.getCurrentUtilisation().then((response: any) => {
    this.cpuUtil = response.cpu;
    this.memUtil = response.memory;
    this.cpuUtilPercentage = (response.cpu / this.cpuAllocated) * 100;
    this.memUtilPercentage = (response.memory / this.memAllocated) * 100;
  });
}
```

We were very conscious of only displaying useful and relevant information to the user on the dashboard. Providing quick access to all of their projects on the homepage allows the user to access the subject of this project very quickly.

`state-management.service.ts`

```typescript
import { Injectable } from '@angular/core';
import axios from 'axios';
import { CookieService } from 'ngx-cookie-service';
import { environment } from '../environments/environment';

@Injectable({
  providedIn: 'root'
})
export class StateManagementService {

  constructor(private cookieService: CookieService) {}

  apiUrl = environment.apiUrl;

  startProject(projectId: string){
    return axios.post(this.apiUrl+`/v1/jobs/${projectId}/start`, {}, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
    .then(response => {
      return response.data;
    });
  }

  stopProject(projectId: string){
    return axios.delete(this.apiUrl+`/v1/jobs/${projectId}?purge=false`, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
    .then(response => {
      return response.data;
    });
  }

  deleteProject(projectId: string){
    return axios.delete(this.apiUrl+`/v1/jobs/${projectId}?purge=true`, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
    .then(response => {
      return response.data;
    });
  }

  restartProject(projectId: string){
    return axios.put(this.apiUrl+`/v1/jobs/${projectId}/restart`, {}, {
      headers: {
        "Content-Type": "application/json",
        'Authorization': `Bearer ${this.cookieService.get("sessionToken")}`
      }
    })
    .then(response => {
      return response.data;
    });
  }
  restartComponent(projectId: string, componentId: string){
    return axios.put(this.apiUrl+`/v1/jobs/${pro
```

The use of these `*.service.ts` handlers really helped in diagnosing any issues we came across, alongside allowing us to test things manually on the fly.

Bootstrap came in super handy in the creation of the dashboard thanks to its simple grid system where it allowed us to define a proper grid to populate each of the cards to be placed equally and legible.

Bootstrap classes that defined a row and columns

```
<div class="row d-flex">
<div class="col-sm-4 mb-2 d-flex flex-column">
<div class="col-sm-3 mb-2 d-flex flex-column">
```

If a user can view their projects, then they must have the ability to create more projects. A project is made up of any number of components (as long as you're under your quota!) and providing a simple way to create these components is vital to the success of the applicaiton.

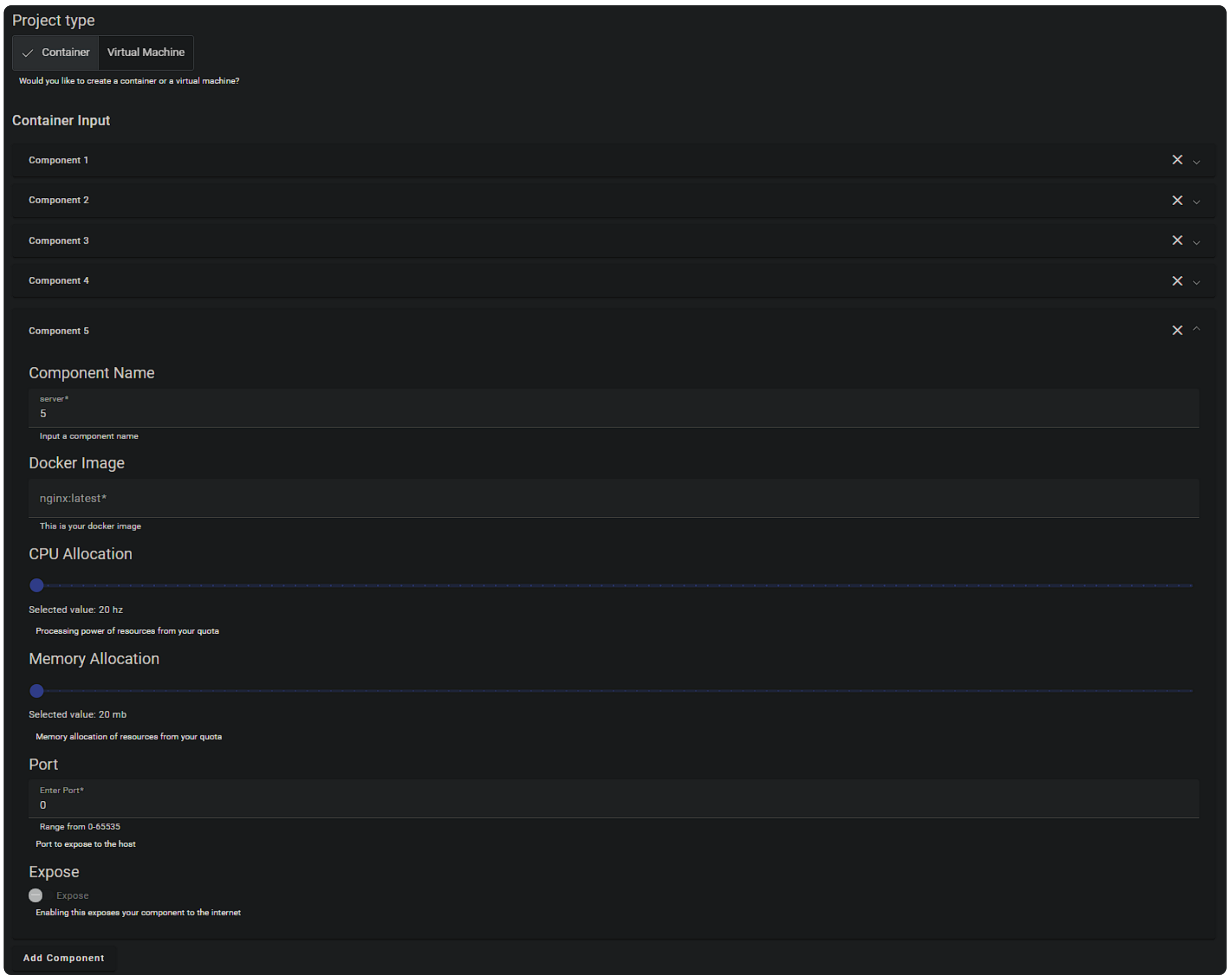
Create Project

```
interface image {
  value: string;
  viewValue: string;
}

@Component({
  selector: 'app-user-create-job',
  standalone: true,
  imports: [MatTooltipModule, MatExpansionModule, MatIconModule, RouterLink, RouterOutlet,
MatProgressBarModule, CommonModule, MatSliderModule, MatSlideToggleModule, MatFormFieldModule,
MatInputModule, MatSelectModule, FooterComponent, UserHeaderComponent, UserSidebarComponent,
MatButtonToggleModule, FormsModule, ReactiveFormsModule, HeaderComponent, MatButtonModule],
  templateUrl: './user-create-job.component.html',
  styleUrl: './user-create-job.component.css'
})

export class UserCreateJobComponent {
  projectName: string;
  instanceType: string;
  formSubmitted: boolean;
  selectedValue: string;

  components: any[] = [];

  images: image[] = [
    {  viewValue: 'Ubuntu' },
    {  viewValue: 'Fedora' },
    {  viewValue: 'Debian' },
  ];

  constructor(private InfoService: InfoService, private router: Router) {
    this.projectName = '';
    this.instanceType = '';
    this.formSubmitted = false;
    this.selectedValue = '';
  }

