deployment # guide

Deploying a Simple Web API

TCSS 460 — Client/Server Programming

Your API works on your laptop. Now it needs to work on the internet. This guide walks you through deploying a simple Express API — no database, no authentication middleware — to a public URL using either Render or Heroku. By the end, anyone with a browser can hit your heartbeat route and get a response.

> **Your Platform, Your Responsibility**
>
> This guide documents Render and Heroku because they are common choices with student-friendly pricing. You are not required to use either one — your team may investigate and deploy to any cloud platform you wish. Regardless of which platform you choose, your group is responsible for its own deployment and troubleshooting. Your instructor will not debug platform-specific issues for you — that is part of the learning.

---

1 Why Deploy Now?

"Deploying to the cloud" means running your application on a server that someone else manages, accessible at a public URL. Instead of `localhost:3000`, your API responds at something like `https://your-app.onrender.com`. Anyone on the internet can reach it — your teammates, your instructor, the group that will eventually build a front-end against your API.

You will study cloud computing concepts — virtual machines, containers, platform abstractions, scaling — in a later reading. Right now, the goal is simpler: get a working URL. You need to prove that your team can ship code to production, and your Sprint 0 deliverable requires a deployed API. Everything else is context for later.

> **Keep It Simple**
>
> This guide covers deploying a stateless Express API — no database, no persistent storage, no environment-specific configuration beyond a port number. When your API needs a database, see the Deploying a Web API with Database guide (coming later in the quarter).

2 What You Need Before You Start

Before deploying, confirm all of the following:

- Your Express API runs locally (`npm run dev` or `npm start` starts the server without errors)
- Your heartbeat route responds (e.g., `GET /health` returns JSON)
- Your `/api-docs` page loads in the browser
- Your code is pushed to a GitHub repository — the `main` branch is up to date
- You have a free account on your chosen platform (Render or Heroku)

If any of these are not true, fix them first. Deploying broken code to the cloud does not make it work — it makes it broken somewhere harder to debug.

---

3 From `.env` to Environment Variables

If you have been using a `.env` file locally, you need to understand what it is actually doing — and why it does not follow your code to production.

3.1 What `.env` Is Actually Doing

When your app starts, the `dotenv` package reads your `.env` file and loads each key-value pair into `process.env`. That is all it does. It is a convenience for local development — a way to set environment variables without typing them into your terminal every time.

```
# .env
PORT=3000
```

```
// Somewhere in your app
const port = process.env.PORT || 3000;
```

Your code reads from `process.env`. It does not know or care whether those values came from a `.env` file, a terminal command, or a cloud platform's dashboard. `process.env` is the interface — everything else is just a way to populate it.

3.2 Why `.env` Does Not Go to Production

Your `.env` file should already be listed in `.gitignore` — the starter repo does this for you. This is intentional:

- **It contains secrets.** API keys, database credentials, and other sensitive values do not belong in version control. Anyone with access to your repo would see them.
- **It is environment-specific.** Your local `PORT` might be `3000`, but Render assigns its own port dynamically. The values are different per environment by design.
- **Cloud platforms have a better mechanism.** Every hosting platform provides a way to set environment variables through their dashboard or CLI. These values are encrypted at rest and never appear in your source code.

The rule is simple: `.env` is for your laptop. Environment variables configured in the platform dashboard are for production.

3.3 The One Variable You Almost Always Need: `PORT`

Cloud platforms do not let you choose which port your server listens on. They assign one dynamically and pass it to your app through the `PORT` environment variable. Your app must read it:

```
const port = process.env.PORT || 3000;
app.listen(port, () => {
    console.log(`Server running on port ${port}`);
});
```

This pattern works in both environments:

Environment	Where `PORT` comes from	Typical value
Local development	Your `.env` file (or missing, so the fallback `3000` is used)	`3000`
Render / Heroku	Set automatically by the platform	Assigned at runtime (e.g., `10000`, `5432`, etc.)

If you hardcode `app.listen(3000)` instead of reading from `process.env.PORT`, your app will fail on every cloud platform. This is one of the most common first-deployment mistakes.

> **Check Your `app.listen` Call**
>
> Search your codebase for `app.listen`. If the port is a hardcoded number, change it to read from `process.env.PORT` with a fallback. The starter repo already does this — but verify.

---

4 Option A: Render

Render is a cloud platform with a free tier that deploys directly from a GitHub repository. It detects Node.js projects automatically and handles the build for you.

