

Lab 1 :

Auteur: Badr TAJINI - DevOps Data for SWE - ESIEE - 2025

An Introduction to Deploying Apps

As you saw in the first chapter of the course, the first step is to run your app on a single server. How do you run an app? What's a server? What type of server should you use? These are the questions we'll address in these labs.

The first place you should be able to deploy your app is locally, on your own computer. This is typically how you'd build the app in the first place, writing and running your code locally until you have something working.

How you run an app locally depends on the technology. Throughout these labs, you're going to be using a simple Node.js sample app.

Example: Run the Sample App Locally

Create a new folder on your computer, perhaps called something like *devops-base*, which you can use to store code for various examples you'll be running throughout the labs. You can run the following commands in a terminal to create the folder and go into it in your `Cygwin` terminal:

```
$ mkdir devops_base
$ cd devops_base
```

In that folder, create a new subfolder for the chapter 1 examples, and a subfolder within that for the sample app:

```
$ mkdir -p ch1/sample-app
$ cd ch1/sample-app
```

The sample app you'll be using is a minimal "Hello, World" Node.js app, written in JavaScript. You don't need to understand much JavaScript to make sense of the app. In fact, one of the nice things about getting started with Node.js is that all the code for a simple web app fits in a single file that's ~10 lines long. Within the *sample-app* folder, create a file called *app.js*, with :

```
$ nano app.js
```

Then copy and paste the script below :

```
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

const port = process.env.PORT || 8080;
server.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
```

This is a “Hello, World” app that does the following:

- [1] Respond to all requests with a 200 status code and the text “Hello, World!”
- [2] Listen for requests on the port number specified via the `PORT` environment variable, or if `PORT` is not set, default to port 8080.

To run it locally, you must first (or already done in `lab0`) [install Node.js](#). Once that’s done, you can start the app with `node app.js` :

```
$ node app.js
Listening on port 8080
```

if node is not recognized, try to source it and run these commands as we did in `lab0` :

```
# source ~/.bash_profile
node -v
# v23.2.0
npm -v
# 10.9.0
```

You can then open <http://localhost:8080> in your browser, and you should see: “Hello, World!”

Congrats, you’re running the app locally! That’s a great start, but if you want to expose your app to users, you’ll need to run it on a server, as discussed next.

Deploying an App on a --Server--

By default, when you run an app on your computer, it is only available on *localhost*. This is a hostname that is configured on every computer to point back to the *loopback network interface*, which means it bypasses any real network interface. In other words, when you're running an app on your computer locally, it typically can *only* be accessed from your own computer, and not from the outside world. This is by design, and for the most part, a good thing, as the way you run apps on a personal computer for development and testing is *not* the way you should run them when you want to expose them to outsiders.

Deploying An App Using PaaS

One of the first cloud computing offerings from AWS was Elastic Compute Cloud (EC2), which came out in 2006, and allowed you to rent servers from AWS. This was the first *Infrastructure as a Service (IaaS)* offering, which gives you access directly to computing resources, such as servers. This dramatically reduced the time it took to do the hardware aspects of operations (e.g., buying and racking servers), but it didn't help as much with the software aspects of operations, such as installing the app and its dependencies, setting up databases, configuring networking, and so on.

About a year later, in 2007, a company called Heroku came out with one of the first *Platform as a Service (PaaS)* offerings. The key difference with PaaS is that the focus is on higher level primitives: not just the underlying infrastructure (i.e., servers), but also application packaging, deployment pipelines, database management, and so on.

The difference between IaaS and PaaS isn't so black and white; it's really more of a continuum. Some PaaS offerings are higher level than Heroku; some are lower level; many IaaS providers have PaaS offerings too (e.g., AWS offers Elastic Beanstalk, GCP offers App Engine); and so on. The difference really comes down to this: PaaS gives you a full, opinionated software delivery process; IaaS gives you low-level primitives to create your own software delivery process. The best way to get a feel for the difference is to try it out. You'll try out a PaaS in this section and an IaaS in the next section.