  ngOnInit() {
    this.step = 0;
  }

  onSubmit() {

    const data = {
      components: this.components.map(component => ({
        image: component.Image,
        name: component.Name,
        network: {
          expose: component.Network.Expose,
          port: parseInt(component.Network.Port)
        },
        resources: {
          cpu: parseInt(component.Resources.cpuValue),
          memory: parseInt(component.Resources.ramValue)
        },
        user_config: {
          ssh_key: component.User_config.ssh_key,
        },
        volumes: component.Volumes.split(',').map((value: string) => {
          return value.trim();
        }),
      })),
      name: this.projectName,
      type: this.instanceType,
    };

    console.log('Form submitted with data', data);
    console.log(this.components)
    // check in place to remove empty volumes since the API should not recieve them
    for (let component of data.components) {
      if (component.volumes.length === 1 && component.volumes[0] === "") {
        delete component.volumes;
      }
    }
    this.InfoService.postJob(data);

    this.formSubmitted = true;
    setTimeout(() => {
      this.router.navigate(['/user-dashboard']);
    }, 2000);
  }

  step = 1;

  setStep(index: number) {
    this.step = index;
  }

  nextStep() {
    this.step++;
  }

  lastStep() {
    this.step;
  }

  componentAdded = false;

  addComponent() {
    console.log(this.step)
    this.componentAdded = true;
    this.components.push({
      Name: '',
      Image: '',
      Network: {
        Port: 0,
        Expose: false,
      },
      Resources: {
        cpuValue: 20,
        ramValue: 20
      },
      User_config: {
        ssh_key: ''
      },
      Volumes: ""
    });
  }

  removeComponent(index: number) {
    console.log('Removing component');
    this.components.splice(index, 1);
  }

  onToggleChange(event: any, index: number) {
    if (event.checked) {
      // The toggle is checked
      console.log('Toggle is on for component', index);
    } else {
      // The toggle is not checked
      console.log('Toggle is off for component', index);
    }
  }

  formatCPULabel(value: number) {
    return value + 'hz';
  }

  formatRAMLabel(value: number) {
    return value + 'MB';
  }

  // manual fix for the form validation
  isFormValid() {
    if (!this.projectName || !this.instanceType) {
      return false;
    }

    for (let component of this.components) {
      if (this.instanceType === 'vm') {
        // Validation rules for 'Virtual Machine'
        if (!component.Name || !component.Image || component.Resources.cpuValue === 0 ||
component.Resources.ramValue === 0) {
          return false;
        }
      } else if (this.instanceType === 'docker') {
        // Validation rules for 'Container'
        // Source : https://regex101.com/r/hP8bK1/1
        let dockerRegex = new RegExp("^(?:(?=[^:\/]{4,253})(?!-)[a-zA-Z0-9-]{1,63}(?<!-)(?:\.
(?!-)[a-zA-Z0-9-]{1,63}(?<!-))*?/)?((?![._-])(?:[a-z0-9._-]*)(?<![._-])(?:/(?![._-])[a-z0-9._-]*
(?<![._-]))*)?$");
        if (!component.Name || component.Resources.cpuValue === 0 ||
component.Resources.ramValue === 0 || (!dockerRegex.test(component.Image) && component.Image)) {
return false;
        }
      }
    }

    return true;
  }
  // manual fix for when toggling between container or vm the form data is reset
  resetForm() {
    this.componentAdded = false;
    this.components = [];
  }

}
```

Here you will see that its not as simple as just making a simple API call to the backend, but rather there is validation in place to not allow an user to submit incomplete projects to Prospector.

```
    for (let component of this.components) {
      if (this.instanceType === 'vm') {
        // Validation rules for 'Virtual Machine'
        if (!component.Name || !component.Image || component.Resources.cpuValue === 0 ||
component.Resources.ramValue === 0) {
          return false;
        }
      } else if (this.instanceType === 'docker') {
        // Validation rules for 'Container'
        // Source : https://regex101.com/r/hP8bK1/1
        let dockerRegex = new RegExp("^(?:(?=[^:\/]{4,253})(?!-)[a-zA-Z0-9-]{1,63}(?<!-)(?:\.
(?!-)[a-zA-Z0-9-]{1,63}(?<!-))*?/)?((?![._-])(?:[a-z0-9._-]*)(?<![._-])(?:/(?![._-])[a-z0-9._-]*
(?<![._-]))*)?$");
        if (!component.Name || component.Resources.cpuValue === 0 ||
component.Resources.ramValue === 0 || (!dockerRegex.test(component.Image) && component.Image)) {
return false;
        }
      }
    }
```

Here you can see an example of how nicely presented each of the components were to the user. In these cards that would expand upon clicking it.



Project type
✓ Container | Virtual Machine
Would you like to create a container or a virtual machine?

Container Input

Component 1
Component 2
Component 3
Component 4
Component 5

Component Name
server*
5
Input a component name

Docker Image
nginx:latest*
This is your docker image

CPU Allocation
Selected value: 20 hz
Processing power of resources from your quota

Memory Allocation
Selected value: 20 mb
Memory allocation of resources from your quota

Port
Enter Port*
0
Range from 0-65535
Port to expose to the host

Expose
Expose
Enabling this exposes your component to the internet

Add Component

Create job takes input from the user, populating fields in the data object ready to be sent to the API in the service handler. Once the request completes, the user is redirected back to their dashboard.

Project Page

Finally there is the project page where you can view logs and edit your components.

`user-project-page.component.ts`

```
@Component({
  selector: 'app-user-project-page',
  standalone: true,
  imports: [CommonModule, UserHeaderComponent,
    UserSidebarComponent,
    FooterComponent,
    MatTabsModule, MatCardModule, MatButtonToggleModule, MatFormFieldModule, MatSelectModule,
MatButtonModule, MatChipsModule, MatIcon
    , FormsModule, ReactiveFormsModule, MatFormFieldModule, MatSliderModule, MatFormField,
MatInputModule, MatSlideToggleModule],
  templateUrl: './user-project-page.component.html',
  styleUrl: './user-project-page.component.css'
})

export class UserProjectPageComponent implements OnInit, OnDestroy {
  id: string = '';
  components: any = [];
  selectedComponent: any;
  logs: string[] = [];

  projectDefinition: any = {};
  componentToBeReplaced: any = {};

  constructor(private route: ActivatedRoute, private InfoService: InfoService, private
elementRef: ElementRef, public dialog: MatDialog, private router: Router, private
StateManagementService: StateManagementService) { }

  ngAfterViewInit() {
  }

  ngOnInit() {
    this.id = this.route.snapshot.paramMap.get('id') ?? '';
    this.getComponents();
    this.getProjectDefintion();
  }

  ngOnDestroy(): void {
  }

  getComponents() {
    this.InfoService.getProjectComponents(this.id).then((data) => {
      this.components = data;
    });
  }

  selectComponent(component: any) {
    this.selectedComponent = component;
    this.findComponentInProjectDefinition();
  }

  getComponentLogs(projectId: string, componentId: string, type: string) {
    this.logs = [];
    this.InfoService.getComponentLogs(projectId, componentId, type).then(async (data) => {
      let stream: any;

      let readStream = async () => {
        const { value, done } = await stream.read();

        if (done) {
          console.log("Stream is done");
          return;
        }
        console.log("Reading stdout");
        const text = new TextDecoder().decode(value);
        const lines = text.split('\n');
        this.logs.push(...lines);
        readStream();
      }

      if (data.status === 200 && data.body) {
        stream = await data.body.getReader();
        readStream();
      }
    });
  }

  restartComponentButton() {
    this.StateManagementService.restartComponent(this.id, this.selectedComponent.id).then((data)
=> {
    });
  }

  getProjectDefintion() {
    this.InfoService.getProjectDefinition(this.id).then((data) => {
      this.projectDefinition = data;
    });
  }