> **These Instructions Are a Reference, Not a Single Source of Truth**
>
> Cloud platform UIs change frequently. The steps below were accurate at the time of writing, but buttons may have moved, options may have been renamed, and workflows may have changed. Use these instructions as a general guide — if something does not match what you see, consult Render's official documentation for the current process.
>
> If you are using an AI coding assistant to help with deployment, tell it to read the platform's most recent documentation. AI models have a training data cutoff and may give you outdated instructions otherwise.

> **Render Free Tier: 30-Day Database Limit**
>
> Render's free PostgreSQL tier expires after **30 days**, with a 14-day grace period to upgrade before deletion. This matters when you think ahead to database hosting in later sprints. If you choose Render, plan for this limitation — you may need to pay for a database or use an external service like Supabase when the time comes.

4.1 Create an Account and Connect GitHub

1. Go to render.com and create a free account
2. When prompted, connect your GitHub account — Render needs access to pull your code

4.2 Create a Web Service

1. From the Render dashboard, click **New** → **Web Service**

2. Select the GitHub repository containing your API
3. Configure the service:

Setting	Value
Name	Your team's app name (e.g., `tcss460-team-5-api`)
Region	Oregon (US West) or whichever is closest
Branch	`main`
Runtime	Node
Build Command	`npm install && npm run build`
Start Command	`npm start`
Instance Type	Free

> **Build and Start Commands**
>
> The **build command** runs once during deployment — it installs dependencies and compiles your TypeScript. The **start command** runs your compiled JavaScript to start the server. These should match what you would run locally: `npm install`, then `npm run build`, then `npm start`. Check your `package.json` to verify what `build` and `start` do.

4.3 Set Environment Variables

Scroll to the **Environment Variables** section (or find it under the service's **Environment** tab after creation). Add any variables your app needs. For a simple API with no database, you may not need any — Render sets `PORT` automatically.

If your app reads other environment variables (even non-secret ones like a log level), add them here. Each entry is a key-value pair, just like a line in your `.env` file.

4.4 Deploy and Verify

Click **Create Web Service**. Render will:

1. Clone your repo

2. Run your build command
3. Start your server
4. Assign a public URL (e.g., `https://your-app.onrender.com`)

Watch the deploy logs in real time. When you see your server's startup message (e.g., `Server running on port 10000`), the deploy is complete.

Verify by hitting your endpoints:

- `https://your-app.onrender.com/health` — your heartbeat route should return JSON
- `https://your-app.onrender.com/api-docs` — Scalar should load and display your documented endpoints

> **Try It Yourself**
>
> 1. Open your deployed URL in a browser — does the heartbeat route respond?
> 2. Open `/api-docs` — does Scalar render your OpenAPI spec?
> 3. Use Scalar's built-in client to send a request to your deployed API — does it work the same as localhost?

4.5 Gotchas

Cold starts on the free tier. Render's free instances spin down after 15 minutes of inactivity. The first request after a cold start takes 30-60 seconds while the server boots back up. Subsequent requests are fast. This is expected behavior on the free tier — not a bug in your code.

Automatic redeploys. By default, Render redeploys every time you push to `main`. This is convenient but means a broken push immediately takes down your live API. Make sure code works locally before merging to `main`.

> **Protect Your `main` Branch**
>
> Consider setting up branch protection rules on your GitHub repository. Requiring pull request reviews before merging to `main` prevents accidental pushes from breaking your deployed API. Some branch protection features require a paid GitHub account — but the basic "require a pull request before merging" rule is available on free accounts for public repositories.

Build failures. If the deploy fails, read the build logs. The most common issues are:

- Missing `build` or `start` script in `package.json`

- TypeScript compilation errors that you did not catch locally
- Dependencies listed in `devDependencies` that are needed at build time (Render runs `npm install` which installs all dependencies by default, but check if you see missing module errors)

---

5 Option B: Heroku

Heroku is one of the original cloud platforms for deploying web applications. It offers credits through the GitHub Student Developer Pack, which makes it effectively free for students.

> **These Instructions Are a Reference, Not a Single Source of Truth**
>
> Cloud platform UIs change frequently. The steps below were accurate at the time of writing, but buttons may have moved, options may have been renamed, and workflows may have changed. Use these instructions as a general guide — if something does not match what you see, consult Heroku's Dev Center for the current process.
>
> If you are using an AI coding assistant to help with deployment, tell it to read the platform's most recent documentation. AI models have a training data cutoff and may give you outdated instructions otherwise.

