

Lab 5: Industry (Optional)

Spring 2022

Contents

1	Introduction	2
1.1	Industry Lab Overview	2
1.2	Key Concepts	2
1.2.1	Cloud Computing	2
1.2.2	Containers (Docker)	2
1.2.3	Microservice Architecture	3
2	Prerequisites	3
3	Getting Started with Google Cloud	3
3.1	Create an Account	3
3.2	Interactions with GCP	4
4	Bruno Chat	4
4.1	Architecture	4
4.1.1	auth	4
4.1.2	chat	5
4.1.3	frontend	5
4.1.4	Ingress and cert-manager	5
4.2	Development Workflow	5
5	Your Tasks	6
5.1	Create a Project and spin up a Kubernetes Cluster (GKE)	6
5.2	Configure Google OAuth	6
5.3	Containerize Services and Upload to GCR	6
5.4	Deploy BrunoChat to GKE	7
5.5	Install ingress-nginx	7
5.6	Associate a Domain with your BrunoChat Instance	7
5.7	Expose Services with Nginx Ingress	8
5.8	Setup cert-manager for HTTPS	8
5.9	Observe Service Diagram with Weave Scope	9
6	Grading	10

1 Introduction

1.1 Industry Lab Overview

As you are probably aware, modern tech giants like Google, Meta, Netflix, etc., rely on distributed systems to keep their services highly available, scalable, and fault-tolerant. From class, you have learned a lot of the fundamentals of building these giant service infrastructures. However, going into the industry as a software developer, you will find most people rarely re-implement these concepts and algorithms. Rather, they build on existing tools and services.

This lab will introduce you to the industry experience of working with distributed systems. You will use many of the widely adopted tools to create a simple chat service: Bruno Chat.

Industry moves fast, especially when it comes to hot areas like cloud computing and microservices. One of the unique challenges of creating a lab like this is that many of the APIs used become deprecated over time. As a result, line-to-line commands may soon become obsolete. Instead, we state the project requirements, point you towards the relevant "Getting Started" guides, and expect you to figure out the rest. Hyperlinks noted **[required]** are considered part of the lab handout, so we expect you to read it.

Additionally, in industry, the community is your friend. As such, **we require ALL industry lab-related questions to be public on Edstem** (since others may encounter the same issue) and seek and provide help to others on Slack channel **#industry-lab**.

WARNING: There are a lot of moving parts involved in creating an application with a microservice architecture. You will likely not find any concepts hard, but you will absolutely face weird errors from Docker, Google Cloud, Kubernetes, etc. Start early, seek help, and plan your time accordingly.

1.2 Key Concepts

1.2.1 Cloud Computing

Cloud computing is a delivery model for computing resources in which various servers, applications, data, and other resources are integrated and provided as a service over the internet. While somewhat pricey, these services make it easy to create applications that perform and scale well, unlike websites that some universities use to allocate tickets for on-campus music festivals. Two videos from [AWS \[Required\]](#) and [Google Cloud](#) provide a nice high-level overview of cloud computing and why you should use it.

1.2.2 Containers (Docker)

Before getting more familiar with Docker and Kubernetes, let's glance at the definition of Containers. Containers are a form of operating system virtualization that provides OS-level virtualization, private namespace, network interface, IP address, etc. A significant difference with VMs is that containers share the host system's kernel with other containers. You may wonder why containers are so popular now? It turns out containers are a solution to the problem of how to get the software to run reliably when moved from one computing environment to another. And put simply, a container solves the problem by consisting of an entire runtime environment: an application, all dependencies, libraries, other binaries, and configuration files needed to run it, bundled into one package. Or using the most famous quote of containers here, "build once, run anywhere."

As we mentioned above, two examples of container applications are **Docker** and **Kubernetes**. [Docker](#) is a suite of software development tools for developing, creating, sharing, and running individual containers. [Kubernetes](#) is a system for operating containerized applications at scale and automating deployment.