  // lets find the component that is selected in the project definition
  findComponentInProjectDefinition() {
    for (let i = 0; i < this.projectDefinition.components.length; i++) {
      if (this.projectDefinition.components[i].name === this.selectedComponent.name) {
        this.componentToBeReplaced = this.projectDefinition.components[i];
        // modify the volumes to be a string
        console.log('Component to be replaced', this.componentToBeReplaced);
        this.componentToBeReplaced.volumes =
this.setVolumeString(this.componentToBeReplaced.volumes);
        console.log('Component to be replaced', this.componentToBeReplaced);
      }
    }
  }

  onSubmit() {
    this.componentToBeReplaced.volumes =
this.componentToBeReplaced.volumes.split(',').map((value: string) => {
      return value.trim();
    });
    this.componentToBeReplaced.resources.cpu =
parseInt(this.componentToBeReplaced.resources.cpu);
    this.componentToBeReplaced.resources.memory =
parseInt(this.componentToBeReplaced.resources.memory);
    this.componentToBeReplaced.network.port = parseInt(this.componentToBeReplaced.network.port);
    this.InfoService.updateProjectDefinition(this.id, this.projectDefinition).then((data) => {
      console.log('Project definition updated');
    });
  }

  outputTest() {
    console.log(this.componentToBeReplaced);
  }

  formatCPULabel(value: number) {
    return value + 'hz';
  }

  formatRAMLabel(value: number) {
    return value + 'MB';
  }

  setVolumeString(volumes: string[]) {
    return volumes.join(', ');
  }

}
```

Here we can break up this page by 2 parts. One the logs and other edit.

The logs componment makes an API call that is upgraded to a websocket and returned. This connection is kept open and streams the logs of a given component to the user in the GUI.

Edit is almost the exact same as create job, the only difference is that there is some extra logic handlers because we want to updat an old component, not create a whole new one. We do this by calling the API for the project definition, populating the frontend form with the selected component definition, and finally sending that back to the API to update the component.

Nomad

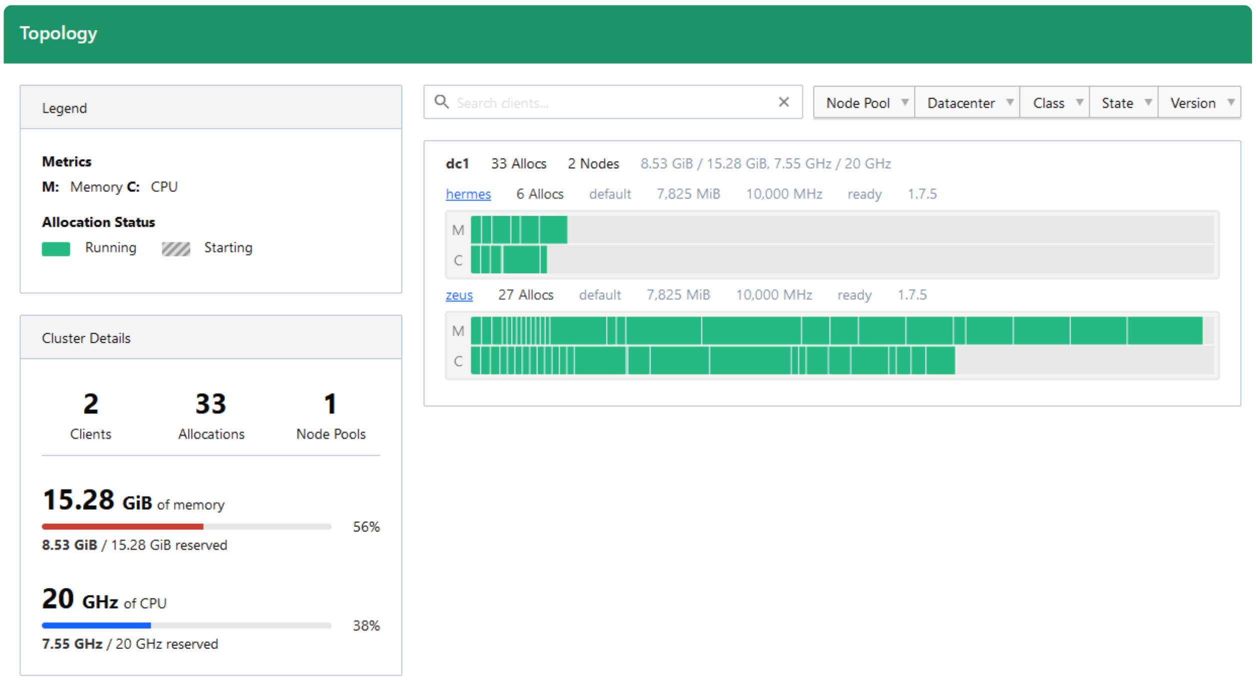
Nomad is an open-source product by HashiCorp (who make Terraform, Vault and Consul also) and is a simple and flexible scheduler and orchestrator for managing containers and virtual machines at scale.

We decided to use Nomad instead of directly communicating with the Docker API or QEMU Hypervisor ourselves, as it provides scalability and availability by default, and for a project like Prospector 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.

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

Topology


Topology

Legend

Metrics
M: Memory **C:** CPU

Allocation Status
🟩 Running ▨ Starting

Cluster Details

2	33	1
Clients	Allocations	Node Pools

15.28 GiB of memory 56%
8.53 GiB / 15.28 GiB reserved

20 GHz of CPU 38%
7.55 GHz / 20 GHz reserved

Search clients...

Node Pool | Datacenter | Class | State | Version

dc1 33 Allocs 2 Nodes 8.53 GiB / 15.28 GiB, 7.55 GHz / 20 GHz

hermes	6 Allocs	default	7,825 MiB	10,000 MHz	ready	1.7.5
zeus	27 Allocs	default	7,825 MiB	10,000 MHz	ready	1.7.5

This is a screenshot of the Topology page from the Nomad Web UI. You can see there are two nodes in this cluster - `hermes` and `zeus`, with 33 allocations across the two hosts. Nomad uses an allocation strategy called "bin packing" to maximise the usage of each connected client node which is why the allocations favour `zeus`. Adding more clients will increase the cluster resources thus increasing the number of allocations (instances of a job) that can be run across the cluster.

Server/Client Configuration

```
datacenter = "dc1"
data_dir = "/opt/nomad"
bind_addr = "0.0.0.0"

advertise {
  http = "zeus.internal"
  rpc  = "zeus.internal"
  serf = "zeus.internal"
}

# expose metrics for collection in prometheus
telemetry {
  collection_interval = "10s"
  disable_hostname = true
  prometheus_metrics = true
  publish_allocation_metrics = true
  publish_node_metrics = true
}

server {
  enabled = true
  bootstrap_expect = 3
}

client {
  enabled = true
}

plugin "raw_exec" {
  config {
    enabled = true
  }
}

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

The server stanza is enabled here, so a node with this configuration will run as a control server for the cluster. Usually there are at least 3 servers, sometimes 5 or 7 depending on uptime and resilience requirements. The reason for an odd number is to prevent split brains during leader election. Any node can be elected leader if it gets enough votes, and if it dies, any other node can run for election. Votes are cast via heartbeat pings to every other server node in the cluster. These heartbeats also carry updates about the cluster state and consensus information.

