Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

Lecture 14: May 19, 2025

Outline

1. Frameworks and Tools
2. Deployment to a Single Node
3. Deployment to Multiple Nodes and Cloud

1 Frameworks and Tools
- Angular
- Vite

2 Deployment to a Single Node

3 Deployment to Multiple Nodes and Cloud

Angular

Angular is a full end-to-end web application framework created by Google. It was first released in 2010.

Angular (contd.)

Angular can be used to develop very complex single-page applications that:

1. have an easy-to-use (ha!), end-to-end toolchain
2. use components and services as modular building blocks
3. provide easy accessibility (a11y), internationalization (i18n), and localization (i10n).

Angular (contd.)

We can create an Angular project with:

```
ng new project-name
```

This is called a **workspace** and it comes with skeleton code. **Angular v2** was one of the first frameworks to include a full suite of CLI tools to create and scaffold projects.

Angular (contd.)

Note that React also provided CLI tools to create projects, but the native commands are deprecated.

Something interesting (and sort of beautiful) is that packages like Next.js and Vite allow creating React projects within their own CLI tools.

```
npm create vite@latest my-app --template react
npx create-next-app@latest
```

Angular: TypeScript First

TypeScript is **required** for Angular.

The other frameworks we have discussed (React, Vue, Svelte) are all JavaScript first, do not require TypeScript but support it.

Thus, Angular is typesafe. That is great to have for predictability in an enterprise app.

Angular: Component-Based Architecture

Similar to React, Angular is component-based—though React arguably popularized the approach more successfully.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent {
  name: string = 'World';

  greet(): string {
    return `Hello, ${this.name}!`;
  }
}
```

In my opinion, this syntax can be difficult to understand.

Angular: Component-Based Architecture (contd.)

`hello.component.html` is the template for the component.

```
<h1>{{ greet() }}</h1>
<input [(ngModel)]="name" placeholder="Enter your name">
```

`hello.component.css` is the style for the component.

```
h1 {
  color: darkblue;
  font-family: Arial, sans-serif;
}
```

Angular: Dependency Injection

Angular has a built-in dependency injection system.

```
// logger.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class LoggerService {
  log(message: string) {
    console.log(`[LoggerService]: ${message}`);
  }
}
```

Angular: Dependency Injection (contd.)

Inject into a component:

```
// app.component.ts
import { Component } from '@angular/core';
import { LoggerService } from './logger.service';

@Component({
  selector: 'app-root',
  template: `<button (click)="sayHello()">Say Hello</button>`
})
export class AppComponent {
  constructor(private logger: LoggerService) {}

  sayHello() {
    this.logger.log('Hello from the component!');
  }
}
```

Angular: Dependency Injection (contd.)

Why dependency injection is useful in Angular:

1. It's **modular**: we do not hardcode service instantiation
2. It's **testable**: You can provide a mock or fake logger in tests.
3. It provides **centralized control**. Logging logic is reusable across many components.

Angular: Dependency Injection (contd.)

References:

- Angular Docs
- Angular Tutorial: Tour of Heroes
- Angular CLI Guide
- StackBlitz
- Angular DevTools
- Angular Architecture Deep Dive
- Angular Style Guide

Vite

Vite (pronounced "Veet", it's French) is a combination of a build tool and lightweight *development* server.

It replaces previous bundlers like Webpack. We discussed Webpack a bit last lecture.

Vite (contd.)

It consists of two main parts:

1. a dev server that provides features useful for quick development
   - TypeScript support
   - Hot module replacement
   - Framework-agnostic, but integrates well with Vue and React
   - JSX
   - CSS as a module with support for preprocessors
   - Much more.
2. a build command that bundles code to create optimized static assets

Vite (contd.)

Unlike traditional **bundlers**, Vite serves source files as native ES modules in the browser.

Dependencies are pre-bundled.

Source code is served on demand, module-by-module.

As a result, the dev server is ready in under 1 second.

Vite (contd.)

Hot module replacement (HMR) is not unique to Vite.

Only updated modules are fetched and replaced in real time.

The page itself does not reload, and state is maintained.

Vite (contd.)

Core ideas:

1. Native ES modules for development
2. Pre-bundle dependencies via `esbuild`
3. `rollup` for production
4. Minimal configuration

Vite (contd.)

Vite is not a framework, but a build tool.

- Vue 3 uses Vite via `create-vite`
- SvelteKit uses Vite under the hood
- React has an official Vite template
- Others: Lit, Preact, Vanilla

Vite (contd.)

To install Vite:

```
npm create vite@latest appname
cd appname
npm install
npm run dev
```

Vite (contd.)

DEMO

Vite (contd.)

Some references:

- **Vite Official Docs**
- **Why Vite**
- **Vite GitHub Repo**
- **Migrating from Webpack to Vite**
- **Vite + React Guide**
- Awesome Vite

1 Frameworks and Tools

2 Deployment to a Single Node

3 Deployment to Multiple Nodes and Cloud

Deployment

So far, we have focused on the skills required to build a web application from styling to providing functionality and interactivity.

But this is pointless unless we can **deploy** our application to the web.

Deployment (contd.)

Deployment can be as easy as installing a web server and copying the files to special directories on the server.

But this is very low-level, and there are tools to help us. We will discuss the following:

1. Deploying to a single server
2. Deploying to multiple processes / cloud / serverless
3. Using CI/CD to manage deployment

Deployment to a Single Server

In most cases, we will deploy to a single server. This is the best option for:

1. Small applications that do not see much traffic and do not require horizontal scaling
2. Development and testing
3. Applications that are not mission-critical

Deployment to a Single Server (contd.)

With Node.js, we have been using a very simple, lightweight server, Express to run our projects.

Using only Express is a great option for development, but not for production. Why?

- It is not very secure
- It is not very performant
- It does not support multiple processes or handle traffic well
- It does not support load balancing

Additionally, we need to ensure **fault tolerance** and our program restarts if it crashes.

Single Node Deployment: `pm2`

`pm2` (Process Manager) is a manager for Node.js applications. It provides several services:

1. Keeps apps alive – restarts them if they crash and supports watch mode
2. Makes it easy to manage multiple apps
3. Log management – aggregates `stdout`/`stderr` with log rotation and timestamping.
4. Monitoring and metrics – RAM, CPU, uptime, status
5. Deployment – zero-downtime to reload app
6. System integration

It is similar to `systemctl` or `supervisord` for Linux but for a specific purpose.

Single Node Deployment: pm2 (contd.)

```
pm2 start app.js
```

PM2 forks (or clusters) the app and runs it in the background. If it crashes, PM2 will restart it. Apps can also start with the system.

It is event-based process management not time-based. Use `cron` etc. for time-based.

Single Node Deployment: pm2 (contd.)

