# the Pulumi book

A developer's guide to programming the cloud with TypeScript and Amazon Web Services

Christian Nunciato

# The Pulumi Book

A developer’s guide to programming the cloud with TypeScript and Amazon Web Services

Christian Nunciato

This book is available at https://leanpub.com/pulumi

This version was published on 2025-09-14

# Contents

# Thanks, and welcome!

# Part 1: Foundations

# Chapter 1: Introducing Pulumi

## This chapter covers:

- Infrastructure as code: what it is and why you should care
- How Pulumi is like (and unlike) other infrastructure-as-code tools
- The parts of Pulumi, what they do, and how to use them
- The typical Pulumi workflow, end to end with a simple project

> Easy things should be easy, and hard things should be possible.
>
> — *Larry Wall, creator of Perl*

I'll be honest: As a developer, I've never quite *loved* having to deal with infrastructure.

By *infrastructure*, I mean the underlying machinery of computing: the rack-mounted servers, virtual machines, networks, storage — most of which usually ends up getting in my way, and at best is a distraction from what I really care about, which is building and shipping web applications. In an ideal world, I'd just write my code, push my commits to GitHub, and sit back happily with the beverage of my choice while the rest of the process *just worked*, and the infrastructure magically took care of itself.

The trouble, of course, is that I'm a *web* developer, and one can't get very far with a web application without at least some kind of infrastructure. So for the majority of my career, I've had to contend with this reality, and it hasn't always been fun. Sometimes it has. But generally, not so much.

But after years of frustration and toil, here's what I've learned: with the right set of tools, managing infrastructure can actually be a delightful experience — even for a cranky developer like me. Pulumi, it turns out, is one of those tools.

Pulumi is an open-source command-line tool and software-development kit (SDK) that lets you build, deploy, and manage cloud applications and infrastructure with general-purpose programming languages like TypeScript, C#, Python, and Go. With Pulumi, you can build just about anything that can be built with the cloud today, from simple static websites and serverless applications to vast constellations of distributed microservices, all managed with code, using the languages and tools that you know, love, and use every day. Pulumi is a relatively new addition to a class of tools that enable *infrastructure as code*, and in this book, you'll learn Pulumi from the ground up by building several different kind of applications with TypeScript and deploying them into the Amazon Web Services (AWS) cloud.

Depending on your background, though, you might be wondering why a tool like Pulumi even exists in the first place. Why would anyone want to write code to manage infrastructure? Isn't it easier just to configure computers by hand? And why should using general-purpose programming languages be such a big deal? Doesn't that make the whole process even more complicated?

Good questions all, and I'm glad that you asked them. Let's begin, then, by taking a step back and having a look at what led to the need for infrastructure as code to begin with, and then later, you'll meet Pulumi and get a sense of what makes it so different.

## Infrastructure: it's complicated

The cloud is a great and powerful thing: limitless computing available on demand, and usable in so many ways it's almost impossible to count them.

It's also a huge, complex, and intimidating beast, and using it effectively can be surprisingly difficult.

As a newcomer especially, just approaching the cloud can be overwhelming, if only because of how *much* of the cloud there is today, and how many providers and services there are to choose from. (I work with this stuff every day, and I can hardly keep up with it all, myself.) AWS alone has hundreds of individual products and services, many of which are obscurely named, confusingly priced, and with so many knobs and switches it can feel like staring at the cockpit of an airliner. Even if you happen to know your way around, it's incredibly easy to get lost in all the complexity.

For example, imagine you're a developer like me — comfortable building web applications, but maybe not so comfortable managing servers. You like solving problems with technology, though, and maybe you're a bit of an entrepreneurial soul as well, so let's say one day, you wake up with an idea for a new online venture: a website aimed at helping dog-owners and dog-walkers get connected. (I'll leave the name of this new venture to you; like many a programmer, I'm terrible with names.)

As you sketch out the details of this nascent little project, you can see the details of the application begin to emerge, and it all seems fairly straightforward: to start, all you'll need are a web app and a database. So you opt to keep it simple and go with what you know: Express.js for the application, say, and MySQL for the database. Easy enough.

For the infrastructure, though, you're in murkier territory. You know you'll need at least one server to host the application, and that a single server would *probably* suffice — but beyond that nebulous requirement, since infrastructure isn't your specialty, you aren't quite sure how best to proceed.

But you're committed, and AWS seems as safe a choice as any, so you decide to roll forward and give it a shot. And on the whole, things go all right: with a few button clicks, and a few hours spent sifting through docs and configuring a new virtual machine with Express and MySQL, your shiny new dog-walking service is live, and people are actually signing up. It's an exciting time, both for you and for many a neighborhood pup. And before long, the service starts to become rather popular.

So popular, in fact, that your little machine's starting to struggle to keep up with all the demand. In an effort to distribute the load a bit better, you move the Express app onto a virtual machine of its own, leaving the database on the original one — moving the database would mean downtime — and after a few more hours spent configuring the second machine (based solely on a shady recollection of what you did for the first), you're all set. Or so you think.

A month or so later, you get an email from an angry customer complaining that the website is down, and indeed you immediately confirm that it is — and not just down, but apparently *gone:* both virtual machines have somehow vanished from the AWS Console, leaving you no choice but to build them all over again, from scratch (and right now) to save your fledgling business. Not fun. (And yes, it happens.)

Or imagine you're faced with an entirely different problem: the website's become *so* popular that you seem to be spending all of your time coping with infrastructure — spinning up servers, patching old ones, rotating logs — and no time developing new features for the site, which is, of course, the whole reason you embarked on this project in the first place.

And this is just one website. Now imagine trying to manage two or three others like this one, all by hand, using only the browser and the command line to do so. Now, imagine ten or twenty. It's a losing proposition: beyond the smallest of scales, managing servers by hand eventually becomes a full-time job, and sooner or later, without some kind of automation, it borders on practically impossible.

And at any scale, you're constantly having to deal with other annoyances:

- **Hand-crafted servers are opaque**. You can't look at AWS Console, for example, and

easily discern the state of a given machine — what's installed on it, what version of a particular application is currently deployed there, what upstream services that application relies on, and so on. The need for this kind of information comes up a lot more often than you might think, and to get it, you generally have to go poking around on the command line.
- **Hand-crafted servers are unique snowflakes**. Just like your laptop, a manually configured machine is the product of countless ad hoc mutations over time, which means you can't just stamp out another copy whenever you need one. And even if you could, any change to one of the copies would have to be made to all of them, which is tedious, error-prone, and not the kind of thing most of us enjoy doing with our time.
- **We humans are awesome, but we mess stuff up — a lot**. We click the wrong buttons in dashboards, we make disastrous mistakes on the command line, we stumble under pressure, and our memories are terrible. Not to mention that we can only do one thing at a time. Everything computers are good at, we aren't. It's just how it is.