The client stanza is also enabled on this node. This means that it is eligible for allocations to be deployed on it. Provided a client has the correct driver (configured with the plugin stanzas) and is not excluded by placement constraints, any allocation can be placed on it. These clients also heartbeat against the servers. If a server notices a client has become unhealthy, it will recreate the unhealthy allocations on another node. In this way, provided there is space for it, an allocation can always be available.

We also enable telemetry on this node so that we can collect information about the state of the cluster and it's clients for displaying on a dashboard for administrators elsewhere.

Example Nomad Job

```
job "nginx" {
  datacenters = ["dc1"]
  type        = "service"
  group "web" {
    network {
      port "http" {
        to = 80
      }
    }
    service {
      name = "nginx"
      port = "http"
      check {
        type     = "http"
        path     = "/index.html"
        interval = "10s"
        timeout  = "2s"
      }
      tags = [
        "traefik.enable=true",
        "traefik.http.routers.nginx.rule=Host(`nginx.prospector.ie`)",
        "traefik.http.routers.nginx.entrypoints=websecure",
        "traefik.http.routers.nginx.tls.certresolver=lets-encrypt"
      ]
    }
    task "server" {
      driver = "docker"
      config {
        image = "nginx"
        ports = ["http"]
      }
    }
  }
}
```

This is an example of a simple Nomad Job that runs an NGINX server. The `service` stanza registers the allocation with Consul, which informs Traefik what tags it should use for this container, in this case `nginx.prospector.ie` is routed to this allocation.

Traefik

Traefik is an open-source reverse proxy and load balancer, which we use for providing SSL certificates for HTTPS and exposing services within Prospector. Traefik provides integration with Docker, Nomad and Consul to discover services deployed on the network and provide routes for accessing these services over the web.

Traefik is deployed and managed by Nomad. The job is similar to the NGINX one posted above, but includes some extra Traefik configuration. See the full file at `src/configs/traefik.hcl`.

```
[api]
  dashboard = true
  insecure = true

[entryPoints]
  [entryPoints.web]
  address = ":80"
  [entryPoints.web.http.redirections.entryPoint]
    to = "websecure"
    scheme = "https"

  [entryPoints.websecure]
    address = ":443"

  [entryPoints.traefik]
    address = ":8081"

[providers.consulCatalog]
  prefix = "traefik"
  exposedByDefault = false
  [providers.consulCatalog.endpoint]
    address = "127.0.0.1:8500"
    scheme  = "http"

[providers.nomad]
  prefix = "traefik"
  [providers.nomad.endpoint]
    address = "http://127.0.0.1:4646"

[certificatesResolvers.lets-encrypt.acme]
  email = "jamesthackett1@gmail.com"
  storage = "local/acme.json"
  [certificatesResolvers.lets-encrypt.acme.tlsChallenge]
```

This file configures Traefik to use both Nomad and Consul as sources for configuration information. This lets it route seamlessly and quickly to any number of running Nomad jobs that are configured to be proxied.

We can see two `entryPoints`, these are named `web` and `websecure`, which are mapped to ports `80` and `443`. These names are used later for configuring routing rules.

Both of the `providers` sections define where Traefik should pull its routing information from. One example of this is using the Nomad provider and tags as specified in the NGINX job above.

CI/CD

CI/CD stands for continuous integration/continuous deployment, it's the practice of always testing your changes via automated testing and having a pipeline that takes a change from development into production.

Our CI/CD is set up using GitLab's offering. Below is a high level view of our pipeline definition.

```yaml
stages:
  - build
  - report
  - review
  - deploy_canary
  - deploy_prod

go:
  stage: build
    trigger:
    include: src/api/.gitlab-ci.yml
    strategy: depend

report-go:
  stage: report

frontend:
  stage: build

deploy_canary:
  stage: deploy_canary

deploy_prod:
  stage: deploy_prod

deploy_review:
  stage: review

stop_review:
  stage: review
```

We'll dive into each stage in more detail, but essentially we build each of our components (frontend and API), deploy a review application if the change has been committed to a branch that isn't master, and stop the review application once the pull request has been merged. If the pipeline runs on the master branch, we deploy to our canary environment to validate changes with production data, but without the production traffic, then we finally promote the canary deployment to production.

Stage `go`

```yaml
image: golang:1.22.0

cache:
  key: gocache
  paths:
    - vendor/go/pkg/mod/
    - vendor/cache
    - vendor/linter-cache

variables:
  GOCACHE: ${CI_PROJECT_DIR}/vendor/cache
  GOPATH: ${CI_PROJECT_DIR}/vendor/go
  GOLANGCI_LINT_CACHE: ${CI_PROJECT_DIR}/vendor/linter-cache

stages:
  - test
  - build

before_script:
  - cd src/api

lint:
  stage: test
  script:
    - go vet ./... # this is golang slang for recursively find all packages

unit-test:
  stage: test
  script:
    - go test -v -race -coverprofile=coverage.out ./...
    - "go tool cover -func coverage.out | sed -n -e '/^total/s/:.*statements)[^0-9]*/: /p' >
coverage.txt"
    - go get github.com/boumenot/gocover-cobertura
    - go run github.com/boumenot/gocover-cobertura < coverage.out > cobertura.xml
    - go install github.com/jstemmer/go-junit-report/v2@latest
    - go test -v 2>&1 ./... | $GOPATH/bin/go-junit-report -set-exit-code > report.xml
  cache:
    - key: "$CI_COMMIT_REF_SLUG-reports"
      paths:
        - src/api/report.xml
        - src/api/coverage.txt
        - src/api/cobertura.xml

build:
  stage: build
  image: docker:stable
  services:
    - name: docker:dind
  variables:
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - echo $CI_GITEA_KEY | docker login -u distro --password-stdin git.dbyte.xyz
    - cd src/api
  script:
    - |
      if [[ "$CI_COMMIT_BRANCH" == "$CI_DEFAULT_BRANCH" ]]; then
        tag=""
        echo "Running on default branch '$CI_DEFAULT_BRANCH': tag = 'latest'"
      else
        tag=":$CI_COMMIT_SHORT_SHA"
        echo "Running on commit '$CI_COMMIT_SHORT_SHA': tag = $tag"
      fi
    - docker build --pull -t "git.dbyte.xyz/distro/prospector/api${tag}" .
    - docker push "git.dbyte.xyz/distro/prospector/api${tag}"
```

This stage of the pipeline is defined in the `src/api` directory, this helps keep the main `.gitlab-ci.yml` file tidy.

First up we define the cache for this stage of the pipeline. This allows future stages to perform their tasks quicker because they can rely on built artefacts from previous stages of the job. In this case we cache Go's package directories to save time on redownloading packages.

