Draft. Please do not include this page.

Single chapter compiled via LaTeX.

Shing Lyu, June 11, 2022. Git commit: 496bf14

Notes for the production team

Whitespace marker

You will see pink underline markers like these: `a␣b␣c`. One marker indicates one whitespace. So there is exactly one whitespace between "a" and "b" and one whitespace between "b" and "c". You'll also see them in code blocks like this:

```
#[derive(StructOpt)]
struct␣Options␣{
␣␣␣␣//␣...
␣␣␣␣#[structopt(short␣=␣"f",␣long␣=␣"file",␣parse(from_os_str))]
␣␣␣␣///␣Load␣the␣cat␣picture␣from␣the␣specified␣file
␣␣␣␣catfile:␣Option<std::path::PathBuf>,
}
```

These markers are only for you to see how many whitespaces there are. Please keep the exact number of whitespaces for inline code and code blocks. **Please do NOT include the markers during layout**.

The layout output should look like this: `a b c`

```
#[derive(StructOpt)]
struct Options {
    // ...
    #[structopt(short = "f", long = "file", parse(from_os_str))]
    /// Load the cat picture from the specified file
    catfile: Option<std::path::PathBuf>,
}
```

Auto line wrapping marker

You'll also see arrows near the end of the line and beginning of the second line:

```
This is a very very very very very very very very very very ↵
    ↳very very very long line that was wrapped automatically.
```

This indicates the original line was too long and it was automatically wrapped. **Please include these arrows.**

CHAPTER 6

Going Serverless

You built a website, a REST API and WebSocket servers in the previous chapters. They work fine when you run them on your local machine and only test with low traffic. But when you need to make them publicly accessible, managing the server becomes a headache. Traditionally you'll have to buy physical servers and run your applications on them. You'll have to take care of every aspect of it, from keeping the operating system and system libraries up-to-date, making sure failed hardware is replaced, and keeping the servers powered even when there is a power outage. Unless you have a big budget and an operations team, this is not a fun job.

If you don't want to handle these troubles yourself, there are many companies that let you outsource their servers. For example, 3rd-party web hosting and virtual private server (VPS) service have existed for a long time. Nowadays, you also have many Infrastructure-as-a-service (IaaS) and Platform-as-a-service (PaaS) providers you can choose from. They manage the servers for you and provide different levels of abstraction, so you can focus on your application. Serverless computing pushes this idea to the extreme. With serverless computing, you just write the functions that handle the business logic. The hardware, OS, and the language runtime are all handled by the service provider. You can also connect them to managed databases, message queues and file storage which are also fully-managed by the service provider.

Most of the big cloud providers offer some form of serverless computing capability. In this chapter, you'll use Amazon Web Service (AWS)'s Lambda as the computation platform. You'll also use DynamoDB, a fully-managed NoSQL database from AWS.

6.1 What are you building

In this chapter, you are going to rebuild the Catdex REST API again in a serverless-fashion. You'll learn to build the following features:

- Running Rust code on an AWS Lambda
- Create a REST API endpoint using the Serverless Application Framework.
- Use the `lambda_http` crate to handle API requests coming from AWS API Gateway.
- Read from DynamoDB through the `Rusoto` AWS SDK.

- Writing to DynamoDB to create a new cat.
- Uploading images directly to S3, an object storage service for storing files.
- Serving the frontend from S3.
- Enable Cross-Origin Resource Sharing (CORS) so the frontend can access the API.

6.2 Registering an AWS account

Since you are going to run the service on Amazon Web Service (AWS), you need to register an account. Visit https://aws.amazon.com in your browser and click the "Create an AWS Account". Follow the steps and sign up for an account. You might need to provide a credit card during the process.

AWS provides one year of free-tier services (usage limitations apply) when you sign up for the first time. This covers most of the services you are going to use: Lambda, DynamoDB, S3. Therefore, you should be able to run most of the examples with minimal to no cost. But remember to clean up all the resources after you finished testing.

6.3 Hello world in Lambda

AWS Lambda is a service that allows you to run code without provisioning any server. AWS manages the underlying hardware, networking, operating system and runtime. As a developer, you only upload a piece of code and it can run and scale automatically. A lambda function can be triggered manually (via the web console or AWS CLI), or by events generated by other AWS services. For REST APIs, it's common to use API Gateway or Application Load Balancer to handle the request and trigger the lambda.

AWS Lambda frees the developers from configuring and managing the servers, so they focus on the code. You are charged by the compute time you consume (in 100ms chunks), so if your function is sitting idle, you don't need to pay anything. Lambda can also scale automatically. If you use Lambda to power a REST API, it can automatically spin up more lambda instances when traffic is high.

AWS Lambda provides many language runtimes like Java, Go, PowerShell, Node.js, C#, Python, and Ruby. It also provides a Runtime API so you can build your custom runtime[1]. AWS has released an experimental runtime for Rust using this runtime mechanism, so you can run Rust code on Lambda.

■ **NOTE**

The underlying technology that powers AWS Lambda is Firecracker VM[2]. Interestingly, Firecracker VM is written in Rust. So even if you write lambdas in other languages, your code is still powered by Rust. The project is released as an open source

[1] See https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html
[2] https://firecracker-microvm.github.io/

project by AWS. You can find ways to contribute to it by visiting its GitHub repository: https://github.com/firecracker-microvm/firecracker.

The first thing you are going to look at is the Hello World lambda from the AWS official blog[3]. You are going to deploy this lambda and test it through the AWS management console. First, create a Rust project by running `cargo new serverless-hello-world --bin` and `cd` into the `serverless-hello-world` folder. Then you need to add the `lambda_runtime` crate by `cargo add lambda_runtime`. There are also some extra dependencies for JSON serialization/deserialization and logging, so you need to add them to the dependency section of `Cargo.toml` as well:

```
lambda_runtime = "0.2.1"
serde = "^1"
serde_json = "^1"
serde_derive = "^1"
tokio = "0.1"
log = "^0.4"
simple_logger = "^1"
simple-error = "^0.1"
```

When you use AWS Lambda custom runtime, the Lambda service will look for a binary named `bootstrap` and execute it when the lambda is triggered. Therefore, you need to add the following lines to your `Cargo.toml`, so when you run `cargo build` it will compile `src/main.rs` as a binary called `bootstrap`:

```
[[bin]]
name = "bootstrap"
path = "src/main.rs"
```

Now your `Cargo.toml` file should look like Listing 6.3.
TODO: Add filename

```
[package]
name = "serverless-hello-world"
version = "0.1.0"
authors = ["Shing Lyu"]
edition = "2018"

# See more keys and their definitions at
# https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
lambda_runtime = "0.2.1"
serde = "^1"
serde_json = "^1"
```

[3]https://aws.amazon.com/blogs/opensource/rust-runtime-for-aws-lambda/

```
serde_derive = "^1"
tokio = "0.1"
log = "^0.4"
simple_logger = "^1.11"
simple-error = "^0.1"

[[bin]]
name = "bootstrap"
path = "src/main.rs"
```

Now with the dependencies in place, you can look at the code. Copy Listing 6.3 into `src/main.rs`.