But here's the thing: while it's true that we humans aren't very good at managing computers by hand, we *are* pretty good at describing *how* a computer (or lots of computers) *ought* to be configured. It might take me the better part of a month, for example, to provision and configure a hundred servers running a massively popular dog-walking website — but it only it only takes a few moments to write *I want a thousand servers running my dog-walking website*, and only a few more to express that *in code*, and let the computers handle the work of provisioning and configuring those servers for me.

## What is infrastructure as code?

Infrastructure as code (IaC) is the practice of using code in combination with specialized tools to produce an infrastructural outcome. In the previous section, the outcome was pretty simple: a couple of virtual machines, one running an application and the other a database, and to have them both be configured to communicate with one another and be accessible over the web. In other scenarios, it might be much more complex, involving dozens, hundreds, or thousands of cloud resources all working together to deliver a globally distributed service like Twitter or Facebook. Ultimately, the goal of IaC is to help you deliver software reliably and safely at *any* scale, and the best way to do that is to let you express what you want, in some sort of text-based form, and use the tools of IaC to turn that textual expression into reality.

IaC tools tend to fall into one of two categories. *Configuration management* tools focus on managing the software and settings of an existing computer — a virtual machine, say, or a "bare-metal" (i.e., physical) server installed in a data center. These tools typically run on their target computers directly, as executable programs, and you can think of them as software-based stand-ins for the kinds of tasks you might perform yourself at the command-line: installing packages like Node.js, creating user accounts and permissions, writing configuration files, or fetching source code to update a web application. Popular configuration management tools include Ansible (https://ansible.com), Chef (https://chef.io), Puppet (https://puppet.com), and SaltStack (https://saltstack.com).

*Provisioning* tools, on the other hand, focus on *creating* cloud infrastructure from scratch. Their job is to communicate with cloud providers like AWS on your behalf, and you can think of them as stand-ins for the kinds of things you might do in the AWS console: creating virtual-machine instances or database clusters, modifying domain-name settings, and the like. These tools target your cloud provider of choice, and generally run either on your computer or as part of a continuous-delivery pipeline. Pulumi falls into this category, and other examples of provisioning tools include HashiCorp Terraform (https://www.terraform.io), AWS CloudFormation (https://aws.amazon.com/cloudformation), and AWS Cloud Development Kit (https://aws.amazon.com/cdk/).

Both kinds of tools have their place your automation toolbox. And both operate on a couple of common fundamental principles:

- The code you write is *declarative*. Whether you're writing in plain text, YAML, or JavaScript, you're generally describing *what* you want your infrastructure to be, not *how* the tool should go about creating it. Unlike a shell script that codifies an imperative set of steps — *do this, then this, then this* — infrastructure-as-code declares simply, *I want this*, and the job of the tool is to figure out how to turn that declaration into reality.
- Operations are *idempotent*. Given some declarative code as input, an IaC tool produces consistent output, regardless of how many times you run it with that same code. Like a `multiply()` function that always returns `12` given a `6` and a `2`, an IaC tool produces an equally reliable and repeatable result, free from unintended side-effects, and changing only what happens to change *in the code* from one run to the next.

Likewise, both types of tools generally follow a similar workflow:

1. You write some code that declares the end state that you wish to achieve.

2. You run that code with a tool that knows how to interpret and translate it.
3. The tool computes what needs to be done, issuing the appropriate commands on the target.

Importantly, though, in order to do this work safely, the tool also needs to know what already exists. If it didn't — to return to our dog-walking example — it'd stand up two *new* virtual machines on every run, violating the principle of idempotency we just established (and racking up lots of AWS charges in the process). To do things right, IaC tools have to incorporate the concept of *state*, which you can think of as a point-in-time snapshot of the inventory of your infrastructure as of the last time the tool ran successfully. This way, the tool can skip creating resources it knows already exist, or change only certain properties of them, doing only the work that's necessary to drive the target to the new desired state expressed in the code.

Figure 1.1 shows the parts of a typical IaC workflow and how they all come together.

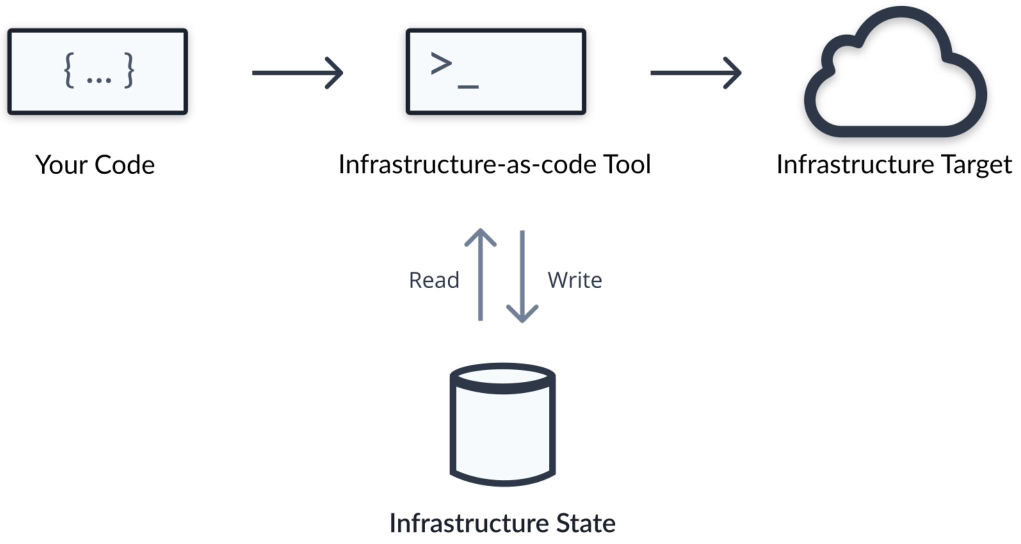


*Figure 1.1: The high-level components of infrastructure as code. IaC tools like Pulumi evaluate your code, compare what's expressed in the code with what currently exists, and drive your infrastructure to some desired state, recording that state as the new, now-current one.*

If you're new to this practice, it can take some getting used to. But the advantages are more than worth the effort. Once you start driving your infrastructure with code, you'll find it hard to go back to poking around in the browser or at the command line manually. And for teams and organizations that need to be able to ship software often and safely, there really is no better way.