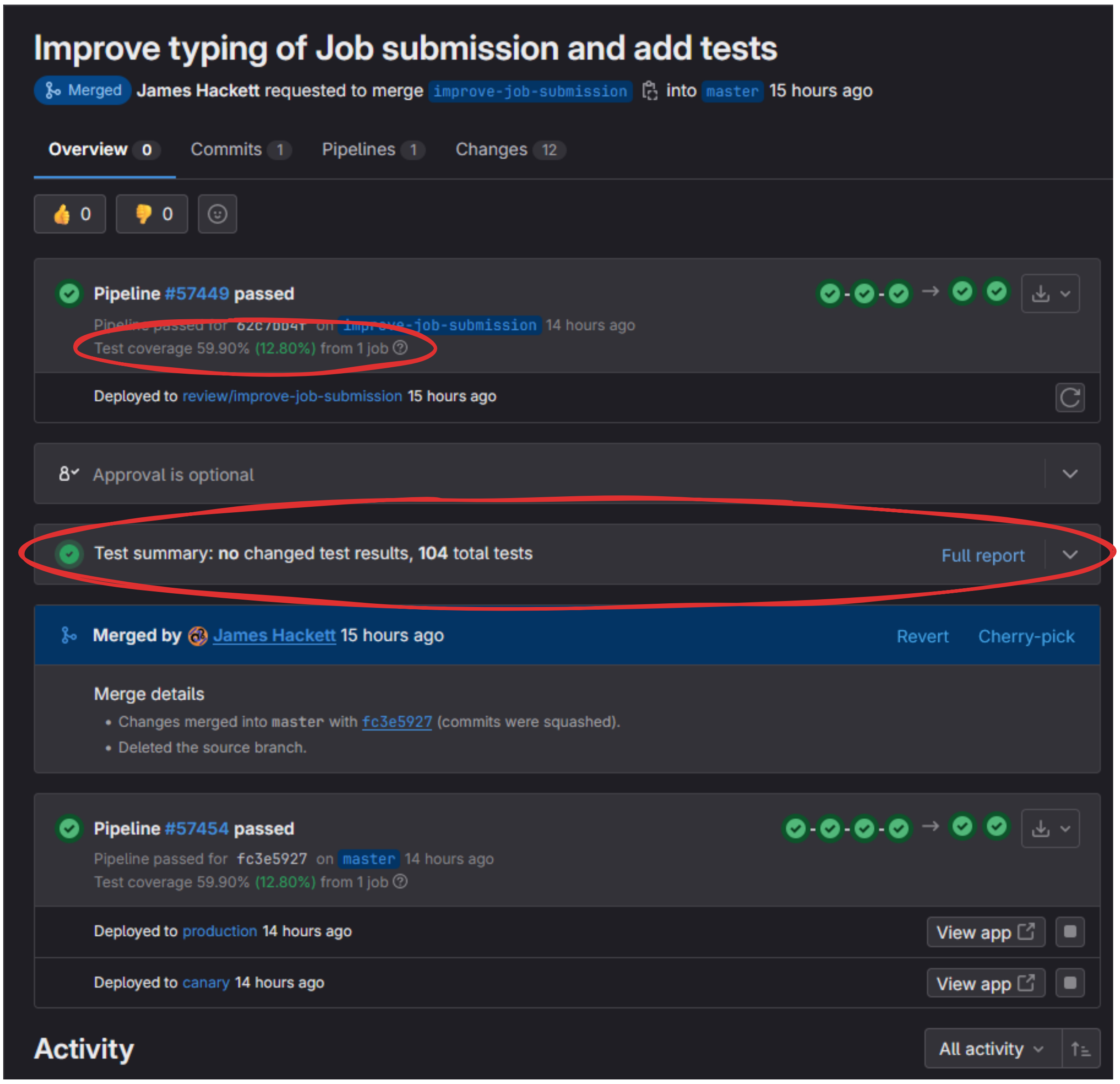
Next we perform a lint and test stage. The lint stage uses Golang's `vet` tool to validate the code superficially conforms to Go's standards. The test stage runs through the API's test suite, performing the tests and outputting the results into a report. Next it generates a coverage percentage of the API's code, a coverage `report.xml` and a `covertura.xml`. These files are parsed by GitLab and are used to show code coverage and test reports on merge requests. These are stored in a keyed cache for use by the parent pipeline later due to a bug that prevents parents accessing reports from children - see here for more.

Finally we build a Docker image for the binary. We tag the image differently based on the branch - `latest` for master branch and the git SHA for branch builds. This is then pushed up to a remote image registry for deployment later.

Stage `report-go`

```yaml
report-go:
  stage: report
  dependencies:
    - go
  script:
    - echo "Generating reports"
    - cat src/api/coverage.txt
  cache:
    key: "$CI_COMMIT_REF_SLUG-reports"
    paths:
      - src/api/report.xml
      - src/api/coverage.txt
      - src/api/cobertura.xml
    policy: pull
  artifacts:
    when: always
    reports:
      coverage_report:
        coverage_format: cobertura
        path: src/api/cobertura.xml
      junit: src/api/report.xml
  coverage: /total:.*\s(\d+(?:\.\d+)?%)/
```

The `report-go` stage grabs the reports generated from the `go` stage and provides them to Gitlab. This allows things like coverage and tests to be shown on the overview of a merge request, and also for test coverage to appear in the merge request diff. This is the second part of the workaround for the bug mentioned above. It grabs the reports from the cache and populates all the necessary fields.



Improve typing of Job submission and add tests

Merged James Hackett requested to merge `improve-job-submission` into `master` 15 hours ago

Overview 0 · Commits 1 · Pipelines 1 · Changes 12

👍 0 · 👎 0

Pipeline #57449 passed
Pipeline passed for 62c7bb4f on `improve-job-submission` 14 hours ago
Test coverage 59.90% (12.80%) from 1 job

Deployed to review/improve-job-submission 15 hours ago

Approval is optional

Test summary: no changed test results, **104** total tests · Full report

Merged by James Hackett 15 hours ago · Revert · Cherry-pick

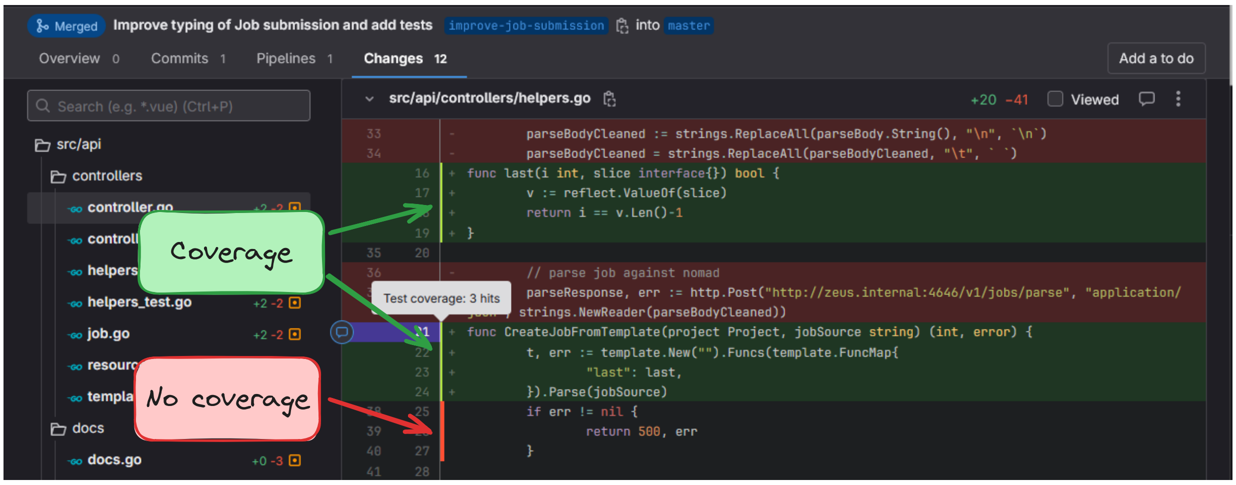
Merge details
- Changes merged into `master` with fc3e5927 (commits were squashed).
- Deleted the source branch.

