Single chapter compiled via latex.

April 17, 2022

git commit: 3d32784

Shing Lyu

CHAPTER 5

REST APIs

Building

~~In the previous chapter, we learned~~ about how to build a server-~~rendered website.~~ ~~However, there are a few drawbacks of using server-side rendering.~~ First, whenever you navigate from one page to another or submit forms, the browser has to request a new page from the server. From the user's perspective, this means the browser will go blank for a second before the next page appears. With the rise of frontend frameworks like React, Angular, or Vue, this problem can be solved by rendering the page in the frontend with JavaScript. The frontend application makes requests to the server to get information or submit forms. The user can still interact with the page while the page is requesting data, thanks to the asynchronous nature of JavaScript HTTP clients (e.g. built-in `fetch`). The server side now only needs to expose an HTTP RESTful API[1].

XmlHttpRequest

A benefit of this architecture for the development team is that the backend and frontend team can work independently. They only need to negotiate an API contract and won't step on each other's toes. The frontend also doesn't need to be served by the application server anymore. Instead, it can be deployed in a separate server or managed service like AWS S3, and serve through a CDN for maximum performance. to offload performance

can work independently

But server-side rendering still has its strength. For example, it works better with SEO (search engine optimization). Although nowadays many search engine's crawlers can partially understand JavaScript rendered pages, a server-side rendered page still works better. Another benefit is that the first page is interactive right away after it's loaded. For a client-side rendered page, the user will receive an empty page and need to wait for the API call to return with data.

In this chapter, ~~we'll show~~ guide in you how to build REST APIs. ~~We'll also discuss many backend topics that we didn't mention in the previous chapter~~, like input validation, error handling, logging, and testing.

[1] You can also use other protocols like SOAP, GraphQL, or gRPC, ~~but we'll stick with REST in this chapter~~ out of scope

5.1 What are you building?

In this chapter, you are going to ~~re~~build the Catdex as a REST API. You'll learn to build the following features:

- A RESTful API that returns the list of cats in JSON format. database?
- A frontend in HTML and JavaScript that consumes the API to display cats.
- Integration tests for the API endpoint.
- An API endpoint that returns a particular cat's detail in JSON, given that cat's ID.
- Input validation to check the ID is valid, and that returns a 400 Bad Request response for invalid IDs.
- Custom error handling to prevent users from seeing unexpected errors from the server.
- Logging using the `Logging` middleware.
- Enabling HTTPS.

~~We'll still~~ You'll be using the `actix-web` framework to build the API. ~~We'll~~ You'll not be using any frontend framework like React, but write the page in vanilla JavaScript. This is because the focus of this chapter is not on the frontend. ~~We'll~~ You'll touch upon how to write the frontend using a Rust framework in Chapter 4.

5.2 Converting the cats list to a REST API

Let's create a new actix-web project by running `cargo new catdex-api`. In `Cargo.toml`, add `actix-web` and other dependencies we'll need in the future (Listing 5.1).

Listing 5.1: `Cargo.toml`

```
[package]
name = "catdex-api"
version = "0.1.0"
edition = "2018"

[dependencies]
actix-web = "2.0.0"
actix-rt = "1.1.1"
actix-files = "0.2.1"
serde = "1.0.110"
serde_json = "1.0.53"
diesel = { version = "1.4.4", features = ["postgres", "r2d2"] }
r2d2 = "0.8.8"
```

In `src/main.rs`, let's first create a static server as shown in Listing 6.2.

Listing 5.2: A basic static server

```
use actix_files::Files;
use actix_web::{App, HttpServer};

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    println!("Listening on port 8080");
    HttpServer::new(move || {
        App::new().service(
            Files::new("/", "static").show_files_listing(),
        )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

■ **TIP** We serve the static files in the `static` folder in the same server that will serve the REST APIs. This is just for ease of development. In production, you should consider serving the static resources (HTML, CSS, JavaScript) from another server (e.g. Nginx) dedicated to serving static files. This gives you a few benefits:

- You can aggressively cache the static resources using a CDN (Content Delivery Network).
- Your static server and API server can scale independently.
- Deployment and maintenance might be easier.

We can also add the static files `static/index.html` (Listing 5.3) and `static/index.css` (Listing 5.4). Since there is no JavaScript in there yet, the page won't show any cats.

Listing 5.3: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Catdex</title>
    <link rel="stylesheet" href="static/css/index.css" type="
      ↪ text/css">
```

```html
  </head>
  <body>
    <h1>Catdex</h1>
    <p>
      <a href="/add.html">Add a new cat</a>
    </p>

    <section class="cats">
      <p>No cats yet</p>
    </section>
  </body>
</html>
```

Listing 5.4: index.css

```css
.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;
}
```

In the previous chapter, the server responds with HTML rendered by Handlebars. But for REST APIs we need to return some structural data so the frontend JavaScript can easily process it. JSON (JavaScript Object Notation) is one of the most popular options. To construct a JSON response, you can use actix-web's `web::Json` helper to turn any serializable (i.e., `impl serde::Serialize`) Rust object into an HTTP Response. For example, a minimal REST API endpoint that returns a hard-coded list of cats can be implemented like Listing 5.5. Notice that because `web::Json` implements the `Responder` trait, so you can simply return a `web::Json` from a handler and actix-web will convert it to a proper HTTP response for you.