```
pm2 start app.js            # Start your app
pm2 list                    # Show all managed processes
pm2 logs                    # Tail all logs
pm2 stop app                # Stop by name
pm2 restart app             # Restart
pm2 reload app              # Reload
pm2 save                    # Save current state for reboot
```

Single Node Deployment: `pm2` (contd.)

PM2 watches a process that scrapes the Mammoth website every 15 minutes. If the scraper crashes, it may bring down the scheduler as well. PM2 will restart the scheduler (and the scraper) if it crashes. PM2 is used for:

- Restarting the scheduler if the scraper crashes
- Restarts processes on reboot
- Provides logging and monitoring so I can see its status

Single Node Deployment: `pm2` (contd.)

DEMO: Mammoth scraper

Deployment to a Single Server: NGINX

Our Node project will now restart on failure, including on boot.

But we are still not serving the project in production!

Deployment to a Single Server: NGINX (contd.)

NGINX is the internet's workhorse. It acts as a smart middleman between users and your app servers.

Deployment to a Single Server: NGINX (contd.)

NGINX is **not just a web server**, but it does excel in that role. NGINX is:

1. a web server
2. a reverse proxy
3. a load balancer
4. a caching server
5. a security layer and SSL/TLS termination

Deployment to a Single Server: NGINX (contd.)

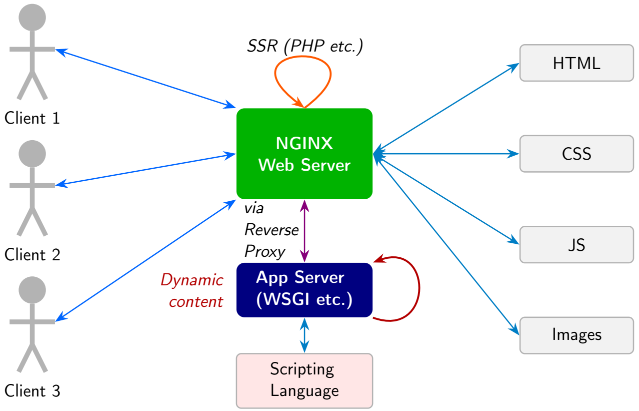
We do not use NGINX as a web server here. Express still handles that. Instead:

- **NGINX acts as a reverse proxy**
- **NGINX handles load balancing**
- NGINX handles SSL/TLS termination
- NGINX handles caching

NGINX can be used to fortify other web servers, such as Apache, Tomcat, and Express.

Deployment to a Single Server: NGINX (contd.)

Standard web server architecture for NGINX:



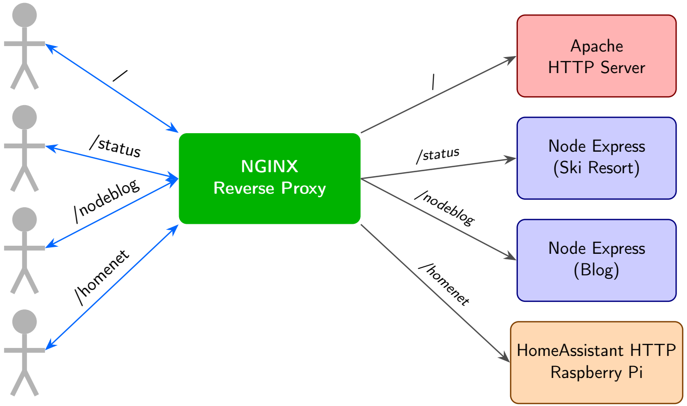
Deployment to a Single Server: NGINX (contd.)

There are three common ways NGINX handles dynamic scripting:

1. Via FastCGI (typically used for PHP) on the server
2. Via CGI modules on the server (for PHP, Python, Ruby, etc.)
3. Via an application server behind a reverse proxy (e.g., Gunicorn)

Deployment to a Single Server: NGINX (contd.)

Back to Node.js. A typical NGINX reverse proxy configuration:



```
// Create a file /etc/nginx/sites-available/chat
// When done, activate to configuration
// sudo ln -s /etc/nginx/sites-available/chat /etc/nginx/sites-enabled/chat
// Test: sudo nginx -t
// Restart NGINX: sudo systemctl restart nginx
// Start node process
// Visit site
server {
    listen 80;
    server_name yourdomain.com;

    # 1. Root path `/` → Apache server on another machine (e.g., 192.168.1.10)
    location / {
        proxy_pass http://192.168.1.10;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # 2. `/status` → Local Node.js process on port 1919
    location /status {
        proxy_pass http://127.0.0.1:1919;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    # ... another here for /nodeblog

    # 4. `/homenet` → Remote server (e.g., 10.0.0.5), but strip `/homenet` from the URL
    location /homenet/ {
        rewrite ^/homenet/(.*)$ /$1 break;
        proxy_pass http://10.0.0.5;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Deployment to a Single Server: NGINX

A worthy aside: **load balancing**. NGINX supports load balancing and a few different algorithms. Before we can load balance, we need to first run a few different Node.js Express projects.

```
pm2 start app.js --name "app-3000" --env PORT=3000
pm2 start app.js --name "app-3001" --env PORT=3001
pm2 start app.js --name "app-3002" --env PORT=3002
```

Note: these processes can also run on separate machines, but that requires updating the NGINX config with the correct IP addresses.