TODO: Add filename

```
use std::error::Error;

use lambda_runtime::{error::HandlerError, lambda, Context};
use log::{self, error};
use serde_derive::{Deserialize, Serialize};
use simple_error::bail;
use simple_logger::SimpleLogger;

#[derive(Deserialize)]
struct CustomEvent {
    #[serde(rename = "firstName")]
    first_name: String,
}

#[derive(Serialize)]
struct CustomOutput {
    message: String,
}

fn main() -> Result<(), Box<dyn Error>> {
    SimpleLogger::new().with_level(log::LevelFilter::Debug).init
    ↪ ()?;
    lambda!(my_handler);

    Ok(())
}

fn my_handler(
    e: CustomEvent,
    c: Context,
) -> Result<CustomOutput, HandlerError> {
    if e.first_name == "" {
        error!(
```

```
            "Empty first name in request {}",
            c.aws_request_id
        );
        bail!("Empty first name");
    }

    Ok(CustomOutput {
        message: format!("Hello, {}!", e.first_name),
    })
}
```

In the `main` function, you can see you set up a `simple_logger` for logging. The log generated by the lambda will be collected in AWS CloudWatch, AWS's logging and metrics service. Then you use the `lambda!` macro to mark the `my_handler()` function as the lambda handler. That means when an event triggers the lambda, it will call the `my_handler()` function and pass the event and some context information.

A lambda can handle different types of events from different sources, like API Gateway, SQS, S3 or DynamoDB stream. Each event has its own structure, so you'll have to write your code accordingly. In this example, you are going to define a custom event format `struct CustomEvent`, which contains a single field called `first_name`. The struct implements the `Deserialize` trait from serde. You also use serde's `rename` feature to rename the field from `firstName` (JSON convention) to `first_name` (Rust convention). Similarly, you define a `CustomOutput` as the lambda's output format.

The `my_handler` function is very straightforward, it first checks if the `first_name` field in the input event is non-empty, otherwise it stops immediately with `bail!()`. If the `first_name` is non-empty, it returns a custom output with a message `"Hello, <first_name>!"`.

Besides the event, the handler also receives a `Context` struct. The `Context` contains information like the function name, function version, the request ID and much more. The ID is unique for each invocation request, so it's very useful to include it in the log to distinguish the logs from different invocations.

There is one more thing to set up before you can compile the code: You need to install a new compile target `x86_64-unknown-linux-musl`[4] by running:

```
rustup target add x86_64-unknown-linux-musl
```

■ **NOTE** If you are cross-compiling on Linux, installing the Musl target is enough. But if you are compiling on MacOS, you also need to install an extra library and to configure the linker using the instructions here: https://aws.amazon.com/blogs/opensource/rust-runtime-for-aws-lambda/

[4] Musl is a lightweight C standard library implementation. It allows us to build fully-statically-linked binaries.

Finally, you can compile `src/main.rs` into `./target/x86_64-unknown-linux-musl/release/bootstrap`, then zip it into a zip file named `rust.zip`:

```
cargo build --release --target x86_64-unknown-linux-musl
zip -j rust.zip ./target/x86_64-unknown-linux-musl/release/
    ↪ bootstrap
```

To test the lambda, you need to do the following:

1. Visit the AWS Management Console https://aws.amazon.com/console/ from your browser. Log in with your credentials.
2. In "Find Services", find "Lambda" and click on the result.
3. In the Lambda console (Figure 6.1), click "Create function"
4. In the function creation page, select "Author from scratch". Set the "Function name" as "hello-world". Select "Custom runtime - Provide your own bootstrap on Amazon Linux 2" in the "Runtime" field. Then click "Create function".
5. Once you are redirected to the `hello-world` function's page, scroll down to the "Function code" section and click "Actions". Then select the "Upload a .zip file" option and upload the `rust.zip` you created previously. (Figure 6.2).

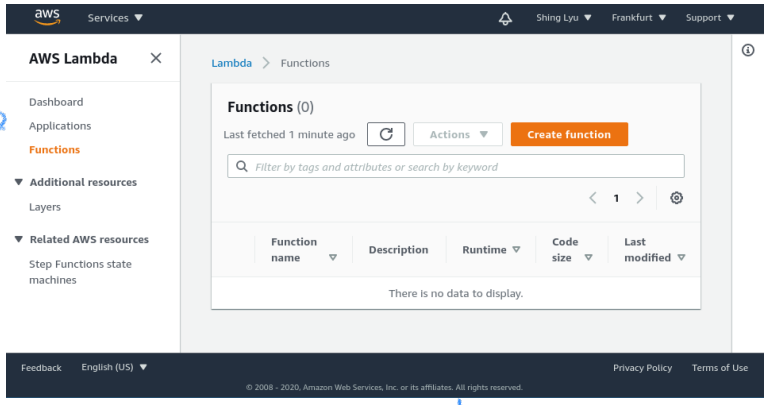

Figure 6.1: Lambda console

To test this lambda, you can click on the "Test" button on the lambda page (Figure 6.3). If it's the first time you are testing it, AWS console will prompt you to create a test event (Figure 6.4). You can give it an "Event name" called "test" add a test event body like so:

```
{
  "firstName": "Shing"
}
```

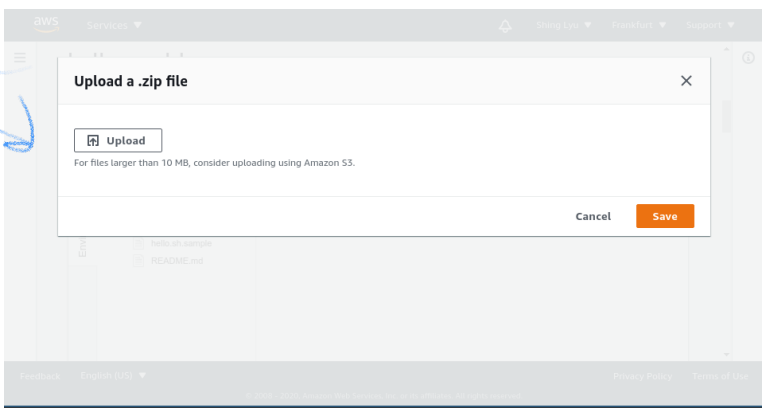

Figure 6.2: Uploading the zip file

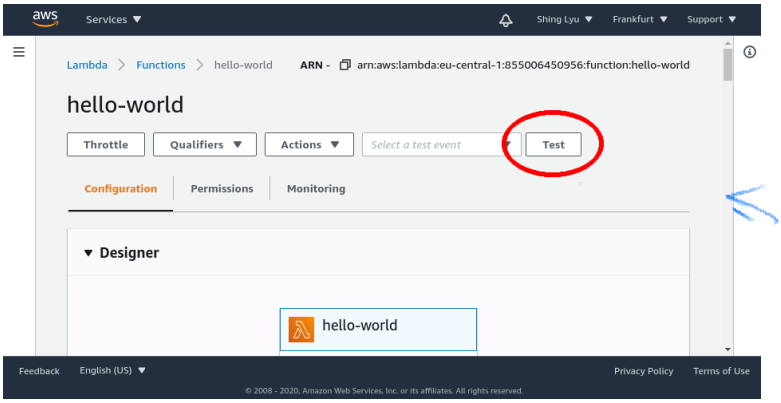

Figure 6.3: ~~The test button~~

Then click "Create". Now the dropdown menu will show a test event named "test". If you click "Test" again, the test event will be sent to the lambda and you should see an output and some logs as shown in Figure 6.5.