1.2.3 Microservice Architecture

Back in the stone ages of the internet when PHP ruled the web, companies used to build *monoliths*. A monolith refers to a type of repository in which all application-level logic is written in one place, in one language. Different components spoke to each other in the language of function calls (and the occasional database call).

But what if one person made some modifications and just wanted to re-deploy their changes? Well, too bad for them! The entire backend application was one behemoth, so *everything* would have to be redeployed. And what if you had some highly opinionated (but objectively wrong) folks who thought Java was the best language, but other (objectively right) folks thought that that award belonged to Go? Some people would go home to their families each day absolutely devastated since they would spent yet another day writing in a language they hated.

What's the solution? Microservices. "Microservice architecture" refers to the pattern of having many logically separated application services that can be written in their own language, deployed independently, and scaled up and down independently, for example. For example, BrunoChat has an auth service (handles login), a chat service (handles the individual messages you send), a MongoDB service (handles all database reads/writes), and a few more. While this approach allows teams and individuals to work according to their own schedules and preferences, coordinating all these pieces gets difficult. Kubernetes, a container orchestration platform, makes the coordination between microservices manageable.

2 Prerequisites

Here is the link to the [Github Classroom assignment](#). This assignment can be done individually or with your project partner ¹.

Before you start this lab, make sure to join the Slack channel [#industry-lab](#).

3 Getting Started with Google Cloud

For this lab, we selected Google Cloud as our cloud provider. You will also hear people refer to them as the Google Cloud Platform (GCP). GCP provides state-of-the-art cloud service infrastructure that also powers the other Google services that we all love, such as Search, Gmail, Google Drive, etc.

Another major benefit of GCP is cost-efficiency. GCP provides a 90-day, [\\$300 free trial to all new customers](#). This policy is especially useful when you are just learning about the cloud, but not running a long-standing service. \$300 is more than sufficient for this lab. However, you should be mindful of your usage, since cloud services get expansive fast. GCP provides a [pricing calculator](#) to estimate your billing.

3.1 Create an Account

Despite our Brown email also qualifies for the \$300 free trial, we recommend creating a new Google account for this lab. You can create a new Google account with [this](#) link.

After you have your new Google Account, apply for the \$300 dollar free credit [here](#). Double-check if you are logged in as your newly created account. You could select the "best describe" question to be "Personal Project" or "Class project/assignment", and the account type to be "Individual".

¹Download your assignment from Github Classroom. Don't clone our stencil repo. You have to work on the repo created by GitHub Classroom when you accept the homework. You will have to push your code to your own repo to get grades.

You will need to leave a credit card here, but fortunately, they won't charge you after the free trial ends.²

3.2 Interactions with GCP

[Google Cloud overview \[Required\]](#) provides a nice high-level overview of Google Cloud, as well as explaining some terms used latter in the handout. All sections are important, but you should pay closer attention to section [projects](#) and [ways to interact with the services](#). In this lab, we will use Cloud Console and `gcloud` (Command-line interface).

4 Bruno Chat

[Bruno Chat](#) is a simple web chatroom application designed to be performant, scalable, and secure³. It resembles practices widely used in the industry. That being said, Bruno Chat was only designed to be a sandbox playground. Do not deploy this for any production use cases.