There are many PaaS offerings out there, including [Heroku](#), [Render](#), [Fly.io](#), [Vercel](#), [Railway](#), [Firebase](#), [Supabase](#), [Platform.sh](#), [Netlify](#), [Dokku](#), [Porter](#), [Adaptable](#), [Koyeb](#), and [Digital Ocean App Platform](#). For the examples in this lab, I wanted a PaaS that met the following requirements:

- There is a free tier, so that readers can try it out at no cost.
- You can deploy application code without having to set up a build system, framework, Docker image, etc. (these are topics you'll learn about later in the course).

- It has a good reputation and track record.

Thus, we'll try out Render. It offers a [free "Hobby" tier](#), so running the examples in this lab shouldn't cost you anything; it [supports a number of languages and frameworks](#), including Node.js; and it's well-regarded by the community, often described as the spiritual successor to Heroku.

Example: deploying an app using Render

Here's what you need to do to use Render to deploy the sample app:

- **Step 1: Sign up for a Render account.** Create a new account on render.com.

If you are new to Git, check out the Git for Beginners tutorial [here](#) that you will probably need for step 2.


- **Step 2: Deploy a new web service.** Head to the [Render Dashboard](#) and click the Deploy a Web Service button. On the next page, select the Public Git Repository tab, and enter the URL of your Public Git Repository, for instance, it could be as follow <https://github.com/BTajini/devops-base>, as shown in [Figure 1-1]. This repo contains the Node.js code from [Example: Run the Sample App Locally] in the *ch1/sample-app* folder, so this lets you deploy the app without creating your own Git repo.

You are deploying a Web Service

Source Code

Git Provider	Public Git Repository	Existing Image
--------------	-----------------------	----------------

PR Previews and Auto-Deploy are available only for repositories configured with render.yaml



Connect →

Figure 1-1. Use this sample code repo to create a web service in Render

Click the Connect button, and then configure your web service in Render by filling in the form with the values shown in [Table 1-1]:

Table 1-1. The values to configure for your web service in Render

Configuration	Value
Name	sample-app
Language	Node
Root Directory	ch1/sample-app
Build Command	# (no build command)
Start Command	node app.js
Instance Type	Free

Leave the other settings (e.g., Project, Branch, Region) at their default values and click the Deploy Web Service button at the bottom of the page to start the deployment.

- **Step 3: Test your app.** After a couple of minutes, the deployment log should say something like “Your service is live”, as shown [Figure 1-2]:

WEB SERVICE

sample-app

Node

Free

[Upgrade your instance →](#)

<https://sample-app-ws.av.onrender.com>

Events

Logs

Disks

Environment

Shell

Previews

Jobs

Metrics

Scaling

Settings

All logs ▾

Q Search

```

Oct 14 05:23:03 PM  ➤ ==> Cloning from https://github.com/brikis90/devops-book
Oct 14 05:23:04 PM  ➤ ==> Checking out commit 472f43fa716cb2ed715ebe153b85cf0f75e00bb4 in branch main
Oct 14 05:23:07 PM  ➤ ==> Using Node.js version 20.15.1 (default)
Oct 14 05:23:07 PM  ➤ ==> Docs on specifying a Node.js version: https://render.com/docs/node-version
Oct 14 05:23:10 PM  ➤ ==> Using Bun version 1.1.0 (default)
Oct 14 05:23:10 PM  ➤ ==> Docs on specifying a bun version: https://render.com/docs/bun-version
Oct 14 05:23:10 PM  ➤ ==> Running build command '#'...
Oct 14 05:23:11 PM  ➤ ==> Uploading build...
Oct 14 05:23:21 PM  ➤ ==> Build uploaded in 9s
Oct 14 05:23:21 PM  ➤ ==> Build successful 🎉
Oct 14 05:23:23 PM  ➤ ==> Deploying...
Oct 14 05:24:28 PM  ➤ ==> No open ports detected, continuing to scan...
Oct 14 05:24:28 PM  ➤ ==> Docs on specifying a port: https://render.com/docs/web-services#port-binding
Oct 14 05:24:31 PM  ➤ ==> Running 'node app.js'
Oct 14 05:24:32 PM  ➤ Listening on port 10000
Oct 14 05:24:34 PM  ➤ ==> Your service is live 🎉

```

Figure 1-2. A completed deployment in Render

At the top-left of the page, you should see a randomly-generated URL for your app of the form `https://<NAME>.onrender.com`. Open this URL in your web browser, and you should see:

Congrats, in just a few steps, you now have an app running on a server!