Still, it can be challenging. With many of these tools, the learning curve can be steep, especially when you're trying to learn the tool itself, its unique expression language, and the conventions of infrastructure-as-code all at the same time. And since managing infrastructure has historically fallen under the umbrella of operations, most of these tools were built by and for operational specialists, so they model their domains in ways that may make sense to an operator, but leave others — application developers, programmers from other disciplines, hobbyist coders — generally lost. Despite how simple it might've seemed on the surface, our little dog-walking app, as currently designed, would actually require *both* classes of IaC tool: a provisioning tool to spin up its virtual machines, and a configuration management tool (run separately afterward) to manage the software installed on both of them. Not so simple after all.

So what do you do? You want to be able to manage your infrastructure as code, but the tools don't seem to be making that very easy for you. Let's have a look at Pulumi to see how it's different.

## What is Pulumi?

Pulumi is an open-source infrastructure-as-code platform that helps you create and manage cloud applications and infrastructure. In some ways, it's a lot like other provisioning tools in that it's powered by a command-line interface (CLI) that runs on multiple platforms (macOS, Linux, and Windows), and its main job is to translate the code that you write into running cloud software.

But it's there that Pulumi begins to diverge from other provisioning tools in a few important ways:

- With Pulumi, you write code in your high-level programming language of choice, not in a tool- or domain-specific language (DSL). In this book, you'll use TypeScript, but as of this writing, Pulumi also supports JavaScript, Python, Go, Java, YAML, and the .NET Core family of languages, including C#, F#, and Visual Basic.
- Pulumi ships with an extensive set of language SDKs — npm packages for Node.js, PyPi for Python, NuGet for .NET, and so on — that model the cloud in a rich, consistent, and idiomatic way, allowing you to program (and compose) at very high levels of abstraction. You can also embed Pulumi into your programs directly, which can enable all sorts of interesting runtime possibilities.
- By default, Pulumi uses the free Pulumi Service to track the state of your infrastructure across projects and environments, while other tools tend expect you to manage

this on your own. The Service is optional, but incredibly helpful, so we'll be using it throughout this book (and you'll learn more about it in chapter 3).

These are just a few of the features that most distinguish Pulumi. You'll see them in action shortly, and we'll put them all to good use over the course of this book. Additionally, it can also help to gain a quick understanding of a few of the ideas behind Pulumi.

## Understanding Pulumi

Despite being an open-source project with a large and growing community and dozens of active contributors, Pulumi is nevertheless guided by a few fundamental principles.

### Easy things are easy

In describing the design goals of the Perl programming language, its creator Larry Wall said something that's always stuck with me: "Easy things should be easy, and hard things should be possible."

I haven't written much Perl, myself, but most of us would probably agree that a well-designed API of any kind should do exactly that: make it easy to accomplish the kinds of tasks we do most often, while still allowing us to handle more complicated situations when the need arises.

That's one of Pulumi's guiding principles as well, and you can see it in the way it builds in high-value, often overlooked features like hosted state management, secrets management (for managing passwords, keys, and the like), and high-level APIs for handling common cloud-native architectures.

### Driven by general-purpose programming languages

Most infrastructure-as-code tools have you declare your resources individually with blocks of static, structured text — JSON, YAML, and the like. This definitely works, and seems easy enough on the surface. But over time, as a project grows, you'll often find that you need more control and flexibility than these formats are able to give you. What if you wanted to run three app servers in production, but only one in test? Or name your resources dynamically, using a third-party library or an HTTP service? While some tools do support simple mechanisms like loops or string interpolation, managing any sort of real complexity with statically written text eventually becomes an exercise in frustration.

Modern programming languages make this immeasurably easier. With Pulumi, since you're writing in your language of choice, you can use all of the facilities available for that language and its ecosystem, including variables, loops, conditional logic, functions, classes, modules, package managers, and unit testing — and with the help of modern IDEs, you can easily refactor your programs with confidence as you go. You'll see all of this at work in the chapters ahead.

### By and for developers

Built by a team of engineers from Microsoft with backgrounds in developer tools, languages, and compilers, Pulumi is the first infrastructure-as-code tool made with the needs of application developers in mind, and with the goal of making it easier for anyone who can write a little code to take advantage of the cloud. As long as you're reasonably comfortable in at least one programming language, can run a few commands in a terminal window, and are willing to spend a little time learning what may be a few new cloud concepts, you've got all you need to pick up start using Pulumi to program the cloud.

## Using Pulumi: A end-to-end walkthrough

Let's combine all we've covered up to now and have a look at something real — well, semi-real, at least; you'll build an application of your own in the next chapter. For now, let's just walk through a simple Pulumi project together from start to finish, just to give you a sense of what the end-to-end workflow typically looks like. The task at hand is ambitious, but achievable: a one-page website deployed on AWS, purpose-built to deliver the programmer's favorite greeting.

*Figure 1.2: A wireframe of the single-page website we'll build and deploy to AWS with Pulumi.*

On AWS, the most common way to manage a static website is with its Simple Storage Service (S3) (https://aws.amazon.com/s3/). With S3, you can store files of any kind in what's called a *bucket*, which you can think of as a uniquely named, unlimited-capacity folder in the cloud. Since a website (a static one, anyway) is ultimately just a collection of files in a folder, and S3 offers built-in support for serving the contents of a bucket as a website, an S3-based website an excellent fit for this kind of project.

So in this walkthrough, we'll:

- Generate a new *project* with the Pulumi CLI
- Write the code to define an S3 bucket and a home page
- Run that code with Pulumi to provision the bucket and configure it as a website
- Verify the result.

These steps align with the workflow of pretty much every Pulumi project, as illustrated in Figure 1.3. Whether you're building a simple website like this one or a complex infrastructure platform consisting of thousands of distributed resources, it's always the same: write the code, run Pulumi to turn it into cloud infrastructure, and verify the result. (Technically, the third step is optional, but it's always a good idea — ideally with some automated testing, which you'll learn more about in part 3.)

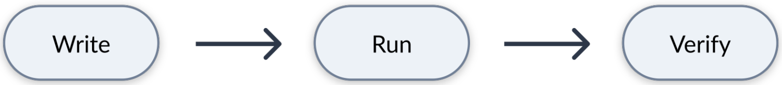


*Figure 1.3: A typical Pulumi workflow: Write some infrastructure code, run Pulumi, and verify the result aligns with what you expected.*

As we go, don't worry about understanding every last detail; we've got the rest of the book to explore everything in depth. At this point, it's enough to understand the broad strokes: what the steps are, what happens at each one, and how they relate to one another to produce the intended result. Beyond that, you might also keep an eye out for the ways that Pulumi applies the principles of infrastructure-as-code we've identified so far: declarative code, idempotently applied, and backed by some form of persistent state.

Let's have a look.