4.1 Architecture

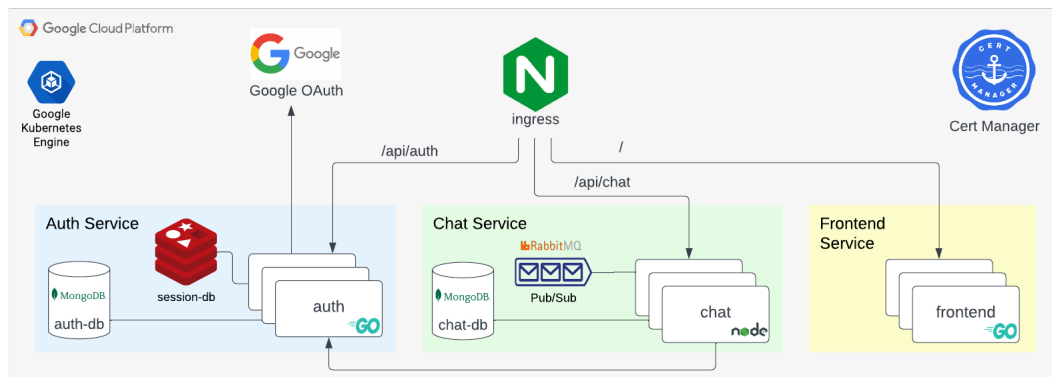


Figure 1: Bruno Chat Service Architecture

Figure 1 shows the architectural diagram for Bruno Chat. Bruno Chat mainly contains three services: auth, chat, and frontend.

4.1.1 auth

The `auth` service handles user authentication and profile. On user login, `auth` utilizes [Google's OAuth 2.0](#) authorization service to authenticate the current user. When the user first authenticates, it generates a unique session token, stores it in [Redis](#) (a high performance key value store), and sends it back to the user with the Set-Cookie HTTP header. We chose to store session tokens in Redis rather than MongoDB, since it is faster, and auth requests happen much more frequently than profile requests.

²GCP has never charged me for the many times I have used their free trial, but this is not the case with AWS. AWS will charge you, and it gets really expensive fast. However, if you do run into an unfortunate case where a cloud provider charges you on services you forgot to shut down, contact customer support and they may refund you.)

³The "secure" here is in the context of educational purposes. We designed Bruno Chat to showcase some of the best practices used in the industry. As you may know, security is only as good as the weakest link. We did not hammer every detail to make Bruno Chat industry-level secure.

The **auth** service also serves the purpose of serving user profiles. **auth** stores and reads and returns user profile information provided by Google to **auth-db**, (powered by **MongoDB**).

4.1.2 chat

The **chat** service provides chatroom functionality for Bruno Chat. **chat** uses WebSocket to achieve real-time communication with the client. All chat requests have the user's session cookie, and **chat** validates the session token with **auth**. All chat messages are stored in **chat-db** (also powered by MongoDB) so that all logged-in users can view historic messages in the chatroom. Lastly, to enable horizontal scaling capabilities, a microservice dedicated to receiving messages from one chat instance and relaying them to all the other chat instance is added. For us, this microservice is **RabbitMQ**, which is aptly called a *message broker*.

4.1.3 frontend

frontend is a just simple Golang service serving static frontend files (that were compiled from a React app, which you can find in the client directory). Ideally, we'd use something optimized for serving static files (like NGINX), but that's a TODO for us TAs :).

4.1.4 Ingress and cert-manager

An **ingress** is an API object that specifies the *rules* for external access to the services in a cluster, typically HTTP. The ingress contains rules to expose HTTP and HTTPS routes from outside the cluster to services within the cluster. This isn't enough, however. In addition to the ingress resource, we also need a **ingress controller** to implement the Ingress. An Ingress controller is responsible for fulfilling the Ingress, usually with a load balancer to help. There are many **available options**. In this lab, we use **ingress-nginx**.

To secure our traffic, we send all our traffic through HTTPS. To do that, we need an TLS certificate. Luckily, **cert-manager** makes it easy for us to obtain, renew, and use an TLS certificate from sources like **Let's Encrypt**.

4.2 Development Workflow

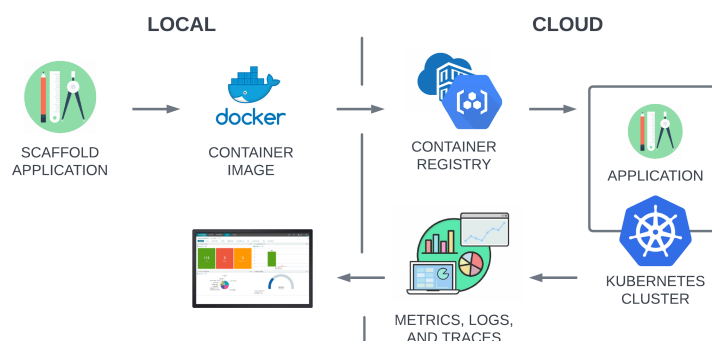


Figure 2: Docker and Kubernetes Workflow

Textually explained, here's what's going on:

1. Locally, you package your application code using Dockerfiles, which create container images. A container image is a file that bakes in all the dependencies needed to run an application. If you have a Go-based application, a container for the Go application will contain a Linux and Go runtime.