Pipeline #57454 passed
Pipeline passed for `fc3e5927` on `master` 14 hours ago
Test coverage 59.90% (12.80%) from 1 job

Deployed to production 14 hours ago · View app

Deployed to canary 14 hours ago · View app

Activity · All activity



Merged · Improve typing of Job submission and add tests · `improve-job-submission` into `master`

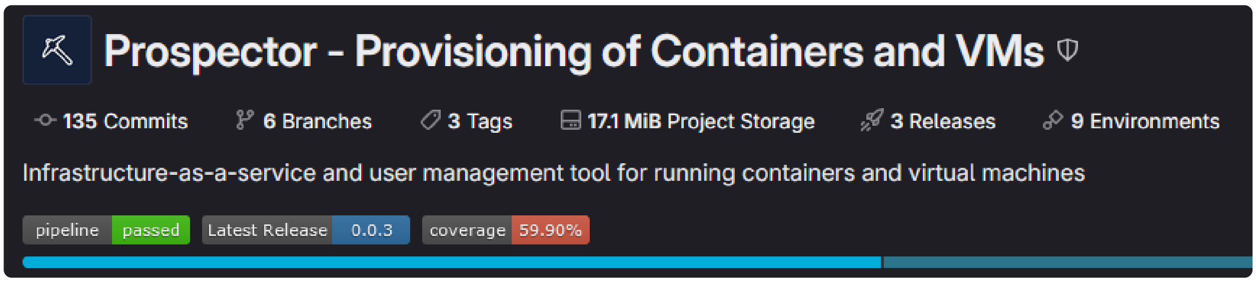
Overview 0 · Commits 1 · Pipelines 1 · Changes 12 · Add a to do

src/api/controllers/helpers.go +20 −41 Viewed

```
-       parseBodyCleaned := strings.ReplaceAll(parseBody.String(), "\n", `\n`)
-       parseBodyCleaned = strings.ReplaceAll(parseBodyCleaned, "\t", ` `)
+ func last(i int, slice interface{}) bool {
+       v := reflect.ValueOf(slice)
+       return i == v.Len()-1
+ }

-       // parse job against nomad
        parseResponse, err := http.Post("http://zeus.internal:4646/v1/jobs/parse", "application/
        strings.NewReader(parseBodyCleaned))
+ func CreateJobFromTemplate(project Project, jobSource string) (int, error) {
+       t, err := template.New("").Funcs(template.FuncMap{
+               "last": last,
+       }).Parse(jobSource)
        if err != nil {
                return 500, err
        }
```

Test coverage: 3 hits

Screenshots showing reporting in merge requests. The coverage percentage is also reported on a badge on the project's repository homepage:


Prospector - Provisioning of Containers and VMs

135 Commits 6 Branches 3 Tags 17.1 MiB Project Storage 3 Releases 9 Environments

Infrastructure-as-a-service and user management tool for running containers and virtual machines

Stage `frontend`

```yaml
stages:
  - build

before_script:
  - cd src/frontend

build:
  stage: build
  image: docker:stable
  services:
    - name: docker:dind
  variables:
    DOCKER_TLS_CERTDIR: ""
  before_script:
    - echo $CI_GITEA_KEY | docker login -u distro --password-stdin git.dbyte.xyz
    - cd src/frontend
  script:
    - |
      if [[ "$CI_COMMIT_BRANCH" == "$CI_DEFAULT_BRANCH" ]]; then
        tag=""
        echo "Running on default branch '$CI_DEFAULT_BRANCH': tag = 'latest'"
      else
        tag=":$CI_COMMIT_SHORT_SHA"
        echo "Running on commit '$CI_COMMIT_SHORT_SHA': tag = $tag"
      fi
    - docker pull git.dbyte.xyz/distro/prospector/frontend:latest || true
    - docker build --cache-from git.dbyte.xyz/distro/prospector/frontend:latest --pull --tag
"git.dbyte.xyz/distro/prospector/frontend${tag}" .
    - docker push "git.dbyte.xyz/distro/prospector/frontend${tag}"
```

The frontend stage builds a container image for the Angular frontend.

Stages `deploy_review` & `stop_review`

```
deploy_review:
  image: git.dbyte.xyz/distro/levant
  stage: review
  dependencies:
    - go
    - frontend
  script:
    - src/tools/deploy-review.sh
  except:
    - master
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: https://$CI_ENVIRONMENT_SLUG.prospector.ie
    on_stop: stop_review

stop_review:
  image: multani/nomad
  stage: review
  before_script:
    - echo "null"
  variables:
    GIT_STRATEGY: none
  script:
    - nomad status -address=http://nomad.service.consul:4646
    - nomad job stop -address=http://nomad.service.consul:4646 -purge
prospector-${CI_ENVIRONMENT_SLUG}
  except:
    - master
  when: manual
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
```

As mentioned before, we deploy review applications for each merge request (really for every commit on every branch that isn't `master`). These two stages are combined here because they cannot work without each other.

The `deploy_review` stage uses a custom built Docker image to run a script on the `gitlab-runner` which is running on the same cluster as the deployment of our project. This allows the script to automatically deploy an application to the Nomad cluster that runs there. The review application will then be available at `https://review-${CI-ENVIRONMENT-SLUG}.prospector.ie`, which is essentially a combination of the branch name and the git SHA of the branch.

```
#!/usr/local/bin/bash

DEPLOY_URL="${CI_ENVIRONMENT_SLUG}.prospector.ie"
echo "Deploy URL: ${DEPLOY_URL}"
levant deploy \
  -var git_sha="${CI_COMMIT_SHORT_SHA}" \
  -var environment_slug="${CI_ENVIRONMENT_SLUG}" \
  -var deploy_url="${DEPLOY_URL}" \
  -address "http://nomad.service.consul:4646" \
  src/tools/templates/prospector-review.hcl
```

This script templates out a Nomad job (that contains the deployment definition of our application). Then it deploys the application.

The `stop_review` stage stops the application either manually or after a merge request has been merged (and the source branch deleted). Notable in the `stop_review` stage is the `GIT_STRATEGY: none` to prevent the runner from failing to pull a non-existent branch.

Stages `deploy_canary` & `deploy_prod`

```yaml
deploy_canary:
  image: git.dbyte.xyz/distro/levant
  stage: deploy_canary
  dependencies:
    - go
    - frontend
  script:
    - src/tools/deploy-canary.sh
  only:
    - master
  environment:
    name: canary
    url: https://canary.prospector.ie

deploy_prod:
  image: multani/nomad
  stage: deploy_prod
  allow_failure: false
  when: manual
  dependencies:
    - go
    - frontend
  script:
    - nomad job promote -address=http://nomad.service.consul:4646 prospector
  only:
    - master
  environment:
    name: production
    url: https://prospector.ie
```

Much like the review stages, these stages are grouped as they both perform similar functions. `deploy_canary` updates the deployment version of the application and, alongside the job specification, creates a "canary" application. A canary application runs using production data as opposed to the sanitised/clean version of data that a review application uses from tests and sample deployments of LDAP etc. This allows us to validate that our application behaves as expected without risking the corruption of all production traffic.

This canary application is served at `https://canary.prospector.ie`, alongside the current production version deployed at `https://prospector.ie`. Once the version has been validated, it can be promoted to production. This removes the `canary` from the proxy URL in the deployment configuration and kills the old container version, ensuring that service is never interrupted.

Ansible

Ansible is an open source IT automation engine that automates provisioning, configuration management, application deployment, orchestration, and many other IT processes. It is used industry wide to automate operations on all kinds of infrastructure.