Listing 5.5: A minimal JSON API that returns hard-coded data

```rust
use actix_files::Files;
use actix_web::{http, web, App, Http, HttpServer, Responder};
use serde::Serialize;
```

```rust
#[derive(Serialize)]
pub struct Cat {
    pub id: i32,
    pub name: String,
    pub image_path: String,
}

async fn cats() -> impl Responder {
    let cats = vec![
        Cat {
            id: 1,
            name: "foo".to_string(),
            image_path: "foo.png".to_string(),
        },
        Cat {
            id: 2,
            name: "bar".to_string(),
            image_path: "bar.png".to_string(),
        },
    ];
    return web::Json(cats);
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    println!("Listening on port 8080");
    HttpServer::new(move || {
        App::new()
            .service(
                web::scope("/api")
                    .route("/cats", web::get().to(cats)),
            )
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Now you can run `cargo run` to start the server. You can test the API using `curl`[2]
:

[2] `curl` might not be installed in your distribution by default. For example, for Ubuntu you can install it with `sudo apt-get install curl`.

```
% curl localhost:8080/api/cats
[{"id":1,"name":"foo","image_path":"foo.png"},
{"id":2,"name":"bar","image_path":"bar.png"}]
```

Of course, we are not satisfied with returning a static response. We need to connect to a database. We can simply reuse the same PostgreSQL database we created in the previous chapter. In the `main()` function, we need to set up the r2d2 connection pool and Diesel connection similar to what we've done before, and copy the `src/models.rs` and `src/schema.rs` from the Catdex project (Listing 5.6). Notice that the `Cat` struct definition has been moved to `src/model.rs`.

Listing 5.6: Setting up the database in `main()`

```rust
#[macro_use]
extern crate diesel;
// ...
use actix_web::{http, web, App, Http, Responder, HttpServer};
use diesel::pg::PgConnection;
use diesel::prelude::*;
use diesel::r2d2::{self, ConnectionManager};
use std::env;

mod models;
mod schema;
use self::models::*;
use self::schema::cats::dsl::*;

type DbPool = r2d2::Pool<ConnectionManager<PgConnection>>;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");
    let manager =
        ConnectionManager::<PgConnection>::new(&database_url);
    let pool = r2d2::Pool::builder()
        .build(manager)
        .expect("Failed to create DB connection pool.");

    println!("Listening on port 8080");
    HttpServer::new(move || {
        App::new()
            .data(pool.clone())
            .service(
                web::scope("/api").route(
                    "/cats",
                    web::get().to(cats_endpoint),
```

```
                ),
            )
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

// src/models.rs
use super::schema::cats;
use serde::{Deserialize, Serialize};

#[derive(Queryable, Serialize)]
pub struct Cat {
    pub id: i32,
    pub name: String,
    pub image_path: String,
}

// src/schema.rs
table! {
    cats (id) {
        id -> Int4,
        name -> Varchar,
        image_path -> Varchar,
    }
}
```

The cats API endpoint is also very similar to the previous `index()` handler (Listing 5.7).

Listing 5.7: The handler for `/api/cats`

```
async fn cats_endpoint(
    pool: web::Data<DbPool>,
) -> Result<HttpResponse, Error> {
    let connection =
        pool.get().expect("Can't get db connection from pool");

    let cats_data = web::block(move || {
        cats.limit(100).load::<Cat>(&connection)
    })
    .await
    .map_err(|_| HttpResponse::InternalServerError().finish())
    ?;
```

```
    return Ok(HttpResponse::Ok().json(cats_data));
}
```

The biggest difference is that we respond with a `HttpResponse::Ok().json(cats_data)`. Because `cats_data` is an array of the `Cats` struct, and `Cats` implements `serde::Serialize`, the `.json()` function can serialize it to a JSON string. We name the function `cats_endpoint` instead of just `cats` because the name conflicts with the table name cats defined by Diesel schema.

If we restart the server and call it with curl again, you can see that the API returns cats from the database:

```
% curl localhost:8080/api/cats
[{"id":1,"name":"British short hair",
"image_path":"/static/image/british-short-hair.jpg"},
{"id":2,"name":"Persian","image_path":"/static/image/persian.jpg"},
{"id":3,"name":"Ragdoll","image_path":"/static/image/ragdoll.jpg"}]
```

If we format it for readability

```
[
    {
        "id":1,
        "name":"British short hair",
        "image_path":"/static/image/british-short-hair.jpg"
    },
    {
        "id":2,
        "name":"Persian",
        "image_path":"/static/image/persian.jpg"
    },
    {
        "id":3,
        "name":"Ragdoll",
        "image_path":"/static/image/ragdoll.jpg"
    }
]
```

Now we can revisit our frontend page and make the page call the API (Listing 5.8).[3]

[3] You'll find a hack in this example. We remove the `static` prefix from the `image_path`. This is because in the server we built for the previous chapter, the images are served under the path `/static/images/`. But in this chapter's example, we serve it under `/images` instead. To avoid having to recreate the database and rebuild the add cat form again, we just use this hack so we can look at the important topics first.

Listing 5.8: Make the frontend call the API

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Catdex</title>
    <link rel="stylesheet" href="static/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('/api/cats')
          .then((response) => response.json())
          .then((cats) => {
             // Clear the "No cats yet"
             document.getElementById("cats").innerText = ""
             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")
               // This is a hack to reuse the test data from previous chapter
               catImage.src = cat.image_path.replace(/\/static/, "")

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

               document.getElementById("cats").appendChild(catElement)
             }
```

```
            })
        })
    </script>
  </body>
</html>
```

We use the `fetch()` API to make the GET call, and draw the cats we received onto the page with a series `document.createElement()` and `element.appendChild()` calls. You can make this more declarative by adopting a frontend framework like React, but that is out of the scope of this chapter. This page now looks like Figure 5.1.