2. Once you create container images, you push these to a container registry, which is essentially Google Drive for container images. We use GCR, Google Container Registry. You can store container images on GCR, see the revision history of an image, download images, etc.
3. When you run services on Kubernetes, you tell Kubernetes how to do so by giving it a URL to a container image (hosted, of course, on a container registry like GCR). Kubernetes then downloads the image and runs it, according to some configuration that you provide (the TA staff has done most of this for you). Container images run in container runtimes, which for us, is Docker.
4. Once a service is running (i.e. a container image is run, therefore yielding a container), you're able to see metrics, logs, and traces from your application. Yay!

5 Your Tasks

5.1 Create a Project and spin up a Kubernetes Cluster (GKE)

1. Create a project named "BrunoChat"
2. Spin up a 5-node standard GKE cluster. You should stick to the default settings, except changing the number of nodes to 5.
3. Connect to your cluster with `kubectl`. You could either connect it from [cloud shell \[required\]](#) or [locally](#).
4. Create [namespace](#) `bruno-chat` by `kubectl apply -f deploy/manifests/1-namespace.yaml`.⁴

5.2 Configure Google OAuth

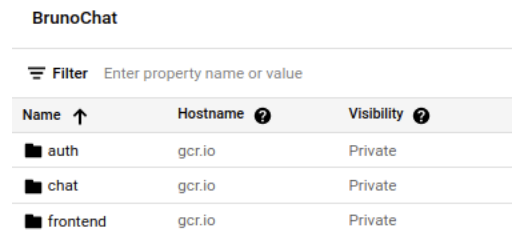
1. In order for users to be able to use Google sign-in for BrunoChat, we need to implement OAuth. To do this, follow Section [Create authorization credentials \[required\]](#) to obtain your client ID and client secret.
2. You can now think of a desired "subdomain" under `brunochat.dev`, and fill `https://<subdomain>.brunochat.dev/api/auth/callback` in field "Authorized redirect URIs". e.g. `https://neil.brunochat.dev/api/auth/callback`. We will associate your domain with your service IP later in Step 5.6.
3. Fill in the same "callback" URL to `OAuthRedirectURL` in `auth/main.go`.
4. Once you get your credentials, you should save these them as Kubernetes [secret](#). We have provided a template for k8s secret in `deploy/secret/oauth.yaml`. **Do NOT push your credentials to GitHub.** They are called "secrets" for a reason :)
5. Apply the secret to your cluster with `kubectl -f deploy/secret/oauth.yaml`

5.3 Containerize Services and Upload to GCR

1. We use [docker](#) as our container platform. To build an image, docker need a series of "instructions" defined in what's called a `Dockerfile`. Follow [this \[required\]](#) tutorial to create a `Dockerfile` for the `auth` service. HINT: Don't forget to expose port 8080. You don't have to worry about Redis and MongoDB mentioned above, since we will use the public images on docker hub [here](#) and [here](#).

⁴Namespaces are generally good practices because it separates resources within the Kubernetes cluster. Not only does it creates better team isolations within a company, but it also gives us a clear view of our services v.s. `nginx-ingress` and `cert-manager` in a smaller cluster like ours.