You can see the lambda is working just as you expected.

6.4 Making a REST API with Lambda

The lambda in the previous section can't serve HTTP requests just yet. To be able to receive HTTP requests, you need to put an API Gateway REST API in front of it. The complete architecture would look like Figure 6.6.

The REST APIs are served through API Gateway. API Gateway handles the HTTP connection, and triggers a lambda for each request. If you are serving two APIs (e.g. `GET /cats` and `POST /cat`), you can have one lambda per API. The database you choose is AWS DynamoDB. DynamoDB is a NoSQL database that is performant and fully-managed. You can directly access DynamoDB from the lambdas with the AWS SDK.

You also have a few frontend files: HTML, CSS, and JavaScript. These files can be served separately from an S3 bucket. An S3 bucket is an object store in which you can store files. S3 also has an option to serve your files through HTTP like a static web server.

The URLs exposed by API Gateway and S3 static file hosting are auto-generated by AWS, so you can't really customize it. You can add a CloudFront CDN and add a custom domain name through Route53, a managed DNS service. This way you have full control over what domain name the API and static files use. But this part is beyond the scope of the book and it's not related to Rust, so you'll not show them here. You can consult the official AWS documentation on how to do this.

6.5 Using the Serverless framework

Configuring all these resources through the web console is not an easy task. It's hard to keep track of what is actually deployed in production. It's also hard to re-create the whole stack from scratch if you destroy it by accident. Infrastructure-as-code (abbreviated as IaC) is a concept to solve this problem. You define your infrastructure and the configuration through code, and the IaC tool of your choice will configure everything according to your code. If you make any change to the definition, the change can also be reflected with a quick deployment. This way you can version control your infrastructure like code, and to fix or re-create the whole stack is just a simple deployment away.

Each cloud platform has its IaC service, like AWS CloudFormation, Azure Resource Manager and Google Cloud Platform Deployment Manager. There are also third-party services that can work cross-platform, like Terraform and Pulumi. In this chapter, you are going to use the Serverless Application Framework, or Serverless for short. Serverless not only manages the infrastructure (using AWS CloudFormation under the hood), but it also helps you manage the whole lifecycle of the application like testing, packaging the

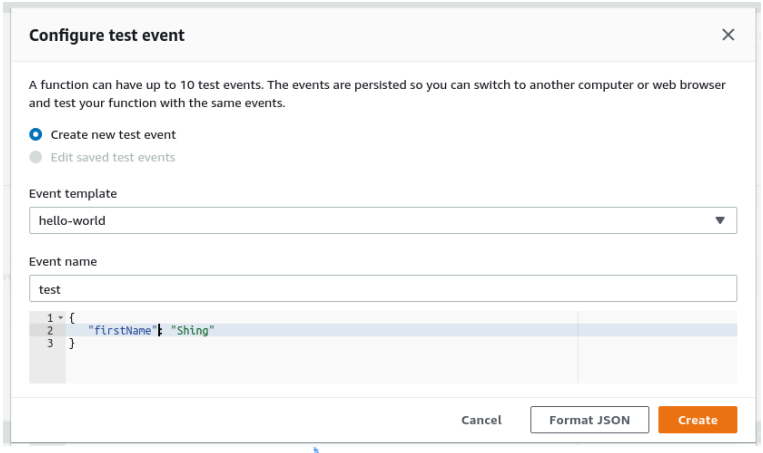

Figure 6.4: Creating a test event

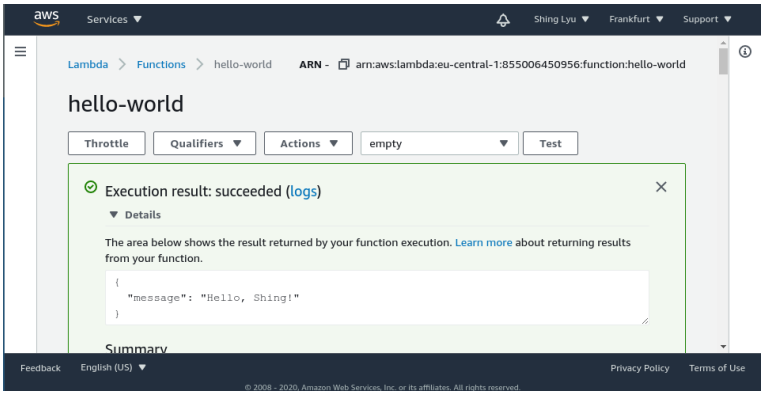

Figure 6.5: Test output

lambda code and logging.

You are going to use the pre-built template `serverless-aws-rust-multi` as the basis of the new serverless catdex. A Serverless framework template contains a Serverless framework configuration and example code to get you started quickly. Because you need to create multiple lambda functions, one for each API, so you choose the `serverless-aws-rust-multi`. This template uses cargo workspaces to manage multiple packages in one repository. To use the template, you first need to install the latest version of Node.js and NPM. The `npm` tool also comes in many different versions. You can use the Node Version Manager (`nvm`[5]) to easily jump between versions. `Nvm` also has an installation script:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/
    ↪ install.sh | bash
```

Once the script finishes successfully, `nvm` should add some commands to your shell profile/configuration file (e.g. `~/.bash_profile`, `~/.bashrc`, `~/.zshrc`) so the `nvm` command can become available:

```
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && \
  printf %s "${HOME}/.nvm" || \
  printf %s "${XDG_CONFIG_HOME}/nvm")"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads
    ↪ nvm
```

If they are not there, please add it yourslef and restart your shell.

Once `nvm` is ready, install the latest Node.js and NPM (I use `v13.11.0`):

```
nvm install v13.11.0
```

Finally, you can run the following command to use the template to create a new project called "serverless-catdex":

```
npx serverless install \
  --url https://github.com/softprops/serverless-aws-rust-multi \
  --name serverless-catdex
```

The `npx` command comes with `npm`. It allows you to run a one-off command (in the case, 'serverless') without installing the package explicitly.

The `serverless install` command creates a project folder named `serverless-catdex`. You can `cd` into the folder and run `npm install` to install all the dependencies.

At the center of this generated project folder is the `serverless.yml` file (Listing 6.5). This file is the main configuration file for the Serverless framework.

TODO: Add filename

```
# ...
service: serverless-catdex
```

[5] https://github.com/nvm-sh/nvm

```
provider:
  name: aws
  runtime: rust
  memorySize: 128
# you can overwrite defaults here
#  stage: dev
#  region: us-east-1

# you can add statements to the Lambda function's IAM Role here
#  ...

package:
  individually: true

plugins:
  - serverless-rust

functions:
  hello:
    # handler value syntax is `{cargo-package-name}.{bin-name}`
    # or `{cargo-package-name}` for short when you are building
    ↪ a
    # default bin for a given package.
    handler: hello

  # ...

  world:
    handler: world
    events:
      - http:
          path: /
          method: get