Practice [Exercise]

Here are a few exercises you can try at home to go deeper:

- Click the Events tab to see events such as deployments.
 - Click the Logs tab to see the logs for your app.
 - Click the Metrics tab to see metrics for your app.
 - Click the Scale tab to change how many servers run your app.
-

IMPORTANT : When you're done experimenting with Render, undeploy your app by clicking the Settings tab, scrolling to the bottom, and clicking the Delete Web Service button.

Deploying an App Using IaaS

For the examples in this lab, I wanted an IaaS provider that met the following requirements:

- There is a free tier, so that readers can try it out at no cost.
- It should provide a wide range of cloud services that support the many DevOps and software delivery examples.
- It has a good reputation and track record. That way, you're learning something you're more likely to use at work.

AWS is a good fit for these criteria: it offers [a generous free tier](#), so running the examples in this lab shouldn't cost you anything; it provides a huge range of reliable and scalable cloud services, including servers, serverless, containers, databases, machine learning, and much more; and it's widely recognized as the dominant cloud provider, with a [31% share of the market](#) and [the leader in Gartner's Magic Quadrant for Strategic Cloud Platform Services for 13 years in a row](#). Let's give AWS a shot.

Example: deploying an app using AWS

Here's what you need to do to use AWS to deploy the sample app:

- **Step 1: Sign up for AWS.** If you don't already have an AWS account, head over to <https://aws.amazon.com> and sign up. When you first register for AWS, you initially sign in as the *root user*. This user account has access permissions to do absolutely anything in the account, so from a security perspective, it's not a good idea to use the root user on a day-to-day basis. In fact, the *only* thing you should use the root user for is to create other user accounts with more-limited permissions, and then switch to one of those accounts immediately, as per the next step. Make sure to store the root user credentials in a secure password manager.
- **Step 2: Create an IAM user.** To create a more-limited user account, you will need to use the *Identity and Access Management (IAM)* service. To create a new *IAM user*, go to the [IAM Console](#) (note: you can use the search bar at the top of the AWS Console to find services such as IAM), click Users, and then click the Create User button. Enter a name for the user, and make sure "Provide user access to the AWS Management Console" is selected, as shown in [Figure 1-4] (note that AWS occasionally makes changes to its web console, so what you see may look slightly different from the screenshots in this lab).

User details

User name

The user name can have up to 64 characters. Valid characters: A-Z, a-z, 0-9, and + = , . @ _ - (hyphen)

☒ Provide user access to the AWS Management Console - *optional*

If you're providing console access to a person, it's a [best practice](#) to manage their access in IAM Identity Center.

Console password

☒ Autogenerated password

You can view the password after you create the user.

☐ Custom password

Enter a custom password for the user.

- Must be at least 8 characters long
- Must include at least three of the following mix of character types: uppercase letters (A-Z), lowercase letters (a-z), numbers (0-9), and symbols ! @ # \$ % ^ & * () _ + - (hyphen) = [] { } | ' "

☐ Show password

☐ Users must create a new password at next sign-in - Recommended

Users automatically get the [IAMUserChangePassword](#) policy to allow them to change their own password.

Figure 1-3. Use the AWS Console to create a new IAM user.

Click the Next button. AWS will ask you to add permissions to the user. By default, new IAM users have no permissions whatsoever and cannot do anything in an AWS account. To give your

IAM user the ability to do something, you need to associate one or more IAM Policies with that user's account. An *IAM Policy* is a JSON document that defines what a user is or isn't allowed to do. You can create your own IAM Policies or use some of the predefined IAM Policies built into your AWS account, which are known as *Managed Policies*.

To run the examples in this lab, the easiest way to get started is to select "Attach policies directly" and add the `AdministratorAccess` Managed Policy to your IAM user (search for it, and click the checkbox next to it), as shown in [Figure 1-4].

Permissions options

☐ **Add user to group**
Add user to an existing group, or create a new group. We recommend using groups to manage user permissions by job function.