5.1 Create an Account and Get Credits

1. Go to heroku.com and create an account
2. If you have the GitHub Student Developer Pack, apply the Heroku credits through the pack dashboard — this gives you enough credit to run a small app for the quarter

> **Heroku Pricing**
>
> Heroku no longer offers a permanent free tier. The GitHub Student Developer Pack provides **$13/month for 24 months** in platform credits. This covers both Eco dynos ($5/month) and a Heroku Postgres Essential-0 database ($5/month) with room to spare — meaning you can run your API **and** a database under the Student Pack for the entire quarter and beyond. Without Student Pack access, the cheapest Eco dynos cost $5/month and the cheapest database is an additional $5/month. If you do not have Student Pack access, Render's free tier may be a better choice for the API itself.

5.2 Create an App and Connect GitHub

1. From the Heroku dashboard, click **New** → **Create new app**
2. Choose an app name (e.g., `tcss460-team-5-api`) and region
3. Under the **Deploy** tab, select **GitHub** as the deployment method
4. Search for and connect your repository
5. Enable **Automatic Deploys** from `main` (optional but recommended)

5.3 Configure the App

Start command. Heroku looks for a `Procfile` in your project root to know how to start your app. Create one:

```
web: npm start
```

That single line tells Heroku: "this is a web process, start it with `npm start`."

> **No Procfile?**
>
> If you do not create a `Procfile`, Heroku falls back to the `start` script in your `package.json`. This usually works, but a `Procfile` makes the intent explicit and is considered best practice.

Environment variables. Go to **Settings** → **Config Vars** to add environment variables. Like Render, Heroku sets `PORT` automatically — you do not need to add it yourself. Add any other variables your app needs.

Buildpacks. Heroku uses buildpacks to detect your project type and install dependencies. It should auto-detect Node.js from your `package.json`. If it does not, go to **Settings** → **Buildpacks** and add `heroku/nodejs`.

5.4 Deploy and Verify

If you enabled automatic deploys, push to `main` and Heroku will build and deploy. Otherwise, go to the **Deploy** tab and click **Deploy Branch** under the Manual Deploy section.

Watch the build log. When it finishes, Heroku assigns a URL (e.g., `https://tcss460-team-5-api-abc123.herokuapp.com`).

Verify the same way:

- Hit your heartbeat route — you should get a JSON response

- Open `/api-docs` — Scalar should render your spec

> **Try It Yourself**
>
> 1. Open your deployed URL in a browser — does the heartbeat route respond?
> 2. Open `/api-docs` — does Scalar render your OpenAPI spec?
> 3. Use Scalar's built-in client to send a request to your deployed API — does it work the same as localhost?

5.5 Gotchas

Eco dynos sleep. Like Render's free tier, Heroku's Eco dynos sleep after 30 minutes of inactivity. The first request after sleeping takes 5-15 seconds. This is platform behavior, not a problem with your code.

Automatic redeploys. If you enabled automatic deploys from `main`, every push triggers a new build. A broken merge means a broken deployment.

> **Protect Your `main` Branch**
>
> Consider setting up branch protection rules on your GitHub repository. Requiring pull request reviews before merging to `main` prevents accidental pushes from breaking your deployed API. Some branch protection features require a paid GitHub account — but the basic "require a pull request before merging" rule is available on free accounts for public repositories.

Build command. Heroku runs `npm install` and then looks for a `build` script in your `package.json`. If your `build` script compiles TypeScript (e.g., `tsc`), it runs automatically. Make sure `typescript` is in your `dependencies` (not just `devDependencies`) or that your build step works within Heroku's install phase.

> **TypeScript in `devDependencies`**
>
> By default, Heroku prunes `devDependencies` after the build step. If `typescript` is only in `devDependencies`, `tsc` will not be available during the build. Either move `typescript` to `dependencies` or set the config var `NPM_CONFIG_PRODUCTION=false` to install all dependencies.

The `PORT` assignment. Heroku sets `PORT` as a config var internally — do not add it yourself in Config Vars. Your code just needs to read `process.env.PORT`, which it should already be doing.

6 Logs and Debugging

When your API ran on your laptop, `console.log` and `console.error` printed to your terminal. In the cloud, that terminal does not exist — but your log output still goes somewhere. Every cloud platform captures your application's standard output and makes it available through a log viewer. Knowing where to find these logs is the difference between debugging a production issue in five minutes and staring at a blank screen wondering what went wrong.

6.1 Build Logs vs. Runtime Logs

There are two kinds of logs, and they appear in different places:

Log Type	When It Runs	What It Shows	Where to Find It
Build logs	During deployment	Dependency installation, TypeScript compilation, build errors	Shown during deploy; accessible from the deploy history
Runtime logs	While your server is running	`console.log` output, request handling, uncaught errors	The platform's live log viewer

If your deploy fails, read the **build logs** — the error is almost always a compilation failure or missing dependency. If your deploy succeeds but your API behaves wrong, read the **runtime logs** — your `console.log` and `console.error` statements are there.