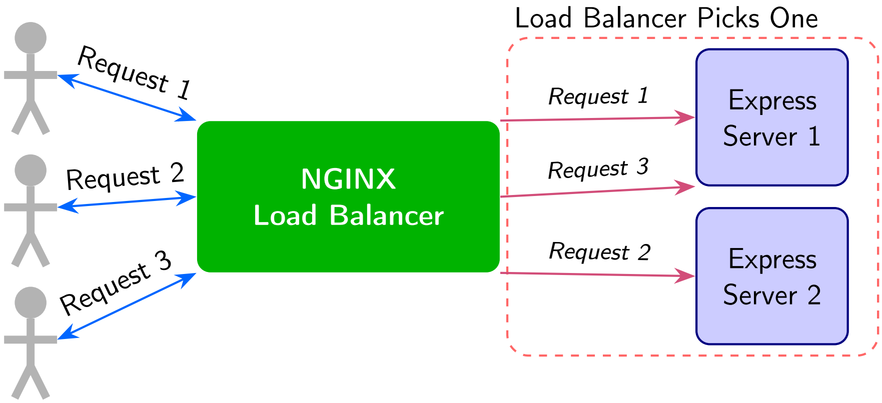
Deployment to a Single Server: NGINX (contd.)

Then in the NGINX config:

```
upstream myapp {
  # round robin is default, or specify ip_hash, least_conn
  server 127.0.0.1:3000;
  server 127.0.0.1:3001;
  server 127.0.0.1:3002;
  server 127.0.0.1:3002 backup;
}

server {
  listen 80;
  location / {
    proxy_pass http://myapp;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
  }
}
```

Deployment to a Single Server: NGINX (contd.)



Deployment to a Single Server: NGINX (contd.)

Question: How would you choose which server to forward to? There are three methods. . .

1. Method 1:
2. Method 2:
3. Method 3:

Deployment to a Single Server: NGINX (contd.)

Question: How can we use this set to load balance to other servers?

Deployment to a Single Server: NGINX (contd.)

Question: Why would we want to load balance to other load balancers?

1 Frameworks and Tools

2 Deployment to a Single Node

3 Deployment to Multiple Nodes and Cloud
- Google App Engine

Deployment to Multiple Nodes / The Cloud

There comes a time (you hope) when your web application becomes so popular that it must scale beyond a single machine.

At that point, your system architecture will typically evolve in one of two directions. . .

Deployment to Multiple Nodes / The Cloud (contd.)

Replication — the application is deployed to multiple nodes as identical or near-identical copies. Each instance handles requests independently, and the system uses load balancing (e.g., NGINX, a cloud load balancer) to distribute traffic. This is ideal for:

- Serving static assets
- Duplicating the app shell (HTML/CSS/JS bundles)
- Stateless APIs or services

Replicating the web app is easy. Replicating the data is hard. We will focus on the web app.

Deployment to Multiple Nodes / The Cloud (contd.)

Sharding — the system is divided into multiple distinct components, each handling a subset of traffic or data. This is common when:

- Different nodes serve different user groups or tenants
- Data volumes are too large for a single node
- Responsibilities are split across specialized services (e.g., auth, billing, recommendations)

Deployment to Multiple Nodes / The Cloud (contd.)

In real-world deployments, a hybrid approach is often used:

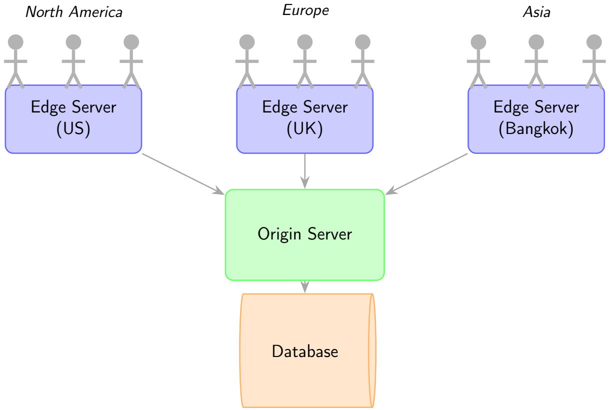
- The app shell and static assets are **replicated** across edge servers or CDNs for low-latency delivery.
- The data layer may be a mix of **replication** (for frequently-read global data) and **sharding** (for large or user-specific datasets).
- Certain services might be horizontally scaled via replication, while others are partitioned by domain responsibility (microservices).

Content Delivery Networks (CDN)

A CDN is a globally distributed network of **edge servers** that cache and deliver content to users based on their geographic location.

When the user visits your app via its domain name, the request is routed to the authoritative DNS server—let's say Cloudflare. That DNS server responds with the IP address of the edge server closest to the user, based on their IP address.