## Generating a new Pulumi project

The easiest way to begin a new Pulumi project is with the `pulumi` command-line tool (CLI). Since in this book, we'll mainly be using using TypeScript and AWS, we'll generally use the `pulumi new` command as follows:

```
$ pulumi new aws-typescript
```

Lots more new-project templates are available for different languages and cloud providers, and you'll even learn how to write your own templates later, when we explore the many ways of packaging and sharing Pulumi projects. After prompting you for a few details about this particular project, the command will complete, leaving you with a couple of new configuration files, a `package.json` file for managing the project's dependencies, and an `index.ts` file containing the TypeScript code for a minimal Pulumi program — conveniently, one that happens to create an S3 bucket.

## Writing the code

Now let's take a look at the contents of the program. Listing 1.1 shows what's inside.

*Listing 1.1: A Pulumi program that creates a storage bucket on Amazon S3.*

```
1  import * as pulumi from "@pulumi/pulumi";
2  import * as aws from "@pulumi/aws";
3  import * as awsx from "@pulumi/awsx";
4
5  const bucket = new aws.s3.Bucket("my-bucket");
6
7  export const bucketName = bucket.id;
```

1 Imports the `@pulumi/pulumi` library for Node.js.
2 Imports `@pulumi/aws`.
6 Declares a new S3 bucket with `@pulumi/aws`.

If you've worked with TypeScript, the first couple of lines should look pretty familiar. The first one imports the open-source `@pulumi/pulumi` `@pulumi/aws` SDK for Node.js, which was installed with npm when we ran `pulumi new`. This package contains most of the JavaScript and TypeScript APIs you'll use to manage the resources of the AWS cloud, including the `s3` module, which defines the TypeScript `Bucket` class.

Line 6 is worth examining a little more closely, both for what it does as well as what it doesn't. Specifically, it creates an instance of the `Bucket` class, passing a name to be used for the bucket resource. Notice, however, that the program doesn't seem to *do* anything with that instance afterward; you don't see any `.create()` method being called anywhere else in the program, for example. How exactly does Pulumi know to *create* an S3 bucket?

We'll get to that in a second. For now, let's add a bit more code to round out the program by configuring the bucket to act like a website. The following listing shows how that's done.

*Listing 1.2: The same program, extended to serve the S3 bucket as a static website.*

```
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.Bucket("hello-world");

const bucketWebsite = new aws.s3.BucketWebsiteConfiguration("bucketWebsite", {
    bucket: bucket.bucket,
    indexDocument: {
        suffix: "index.html",
    },
});

const bucketOwnershipControls = new aws.s3.BucketOwnershipControls("bucket-owne\
rship-controls", {
    bucket: bucket.id,
    rule: {
        objectOwnership: "ObjectWriter",
    },
});

const bucketPublicAccessBlock = new aws.s3.BucketPublicAccessBlock("bucket-publ\
ic-access-block", {
    bucket: bucket.id,
    blockPublicAcls: false,
    blockPublicPolicy: false,
    ignorePublicAcls: false,
    restrictPublicBuckets: false,
}, { dependsOn: [bucketOwnershipControls] });

const bucketPolicy = new aws.s3.BucketPolicy("bucket-policy", {
    bucket: bucket.id,
    policy: bucket.id.apply(bucketId => JSON.stringify({
        Version: "2012-10-17",
        Statement: [{
            Effect: "Allow",
            Principal: "*",
            Action: "s3:GetObject",
            Resource: `arn:aws:s3:::${bucketId}/*`,
        }],
    })),
}, { dependsOn: [bucketPublicAccessBlock] });

const homepage = new aws.s3.BucketObject("index.html", {
    bucket: bucket.id,
    content: `
        <html>
            <body>Hello, world!</body>
        </html>
    `,
49      contentType: "text/html",
50  }, { dependsOn: [bucketPolicy] });
51
52  export const url = bucketWebsite.websiteEndpoint;
```

| | |
|---|---|
| `3` | Declares an S3 bucket. |
| `5-10` | Creates a bucket website configuration that designates `index.html` as the home page. |
| `12-17` | Sets up bucket ownership controls required by AWS for public access. |
| `19-25` | Configures the bucket to allow public access by disabling AWS security blocks. |
| `27-38` | Creates a bucket policy that grants public read access to all objects in the bucket. |
| `40-48` | Declares the `index.html` file with HTML content for the home page. |
| `50` | Exports the website URL for easy access after deployment. |

As you can see, creating a public S3 website requires several security-related resources in addition to the basic bucket. This is because AWS has tightened its security defaults over the years to prevent accidental exposure of private data. Let me briefly explain what each does:

The `BucketWebsiteConfiguration` resource tells S3 to serve the bucket contents as a website and specifies which file to use as the home page. The `BucketOwnershipControls` resource enables the bucket to accept public access configurations — a prerequisite for hosting public websites. The `BucketPublicAccessBlock` resource explicitly allows public access by disabling AWS's protective blocks. Finally, the `BucketPolicy` uses AWS's JSON-based Identity and Access Management (IAM) language to grant read permissions to anyone on the internet.

While this may seem like a lot of configuration for a simple website, these resources work together to ensure your website is both publicly accessible and secure. We'll dive deeper into AWS security patterns in chapter 5, but for now, it's enough to know that these resources are required by AWS's modern security model. The good news is that once you have this pattern working, you can reuse it for any static website.

For the home page itself, we supply a snippet of HTML directly in the code, but we could also have written a more elaborate home page in a separate file and imported it programmatically — using the built-in Node.js `FileSystem` module, say (another example of the benefits of using a general-purpose programming language). For example:

*Listing 1.3: The same program, modified to read the HTML file from the local filesystem.*

```
import * as aws from "@pulumi/aws";
import * as fs from "fs";

const bucket = new aws.s3.Bucket("hello-world");

const bucketWebsite = new aws.s3.BucketWebsiteConfiguration("bucketWebsite", {
    bucket: bucket.bucket,
    indexDocument: {
        suffix: "index.html",
    },
});

const bucketOwnershipControls = new aws.s3.BucketOwnershipControls("bucket-owne\
rship-controls", {
    bucket: bucket.id,
    rule: {
        objectOwnership: "ObjectWriter",
    },
});

const bucketPublicAccessBlock = new aws.s3.BucketPublicAccessBlock("bucket-publ\
ic-access-block", {
    bucket: bucket.id,
    blockPublicAcls: false,
    blockPublicPolicy: false,
    ignorePublicAcls: false,
    restrictPublicBuckets: false,
}, { dependsOn: [bucketOwnershipControls] });

const bucketPolicy = new aws.s3.BucketPolicy("bucket-policy", {
    bucket: bucket.id,
    policy: bucket.id.apply(bucketId => JSON.stringify({
        Version: "2012-10-17",
        Statement: [{
            Effect: "Allow",
            Principal: "*",
            Action: "s3:GetObject",
38                Resource: `arn:aws:s3:::${bucketId}/*`,
39            }],
40        })),
41    }, { dependsOn: [bucketPublicAccessBlock] });
42
43    const homepage = new aws.s3.BucketObject("index.html", {
44        bucket: bucket.id,
45        content: fs.readFileSync("./index.html").toString(),
46        contentType: "text/html",
47    }, { dependsOn: [bucketPolicy] });
48
49    export const url = bucketWebsite.websiteEndpoint;
```

2 Imports Node.js's built-in `fs` module.
43 Reads an `index.html` file from the current directory instead of defining HTML inline.