6.2 Render Logs

Build logs: Click on any deploy in the **Events** tab of your service to see the full build output.

Runtime logs: Click the **Logs** tab on your service page. This shows a live stream of everything your application writes to `stdout` and `stderr` — every `console.log`, every `console.error`, every uncaught exception. You can also filter by time range.

6.3 Heroku Logs

Build logs: Visible during deployment in the **Activity** tab. Click on a build to expand the log output.

Runtime logs: You have two options:

- **Dashboard:** Go to **More** → **View logs** in the top-right of your app's dashboard page
- **CLI:** If you have the Heroku CLI installed, run:

```
heroku logs --tail --app your-app-name
```

The `--tail` flag streams logs in real time, similar to watching your local terminal.

6.4 What to Look For

When something goes wrong, check these in order:

1. **Did the build succeed?** If not, read the build log — the last error message is usually the one that matters.
2. **Did the server start?** Look for your startup message (e.g., `Server running on port 10000`). If it is missing, your app crashed on startup.
3. **Is the route responding?** If the server started but a route returns an error, look for your `console.error` output in the runtime logs. The stack trace tells you what failed.

> **Gen AI & Learning: Debugging Deployments with AI**
>
> AI coding assistants can be genuinely helpful for diagnosing deployment failures — paste your build log or error message and ask what went wrong. But there is a catch: AI models have a training data cutoff, and cloud platform behavior changes frequently. A model trained on 2024 data may give you advice about Render or Heroku settings that no longer exist.
>
> When asking an AI to help with deployment, tell it to read the platform's current documentation first. Most AI coding tools can fetch web pages — point them at `docs.render.com` or `devcenter.heroku.com` and ask them to verify their suggestions against the latest docs. This one step eliminates most of the "the AI told me to click a button that doesn't exist" frustration.

---

7 Think Ahead: Database Hosting

Right now, your API is stateless — it does not store anything. In a few sprints, you will add a PostgreSQL database. When choosing a platform today, consider whether it also offers

database hosting:

Platform	Database Option	Notes
Render	Render PostgreSQL	Free tier available, but expires after 30 days
Heroku	Heroku Postgres	Essential-0 ($5/month) covered by Student Pack credits
Supabase	Managed PostgreSQL	Free tier with generous limits; works with any hosting platform

You do not need a database now. When your platform supports one, the migration later is simpler — you add a database instance, set the `DATABASE_URL` environment variable, and your API connects. If your platform does not offer a database (or the free tier expires), you will use an external service like Supabase and configure the connection string as an environment variable.

This is not a decision you need to make today. It is a reason to think for ten minutes before signing up for a platform.

---

8 Summary

Concept	Key Point
Cloud deployment	Running your app on a managed server with a public URL — not localhost
`.env` files	A local convenience for setting environment variables — never deployed, never committed
`process.env`	The universal interface — your code reads from it regardless of where values come from
`PORT`	Assigned by the platform at runtime — read it from `process.env.PORT` with a local fallback

Concept	Key Point
Platform env vars	The production replacement for `.env` — set through the dashboard or CLI, encrypted at rest
Render	Free tier, auto-deploys from GitHub, cold starts after 15 min inactivity, 30-day DB limit
Heroku	Student Pack credits ($13/mo), Procfile-based, eco dynos sleep after 30 min, DB included in credits
Build logs	Deployment output — check here when builds fail
Runtime logs	Your `console.log` output in production — check here when the app misbehaves
Database hosting	Not needed yet — but factor it into your platform choice now

---

9 References

Official Documentation:

- Render — Deploy a Node.js App — Render's official guide for Express deployment
- Render — Environment Variables — How to configure env vars in the Render dashboard
- Render — Logging — How to access build and runtime logs on Render
- Heroku — Getting Started with Node.js — Heroku's Node.js deployment guide
- Heroku — Configuration and Config Vars — How to manage environment variables on Heroku
- Heroku — Procfile — What a Procfile is and how Heroku uses it
- Heroku — Logging — How to access application logs on Heroku
- GitHub Student Developer Pack — Free developer tools for students, including Heroku credits
- GitHub — Branch Protection Rules — Protecting your `main` branch from accidental pushes

---

10 Further Reading

External Resources

- The Twelve-Factor App — Config — The industry-standard principle behind environment variable configuration
- The Twelve-Factor App — Logs — Why logs are event streams, not files
- Render vs Heroku Comparison — Render's comparison of the two platforms
- Node.js `process.env` Documentation — Official Node.js docs on environment variables

---

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.