# you can add CloudFormation resource templates here
#resources:
#  ...
```

Most of the field names in the `serverless.yml` file are self-explanatory. However, there are a few fields you want to highlight:

- `provider.region`: You can choose a region close to you to reduce the network latency. For example, I use `eu-central-1` (Frankfurt, Germany).
- `plugins: serverless-rust`: This allows us to use the Rust runtime for the lambda.

- `functions`: There are two functions, `hello` and `world`, each is a cargo package in the top-level folder of the same name.

The `hello` lambda is a free-standing one that has no event trigger configured. The `world` lambda, on the other hand, receives events from API Gateway, which is specified by the `events.http` configuration. Also notice that the `world` lambda uses the `lambda_http` crate instead of the `lambda` crate. The `lambda_http` crate is a specialized crate for building API Gateway event focused lambda.

The serverless framework needs to have access to your AWS account so it can create resources on your behalf. You can follow the step-by-step instructions to set it up: https://www.serverless.com/framework/docs/providers/aws/guide/credentials/. To summarize, you need to

1. Log in to the AWS console with your root account and go to the Identity & Access Management (IAM) page.
2. Create a new user with programmatic access. Attach the `AdministratorAccess` policy[6] to it.
3. Copy the newly-created user's Access Key and Secret Access Key.

Then run the following command to give serverless access:

```
npx serverless config credentials \
  --provider aws \
  --key <YOUR-ACCESS-KEY-HERE> \
  --secret <YOUR-SECRET-ACCESS-KEY-HERE>
```

6.6 Building the `/cats` API

You are finally ready to build a REST API on AWS lambda and the Serverless framework. You are going to re-purpose the `world` lambda to be the `/cats` API. First, let's rename the folder:

```
mv world cats
```

Then you need to change the package name in `cats/Cargo.toml`:

```
[package]
name = "cats"
#...
```

In the root-level folder, you need to also change the `Cargo.toml` file and `serverless.yml`.

[6] It's a bad idea to give your IAM use administrator access in production. You should grant permission based on the principle of least privilege. But for demonstration purpose it's easier to give full access.

```
# Cargo.toml
[workspace]
members = ["hello", "cats"]
```

```
# ...
functions:
  hello:
    handler: hello

  cats: # Rename this
    handler: cats # And this
    events:
      - http:
          path: /cats # Add this
          method: get
```

You need a database to store the cat's information. You could use Amazon Relational Database Service (RDS) to run a PostgreSQL database, so you can reuse the same code from the previous chapter. However, in order to show how AWS SDK works, you are going to use DynamoDB, a NoSQL database provided by AWS.

To provision the database, you need to declare it in the `serverless.yml` file, as shown in Listing 6.6.

TODO: Add filename

```
service: serverless-catdex
provider:
  name: aws
  runtime: rust
  memorySize: 128
  region: eu-central-1
  iamRoleStatements:
      - Effect: "Allow"
        Action:
          - "dynamodb:Scan"
        Resource:
          Fn::Join:
            - ""
            - - "arn:aws:dynamodb:*:*:table/"
              - "Ref": "CatdexTable"

# ...
functions:
# ...

resources:
  Resources:
    CatdexTable:
```

```
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: shing_catdex
        AttributeDefinitions:
          - AttributeName: name
            AttributeType: S
        KeySchema:
          - AttributeName: name
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
```

Let's first focus on the `resources` section. You declared a resource called `CatdexTable`, which has the `Type AWS::DynamoDB::Table`. Serverless framework will use CloudFormation, an infrastructure-as-code service, to create the DynamoDB for us. You also defined a few properties of the table, like the name `shing_catdex` that is used to identify the table in AWS. Due to the design of DynamoDB, the data is partitioned into multiple physical storage units. Therefore, you must define a unique *partition key* for each item so they can be partitioned properly. You define an attribute called `name` and mark it as the partition key using `KeySchema`. Finally, you provision the desired throughput of the table. Since this is just a demo database, you set both the read and write capacity to 1 to minimize cost.

■ **NOTE** CloudFormation is an infrastructure-as-code service. It allows you to declare the AWS resources (i.e., your infrastructure) you need in a JSON or YAML format template. CloudFormation will create, update or delete the resources on your behalf to match the declared template. This allows you to manage complex infrastructure without the need to click hundreds of buttons on the AWS console. You can also utilize all the coding best practices like version control and code review on your infrastructure configuration.

By default, the lambda does not have permission to use the DynamoDB. You need to explicitly allow this using AWS Identity and Access Management (IAM). Therefore, in the `provider` section, you add a `iamRoleStatement`, which grants the lambda permission to do a `dynamodb::Scan` operation on the `CatdexTable`. Now you can run `npx serverless deploy` and serverless will create the DynamoDB in your AWS account.

To access the DynamoDB from code, you need to use the Rusoto[7] AWS SDK. Because AWS has over 175 services[8], Rusoto has one crate for each service. So you need to add the `rusoto_core` and `rusoto_dynamodb` crates to the `cats` crate. You can achieve

[7] https://github.com/rusoto/rusoto

[8] As of August, 2020

this by:

```
cd cats
cargo add rusoto_core rusoto_dynamodb
```

Once the dependencies are ready, you can finally write the code for `cats/src/main.rs` Listing 6.6.

TODO: Add filename

```
use lambda_http::http::{Response, StatusCode};
use lambda_http::{
    handler, lambda, Context, IntoResponse, Request,
};
use rusoto_core::Region;
use rusoto_dynamodb::{DynamoDb, DynamoDbClient, ScanInput};
use serde_json::json;
use std::collections::HashMap;

type Error = Box<dyn std::error::Error + Sync + Send + 'static>;

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda::run(handler(cats)).await?;
    Ok(())
}

async fn cats(
    _: Request,
    _: Context,
) -> Result<impl IntoResponse, Error> {
    let client = DynamoDbClient::new(Region::EuCentral1);

    let scan_input = ScanInput {
        table_name: "shing_catdex".to_string(),
        limit: Some(100),
        ..Default::default()
    };

    let response = match client.scan(scan_input).await {
        Ok(output) => match output.items {
            Some(items) => json!(items
                .into_iter()
                .map(|item| item
                    .into_iter()
                    .map(|(k, v)| (k, v.s.unwrap()))
                    .collect())
                .collect::<Vec<HashMap<String, String>>>())
            .into_response(),
```

```
            None => Response::builder()
                .status(StatusCode::NOT_FOUND)
                .body("No cat yet".into())
                .expect("Failed to render response"),
        },
        Err(error) => Response::builder()
            .status(StatusCode::INTERNAL_SERVER_ERROR)
            .body(format!("{:?}", error).into())
            .expect("Failed to render response"),
    };

    Ok(response)
}
```

The `main()` function simply calls `lambda::run(handler(cats))` and awaits on it. The `cats` function is where the magic happens. The first thing you do in the `cats` function is creating a `DynamoDBClient` provided by Rusoto[9]. You want to use `client.scan()` to get a list of cats. Because the `scan` operation allows you to filter the results, you need to specify those filtering criteria as a `ScanInput`, and pass it to `client.scan()`. The `ScanInput` specifies the `table_name` you want and the `limit` of 100, so you'll get at most 100 cats. For the other optional fields in `ScanInput` you simply use the default values.

■ **NOTE** DynamoDB supports two major ways for querying data: query and scan. When you query, you need to specify the partition key so DynamoDB can directly find the item. Scan, on the other hand, needs to scan through the whole table. You can specify filtering criteria to further refine the result.

Scan is significantly slower than query, but it's useful for situations when you don't know the partition key in advance. If you already know the partition key you are trying to find, always prefer query over scan.

You then call `client.scan()` and `await` on the result. You use a `match` block to handle possible errors. If the scan result is an `Err`, then something unexpected has happened. You use the `Response::builder()` to construct a 500 Internal Server Error.

If the scan returns `Ok`, then you can see if the output (which has the type `ScanOutput`) contains anything in its `items` field. If there is none, you return 404 Not Found. If there are some items, you need to convert their format before you return them. The `items` field has the structure[10]:

[9] You hard-coded the region to `Region::EuCentral1` to match the configuration in `serverless.yml`. However, if you are deploying the same API in multiple regions for resiliency, you should pass the region by environment variables or other dynamic ways into the lambda.

[10] The `s` key stands for "string" type