Content Delivery Networks (CDN) (contd.)



Content Delivery Networks (CDN) (contd.)

An **edge server** is simply a server that sits at the edge of the organization's network, and is thus closest to the user.

Think of it as talking with security or reception at a corporate office. You don't know (or need to know) what happens behind the scenes—they just give you the information you need.

It's much faster than having to go through the entire building or network to find what you're looking for.

Content Delivery Networks (CDN) (contd.)

Some tools:

CDN Provider	**Edge Scripting**	**Notes**
Cloudflare	`Workers`	Edge compute + static delivery
Fastly	`VCL / Compute@Edge`	Custom logic at the edge
AWS CloudFront	`Lambda@Edge`	Often used with S3 or API Gateway
Netlify / Vercel	Built-in CDN + SSR	Automatic for frontend frameworks

Table: CDN Providers and Their Edge Scripting Capabilities

Intermission: Critically Important

Your GCP credits are a finite resource. Please check your usage and billing using the following link.

`Google Cloud Console -> Billing -> Credits (left menu)`

Please get ahead of this and let us know if you are running low.

You can see how much you use a week at `Billing -> Last 7 Days`.

Intermission: Critically Important (contd.)

If you run out of credits and do not have a backup billing account (more credits), Google may delete all resources.

Thus, please immediately download any data you have on GCE you want to save.

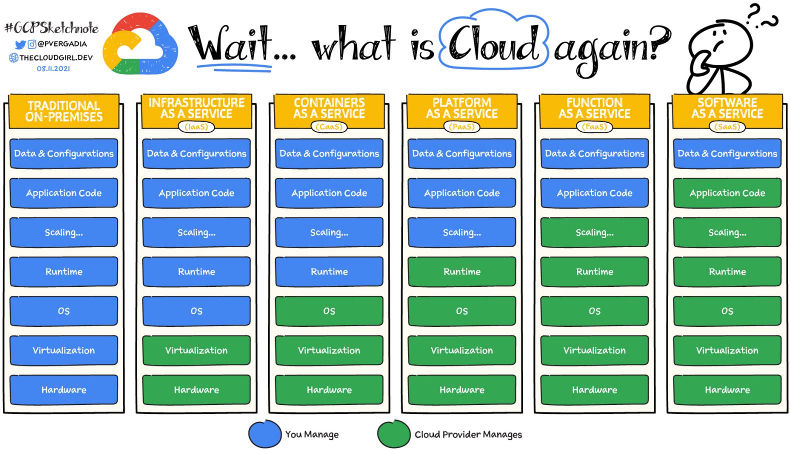
Since we are using Github to deploy code, you should not run into additional data loss.

Deploying on the Cloud

So far we have used GCE for everything. The cloud provides a lot of managed services:

- **Infrastructure as a Service (IaaS)** —- GCE, AWS EC2, Azure VMs
- **Platform as a Service (PaaS)** -— Google App Engine, AWS Elastic Beanstalk, Azure App Service
- **Containers as a Service (CaaS)** —- Google Kubernetes Engine, AWS ECS, Azure AKS
- **Software as a Service (SaaS)** —- Google Workspace, AWS WorkDocs, Azure DevOps
- **Function as a Service (FaaS)** —- Google Cloud Functions, AWS Lambda, Azure Functions

Google App Engine


#GCPSketchnote
@PVERGADIA
THECLOUDGIRL.DEV
08.11.2021
Wait... what is Cloud again?

TRADITIONAL ON-PREMISES	INFRASTRUCTURE AS A SERVICE (IaaS)	CONTAINERS AS A SERVICE (CaaS)	PLATFORM AS A SERVICE (PaaS)	FUNCTION AS A SERVICE (FaaS)	SOFTWARE AS A SERVICE (SaaS)
🟦 Data & Configurations	🟦 Data & Configurations	🟦 Data & Configurations	🟦 Data & Configurations	🟦 Data & Configurations	🟦 Data & Configurations
🟦 Application Code	🟦 Application Code	🟦 Application Code	🟦 Application Code	🟦 Application Code	🟩 Application Code
🟦 Scaling...	🟦 Scaling...	🟦 Scaling...	🟦 Scaling...	🟩 Scaling...	🟩 Scaling...
🟦 Runtime	🟦 Runtime	🟦 Runtime	🟩 Runtime	🟩 Runtime	🟩 Runtime
🟦 OS	🟦 OS	🟩 OS	🟩 OS	🟩 OS	🟩 OS
🟦 Virtualization	🟩 Virtualization	🟩 Virtualization	🟩 Virtualization	🟩 Virtualization	🟩 Virtualization
🟦 Hardware	🟩 Hardware	🟩 Hardware	🟩 Hardware	🟩 Hardware	🟩 Hardware