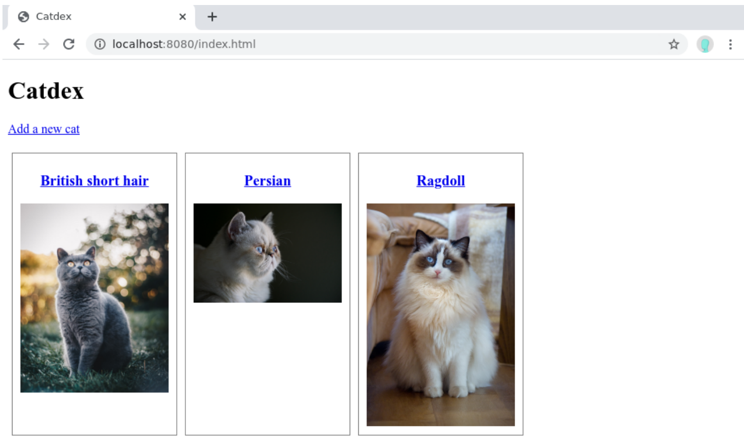

Figure 5.1: The client rendered index.html

5.3 API testing

So far we've been testing our APIs manually. Automating this test process will not only help you reduce human labor, but also urges the developer to test more often and provide quick feedback. Rust comes with unit testing capability. You can unit test all your functions individually with it, and you can learn about it from the official Rust Book[4]. In this book, instead, we'll be focusing on the integration test, in which you spin up a real HTTP server and test it with test requests.

Actix-web provides a few helper functions to set up the test server and create test requests. A simple test that calls the `/api/cats` API should look like Listing 5.9.

Listing 5.9: An integration test that calls the `/api/cats` API

[4]https://doc.rust-lang.org/book/ch11-00-testing.html

```rust
// ...

fn setup_database() -> DbPool {
    let database_url = env::var("DATABASE_URL")
        .expect("DATABASE_URL must be set");
    let manager =
        ConnectionManager::<PgConnection>::new(&database_url);
    r2d2::Pool::builder()
        .build(manager)
        .expect("Failed to create DB connection pool.")
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let pool = setup_database();
    // ...
}

#[cfg(test)]
mod tests {
    use super::*;
    use actix_web::{test, App};

    #[actix_rt::test]
    async fn test_cats_endpoint_get() {
        let pool = setup_database();
        let mut app = test::init_service(
            App::new().data(pool.clone()).route(
                "/api/cats",
                web::get().to(cats_endpoint),
            ),
        )
        .await;
        let req = test::TestRequest::get()
            .uri("/api/cats")
            .to_request();
        let resp = test::call_service(&mut app, req).await;
        assert!(resp.status().is_success());
    }
}
```

There are a few things to focus on in this example. First, we create a test module (`mod tests`) and add test cases as `async` functions. The test case functions need to be annotated with `#[actix_rt::test]`, so they will be run in the Actix runtime. Before running the test, you need to add the `actix_rt` crate using the command `cargo add actix_rt`.

Since we are doing an integration test, which involves starting a real HTTP server that communicates to a real database (as opposed to stubbing/mocking), we can reuse the code that sets up the database and connection pool by extracting it into a function named `setup_database`.

To start the test server, you construct an `App` instance as you would do in the `main()` function and pass it to `test::init_service()`. Of course, you can omit unrelated routes to make the code more readable and easier to debug. Then you can use the `test::TestRequest` builder to create a test request. Here we create a GET request for `/api/cats`. You can make the call with `test::call_service` and get the response. Finally, we can check if the response is a success (i.e., status code is in the 200-299 range) with an `assert!()`.

■ **TIP** For a test run to not interfere with any future test runs, you need to clean the database between every test run. You could create a test PostgreSQL database and use Rust code to set up and clean up before and after each test. But since we are using Docker and it's relatively easy to create new databases, you can consider creating a fresh PostgreSQL container for every test run, and destroy it after the test finishes.

You might notice that the code that sets up the `/api/cats` route is duplicated in the main() function and in the test function. As your service gets more and more routes, this repetition will start making maintenance hard. Actix-web provides a way to reuse configurations using the `App::configure` function. You pass a configuration function to `App::new().configure()`. The function needs to take one parameter of the type `web::ServiceConfig`. The `ServiceConfig` struct has the same interface as `App`, which has the methods `data()`, `service`, `route()`, etc. We can create a function called `api_config` that sets up everything under the `/api` scope. This function can then be reused in the `main()` function and the integration test, as shown in Listing 5.10. The `api_config()` function can also be extracted into a separate module. So you can keep the configuration in a separate file to improve readability.