```
[
  {
    image_path: {
      s: Some("/image/persian.png")
      //...
    },
    "name":{
      s: Some("Persian")
      //...
    }
  }
]
```

With a series of `map` and `collect`, you can convert it to:

```
[
  {
    "image_path": "/image/persian.png",
    "name": "Persian"
  }
]
```

Finally, you use `json!()` to convert it to JSON (`serde_json::value::Value`). And because the `lambda_http` crate implements the `IntoResponse` trait for `Value`, you can convert it to an HTTP response easily by calling `.into_response()`.

To test the new API, run `npx serverless deploy`. In the end of the log, you will see the URL for your new API:

```
$ npx serverless deploy
Serverless: Building Rust hello func...
Serverless: Running containerized build
//...
Serverless: Packaging service...
Serverless: Creating Stack...
Serverless: Checking Stack create progress...
.....
Serverless: Stack create finished...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Uploading service hello.zip file to S3 (1001.3 KB)
    ↪...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
.....
Serverless: Stack update finished...
Service Information
```

```
service: serverless-catdex
stage: dev
region: eu-central-1
stack: serverless-catdex-dev
resources: 10
api keys:
  None
endpoints:
  GET - https://abc0123def.execute-api.eu-central-1.amazonaws.
    ↪ com/dev/cats
functions:
  hello: serverless-catdex-dev-hello
layers:
  None
```

You can use cURL to test it:

```
curl https://abc0123def.execute-api.eu-central-1.amazonaws.com/
    ↪ dev/cats
```

But for now, you don't have any data in the database, so you should see it return an empty object.

6.7 Building the upload API

Let's build the `POST /cat` API so you can create a new cat and upload a new image. Since the `hello` lambda is not useful, let's remove the folder (`rm -rf hello`) and remove it from the root-level `Cargo.toml` and `serverless.yml`.

To create a new lambda for the new API, you can simply copy the `cats` API by `cp -r cats cat_post`. Then you need to change or add the name `cat_post` to a few places:

- Root-level `Cargo.toml`

```
# Cargo.toml
[workspace]
members = ["cats", "cat_post"]
```

- `serverless.yml`

```
# ...
functions:
  cats:
    handler: cats
    events:
      - http:
          path: /cats
```

```
          method: get
      cat_post:
        handler: cat_post
        events:
          - http:
              path: /cat
              method: post
```

- `cats_post/Cargo.toml`

```
[package]
name = "cat_post"
//...
```

In this new API, you need to write data to the DynamoDB, so you need to add the `dynamodb:PutItem` permission to the IAM role in `serverless.yml`:

```
provider:
  # ...
  iamRoleStatements:
      - Effect: "Allow"
        Action:
          - "dynamodb:Scan"
          - "dynamodb:PutItem"
        Resource:
          Fn::Join:
            - ""
            - - "arn:aws:dynamodb:*:*:table/"
              - "Ref": "CatdexTable"
```

With the new permission in place, you can write the code in `cat_post/src/main.rs` (Listing 6.7). The new code uses the `serde` and `serde_json` crates for JSON serialization/deserialization, so remember to run `cargo add serde serde_json` in the `cat_post` folder.

TODO: Add filename

```
use lambda_http::http::{Response, StatusCode};
use lambda_http::{
    handler, lambda, Context, IntoResponse, Request, RequestExt,
};
use rusoto_core::Region;
use rusoto_dynamodb::{
    AttributeValue, DynamoDb, DynamoDbClient, PutItemInput,
};
use serde::Deserialize;
use serde_json::json;
use std::collections::HashMap;
```

```
type␣Error␣=␣Box<dyn␣std::error::Error␣+␣Sync␣+␣Send␣+␣'static>;

#[derive(Deserialize)]
struct␣RequestBody␣{
␣␣␣␣name:␣String,
}

#[tokio::main]
async␣fn␣main()␣->␣Result<(),␣Error>␣{
␣␣␣␣lambda::run(handler(cat_post)).await?;
␣␣␣␣Ok(())
}

async␣fn␣cat_post(
␣␣␣␣request:␣Request,
␣␣␣␣_:␣Context,
)␣->␣Result<impl␣IntoResponse,␣Error>␣{
␣␣␣␣let␣body:␣RequestBody␣=␣match␣request.payload()␣{
␣␣␣␣␣␣␣␣Ok(Some(body))␣=>␣body,
␣␣␣␣␣␣␣␣_␣=>␣{
␣␣␣␣␣␣␣␣␣␣␣␣return␣Ok(Response::builder()
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣.status(StatusCode::BAD_REQUEST)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣.body("Invalid␣payload".into())
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣.expect("Failed␣to␣render␣response"))
␣␣␣␣␣␣␣␣}
␣␣␣␣};

␣␣␣␣let␣client␣=␣DynamoDbClient::new(Region::EuCentral1);

␣␣␣␣let␣mut␣new_cat␣=␣HashMap::new();
␣␣␣␣new_cat.insert(
␣␣␣␣␣␣␣␣"name".to_string(),
␣␣␣␣␣␣␣␣AttributeValue␣{
␣␣␣␣␣␣␣␣␣␣␣␣s:␣Some(body.name.clone()),
␣␣␣␣␣␣␣␣␣␣␣␣..Default::default()
␣␣␣␣␣␣␣␣},
␣␣␣␣);

␣␣␣␣let␣put_item_input␣=␣PutItemInput␣{
␣␣␣␣␣␣␣␣table_name:␣"shing_catdex".to_string(),
␣␣␣␣␣␣␣␣item:␣new_cat,
␣␣␣␣␣␣␣␣..Default::default()
␣␣␣␣};

␣␣␣␣match␣client.put_item(put_item_input).await␣{
```

```
        Ok(_) => (),
        _ => {
            return Ok(Response::builder()
                .status(StatusCode::INTERNAL_SERVER_ERROR)
                .body("Something went wrong when writing to the
    ↪ database".into())
                .expect("Failed to render response"))
        }
    }

    Ok(json!(format!("created cat {}", body.name))
        .into_response())
}
```

The `cat_post()` function does a few things for now:

1. Extract the request body (i.e., payload) to get the cat's name
2. Create the DynamoDB client
3. Create a `PutItemInput`, which will create the new cat in the database when passed to `client.put_item()`
4. Call `client.put_item()` to create the DynamoDB item

In the example in the previous chapter, you uploaded the cat's image through the API. However, API Gateway has a payload size limit of 10 MB, so the image needs to be smaller than that. In order to overcome that, we're going to use the S3 pre-signed URL, which you'll discuss shortly. For now this example doesn't contain the file upload part.

Notice that the `cat_post()` function now takes an event (the first parameter) of the type `Request`; this is provided by `lambda_http` crate. You can call `request.payload()` to get the request body. You expect the body to have the form:

```
{
  "name": "Persian"
}
```

So you define a `RequestBody` struct, which derives the `serde::Deserialize` trait, to tell Rust how to deserialize it. When you call `request.payload()`, if the return value is a `Some(RequestBody)`, you can assign it to a variable `body`.

Next, you create the DynamoDB client and prepare the `PutItemInput`. The `PutItemInput` expects the table name and a new item (as a `HashMap`). Therefore, you use the cat's name specified in `body` for the new cat's name. For every place that might fail (e.g. parsing payload, calling `put_item()`), you use `match` to handle the errors and return an appropriate HTTP response.

6.8 Uploading the image using S3 pre-signed URL

As mentioned, API Gateway has a 10 MB request size limit, so you can't upload image files larger than that. To overcome this limitation, you can use the S3 pre-signed PUT URL. You can use the AWS API to upload a file to S3, but since the S3 bucket is private by default, you need to provide valid credentials so AWS can verify your identity and check if you have the proper access to the bucket. However, there is no secure way to store the AWS credentials on the frontend page. A pre-signed URL can solve this problem. A pre-signed URL allows anyone to upload files to the pre-defined S3 location within a limited time, without the need to provide AWS credentials. When creating the pre-signed URL, you provide AWS credentials, so the user of the URL will get the same permission as the credentials used to sign it. The pre-signed URL generation takes place in the backend (i.e., in the lambda function), so the credentials is never exposed to the frontend.

In the use case, you can let the frontend call the `POST /cat` endpoint, to create the cat in the DynamoDB. Then, the `POST /cat` API needs to generate a pre-signed URL and return it to the frontend. Then the frontend uses this pre-signed URL to upload the cat image directly to S3. Figure 6.7 shows a sequence diagram for this flow. Since this is a demo, the image will then directly be served through the S3 built-in server. But in production, you might want to upload the file to a separate bucket, then sanitize the image, before putting it into the bucket that serves the static files.

This approach has a few advantages over uploading through API Gateway and Lambda. First, S3 allows you to upload files up to 5 GB[11]. Second, you save bandwidth going through API gateway, and you also saved processing time and memory usage in the lambda, potentially saving some costs.

To be able to generate a pre-signed URL, you need to add the `rusoto_s3` and `rusoto_credentials` crates by running `cargo add rusoto_s3 rusoto_credentials`. Then you need to create an S3 bucket in `serverless.yml`:

```
# ...
resources:
  Resources:
    CatdexTable:
      # ...
    FrontendBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: shing-catdex-frontend
        AccessControl: Private
```

[11] If you use the multipart upload the limit can be increased to 5 TB.

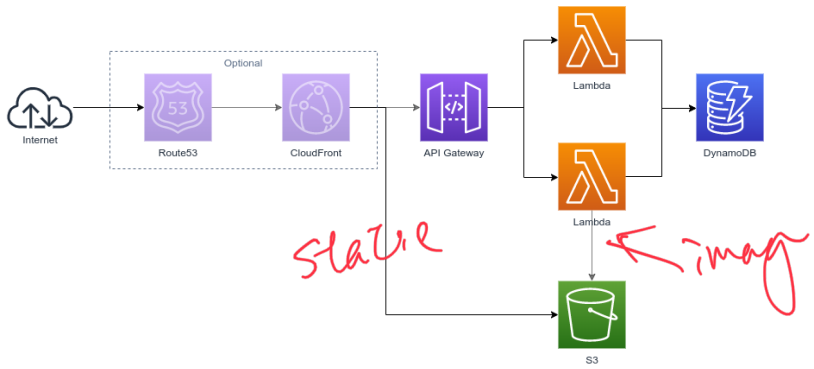

Figure 6.6: A simple REST API architecture

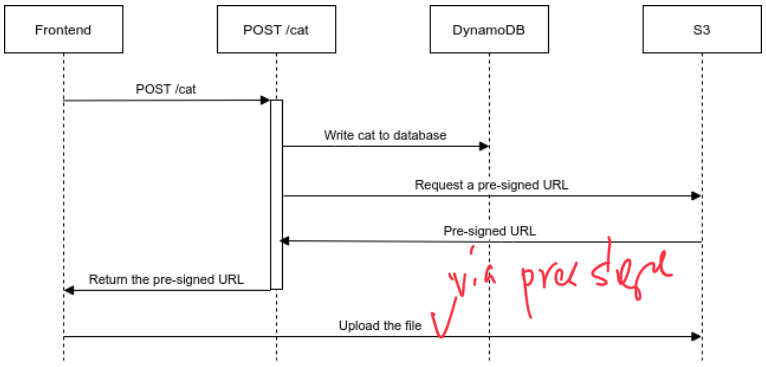

Figure 6.7: Sequence diagram for adding a new cat using the pre-signed URL

■ **TIP** An S3 bucket name must be globally unique, even across accounts. So you need to choose a different bucket name, e.g. your-name-catdex-frontend.

Because the pre-signed URL will get the same permission as the AWS role which creates it, you need to add the `PutObject` permission to the IAM role so the pre-signed URL can upload files. Change the IAM role in `serverless.yml` like so:

```
␣␣iamRoleStatements:
␣␣␣␣␣␣-␣Effect:␣"Allow"
␣␣␣␣␣␣␣␣Action:
␣␣␣␣␣␣␣␣␣␣␣␣-␣"dynamodb:Scan"
␣␣␣␣␣␣␣␣␣␣␣␣-␣"dynamodb:PutItem"
␣␣␣␣␣␣␣␣Resource:
␣␣␣␣␣␣␣␣␣␣Fn::Join:
␣␣␣␣␣␣␣␣␣␣␣␣-␣""
␣␣␣␣␣␣␣␣␣␣␣␣-␣-␣"arn:aws:dynamodb:*:*:table/"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣"Ref":␣"CatdexTable"
␣␣␣␣␣␣-␣Effect:␣"Allow"
␣␣␣␣␣␣␣␣Action:
␣␣␣␣␣␣␣␣␣␣␣␣-␣"s3:PutObject"
␣␣␣␣␣␣␣␣␣␣␣␣-␣"s3:PutObjectAcl"
␣␣␣␣␣␣␣␣Resource:
␣␣␣␣␣␣␣␣␣␣Fn::Join:
␣␣␣␣␣␣␣␣␣␣␣␣-␣""
␣␣␣␣␣␣␣␣␣␣␣␣-␣-␣"arn:aws:s3:::"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣"Ref":␣"FrontendBucket"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣"/*"
```

As you can see you added an `Allow` block that grants `s3:PutObject` and `s3:PutObjectAcl` permission for everything in the S3 bucket. Let's add some code to `cat_post/src/main.rs` so you can generate the pre-signed URL (Listing 6.8).

TODO: Add filename