Finally, the last line of the program exports the `websiteEndpoint` property of the `bucketWebsite` instance — the URL of the website — which AWS will compute and expose to Pulumi once the bucket is created and configured. There's a good bit of detail packed into that statement, but for now, it's enough to note that by exporting this value, Pulumi will print the value of that property to the terminal, giving us something to navigate to.

With that, we're ready to deploy.

## Deploying to AWS

In addition to serving as a helpful way to bootstrap a new project, the Pulumi CLI is responsible for converting your code into running cloud infrastructure. You'll generally divide your time with Pulumi between your editor, writing programs, and the CLI, running commands like the ones we'll look at next:

- `pulumi preview`, which tells you what your program *would* do if you were to deploy it, and
- `pulumi update` (or `pulumi up`, for short), which performs an actual deployment.

When you're comfortable enough with a program to give it a trial run, you can open your terminal, change to your project folder, and run `pulumi preview`. The output below shows the result for our working example.

*Previewing the infrastructure to be created for the hello-world program.*

```
 1  $ pulumi preview
 2  
 3  Previewing update (dev):
 4       Type                                       Name                       Plan
 5   +   pulumi:pulumi:Stack                        hello-dev                  crea\
 6  te
 7   +   ├─ aws:s3:Bucket                          hello-world                create
 8   +   ├─ aws:s3:BucketWebsiteConfiguration      bucketWebsite              create
 9   +   ├─ aws:s3:BucketOwnershipControls         bucket-ownership-controls  create
10   +   ├─ aws:s3:BucketPublicAccessBlock         bucket-public-access-block create
11   +   ├─ aws:s3:BucketPolicy                    bucket-policy              create
12   +   └─ aws:s3:BucketObject                    index.html                 create
13  
14  Outputs:
15      url: output<string>
16  
17  Resources:
18      + 7 to create
```

| | |
|---|---|
| 5 | The `dev` stack resource. |
| 6 | The S3 `Bucket` resource. |
| 7 | The bucket website configuration. |
| 8 | The bucket ownership controls for public access. |
| 9 | The public access block configuration. |
| 10 | The bucket policy granting public read access. |
| 11 | The S3 `BucketObject` containing the home page. |
| 14 | The not-yet-determined URL of the website-to-be. |

There are several interesting things to note about this output. First, after evaluating the code, Pulumi's determined that it needs to create *seven* new resources — more than you might have expected from the code! While we explicitly declared six resources (`Bucket`, `BucketWebsiteConfiguration`, `BucketOwnershipControls`, `BucketPublicAccessBlock`, `BucketPolicy`, and `BucketObject`), Pulumi also creates a `pulumi:pulumi:Stack` resource automatically.

We'll cover *stacks* in much more detail in chapter 3, but you can think of a stack as a logical grouping of cloud resources that makes it easier to manage your deployments. You can create as many stacks as you like for a given project, name them however you like, and configure them independently of one another, as you'll see in chapter 3. By default, Pulumi offers to create a `dev` stack for every new project; we accepted that name when we ran `pulumi new`.

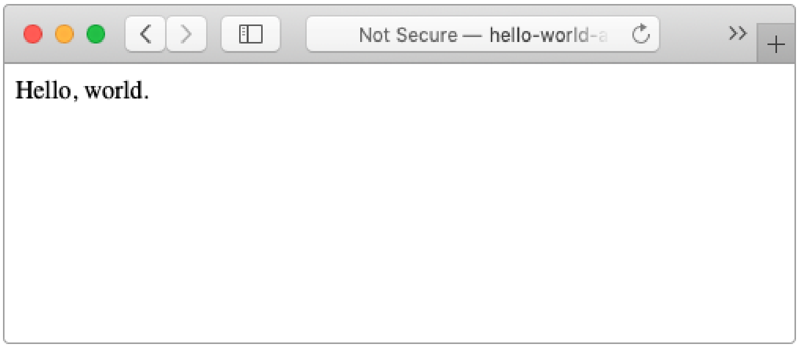
Now let's see what happens when we run the update for real.

*Deploying the hello-world program for the first time.*

```
 1  $ pulumi up
 2
 3  View Live: https://app.pulumi.com/cnunciato/hello/dev/updates/1
 4
 5  Updating (dev):
 6       Type                    Name         Status
 7   +   pulumi:pulumi:Stack     hello-dev    created
 8   +   ├─ aws:s3:Bucket        hello-world  created
 9   +   └─ aws:s3:BucketObject  index.html   created
10
11  Outputs:
12      url: "hello-world-ae6198e.s3-website-us-west-2.amazonaws.com"
13
14  Resources:
15      + 3 created
16
17  Duration: 18s
```

| | |
|---|---|
| 3 | A permanent link to the update in the Pulumi Service. |
| 7 | The newly created `dev` stack. |
| 8 | The S3 bucket containing our website. |
| 9 | The home page of the website as an S3 bucket object. |
| 12 | The URL of the website, exported using the bucket's `websiteEndpoint` property. |

The output shows Pulumi did what it said it would: All three resources have now been created, and the `export` statement we declared in the program has produced a `url` property listed under `Outputs`. Navigating to that URL reveals the final product, in all its minimalist glory.



Hello, world.