Listing 5.10: Reusing configuration using App::configure()

```
// ...

fn api_config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/api")
            .route("/cats", web::get().to(cats_endpoint)),
    );
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let pool = setup_database();
```

```
    // ...
    HttpServer::new(move || {
        App::new()
            .data(pool.clone())
            .configure(api_config) // Used here
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

#[cfg(test)]
mod tests {
    use super::*;
    use actix_web::{test, App};

    #[actix_rt::test]
    async fn test_cats_endpoint_get() {
        let pool = setup_database();
        let mut app = test::init_service(
            App::new().data(pool.clone()).configure(api_config)
    ↪ ,
        )
        .await;
        let req = test::TestRequest::get()
            .uri("/api/cats")
            .to_request();
        let resp = test::call_service(&mut app, req).await;
        assert!(resp.status().is_success());
    }
}
```

.

5.4 Building the cat detail API

The `cats` API is too simple for demonstrating advanced use cases like query parameter, input validation, and error handling, so we are going to rebuild the `cat` API that returns a single cat's detail.

First, let's take a look at how the frontend is supposed to call the API. You might have noticed that in Listing 5.8, each cat's name is a link that points to `/cat.html?id=${cat.id}`. This page doesn't exist yet, so you need to create it in `static/cat.html` and paste the code in Listing 5.11 into it.

Listing 5.11: Single cat detail page

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Cat</title>
    <link rel="stylesheet" href="/static/css/cat.css" type="
    ↪ text/css">
  </head>
  <body>
    <h1 id="name">Loading...</h1>
    <img id="image" />
    <p>
      <a href="/index.html">Back</a>
    </p>
    <script charset="utf-8">
      const urlParams = new URLSearchParams(window.location.
    ↪ search)
      const cat_id = urlParams.get("id")
      document.addEventListener("DOMContentLoaded", () => {
        fetch(`/api/cat/${cat_id}`)
          .then((response) => response.json())
          .then((cat) => {
            document.getElementById("name").innerText = cat.
    ↪ name
            document.getElementById("image").src = cat.
    ↪ image_path
            document.title = cat.name
          })
      })
    </script>
  </body>
</html>
```

The link above opens the `cat.html` page and pass a query parameter (e.g., `?id=1`). This `id` query parameter is extracted as an object in JavaScript by creating a `new URLSearchParams(window.location.search)` and then call the `.get()` function on it. With the cat's ID at hand, we can call the `/api/cat/${cat_id}` API using `fetch`. The API has one path parameter for the ID, and it should return the cat's detail (including the name and the image path) in JSON format.

The most naive implementation for this API would be like Listing 5.12.

Listing 5.12: A naive implementation of the `cat` API

```
// ...

#[derive(Deserialize)]
```

```
struct CatEndpointPath {
    id: i32,
}

async fn cat_endpoint(
    pool: web::Data<DbPool>,
    cat_id: web::Path<CatEndpointPath>,
) -> Result<HttpResponse, Error> {
    let connection =
        pool.get().expect("Can't get db connection from pool");

    let cat_data = web::block(move || {
        cats.filter(id.eq(cat_id.id)).first::<Cat>(&connection)
    })
    .await
    .map_err(|_| HttpResponse::InternalServerError().finish())
    ↪ ?;

    Ok(HttpResponse::Ok().json(cat_data))
}

// ...

fn api_config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/api")
            .route("/cats", web::get().to(cats_endpoint))
            .route("/cat/{id}", web::get().to(cat_endpoint)),
    );
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    // ...
}
```

This code is very similar to the one we saw in the previous chapter. It extracts the `cat_id` using the `web::Path<CatEndpointPath>` extractor and tries to find it in the PostgreSQL database. But there are a few issues with this implementation:

- If it fails to get a connection from the connection pool, it will `panic!` due to the `expect` and returns a 500 error.
- If the ID does not exist in the database, we get a 500 Internal Server Error.
- If the ID in the path is not an integer (e.g., `/api/cat/abc`), it will return a 404

error with a message `can not parse "abc" to a i16`.

- If the ID is an integer, but is not in the correct range (e.g. negative number), we get a 400 Bad Request error.
- It's not very obvious where and why the error occurs in the source code.

500 Internal Server Error is not very informative for the frontend. The frontend only knows that something went wrong on the server-side, but it can't generate a helpful error message that will help the user to work around the problem. There are a few ways to do it better:

- Return a 400[5] error when the ID is invalid (e.g., not a number, out of bound).
- Return a 404 error when the ID doesn't exist in the database.
- Return a 500 error when we can't get a connection from the pool.
- Be able to customize the error message ourselves.
- Make it clear in the code where and why an error occurs.

5.5 Input validation

Let's deal with the input validation first. We know that the cat's ID can be wrong in many ways. If it's not an integer, Actix-web's type-safe extractor will return a 404 error. This error can be customized, but we'll get back to it later. Let's first handle the case where the ID is an integer, but it's not in the sensible range.

Because our cat ID has the schema `id SERIAL PRIMARY KEY`, we know that PostgreSQL will start with 1 and increase it by 1 every time we insert a new row. Therefore, the ID can't go below 1. Also for the sake of the example, if we only allow a user to add unique cat breeds to the website, then there are only 71 standardized breeds recognized by The International Cat Association (TICA). If we keep some buffer and assume that the cat breeds might double in the future, then we will have about $71 \times 2 = 142 \approx 150$ breeds. Therefore, we can check if the cat's ID is between 1 and 150 (inclusive), otherwise we can simply reject the request without even querying the database.

To validate the input parameter in a more declarative way, you can use the `validator` and `validator_derive` crates. Add the crates with the command `cargo add validator validator_derive`. Let's apply that onto the cat's ID, as shown in Listing 5.13.

Listing 5.13: Using `validator` on cat's ID

```
use validator::Validate;
```

[5] There are many debates about whether a 400 or a 422 is more appropriate in this case. We'll stick with the more generic 400 error.