```
use␣lambda_http::http::{HeaderValue,␣Response,␣StatusCode};
use␣lambda_http::{
␣␣␣␣handler,␣lambda,␣Context,␣IntoResponse,␣Request,␣RequestExt,
};
use␣rusoto_core::Region;
use␣rusoto_credential::{ChainProvider,␣ProvideAwsCredentials};
use␣rusoto_dynamodb::{
␣␣␣␣AttributeValue,␣DynamoDb,␣DynamoDbClient,␣PutItemInput,
};
use␣rusoto_s3::util::PreSignedRequest;
use␣rusoto_s3::PutObjectRequest;
use␣serde::Deserialize;
use␣serde_json::json;
use␣std::collections::HashMap;
```

```
type Error = Box<dyn std::error::Error + Sync + Send + 'static>;

#[derive(Deserialize)]
struct RequestBody {
    name: String,
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda::run(handler(cat_post)).await?;
    Ok(())
}

async fn cat_post(
    request: Request,
    _: Context,
) -> Result<impl IntoResponse, Error> {
    let body: RequestBody = match request.payload() {
        Ok(Some(body)) => body,
        _ => {
            return Ok(
                //... generate the bad request response
            );
        }
    };

    let client = DynamoDbClient::new(Region::EuCentral1);

    let mut new_cat = HashMap::new();
    new_cat.insert(
        "name".to_string(),
        AttributeValue {
            s: Some(body.name.clone()),
            ..Default::default()
        },
    );
    let image_path = format!("image/{}.jpg", &body.name);
    new_cat.insert(
        "image_path".to_string(),
        AttributeValue {
            s: Some(image_path.clone()),
            ..Default::default()
        },
    );
```

```
    let put_item_input = PutItemInput {
        table_name: "shing_catdex".to_string(),
        item: new_cat,
        ..Default::default()
    };

    match client.put_item(put_item_input).await {
        Ok(_) => (),
        _ => {
            return Ok(
                //... generate internal server error response
            );
        }
    }

    let credentials =
        ChainProvider::new().credentials().await.unwrap();

    let put_request = PutObjectRequest {
        bucket: "shing-catdex-frontend".to_string(),
        key: image_path,
        content_type: Some("image/jpeg".to_string()),
        ..Default::default()
    };

    let presigned_url = put_request.get_presigned_url(
        &Region::EuCentral1,
        &credentials,
        &Default::default(),
    );

    let mut response =
        json!({ "upload_url": presigned_url }).into_response();
    Ok(response)
}
```

The first thing you changed is adding an `image_path` to the `new_cat` that will be inserted into the database. The `image_path` is hard-coded to be the `image/<cat_name>.jpg`[12]. When you generate the pre-signed URL later, the name will be fixed, so no matter what file the user uploads, it will be renamed as `image/<cat_name>.jpg` in S3.

To create the pre-signed URL, you need to create a `PutObjectRequest` first. The `PutObjectRequest` represents an attempt to upload a file to S3. You can get the pre-signed URL by calling `.get_presigned_url()` on the `PutObjectRequest`.

[12] You could also support more file extensions (e.g., `.png`, `.bmp`). But for simplicity you only allow user to upload JPEG files.

You need to provide an AWS credentials to `.get_presigned_url()`. Whoever uses this URL will get the same permission as the credentials that was used to sign it. Therefore, you get the lambda's execution role credentials using the `rusoto_credentials::ChainProvider`. Using these credentials, the user will get the `s3::PutObject` permission you defined in the `serverless.yml` file. You then add the URL to the response body so the frontend can receive it.

■ **NOTE** The `ChainProvider` will try to find the AWS credentials from multiple sources using a priority order. For example by looking for the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY_ID` environment variable, or the AWS credentials file, or the IAM instance profile in EC2. The lambda runtime will provide the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variable (for the lambda's execution role) to the lambda by default, so the `ChainProvider` can get the credentials.

You can deploy to AWS with `npx serverless deploy`. If you call the API with curl, you should receive the presigned URL in the response body:

```
$ curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"name": "Persian"}' \
  https://abc0123def.execute-api.eu-central-1.amazonaws.com/dev/
    ↪ cat

{
  "upload_url":"https://s3.eu-central-1.amazonaws.com/shing-
    ↪ catdex-frontend/
                  image/Persian.jpg?X-Amz-Algorithm=AWS4-HMAC-
    ↪ SHA256&X-Amz-credentials=...&
                  X-Amz-Date=20200819T095109Z&X-Amz-Expires=3600&X
    ↪ -Amz-Security-Token=...&
                  X-Amz-Signature=...&X-Amz-SignedHeaders=host"
}
```

You can use this URL to upload the file like so:

```
$ curl -X PUT -T persian.jpg -L -v "https://s3.eu-central-1.
    ↪ amazonaws.com/
    shing-catdex-frontend/image/Persian.jpg?X-Amz-Algorithm=AWS4
    ↪ -HMAC-SHA256&
    X-Amz-credentials=...&X-Amz-Date=20200819T095109Z&X-Amz-
    ↪ Expires=3600&
    X-Amz-Security-Token=...&X-Amz-Signature=...&X-Amz-
    ↪ SignedHeaders=host"
```

This uploads a file on the local machine named `persian.jpg`. A few fields like `X-Amz-credentials` and `X-Amx-Security-Token` are omitted because they change every time you generate a new URL.

6.9 Adding the frontend

Now with the API ready, you can also serve the HTML, JavaScript and CSS on AWS. You can upload the files to an S3 bucket and enable "static website hosting" on that S3 bucket. To automate this process, you will use the `serverless-finch` plugin. Such a plugin uploads the files for us and makes all the necessary configuration to enable static website hosting.

To install this plugin, run `npm install --save serverless-finch`. After installing the plugin, modify `serverless.yml` to Listing 6.9 so the plugin will use the `shing-catdex-frontend` S3 bucket you created before to serve the static files.

TODO: Add filename

```
# ...
plugins:
  - serverless-rust
  - serverless-finch

resources:
  Resources:
    # ...
    FrontendBucket:
      Type: AWS::S3::Bucket
      Properties:
        BucketName: shing-catdex-frontend  # Change this to your
        ↪ -name-catdex-frontend
        AccessControl: Private

custom:
  client:
    bucketName: shing-catdex-frontend
```

By default, the `serverless-finch` plugin looks for files in the `client/dist` folder and uploads them to S3. So you need to create the folder using `mkdir -p client/dist`. Then create the following files in it:

- index.html: the cats overview page (Listing 6.9).
- `css/index.css`: CSS stylesheet for `index.html` (Listing 6.9).
- add.html: the add new cat form (Listing 6.9).

TODO: Add filename

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Catdex</title>
    <link rel="stylesheet" href="css/index.css" type="text/css">
  </head>
  <body>
    <h1>Catdex</h1>
    <p>
      <a href="/add.html">Add a new cat</a>
    </p>

    <section class="cats" id="cats">
      <p>No cats yet</p>
    </section>
    <script charset="utf-8">
      document.addEventListener("DOMContentLoaded", () => {
        fetch('https://abc0123def.execute-api.eu-central-1.
    ↪ amazonaws.com/dev/cats')
          .then((response) => response.json())
          .then((cats) => {
            document.getElementById("cats").innerText = "" //
    ↪ Clear the "No cats yet"
            for (cat of cats) {
              const catElement = document.createElement("
    ↪ article")
              catElement.classList.add("cat")
              const catTitle = document.createElement("h3")
              const catLink = document.createElement("a")
              catLink.innerText = cat.name
              catLink.href = `/cat.html?id=${cat.id}`
              const catImage = document.createElement("img")
              catImage.src = cat.image_path

              catTitle.appendChild(catLink)
              catElement.appendChild(catTitle)
              catElement.appendChild(catImage)

              document.getElementById("cats").appendChild(
    ↪ catElement)
            }
          })
      })
    </script>
  </body>
```

```
</html>
```