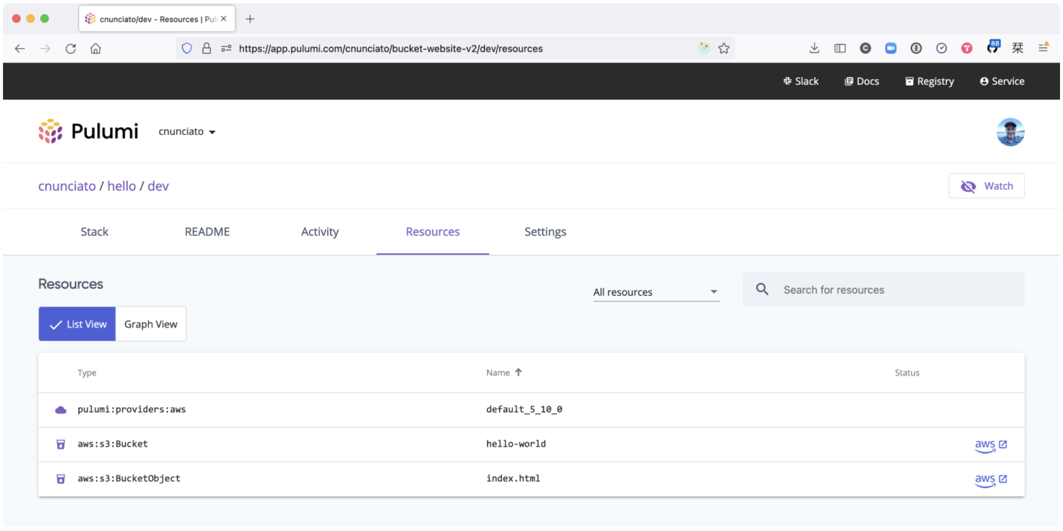
*Figure 1.4: The finished product, now deployed as a static website on AWS.*

You’ll find the source code for this example, along with all others contained in this book, on GitHub at https://github.com/pulumibook.

Let’s conclude this excursion by following that link to the Pulumi Service, which Pulumi uses by default to manage the state of your infrastructure, along with all of your projects, stacks, and integrations with third-party services like GitHub (which you’ll learn more about in part 3, when we explore continuous integration and delivery).

## Verifying the result

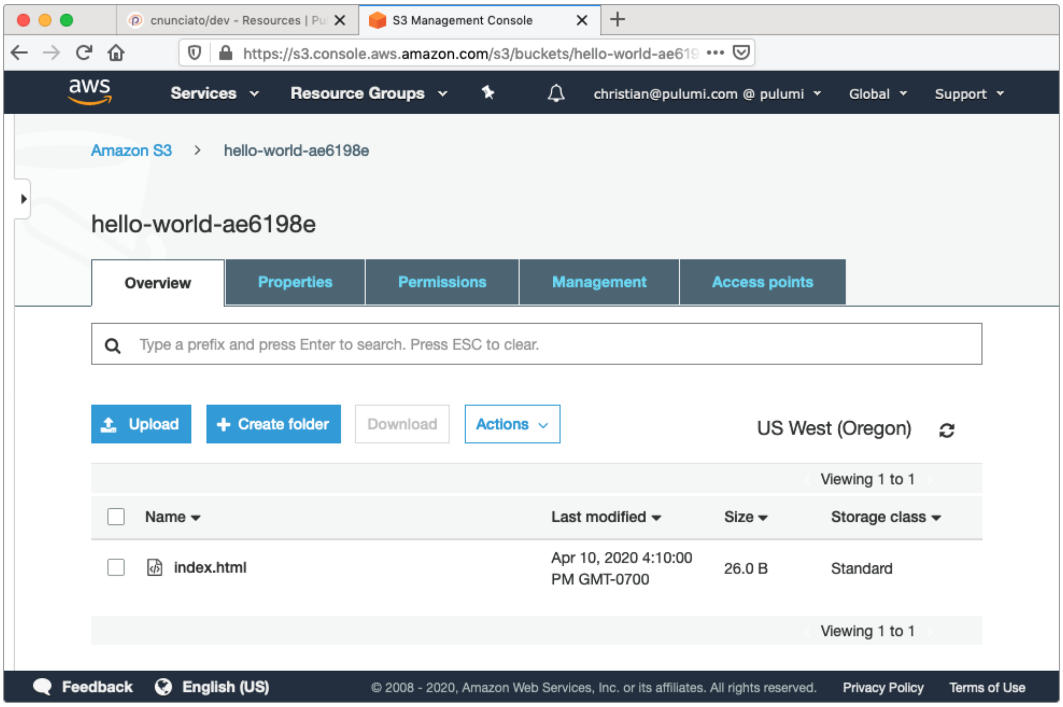
Our project has a permanent home in the Pulumi Service. There, you’ll see a history of all of the changes made to your each of your Pulumi stacks, along with the resources belonging to each one. Figure 1.5 shows the Resources tab of the `hello` project’s `dev` stack, which lists the bucket and bucket object, each with a link to their respective locations in the AWS Management Console.



| Type | Name | Status |
|---|---|---|
| pulumi:providers:aws | default_5_10_0 | |
| aws:s3:Bucket | hello-world | aws |
| aws:s3:BucketObject | index.html | aws |

*Figure 1.5: The Pulumi Service, which helps you manage your Pulumi projects, stacks, teams, and more.*

When you follow the link to the bucket in the AWS Console, you can inspect the properties of the bucket in more detail, as shown in Figure 1.6.


cnunciato/dev - Resources | Pu
S3 Management Console
https://s3.console.aws.amazon.com/s3/buckets/hello-world-ae619
aws
Services
Resource Groups
christian@pulumi.com @ pulumi
Global
Support
Amazon S3
hello-world-ae6198e
Overview
Properties
Permissions
Management
Access points
Type a prefix and press Enter to search. Press ESC to clear.
Upload
Create folder
Download
Actions
US West (Oregon)
Viewing 1 to 1
Feedback
English (US)
© 2008 - 2020, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

hello-world-ae6198e

| Name | Last modified | Size | Storage class |
|---|---|---|---|
| index.html | Apr 10, 2020 4:10:00 PM GMT-0700 | 26.0 B | Standard |

*Figure 1.6: The bucket and bucket object, as seen in the AWS Management Console.*

## Reviewing what happened

We covered a lot, so let's briefly review.

We began with the `pulumi new` command, which generated a Pulumi project for us, along with a complete, ready-to-run TypeScript program defining an S3 bucket. We added a few lines of code to turn that bucket into a website, ran `pulumi preview` to get a glimpse of the cloud resources the program would create, and ran `pulumi up` to create them. And finally, we verified the result by viewing the website in a browser, the project and stack in the Pulumi Service, and the resources themselves on AWS. If we wanted, we could make a change to the homepage, run `pulumi up`, and have that change deployed instantly, or even delete the website entirely with another command, `pulumi destroy`, which you'll use to tidy up after the project you build in the next chapter.

You also learned that by default, the Pulumi CLI uses the free Pulumi Service to manage the state of your infrastructure over time, and that the Pulumi Service offers a convenient way to keep track of your projects, stacks, and cloud resources all in one place. You'll learn much more about the Service and its various features in the chapters ahead.

## A peek under the hood

Let's expand our working model now into a more detailed one to carry with you as you move forward. Again, don't worry if you don't understand everything here; we'll return to this model repeatedly throughout the book. You've learned enough at this point, though, that it'd be good to get a more complete sense of how Pulumi really works.