```
use validator_derive::Validate;

// ...

#[derive(Deserialize, Validate)]
struct CatEndpointPath {
    #[validate(range(min = 1, max = 150))]
    id: i32,
}

async fn cat_endpoint(
    pool: web::Data<DbPool>,
    cat_id: web::Path<CatEndpointPath>,
) -> Result<HttpResponse, Error> {
    cat_id
        .validate()
        .map_err(|_| HttpResponse::BadRequest().finish())?;

    // ... getting a connection and query from database

    Ok(HttpResponse::Ok().json(cat_data))
}
```

In this code snippet, the `web::Path` extractor now tries to extract the `CatEndpointPath` struct from the URL. The `CatEndpointPath` is marked to have a `Validate` auto-derive trait provided by the `validate_derive` crate. This means you can call `CatEndpointPath.validate()` to validate all its fields. Each field's validation rule can be annotated on them individually. For our `id` we specify that it should be a number in the range of 1 to 150: `#[validate(range(min=1, max=150))]`. The `validator` crate also provides some common checks like whether the field is an email, IP, URL or having a certain length.

Inside the `cat_endpoint` handler, we call `cat_id.validate()` to validate. If the validation passes, it returns an `Ok<()>` and we just allow the code to continue. If the validation fails, it returns an `Err<ValidationError>`, and we convert it to a `HttpResponse::BadRequest` and force it to early return with the `?` operator.

Now if you start the server again with `cargo run` and make a call to the API with an ID outside of the range (e.g. `curl -v localhost:8080/api/cat/9999` or `curl -v localhost:8080/api/cat/-1`)[6], you should see the 400 Bad Request response.

```
% curl -v localhost:8080/api/cat/9999
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
```

[6] The '-v' option is an abbreviation of `--verbose`. It will make cURL print extra information like HTTP status code.

```
> GET /api/cat/9999 HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 400 Bad Request
< content-length: 0
< date: Tue, 21 Jul 2020 10:05:21 GMT
<
* Connection #0 to host localhost left intact
```

5.6 Error handling

You might notice that even this simple `cat_endpoint` handler can fail at many different points:

- The parameter validation might fail.
- Getting a connection from the connection pool might fail.
- Querying the cat from the database might fail because:
  - `web::block()` might fail for unexpected reasons.
  - Diesel ORM might fail for unexpected reasons
  - The Diesel query might fail because the cat doesn't exist.[7]

Each of these errors might come from different libraries (`actix-web`, `r2d2`, `diesel`), and we've been converting them to HTTP response with `.map_err()` and `?`. But it's worth taking a step back and look at how Actix-web handles errors.

Let's first look at what is an API endpoint handler's response: `Result<HttpResponse, Error>`. The `Error` here refers to Actix-web's own `actix_web::Error`[8], rather than the standard library `std::error::Error`. An `actix_web::Error` contains a trait object of the trait `ResponseError`. The `ResponseError` contains metadata (e.g., status code) and helper functions to construct a HTTP Response, so Actix-web can easily convert a `actix_web::Error` into an HTTP error response.

Since most of the errors returned by our dependent libraries are not `actix_web::Error`, if we have to handle them with `match` and construct an `actix_web::Error` by hand, the control flow will soon be very verbose. But in our previous ex-

[7] Although we make sure the ID is within 1 to 150, but we might only have 70 cats in the database and someone tries to find a cat with ID 71.

[8] It's actually a re-export of `actix_http::error::Error`. It's re-exported by `actix_web` for convenience. `actix_web::error::Error` is also the same thing.

ample we could do something like `.map_err(error::ErrorBadRequest)?;` or `.map_err(|_|␣␣HttpResponse::InternalServerError().finish()␣)?;`. How did they work?

Actix-web provides many helper functions and implicit type conversions to help you handle errors more fluently. But because there are so many ways, it can get confusing at times. So we'll break them down into three main categories:

- Using a `ResponseBuilder` object or a `Response` object.
- Using the `actix_web::error` helper functions like `actix_web::error::ErrorBadRequest`.
- Using a generic error that has implemented the `ResponseError` trait
- Using a custom-built error type.

Using a `ResponseBuilder` or `Response`

The first way is the one we saw in Listing 5.13 and previous examples. You'll often see this style of code in Actix-web examples:

```
async fn cat_endpoint(
    pool: web::Data<DbPool>,
    cat_id: web::Path<CatEndpointPath>
) -> Result<HttpResponse, Error> {
    cat_id
        .validate()
        .map_err(|_| HttpResponse::BadRequest().finish())?;
    // ...
}
```

The `validate()` function returns a `Result<(),␣ValidationErrors>`. We use the `.map_err()` function to convert the `ValidationError` into a `HttpResponse::BadRequest().finish()`. You might be surprised that we convert an error into a `Response`. At a first glance, we are changing the return value to `Result<HttpResponse,␣Response>`. But in fact, because the `actix_web::error` module implements `impl␣From<Response<Body>>␣for␣Error`, so a `Response` can be converted to an `Error` with `Error::from(response)` (or `response.into()`). When we use the `?` operator to make the line return early in case of `Err`, the `?` operator will implicitly use `From` to convert the `Response` into an `Error`. So although we seem to return a Response, it is converted to an `actix_web::Error`.

There is also an implementation of `impl␣From<ResponseBuilder>␣for␣Error`. So even if you omit the `.finish()` call it will still work:

```
cat_id
    .validate()
    .map_err(|_| HttpResponse::BadRequest())?;
```

■ **NOTE** If you are not familiar with the `.map_err()` function, its purpose is to convert the `Err` value of a `Result` from one type to another, leaving the `Ok` value unchanged. For example, if we pass a function that converts a value of type `E` to type `F`, the `.map_err()` will convert a `Result<T, E>` to `Result<T, F>`. This is useful for passing through the `Ok` value and handle the `Err`. In our example, we use it to convert the error to a type that Actix-web accepts.

Figure 5.2 visualizes the error handling flow we have so far using this method.

Figure 5.2: The current error handling flow

Using the `actix_web::error` helpers

The first, and probably most straightforward method is to use the `actix_web::error` helpers. In the `actix_web::error` module there are helper functions for most of the commonly used HTTP status codes. For example:

- `ErrorBadRequest()`: 400
- `ErrorNotFound()`: 404
- `ErrorInternalServerError()`: 500
- `ErrorBadGateway()`: 502

These error helpers wraps any error and returns a `actix_web::Error`. For example the signature of `ErrorBadRequest` is as follows:

```
pub fn ErrorBadRequest<T>(err: T) -> Error
where
    T: Debug + Display + 'static,
```

Therefore, if we make a function call which may return a `Result<T, E>`, we can use the `.map_err()` function to convert the `E` into an `actix_web::Error`. Then, we can use the `?` operator to force the handler function to return early with the converted `actix_web::Error`.

```
cat_id
    .validate()
    .map_err(|e| error::ErrorBadRequest(e))?;
```

Or simply replace the closure with the helper function:

```
cat_id
    .validate()
    .map_err(error::ErrorBadRequest)?;
```

Using a generic error that has implemented `ResponseError` trait

The two methods above converts (or wraps) the error we got into an `actix_web::Error`. But the type definition of `Responder` only requires the error to be `Into<Error>`. And since there is an implementation of `impl<T: ResponseError + 'static> From<T> for Error`, you can return anything that implements the `ResponseError` trait.

Actix-web already implements `ResponseError` for many of the common error types you'll encounter in web services. For example,

- `std::io::error::Error`: when reading files.
- `serde_json::error::Error`: when serialize/deserialize JSON.
- `openssl::ssl::error::Error`: when making HTTPS connections.

Therefore, if you have some very simple handlers that have only one error, you can just return it directly. For example, if we are serving the `index.html` by reading it in the handler with `NamedFile::open`, then we can simply return `std::io::Result<T>` (i.e., `Result<T, std::io::error::Error>`) and the `io::error::Error` can be converted to an HTTP Response error without you writing anything extra (Listing 5.14).

Listing 5.14: Returning a `io::Result`, which implements `ResponseError`

```
use actix_files::NamedFile;
use std::io;

fn index(_req: HttpRequest) -> io::Result<NamedFile> {
    Ok(NamedFile::open("static/index.html")?)
}
```

Using a custom-built error type

The built-in implementation of `impl ResponseError for T` and `impl From<T> for Error` are helpful if you want to quickly return some error and don't want to deal with the conversion. But because many of the error types can be converted too easily, you might accidentally return some error that exposes too much detail to the user. When building an API you need to carefully choose how much detail you expose to the user. A very detailed error is useful for debugging, but it may expose too much implementation detail and give attackers some hint on hacking your system. For example, if the application server fails to connect to the database, it might be tempting to respond with an error describing why the database connection failed, what is the database IP and port, or if you are really not careful, what is the database username and password. All these are useful information for the attacker to plan an attack based on the known vulnerability of the kind of database you use. Instead, you should just return a generic 500 Internal Server Error and don't let the client know why. In other words, it's important to distinguish the internal error (e.g. database connection failed for a particular reason) and the user-facing error (e.g. 500 Internal Server Error).

To achieve this separation, we can implement our custom error type that implements the `ResponseError` trait. The error type can be an enum with a detailed reason that helps debugging, but the `ResponseError` implementation can convert these detailed errors into generic user-facing errors. We can also customize the error message, instead of relying on the default provided by the `actix_web::error` helpers or `ResponseBuilder`.

To define our custom error, let's first create a new file called `src/errors.rs` and create an `enum` called `UserError`, as shown in Listing 5.15.

Listing 5.15: Custom error definition

```
#[derive(Debug)]
pub enum UserError {
    ValidationError,
    DBPoolGetError,
    NotFoundError,
    UnexpectedError,
}
```

Then let's declare this module in `src/main.rs` and use them in our cat_endpoint (Listing 5.16).

Listing 5.16: Declaring and using the `UserError` in the `cat_endpoint`

```
// ...