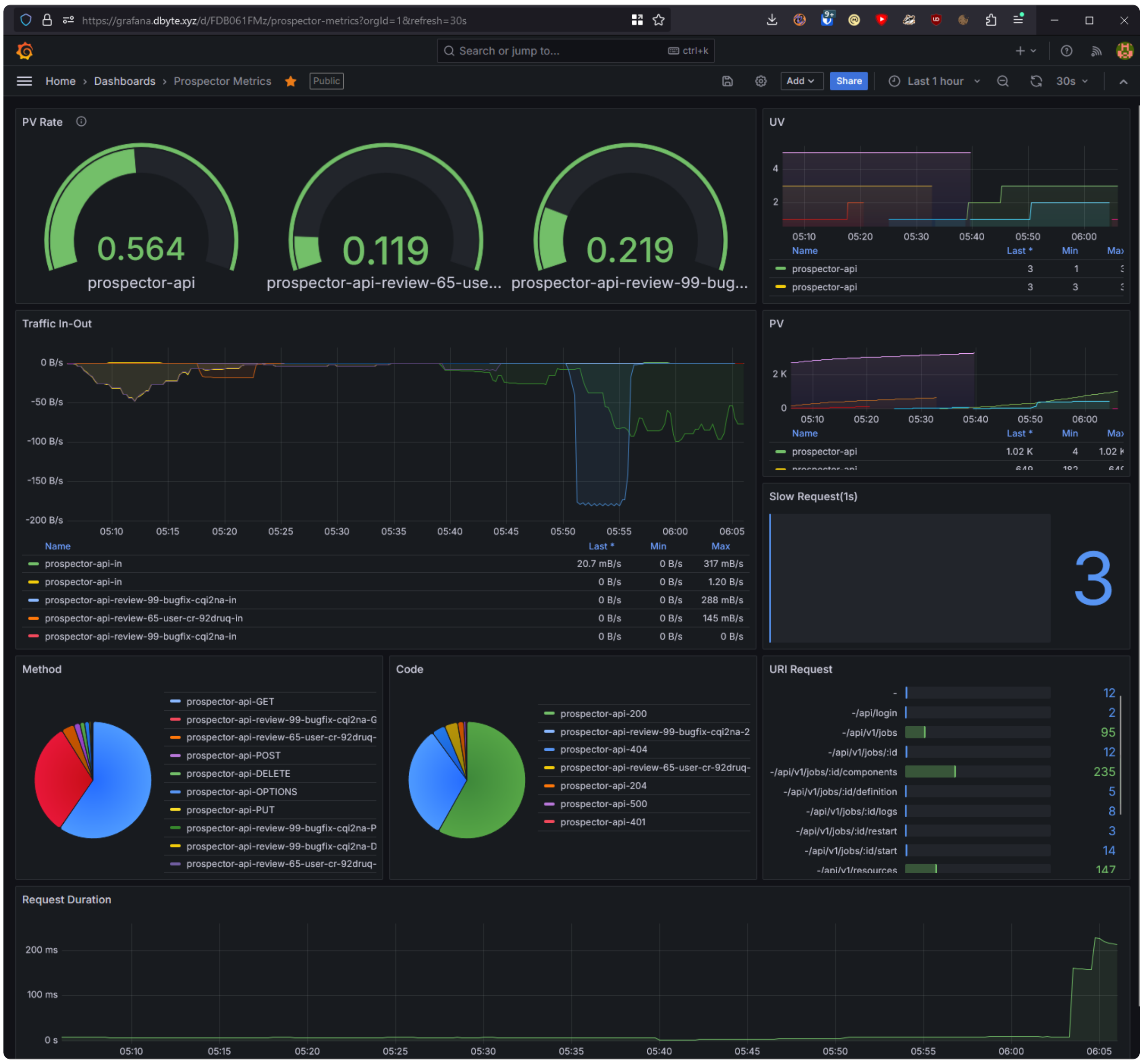
Ansible was chosen over something like Puppet or Chef due to it's push-style architecture - the client connects to each host and performs actions, only depending on Python to do so. This differs from other tools' approach which requires a "controller" server to control and serve the configurations to hosts, which grab the configs and apply them periodically.

For Prospector, Ansible is used to provision new hosts that can join the cluster as clients. The idea behind this is to make scaling the available resources as easy as possible. This follows our goal of reducing the time that administrators spend provisioning deployments for users.

We won't include the Ansible configuration files here, there's too many and they're too long to be of any use. You can view them in the `src/ansible` directory on the Gitlab repo.

Metrics

We created a Grafana dashboard for the backend, this displays metrics from the last hour on all deployments of Prospector to the public. For logged in users, the time range can be expanded. Incorporating this was very useful to see what endpoints were slow, what our error rate was and how much traffic we were sending and receiving.


https://grafana.dbyte.xyz/d/FDB061FMz/prospector-metrics?orgId=1&refresh=30s
Search or jump to...
ctrl+k
Home › Dashboards › Prospector Metrics
Public
Add
Share
Last 1 hour
30s
PV Rate
0.564
prospector-api
0.119
prospector-api-review-65-use...
0.219
prospector-api-review-99-bug...
UV
PV
Traffic In-Out
Slow Request(1s)
3
Method
Code
URI Request
Request Duration

UV

Name	Last *	Min	Max
prospector-api	3	1	3
prospector-api	3	3	3

PV

Name	Last *	Min	Max
prospector-api	1.02 K	4	1.02 K

Traffic In-Out

Name	Last *	Min	Max
prospector-api-in	20.7 mB/s	0 B/s	317 mB/s
prospector-api-in	0 B/s	0 B/s	1.20 B/s
prospector-api-review-99-bugfix-cqi2na-in	0 B/s	0 B/s	288 mB/s
prospector-api-review-65-user-cr-92druq-in	0 B/s	0 B/s	145 mB/s
prospector-api-review-99-bugfix-cqi2na-in	0 B/s	0 B/s	0 B/s

Method

- prospector-api-GET
- prospector-api-review-99-bugfix-cqi2na-G
- prospector-api-review-65-user-cr-92druq-
- prospector-api-POST
- prospector-api-DELETE
- prospector-api-OPTIONS
- prospector-api-PUT
- prospector-api-review-99-bugfix-cqi2na-P
- prospector-api-review-99-bugfix-cqi2na-D
- prospector-api-review-65-user-cr-92druq-

Code

- prospector-api-200
- prospector-api-review-99-bugfix-cqi2na-2
- prospector-api-404
- prospector-api-review-65-user-cr-92druq-
- prospector-api-204
- prospector-api-500
- prospector-api-401

URI Request

-	12
-/api/login	2
-/api/v1/jobs	95
-/api/v1/jobs/:id	12
-/api/v1/jobs/:id/components	235
-/api/v1/jobs/:id/definition	5
-/api/v1/jobs/:id/logs	8
-/api/v1/jobs/:id/restart	3
-/api/v1/jobs/:id/start	14
-/api/v1/resources	147

Should this application be deployed with SLAs (service level agreements), these metrics can help inform the 4 golden signals and ensure that agreements are being met.

Problems Solved

Frontend Routing

There was a period where Alexandru Dorofte would constantly forget to change the api url, from local to production, upon pushing his code to gitlab. This happened constantly that there had to be something done.

```
export const environment = {
    production: false,
    apiUrl: "http://localhost:3434/api",
};
```

```
export const environment = {
    production: true,
    apiUrl: "/api",
};
```

So we implemented environment variables where if the build was run in a production environment it would run using a different url, same as running locally.

Networking

Container Networking

A key feature of this project was offering users the ability to expose their containers to the internet. Traefik does the work here regarding the routing, but informing Traefik what traffic to route where would be the challenge. Traefik might advertise itself as a reverse proxy for containers, but it really only does http traffic. It struggles with anything more than that. We decided to restrict the methods of access to just HTTP, but include ways for users to route between containers. This would allow users to connect to databases with a frontend application or similar. We felt this was a good compromise.