Figure 1.7 zooms in on the Pulumi CLI to reveal a couple of new but important parts: the *language host*, which actually runs the code that you write, and the *resource provider*, which handles communicating with the cloud service to create, read, update, or delete the resources you define in your Pulumi programs.

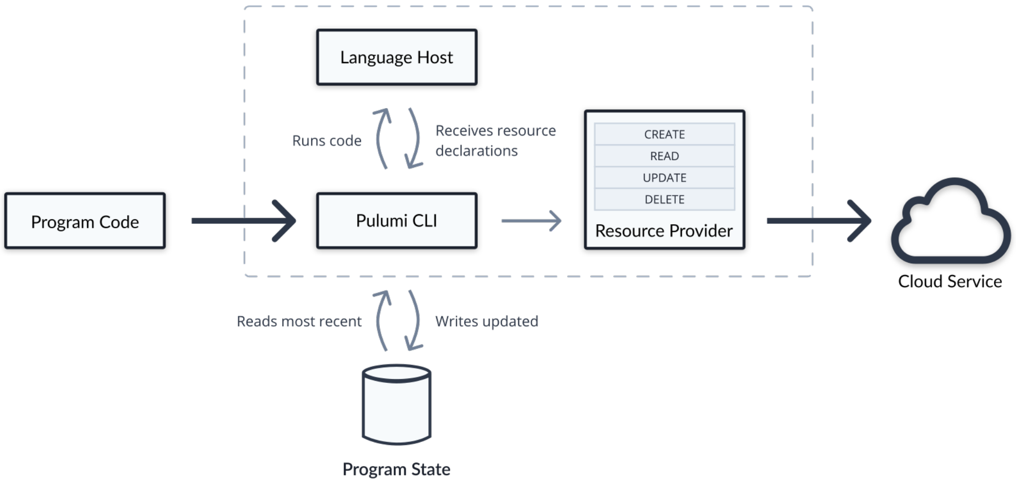


*Figure 1.7: A closer look at the Pulumi CLI. Pulumi passes your code to a language host (such as Node.js), waits to be notified of resource registrations, assembles a model of your desired state, and calls on the resource provider to produce that state by communicating directly with cloud services like AWS.*

Earlier I mentioned we'd come back to the part about how Pulumi knew to create the S3 bucket and homepage, given that all we'd done was instantiate a couple of objects. That's where the language host comes in. When we ran `pulumi preview` and later `pulumi up`, the Pulumi CLI determined that since our chosen language was TypeScript, our program code should be run with Node.js. (If we'd written a program in Python, it would've used Python; if C#, .NET Core, and so on.) So Pulumi launched Node implicitly, passed it our code, and waited to be notified of resource declarations.

Our calls to construct `new Bucket()` and `new BucketObject()` instances were exactly that — declarations to Pulumi of a desire to have these two resources exist in our infrastructure stack. Importantly, the resources weren't actually created on AWS in that moment; rather, their details were simply captured by Pulumi, collected, and compared with our existing state. Since that state was empty (as new projects are), Pulumi computed that these resources should be created, so it assembled a list of operations to perform — *create a new bucket*, *configure the bucket as website*, *create a new file in that bucket* — and passed that list to the resource provider, which translated them into API requests to perform on the AWS service itself. When the work of the resource provider was complete and our resources created, Pulumi updated our stack's state accordingly, saving it back to the Pulumi Service for use on the next run.

Every Pulumi project follows the very same workflow, from simple websites like this one, to serverless web applications, to the most complex cloud-native deployments involving hundreds or thousands of resources: one tool, one workflow, and one language, start to finish. And it's only a short hop from here to the Git-driven workflow I mentioned at the beginning of the chapter — which, as it happens, you'll learn how to do in part 3.

## Why should I use Pulumi?

Let's round out this chapter by highlighting a few of Pulumi's unique strengths.

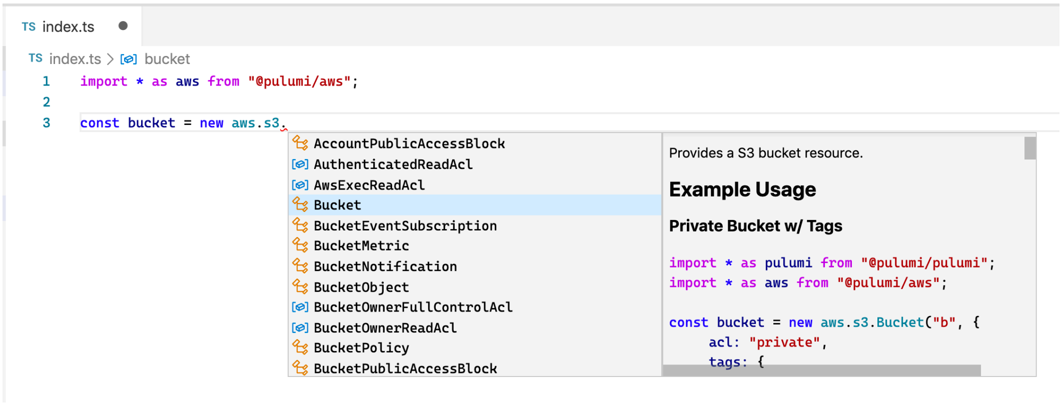
### Building on what you know

We all come to the cloud with different backgrounds and experiences. I mentioned I'm a web developer; you might be, too — or you might be a data scientist looking to move a computational workload into AWS, or a release engineer working to improve the way your company ships software. Whoever you are, if you're reading this book, there's a decent chance you *aren't* an expert in cloud operations, and an equally good chance you're at least moderately comfortable with at least one high-level programming language, whatever that language happens to be.

Pulumi was built with this in mind. Rather than expect you to learn a new discipline, or vendor- or tool-specific language, it lets you start with a language you already know well, and go from there. In practice, that means you can generally hit the ground running, applying the skills you already have to what might well be a totally new domain for you. (As it was for me.)

It also means being able to bring all of the benefits of your chosen language and its ecosystem along for the ride. As you've seen, a Pulumi program is just a regular program — a Node.js script, a Go program with a `main` entrypoint — so you can use whatever organizational constructs, language features, or third-party libraries you like with Pulumi, too.

And if you're a fan, as I am, of tools like Visual Studio Code, you've probably come to appreciate features like type checking, IntelliSense, and head's up documentation that make it easier to explore and use APIs you may still be learning. Figure 1.8 shows what it's like to develop with the Pulumi SDK for Node.js, where code hinting and inline documentation help to keep you productive and on the happy path.



```
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.
```

```
AccountPublicAccessBlock
AuthenticatedReadAcl
AwsExecReadAcl
Bucket
BucketEventSubscription
BucketMetric
BucketNotification
BucketObject
BucketOwnerFullControlAcl
BucketOwnerReadAcl
BucketPolicy
BucketPublicAccessBlock
```

Provides a S3 bucket resource.