mod errors;
use self::errors::UserError;

// ...

async fn cat_endpoint(
    pool: web::Data<DbPool>,
    cat_id: web::Path<CatEndpointPath>,
) -> Result<HttpResponse, UserError> {
    cat_id.validate().map_err(|_| UserError::ValidationError)?;
    let connection =
        pool.get().map_err(|_| UserError::DBPoolGetError)?;

    let query_id = cat_id.id.clone();
    let cat_data = web::block(move || {
        cats.filter(id.eq(query_id)).first::<Cat>(&connection)
    })
    .await
    .map_err(|e| match e {
        error::BlockingError::Error(
            diesel::result::Error::NotFound,
        ) => UserError::NotFoundError,
        _ => UserError::UnexpectedError,
    })?;
    Ok(HttpResponse::Ok().json(cat_data))
}
// ...
```

Notice that the `cat_endpoint` now returns the type `Result<HttpResponse, UserError>`. All the `.map_err()` now converts the errors into `UserError`, instead of the error helper or `ResponseBuilder`. We also make a `match` in the `.map_err()` of the database query call, so we can isolate the special case where Diesel reports it can't find the cat (`diesel::result::Error::NotFound`.

The `UserError` has not implemented the `ResponseError` trait yet, so it can't be turned into an HTTP Response. We can implement it in `src/errors.rs`, as shown in Listing 5.17. You'll also notice that we used the `derive_more` crate so we can auto-derive the `Display` trait on the `UserError` enum. You can add this crate by running `cargo add derive_more`.

Listing 5.17: Implementing `ResponseError` for `UserError`

```
use actix_web::http::StatusCode;
use actix_web::{error, HttpResponse};
use derive_more::Display;
use serde_json::json;

#[derive(Display, Debug)]
pub enum UserError {
    #[display(fmt = "Invalid input parameter")]
    ValidationError,
    #[display(fmt = "Internal server error")]
    DBPoolGetError,
    #[display(fmt = "Not found")]
    NotFoundError,
    #[display(fmt = "Internal server error")]
    UnexpectedError,
}

impl error::ResponseError for UserError {
    fn error_response(&self) -> HttpResponse {
        HttpResponse::build(self.status_code())
            .json(json!({ "msg": self.to_string() }))
    }
    fn status_code(&self) -> StatusCode {
        match *self {
            UserError::ValidationError => {
                StatusCode::BAD_REQUEST
            }
            UserError::DBPoolGetError => {
                StatusCode::INTERNAL_SERVER_ERROR
            }
            UserError::NotFoundError => StatusCode::NOT_FOUND,
            UserError::UnexpectedError => {
                StatusCode::INTERNAL_SERVER_ERROR
```

```
            }
        }
    }
}
```

An HTTP Response has two key elements: the status code and the body. The status code is determined by the `status_code()` function. The function is a simple `match` that converts the enum variant to the appropriate status code. For the body we want to respond with a JSON of the format:

```
{
  "msg": "An error message"
}
```

The HTTP response is generated in the `error_response()` function using the `HttpResponse` builder. The message body is created by calling `self.to_string()`. We derive the `Display` trait on the enum and annotate each variant with `#[display(fmt="...")]`, so that the `.to_string()` function will convert the enum variant to the string we specified. The JSON body is serialized using `json!()` macro from `serde_json`.

With this custom error, we can create as many internal errors as we want, but converting them to something general for the user. Also because the return type is `Result<HttpResponse,UserError>`, type check will prevent you from accidentally returning an error that happens to implement `ResponseError`.

Figure 5.3 visualizes the new error handling flow after using `UserError`.

5.7 Customize the `web::Path` extractor error

We now have control over most of the errors, but we missed one case. If the ID can not be converted to `i32`, the `web::Path` extractor will return a 404 Not Found with a default error message. But that error can also be customized through `web::PathConfig::error_handler()`. When we construct the `App` (or a `ServiceConfig`), we can define a custom error handler for `web::Path` extractors that returns custom errors. We can add it to the `api_config()` function as shown in Listing 5.18.

Listing 5.18: Custom error handler for `web::Path` extractor error

```
fn api_config(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/api")
            .app_data(web::PathConfig::default().error_handler(
                |_, _| UserError::ValidationError.into(),
            ))
            .route("/cats", web::get().to(cats_endpoint))
```

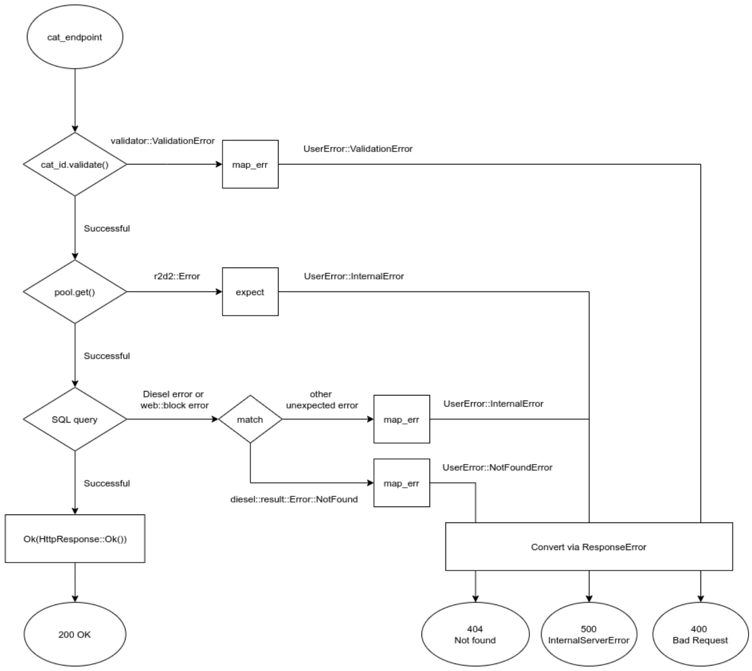

Figure 5.3: The error handling flow after using `UserError`