☐ **Copy permissions**
Copy all group memberships, attached managed policies, and inline policies from an existing user.

☒ **Attach policies directly**
Attach a managed policy directly to a user. As a best practice, we recommend attaching policies to a group instead. Then, add the user to the appropriate group.

Permissions policies (1/2050)

Choose one or more policies to attach to your new user.

Filter by Type All types 4 matches < 1 >

<input type="checkbox"/>	<input type="button" value="+"/> Policy name <input type="button" value="Link"/>	Type	Attached entities
<input checked="" type="checkbox"/>	<input type="button" value="+"/> AdministratorAccess	AWS managed - job function	2
<input type="checkbox"/>	<input type="button" value="+"/> AdministratorAccess-Amplify	AWS managed	0
<input type="checkbox"/>	<input type="button" value="+"/> AdministratorAccess-AWS...	AWS managed	0
<input type="checkbox"/>	<input type="button" value="+"/> AWSAuditManagerAdmini...	AWS managed	0

Figure 1-4. Add the `AdministratorAccess` Managed IAM Policy to your IAM user.

Click Next again and then the "Create user" button. AWS will show you the security credentials for that user, which consist of (a) a sign-in URL, (b) the user name, and (c) the console password. Save all three of these immediately in a secure password manager (e.g., 1Password), especially the console password, as it will *never* be shown again.

- **Step 3: Login as the IAM user.** Now that you've created an IAM user, log out of your AWS account, go to the sign-in URL you got in the previous step, and enter the user name and console password from that step to sign in as the IAM user.
- **Step 4: Deploy an EC2 instance.** You can use the *Elastic Compute Cloud (EC2)* service to deploy servers called *EC2 instances*. Head over to the [EC2 Console](#) and click the "Launch

instance” button. This will take you to a page where you configure your EC2 instance, as shown in [Figure 1-5].

Launch an instance [Info](#)

Name and tags [Info](#)


Name

[Add additional tags](#)


▼ Application and OS Images (Amazon Machine Image) [Info](#)

Quick Start


Amazon Linux




macOS




Ubuntu




Windows




Red Hat



SUSE Linux




[Browse more AMIs](#)
Including AMIs from
AWS, Marketplace and
the Community

Amazon Machine Image (AMI)

Amazon Linux 2023 AMI

ami-0900fe555666598a2 (64-bit (x86), uefi-preferred) / ami-08789139447cee751 (64-bit (Arm), uefi)
Virtualization: hvm ENA enabled: true Root device type: ebs

Free tier eligible ▼

Figure 1-5. Configure the name and AMI to use for your EC2 instance.

Fill in a name for the instance. Below that, you need to pick the operating system image to use (the Amazon Machine Image or *AMI*). For now, you can stick with the default, which should be Amazon Linux. Below that, configure the instance type and key pair, as shown in [Figure 1-6]:

▼ Instance type [Info](#) | [Get advice](#)

Instance type

t2.micro

Free tier eligible

Family: t2 1 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0116 USD per Hour
On-Demand SUSE base pricing: 0.0116 USD per Hour
On-Demand Windows base pricing: 0.0162 USD per Hour
On-Demand RHEL base pricing: 0.0716 USD per Hour

▼

☒ All generations

[Compare instance types](#)

[Additional costs apply for AMIs with pre-installed software](#)

▼ Key pair (login) [Info](#)

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

Proceed without a key pair (Not recommended)

Default value ▼


 [Create new key pair](#)

Figure 1-6. Configure the instance type and key pair to use for your EC2 instance.

The *instance type* specifies what type of server to use: that is, what sort of CPU, memory, hard drive, etc. it'll have. For this quick test, you can use the default, which should be `t2.micro` or `t3.micro`, small instances (1 CPU, 1GB of memory) that are part of the AWS free tier. The *key pair* can be used to connect to the EC2 instance via SSH, a topic you'll learn more about in [Chapter 7]. You're not going to be using SSH for this example, so select "Proceed without a key pair." Scroll down to the network settings, as shown in [Figure 1-7].