🟦 You Manage 🟩 Cloud Provider Manages

Google App Engine

Google App Engine (GAE) is a platform-as-a-service (PaaS) offering by Google Cloud that allows you to deploy web applications without managing the underlying infrastructure. It provides:

- automatic scaling
- load balancing
- a variety of runtime environments (Node.js, Python, etc.)

We will focus on the **standard environment** which has a free tier and is sufficient for Node.js.

Google App Engine (contd.)

What you need:

- your Google Cloud account (done!)
- the Google Cloud SDK (`gcloud`) (done!)
- a Google Cloud project (your team will need to work on this)
- create a `app.yaml` file

Google App Engine (contd.)

Using your Google Cloud `PROJECT_ID`:

```
gcloud services enable appengine.googleapis.com
gcloud app create --project=<PROJECT_ID>

# Choose us-west1
```

Google App Engine (contd.)

To deploy, we simply create a YAML file called `app.yaml` in the root of the Node project.

```
runtime: nodejs22
env: standard
instance_class: F1
automatic_scaling:
  max_instances: 1
  min_instances: 1
```

Note that your `package.json` **must** have a `start` script.

Google App Engine (contd.)

Then call `gcloud app deploy -version 1` in the root of the project. Make a note of the `target url`. That is where you will view your app. The version prevents Cloud from creating a new version at each deployment.

Google App Engine (contd.)

If you get a permissions error involving a Cloud Storage bucket:

```
gsutil iam ch \
  serviceAccount:<PROJECT_ID>@appspot \
  .gserviceaccount.com:roles/storage.objectAdmin \
  gs://staging.<PROJECT_ID>.appspot.com
```

This grants your user account the necessary permissions to deploy to GAE.

Google App Engine (contd.)

Progress looks as follows:

```
Beginning deployment of service [default]...

| Uploading X files to Google Cloud Storage                |

File upload done.
Updating service [default]...done.
Setting traffic split for service [default]...done.
Deployed service [default] to [https://<PROJECT_ID>.uw.r.appspot.com]

You can stream logs from the command line by running:
  $ gcloud app logs tail -s default

To view your application in the web browser run:
  $ gcloud app browse
```

Read the output carefully as it gives you useful commands.

Google App Engine (contd.)

When you are done testing your app, **disable it** to stop billing.

Use this link and click on Disable Application.

Note that you will continue to be billed for any other resources you use.

Google App Engine (contd.)

GAE billing is sneaky. In the free tier, you get 28 instance hours per day on an F instance. **Big caveat**.

You are charged per instance. For this class, you should never need more than one instance on GAE.

If your app is seeing consistent traffic (15 minute cadence), your instance will run continuously and you are charged for 24 instance hours. Free.

Otherwise (which is our case), your app will shut down after 15 minutes. At the next request, an instance is created **and you are billed for 15 minutes.** Isn't that convenient?

Google App Engine (contd.)

So, there is a penalty if your app does not see enough traffic in a 15 minute period.

You should only deploy the app to check that it works. Then undeploy it when you are not actively working on it.

If you exceed the free tier, you will be billed against your coupon, so all is not lost.

Google App Engine (contd.)

Things to remember:

1. Disable your GAE app when you are not actively testing it.
2. You can view your app at `https://<PROJECT_ID>.uw.r.appspot.com`.
3. Disable your GAE app when you are not actively testing it.
4. You can view your app logs at `gcloud app logs tail -s default`.
5. Disable your GAE app when you are not actively testing it.
6. You cannot use custom ports. You must use 8080.
7. Disable your GAE app when you are not actively testing it.
8. Visit https://console.cloud.google.com/appengine/versions and verify only ONE version is deployed. Delete others.