**Example Usage**

**Private Bucket w/ Tags**

```
import * as pulumi from "@pulumi/pulumi";
import * as aws from "@pulumi/aws";

const bucket = new aws.s3.Bucket("b", {
    acl: "private",
    tags: {
```

*Figure 1.8: Deep support for programming languages enables syntax highlighting, code completion, inline documentation, automated refactoring, and more, with popular tools like Visual Studio Code.*

## Meeting you where you are

As a developer, I'm used to working at a fairly high level. The languages I use most often are built to help me work productively, and at a level of abstraction that allows me to get big things done fast. TypeScript, for instance, is a great fit for web development work. C, on the other hand, is not — it's too low-level, too "close to the metal". If building web applications meant having to learn and use C every day, there'd be a lot fewer web applications in the world, and a lot fewer web developers, too. It'd just be too difficult for most people to work with.

And yet that's what a lot of infrastructure tools seem to expect from us — to descend from our high-level cruising altitudes back down to sea level just to manage some infrastructure.

Pulumi's use of general-purpose, high-level languages like JavaScript flips this upside down, bringing infrastructure *up* to a level that developers are not only able to work with, but are comfortable doing so. We've already seen how with a few lines of TypeScript, you can deploy a static website — but as you'll see as we move through this chapter and beyond, even a much more complex application needn't require a whole lot more code than that.

When you want a bucket, you can ask for a bucket — but when you want a serverless REST API, you can also just ask for a serverless REST API, and the Pulumi SDK will likely be there to help you. That a REST API might consist of a dozen or more cloud resources — a network, gateway, multiple cloud functions and access policies, etc. — is an implementation detail that Pulumi knows you probably don't care all that much about, so long as it works, is appropriately secure, and you can customize and extend it when you need to. Easy things are easy, hard things are possible.

## Using the whole cloud

A lot of developer-focused cloud tools do a few things really well, like help you build and deploy static websites and distribute them globally, or make it easy to add serverless API endpoints for those websites with a little configuration. Many of these tools are truly great at what they do — but when you find yourself hitting the edges of what they're capable of, like when it's time to add a relational database, or a horizontally scalable API service, you usually have figure those things out on your own, which often means having to go back to clicking buttons in a dashboard, or using some other tool to set those additional components up for you.

The Pulumi SDK gives you a rich set of language-specific APIs that aim to model the whole cloud, consistently and idiomatically — all major clouds, by the way; not just AWS — which means you can write programs that create static websites, serverless APIs, database clusters, or containerized services as easily as you can a single S3 bucket — and even wire them all together within the scope of a single program.

## Built for complexity

In this book, we're focused mainly on learning Pulumi by building small applications. But it's important to know that Pulumi is made to handle the needs of large organizations as well, where responsibilities are often split between teams of developers and operators, and where operators are charged with the task of building secure, reliable cloud *platforms* of their own that can handle constant change and unbounded scale.

General-purpose programming languages are a powerful solution to this problem, in part because of how good they are at handling complexity. JavaScript, for example, is not only a flexible and productive language with a massive community and ecosystem; it also lends itself well to modern software-engineering practices like linting, testing, debugging, and refactoring, among others — practices that help us keep our code under control as our systems grow larger and ever more complicated. Nothing is worse than having to work in a codebase that's huge, brittle, and hard to understand. With Pulumi, you'll manage your infrastructure as software, refactoring as you go to keep things clean and under control.

> Programs must be written for people to read, and only incidentally for machines to execute.
>
> — *Harold Abelson and Gerald Jay Sussman, Preface to the Second Edition, Structure and Interpretation of Computer Programs*

You've learned quite a bit about Pulumi in this chapter, and I hope you're excited to keep learning more. Now would be a good time to take a break with your own beverage of choice before heading into chapter 2, where you'll build your first cloud-native application with Pulumi.

## Summary

- Pulumi is an open-source infrastructure-as-code platform that lets you build, deploy, and manage cloud applications and infrastructure with modern programming languages like TypeScript, Python, C#, and Go.
- Pulumi is made up of three parts: a command-line tool that runs on all major platforms; a collection of language-specific SDKs that model the resources of the cloud; and the optional Pulumi Service, which helps you manage your projects, stacks, and infrastructure state more effectively.
- With Pulumi, you write programs that declare cloud resources as collections of objects, and Pulumi translates those programs into running cloud software.
- With the Pulumi CLI, you can generate new projects, preview infrastructure changes before you make them, and modify your infrastructure safely and idempotently.
- Pulumi is built to make developing cloud applications and infrastructure as accessible as possible, while at the same time addressing the complex needs of large teams and organizations.

# Chapter 2: Your first Pulumi program

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### This chapter covers:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Enter serverless

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Building a serverless cloud notifier

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Configuring your command-line tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

#### Node.js

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### The AWS command-line interface (CLI)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### The Pulumi CLI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Creating the project and stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Specifying your project settings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Understanding the new project layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### index.ts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### package.json, package-lock.json and node_modules

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### tsconfig.json

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Pulumi.yaml and Pulumi.dev.yaml

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Writing the code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Declaring the topic and subscription

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Scheduling and handling events

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Deploying the application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Tailing the logs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Completing the program

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

#### Deploy-time vs. runtime: a brief digression

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Updating the stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Browsing the stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Tidying up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

# Chapter 3: Projects and stacks

### This chapter covers:

## Monitoring a website’s uptime serverlessly

### Sketching it out

### Choosing the right set of cloud resources

## Creating the project and stack

## Thinking in projects and stacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Developing the application

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Scheduling and handling an event

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Adding an HTTP client

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Wiring up notifications

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Posting to Slack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Taking stock

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Making the program configurable

## Adding a production stack

### Listing and creating stacks

### Deploying the production stack

## Stepping back, and looking ahead

## Tidying up

## Summary

# Chapter 4: Configuration and secrets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### This chapter covers:

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## A quick recap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Introducing Pulumi config

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Configuring the dev stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Configuring with more than just strings

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Introducing Pulumi secrets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Updating the program to use the secret

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Keeping secrets secret

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Configuring with AWS environment variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

## Updating and deploying the production stack

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Configuring items individually

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Configuring multiple items

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/pulumi.

### Copying configuration values between stacks

## Tidying Up

### Deleting stack-configuration items

### Deleting stacks

## Summary

# Part 2: Building and managing applications with Pulumi

# Chapter 5: Static Websites

### This chapter covers:

## First, you need a website

# Appendix A: Setting Things Up