2. You can now also create images for `chat` and `frontend`. Don't worry, we have `Dockerfiles` created for you this time.
3. Now it's time to push our images on GCR (gcr.io). Follow [this \[required\]](#) tutorial to push all three of your images to GCR. You are free to name your container images to anything, but we recommend "chat", "auth", and "frontend". (You will need to make changes to YAML files if you decide to name your container images differently.)
4. You should be able to see something like Figure 3 in your [GCR console page](#):



Filter Enter property name or value		
Name ↑	Hostname ?	Visibility ?
auth	gcr.io	Private
chat	gcr.io	Private
frontend	gcr.io	Private

Figure 3: GCR Screenshot with 3 images uploaded

5.4 Deploy BrunoChat to GKE

1. Now it's time to deploy our services to GKE! As you may have seen in step 1 and 2, kubernetes use YAML files to define and manage all resources (namespace, deployment, service, secret, etc...). These files are often called the [Kubernetes Manifest](#). Your task is to fill in `deploy/manifests/6-chat.yaml` and `deploy/manifests/7-chat-db.yaml`. HINT: checkout the manifest files for `frontend` and `auth-db`.
2. If your GCR image tag does not start with `gcr.io/brunochat/` or you name your images differently, you will need to replace the "image" field of `6-chat.yaml` and `8-frontend.yaml` to the correct image tag.
3. Then you can apply manifest 2-8 using command `kubectl apply -f <filename>`.
4. Congratulation! You have now successfully deployed Bruno Chat. Check all deployments are running OK under [Workload](#) tab and all services are OK under Services & Ingress tab.

5.5 Install ingress-nginx

Though all our microservices are up and running, we still can't access them because they are isolated within the cluster. We need an Ingress Controller to bridge our services to the outside Internet. As explained earlier, we use [ingress-nginx](#).

1. You first need to install `ingress-nginx` following [this \[required\]](#) tutorial.
2. After a successful installation, you should be able to get your external ip with command:

```
kubectl get services -n ingress-nginx
```

5.6 Associate a Domain with your BrunoChat Instance

If you want users to easily access your chat service, you need to associate it with a domain. To do this, you would buy a domain from somewhere like GoDaddy or Google Domains, who would then allow you to associate, through DNS, an IP address with your domain. The IP address would be what GCP assigned to your Kubernetes Ingress controller.

However, since we don't want to burden you with buying a domain, we'll be providing you with *subdomains* under `brunochat.dev`. Fill out [this form](#) with your desired subdomain (from Step 5.1) and IP address of your Ingress controller. We'll send you an email when we've configured the DNS.

5.7 Expose Services with Nginx Ingress

Now we have almost everything setup! We just need to configure `ingress-nginx` to correctly route our traffic. We've provided `deploy/manifests/9-ingress.yaml` as a template, and you will need to fill in `rules:`. The following table shows our path to service mapping:

Path	Service Name	Service Port
/	frontend	8080
/api/chat	chat	8080
/api/auth	auth	8080

HINT: [This \[required\]](#) example from `ingress-nginx` provides a good reference. Note the `apiVersion` we are using in the template. Also don't forget to apply the ingress to our k8s cluster.

You should now be able to visit Bruno Chat with your custom domain. Chrome will likely complain about self-signed TLS certificate, but you can bypass it by typing `thisisunsafe`.

5.8 Setup cert-manager for HTTPS

Since setting up HTTPS is often complex and painful, we will provide line-to-line instructions here just to save you the trouble.

1. We need to first install `cert-manager`. `cert-manager` works with Kubernetes to request a certificate and respond to the challenge to validate it.

```
kubectl apply -f https://tinyurl.com/2p834bkm
```

2. We will then set up two issuers for `Let's Encrypt`. The Let's Encrypt production issuer has [very strict rate limits](#). When you are experimenting and learning, it is very easy to hit those limits, and confuse rate limiting with errors in configuration or operation. Because of this, we will start with the Let's Encrypt staging issuer, and once that is working switch to a production issuer. We can create and deploy both staging and product issuers with the two provided YAML files. ⁵

```
kubectl apply -f deploy/cert
```

Check on the status of the issuer after you create it. You should be able to see both issuers

```
kubectl describe issuer --namespace=bruno-chat
```