TODO: Add filename

```
.cats {
  display: flex;
}

.cat {
  border: 1px solid grey;
  min-width: 200px;
  min-height: 350px;
  margin: 5px;
  padding: 5px;
  text-align: center;
}

.cat > img {
  width: 190px;
}
```

TODO: Add filename

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Catdex</title>
    <link rel="stylesheet" href="static/css/index.css" type="
    ↪ text/css">
  </head>
  <body>
    <script>
      async function submitForm(e) {
        e.preventDefault()

        const cat_name = document.getElementById('name').value

        const cat_post_response = await fetch('https://
        ↪ abc0123def.execute-api.eu-central-1.amazonaws.com/dev/cat
        ↪ ', {
          method: 'POST',
          mode: 'cors',
          headers: {
            'Content-Type': 'application/json'
          },
          body: JSON.stringify({ name: cat_name })
        })
```

```
        const image_upload_url = (await cat_post_response.json()
    ↪ ).upload_url

        const image = document.getElementById("image").files[0]

        const image_upload_response = await fetch(
    ↪ image_upload_url, {
            method: 'PUT',
            body: image,
        })

        if (image_upload_response.status === 200) {
            alert("Success")
        } else {
            alert("Failed")
        }
        return false
    }
    </script>
    <h1>Add a new cat</h1>

    <form onsubmit="return submitForm(event)">
      <label for="name">Name:</label>
      <input type="text" name="name" id="name" value="" />
      <label for="image">Image:</label>
      <input type="file" name="image" id="image" value="" />
      <button type="submit">Submit</button>
    </form>

  </body>
</html>
```

The `index.html` and `index.css` are mostly the same as the one in the previous chapters. The `add.html` has a slightly different logic than before. Instead of just calling the `POST /cat` API, you also make a second `PUT` call to update the image.

You can now run `npx serverless client deploy` to upload these static files to S3. Once deployed, you should see the URL in the log output:

```
$ npx serverless client deploy
Serverless: This deployment will:
Serverless: - Upload all files from 'client/dist' to bucket '
    ↪ shing-catdex-frontend'
# ...
Serverless: Success! Your site should be available at
  http://shing-catdex-frontend.s3-website.eu-central-1.amazonaws
    ↪ .com/
```

However, if you open the website now, you'll notice that the API calls are failing. This is because of the same-origin policy. Under that policy, the web page cannot access APIs under a different origin, which is the combination of URI scheme, hostname and port. Because the web page is served under `http://shing-catdex-frontend.s3-website.eu-central-1.amazonaws.com/`, but the API is under `https://abc0123def.execute-api.eu-central-1.amazonaws.com/`, same-origin policy will block the API call. The same-origin policy is a security feature that can block many kinds of attacks.

Since you control both the frontend and the backend API, you can use Cross-origin resource sharing (CORS) to overcome this restriction. With CORS enabled on the backend API, it can grant access to the frontend serving from a different origin.

To enable CORS, first you need to add `cors: true` to all the API endpoints in `serverless.yml`:

```
# ...
functions:
  cats:
    handler: cats
    events:
      - http:
          path: /cats
          method: get
          cors: true
  cat_post:
    handler: cat_post
    events:
    - http:
        path: /cat
        method: post
        cors: true
```

Second, both the APIs need to respond with an `Access-Control-Allow-Origin` header. This header specifies the origin that is allowed to access it. For simplicity you simply specify `Access-Control-Allow-Origin: *`, which allows every origin. This is of course not very secure. If you are running production workloads, always explicitly specify the exact host.

To add this header to the API, you can tweak the `cats/src/main.rs` like Listing 6.9.

TODO: Add filename

```
use lambda_http::http::{HeaderValue, Response, StatusCode};
//...

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda::run(handler(cats)).await?;
```

```
    Ok(())
}

async fn cats(
    _: Request,
    _: Context,
) -> Result<impl IntoResponse, Error> {
    //...

    let mut response = match client.scan(scan_input).await {
        //...
    };

    response.headers_mut().insert(
        "Access-Control-Allow-Origin",
        HeaderValue::from_static("*"),
    );

    Ok(response)
}
```

Do a similar change for `cat_post/src/main.rs`, as shown in Listing 6.9.
TODO: Add filename

```
use lambda_http::http::{HeaderValue, Response, StatusCode};
//...

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda::run(handler(cat_post)).await?;
    Ok(())
}

async fn cat_post(
    request: Request,
    _: Context,
) -> Result<impl IntoResponse, Error> {
    //...
    //
    let mut response =
        json!({ "upload_url": presigned_url }).into_response();

    response.headers_mut().insert(
        "Access-Control-Allow-Origin",
        HeaderValue::from_static("*"),
    );
```

```
    Ok(response)
}
```

Finally, there is a small issue with the default CORS setting set by `serverless-finch`. It allows PUT requests from `https://*.amazonaws.com`, but the frontend is served using `HTTP`, not `HTTPS`. Therefore, you need to manually re-configure the CORS setting using the AWS console:

1. Open the AWS Console.
2. Go to S3.
3. Click on the `shing-catdex-frontend` bucket.
4. Go to the "Permissions" tab, then click "CORS configuration.
5. Change the `https://*.amazonaws.com` to `http://*.amazonaws.com` (Listing 5).

   TODO: Add filename

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc
    ↪ /2006-03-01/">
<CORSRule>
    <AllowedOrigin>http://*.amazonaws.com</AllowedOrigin>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <MaxAgeSeconds>0</MaxAgeSeconds>
    <AllowedHeader>*</AllowedHeader>
</CORSRule>
<CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <MaxAgeSeconds>0</MaxAgeSeconds>
    <AllowedHeader>*</AllowedHeader>
</CORSRule>
</CORSConfiguration>
```

.

Because `serverless-finch` will override your CORS configuration by default, you need to deploy it with an extra flag `--no-cors-change`. Also, `serverless-finch` will remove all the files in the bucket during deployment, so all the uploaded cat images will be lost. You can use the `--no-delete-contents` flag to tell `serverless-finch` to keep the files. Therefore, the command to deploy the frontend now becomes:

```
npx serverless client deploy --no-delete-contents --no-cors-
    ↪ change
```

Now if you run the commands `npx serverless deploy` and `npx serverless client deploy --no-delete-contents --no-cors-change`, the catdex website should work just like the one in the previous chapters.

6.10 Other Alternatives

In the top three cloud providers (AWS, Google Cloud Platform, Microsoft Azure), only AWS has experimental support for Rust, as we've introduced in this chapter. There is an unofficial, community-driven Azure SDK for Rust[13]. And there were attempts to run Rust in Azure Function[14], but it's also unofficial.

[13] https://github.com/Azure/azure-sdk-for-rust
[14] https://robertohuertas.com/2018/12/22/azure-function-rust/