▼ Network settings Info

Edit

Network Info

vpc-deb90eb6

Subnet Info

No preference (Default subnet in any availability zone)

Auto-assign public IP Info

Enable

Additional charges apply when outside of free tier allowance

Firewall (security groups) Info

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

☒ Create security group

☐ Select existing security group

We'll create a new security group called 'launch-wizard-18' with the following rules:

☐ Allow SSH traffic from

Helps you connect to your instance

☐ Allow HTTPS traffic from the internet

To set up an endpoint, for example when creating a web server

☒ Allow HTTP traffic from the internet

To set up an endpoint, for example when creating a web server

Figure 1-7. Configure the network settings for your EC2 instance.

You'll learn about networking in [Chapter 7]. For now, you can leave most of these settings at their defaults: the Network should pick your Default VPC (in [Figure 1-7], my Default VPC has the ID `vpc-deb90eb6`, but your ID will be different), the Subnet should be "No preference," and Auto-assign public IP should be "Enable." The only thing you should change is the Firewall (security groups) setting, selecting the "Create security group" radio button, disabling the "Allow SSH traffic from" setting, and enabling the "Allow HTTP traffic from the internet" setting, as shown in [Figure 1-7]. By default, EC2 instances have firewalls, called *security groups*, that don't allow any network traffic in or out. Allowing HTTP traffic tells the security group to allow inbound TCP traffic on port 80 so that the sample app can receive requests and respond with "Hello, World!"

Next, open up the "Advanced details" section, and scroll down to "User data," as shown in [Figure 1-8].

User data - optional [Info](#)

Upload a file with your user data or enter it in the field.

 **Choose file**

```
#!/usr/bin/env bash

set -e

# Install Node.js on Amazon Linux
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
source ~/.bashrc
nvm install --lts

# Create the Node.js app
tee app.js > /dev/null << "EOF"
const http = require('http');

// Respond to all requests with "Hello, World!"
const server = http.createServer((req, res) => {
```

☐ User data has already been base64 encoded

Figure 1-8. Configure user data for your EC2 instance.

User data is a script that will be executed by the EC2 instance the very first time it boots up. Copy and paste the script shown in [Example 1-2] into user data:

Example 1-2. User data script

```
#!/usr/bin/env bash

set -e

# [1]
curl -fsSL https://rpm.nodesource.com/setup_21.x | bash -
yum install -y nodejs

# [2]
tee app.js > /dev/null << "EOF"
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

# [3]
const port = process.env.PORT || 80;
server.listen(port, () => {
  console.log(`Listening on port ${port}`);
});
EOF

# [4]
nohup node app.js &
```

You should also save a local copy of this script in *ch1/ec2-user-data-script/user-data.sh*, so you can reference it later. This Bash script will do the following when the EC2 instance boots:

- [1] Install Node.js.
- [2] Write the sample app code to a file called *app.js*. This is the same Node.js code you saw earlier in the chapter, with one difference, as described next.
- [3] The only difference from the sample app code you saw earlier is that this code defaults to listening on port 80 instead of 8080, as that's the port you opened up in the security group.
- [4] Run the app using `node app.js`, just like you did on your own computer. The only difference is the use of `nohup` and ampersand (`&`), which allows the app to run permanently in the background, while the Bash script itself can exit.

Watch out for snakes: these examples have several problems

The approach shown here with user data has a number of drawbacks, as explained in [Table 1-2] :

Table 1-2. Problems with the simplified example

Problem	What the example app does	What you should do instead
Root user	User data scripts run as the root user, so running the app from user data means it runs as the root user, too.	Run apps using a separate OS user with limited permissions.
Port 80	The app listens on port 80, which requires root user permissions.	Have apps listen on a port greater than 1024.
User data limits	The app crams all of its code and dependencies into the user data script. User data scripts are limited to 16 KB.	Use configuration management or server templating tools.
Process supervision	The user data script starts the app, but user data scripts only run on the very first boot.	Use process supervisors to restart your app if it crashes or the server reboots.
Node.js specifics	The user data script runs just a single Node.js process, in development mode, using only one CPU core.	Run multiple Node.js processes to use all CPU cores, all in production mode.

You'll see how to address all of these limitations in [Chapter 3]. Since this is just a first example for learning, it's OK to use this simple, insecure approach for now, but make sure not to use this approach in production.