3. With all the prerequisite configuration in place, we can now do the pieces to request the TLS certificate. We will add annotations to the ingress, and take advantage of `ingress-shim` to have it create the certificate resource on our behalf. After creating a certificate, the `cert-manager` will update or create a ingress resource and use that to validate the domain. Once verified and issued, `cert-manager` will create or update the secret defined in the certificate. Edit `deploy/manifests/9-ingress.yaml` add the annotations that were commented out in the previous step. `Cert-manager` will read these annotations and use them to create a certificate, which you can request and see:

```
> kubectl get certificate --namespace=bruno-chat
NAME          READY    SECRET          AGE
brunochat-tls True     brunochat-tls   6s
```

⁵These files (very unfortunately) contain my personal email. These are necessary for domain verification purposes. Plz do not spam me :(

4. Once the READY state of certificate is True, you could verify the certificate by going to your Bruno Chat, and check the certificate with the little 'lock' icon in the chrome address bar. You might need to hard reset the page using Shift + Command (Ctrl) + R. Don't worry if it still shows invalid certificate. This will go away once we use the prod issuer. Check the certificate is now issued by Let's Encrypt.
5. Now that we have confidence that everything is configured correctly, you can update the annotations in the ingress to specify the production issuer:

```
cert-manager.io/issuer: "letsencrypt-prod"
```

, and apply the ingress manifest.

6. You will also need to delete the existing secret, which cert-manager is watching and will cause it to reprocess the request with the updated issuer.

```
> kubectl delete secret brunochat-tls --namespace=bruno-chat  
secret "brunochat-tls" deleted
```

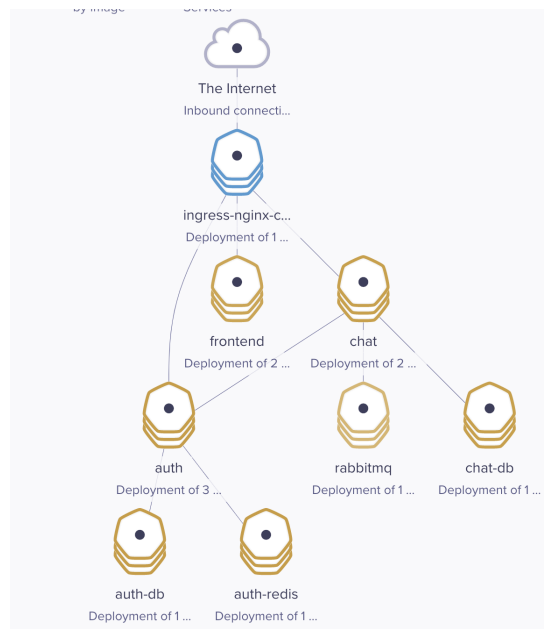
This will start the process to get a new certificate, and using describe you can see the status. Once the production certificate has been updated, you should see the Bruno Chat running at your domain with a signed TLS certificate. Yeyyyy!

5.9 Observe Service Diagram with Weave Scope

Kudos to you, your Bruno Chat is now (finally) successfully deployed! I know it's been quite a journey. Now we can use an open-source tool called **Weave Scope** to observe the network topological structure of our service (also referred to as a service graph).

Installing Weave Scope is easy. Follow [this \[required\]](#) tutorial to set up, and open <http://localhost:4040> on your local browser. If you are using Cloud Shell, click the address on the console output (Google will forward to the port for you automatically).

On the bottom left, you could filter out services to only show namespace bruno-chat and ingress-nginx. Then play around with Bruno Chat to generate an load. You should be able to see something similar to this:



6 Grading

This lab is graded on completion, and you will get a 2% extra credit on the total course grade. We will grade you interactively in ta hours. To get full credit, you are expected to show us

1. All Deployments and Services running in the Google Kubernetes Engine Console.
2. Accessing Bruno Chat using your custom domain with HTTPS and a valid TLS certificate.
3. Your service graph generated by Weave Scope.
4. One potential improvement you could make to Bruno Chat.

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out [the anonymous feedback form](#).