Virtual Machine Networking

The challenges involved with configuring and running virtual machines is part of the reason that cloud providers became so popular. We aimed to replicate their ability to provision virtual machines on the fly through code, and I believe we achieved that, however some of the major issues we faced were around connecting the VMs to the network. After days of trial and error, we were finally able to ssh to our VM with credentials provided via `cloud-init`. This took a combination of host operating system configuration and QEMU argument tuning.

```
distro@desktop:~$ ssh distro@192.168.1.106
Linux distro-vm 6.1.0-20-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.85-1 (2024-04-11) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Nov 27 12:15:05 2023 from 192.168.1.10
distro@distro-vm:~$ echo "success!"
success!
```

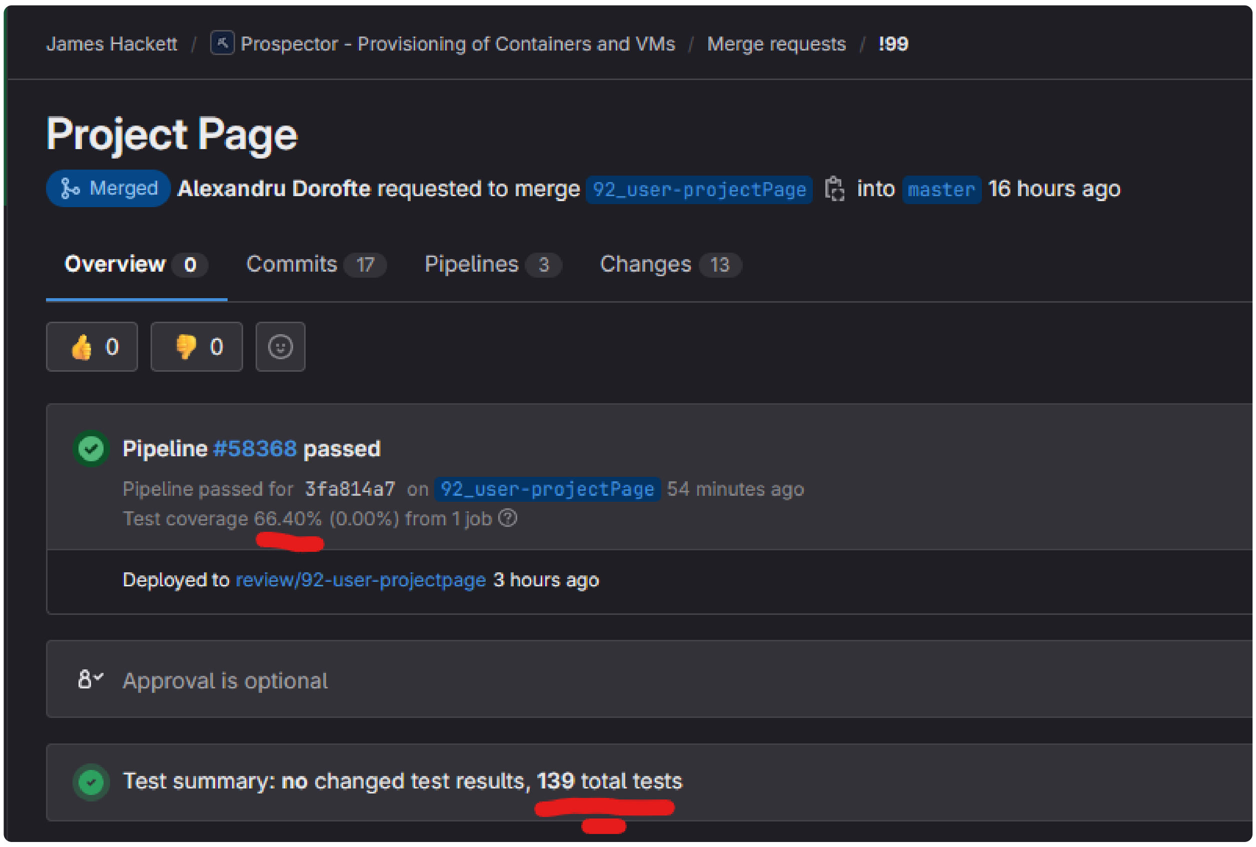
We are able to bind these containers to any network that a host is on, which means that we can bind these virtual machines directly to public IPv4 addresses, providing a virtual machine for anyone on the internet that is signed up, making private clouds a little easier to deploy.

GitLab Quirks

As mentioned, we used GitLab's platform to develop, test and manage our deployment scripts for Prospector. We found that setting it up was relatively easy, as it was quite well documented, however some features weren't quite as documented.

We used "child pipelines" in our pipeline definition to seperate concerns out between the frontend and the backend, keeping shared stages in the root of the repo. This worked great for making changes to the respective pipeline definitions, but came back to haunt us when we wanted to display testing and coverage information on merge requests, we ran into a bug with how GitLab handles artifacts from children.

Options to solve this problem were to created a shared remote S3 compatible storage somewhere something we didn't want to do because everything up to this stage was local, making it pretty fast. We could make the reports artifacts of the child pipelines and pull the information back down in a later stage. Neither of these seemed like great options, adding extra complexity to a simple testing report. We solved it in the end by creating a keyed cache that would be mounted onto the self hosted runner, adding the testing information to it from the child, then once the child had completed, we could access the information in the parent pipeline and display it as normal on the merge request!


James Hackett / Prospector - Provisioning of Containers and VMs / Merge requests / !99

Project Page

Merged Alexandru Dorofte requested to merge 92_user-projectPage into master 16 hours ago

Overview 0 Commits 17 Pipelines 3 Changes 13

Pipeline #58368 passed

Pipeline passed for 3fa814a7 on 92_user-projectPage 54 minutes ago

Test coverage 66.40% (0.00%) from 1 job

Deployed to review/92-user-projectpage 3 hours ago

Approval is optional

Test summary: no changed test results, 139 total tests

Future Work

While we feel the current implementation of this container orchestration and virtual machine manager platform, Prospector, provides a solid foundation for developers and system administrators alike, there are several areas that can be improved upon in future iterations.

Scaling container and virtual machine workloads is no easy task, every time another container is added, things like routing get just a little slower. Managing the performance and scalability of Prospector should help mitigate this slightly. Things like adding a cache between our application and the task scheduler, Nomad, will reduce workload on the clients that manage these projects. Horizontally scaling Prospector can help with this, but at some point, slow code is really just slow code.

The current observability into the application is okay, but it can always be improved. For example, we currently have metrics in Prometheus and displayed in Grafana for error codes and we have metrics for endpoints, but developers and system administrators of Prospector could benefit from mapping error codes to endpoints to provide more insight into bad error handling or problematic endpoints. Improved logging and collection through an Elastic, Kibana and Logstash (ELK) cluster would also greatly improve the observability of the application.

The benefit of cloud-based tools is that you can create and destroy remote environments relatively easily. This is especially useful for CI/CD workflows. Integrating Prospector with Jenkins, Github/GitLab Actions would make this platform very viable for developers that want a runner but don't want to or know how to set one up themselves.

Further integration with a global cloud provider like AWS, GCP or Azure might also provide value. Developers using Prospector's frontend or CLI with the backing of a cloud provider would free up more time for developers and system administrators alike.

Finally, ongoing security and validation improvements for every aspect of the application will be vital in keeping users' containers and virtual machines protected from exploits and bad actors. Integrating a method for vulnerability scanning both container and virtual machine images, improving access controls and logging around access, and keeping up with the best standards for security are all elements that will safeguard and protect Prospector and its users.

By addressing these areas of future work, Prospector can continue to evolve and meet the needs of developers and system administrators, providing them with a robust and secure container and virtual machine management platform.