Leave all the other settings at their defaults and click "Launch instance." Once the EC2 instance has been launched, you should see the ID of the instance on the page (something like `i-05565e469650271b6`). Click on the ID to go to the EC2 instances page, where you should see your EC2 instance booting up. Once it has finished booting (you'll see the instance state go from "Pending" to "Running"), which typically takes 1-2 minutes, click on the row with your instance, and in a drawer that pops up at the bottom of the page, you should see more details about your EC2 instance, including its public IP address, as shown in [Figure 1-9].

Instances (1/1) Info

↺

Connect

Instance stat

🔍 Find Instance by attribute or tag (case-sensitive)

Instance ID = i-05565e469650271b6 ✕

Clear filters

✓	Name ✎	Instance ID	Instance stat
✓	sample-app	i-05565e469650271b6	🟢 Running

Instance: i-05565e469650271b6 (sample-app)

Details

Status and alarms New

Monitoring

Security

Networking

▼ Instance summary Info

Instance ID

📄 i-05565e469650271b6 (sample-app)

IPv6 address

—

Public IPv4 address

📄 3.22.99.215 | [open address](#) ↗

Instance state

🟢 Running

Figure 1-9. Find the public IP address for your EC2 instance.

Copy and paste that IP address, open `http://<IP>` in your browser (note: you have to actually type the `http://` portion in your browser, or the browser may try to use `https://` by default, which will *not* work), and you should see:

Congrats, you now have your app running on a server in AWS!

Practice [Exercise]

Here are a few exercises you can try at home to go deeper:

- Try restarting your EC2 instance. Does the sample-app still work after the reboot? If not, why not?
- Find CloudWatch in the AWS Console and use it to look at the logs and metrics for your EC2 instance. How does this compare to the monitoring you got with Render?

IMPORTANT :

When you're done experimenting with AWS, you should undeploy your app by selecting your EC2 instance, clicking "Instance state," and choosing "Terminate instance" in the drop down, as shown in [Figure 1-10]. This ensures that your account doesn't start accumulating any unwanted charges.

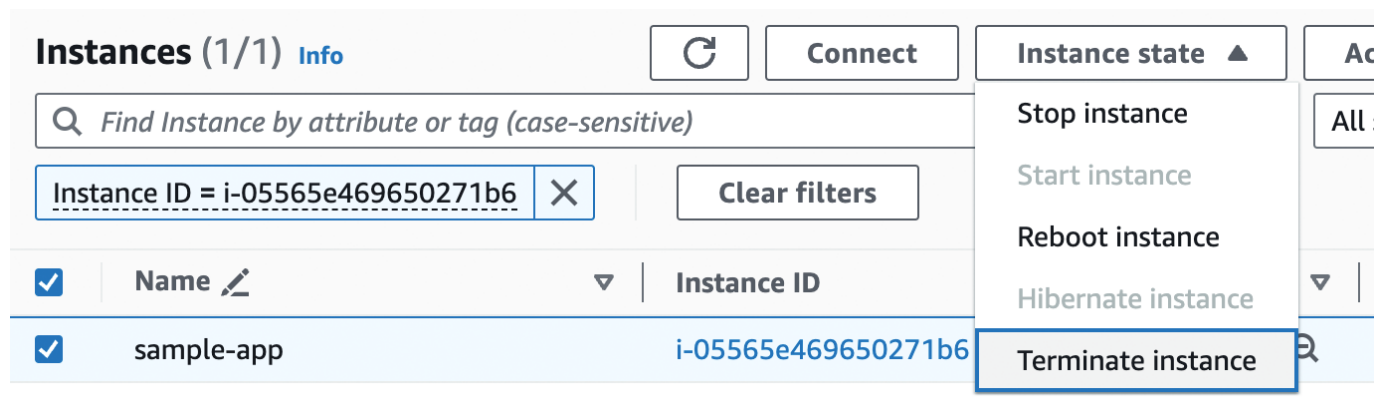


Figure 1-10. Make sure to terminate your EC2 instance when you're done testing.

Conclusion

You now know the basics DevOps, software delivery, and deploying apps.