```
            .route("/cat/{id}", web::get().to(cat_endpoint)),
    );
}
```

We configured a custom error handler that returns a `UserError::ValidationError`, which will be converted to a 400 Bad Request thanks to our `ResponseError` implementation.

5.8 Logging

Good error handling helps us provide meaningful error status codes and messages to the frontend. But to really understand what happened, we need to rely on logging. When the server is small and the business logic is simple, you can easily try a few requests and reproduce a bug. But when you have thousands of concurrent users, all going through different code paths, it's hard to pinpoint where the bug is. With proper logging, you can gain visibility into what happened to the requests and easily identify problems and bugs. It might also give you a view into user behavior and trends.

There is a key concept you need to understand before jumping into logging: logging facade vs. logging implementation. A logging facade defines an "interface" for logging. A logging implementation adopts that "interface" and does the actual logging (e.g. writing to STDOUT; writing to file). A logging facade gives an extra layer of abstraction so you can swap different implementations without rewriting the whole code. This is particularly useful when building libraries. A Rust library can log using a logging facade but don't choose a concrete implementation. An application that uses libraries can choose an implementation, and as long as all the libraries adopt the same logging facade, they end up using the same implementation.

A commonly used facade is the `log` crate, and `env_logger` is a simple but effective logging implementation. The `env` in the name suggests that you can configure the logging level using environment variables. Actix-web also provides a `Logger` middleware that produces access logs using the `log` facade.

To enable the `Logger`, you `.wrap()` the `App` with the `Logger` middleware as shown in Listing 5.19.

Listing 5.19: Using the `Logger` middleware

```
//...
use actix_web::middleware::Logger;
// ...

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    env_logger::init();
    // ...
    HttpServer::new(move || {
        App::new()
```

```
            .wrap(Logger::default())
            .data(pool.clone())
            .configure(api_config)
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

The `Logger` middleware uses the `log` facade, but you need to provide a logger implementation for it to work. For that, we need to add the `env_logger` crate to our dependency (`cargo add log env_logger`) and initialize it at the beginning of `main()`:

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    env_logger::init();
    // ...
}
```

In the example we use `Logger::default()` to get the default format. But you can also customize the log format when you initialize it.

The `log` facade defines five log levels, ordered by priority:

- Error: Designates very serious errors.
- Warn: Designates hazardous situations.
- Info: Designates useful information.
- Debug: Designates lower priority information.
- Trace: Designates very low priority, often extremely verbose, information.

When you choose a log level, any log that has priority above it and includes that level will be shown. Because the `env_logger`'s log level is configured through environment variables, we can run the server with log level set to `debug` in this way:

```
RUST_LOG=debug cargo run
```

When you try calling the `http://localhost:8080/api/cats` API, the `Logger` middleware should log this request[9]:

[9] You can see the request for `favicon.ico` results in 404 Not Found. Favicon is an icon that most browser will fetch automatically, it can be used as the icon on the browser tab, favorite list, and URL bar. We didn't add this icon so it is normal that you see a 404 Not Found.

```
[2020-07-21T11:40:32Z INFO  actix_server::builder] Starting 4 workers
[2020-07-21T11:40:32Z INFO  actix_server::builder]
    Starting "actix-web-service-127.0.0.1:8080" service on 127.0.0.1:8080
[2020-07-21T11:41:58Z INFO  actix_web::middleware::logger]
    127.0.0.1:38278 "GET /api/cats HTTP/1.1" 200 764 "-"
    "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/79.0.3945.88 Safari/537.36" 0.008303
[2020-07-21T11:41:59Z DEBUG actix_files]
    Files: Failed to handle /favicon.ico: No such file or directory (os error 2
[2020-07-21T11:41:59Z INFO  actix_web::middleware::logger]
    127.0.0.1:38278 "GET /favicon.ico HTTP/1.1" 404 0
    "http://localhost:8080/api/cats" "Mozilla/5.0 (X11; Linux x86_64)
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36"
    0.000438
```

You can also log custom log messages. The `log` crate exposes logging macros for logging at a particular level: `error!()`, `warn!()`, `info!()`, `debug!()`, and `trace!()`. We can add logs to all the places where errors are handled (Listing 5.20).

Listing 5.20: Custom logging

```rust
//...
use log::{error, info, warn};

// ...

async fn cats_endpoint(
    pool: web::Data<DbPool>,
) -> Result<HttpResponse, UserError> {
    let connection = pool.get().map_err(|_| {
        error!("Failed to get DB connection from pool");
        UserError::InternalError
    })?;

    let cats_data = web::block(move || {
        cats.limit(100).load::<Cat>(&connection)
    })
    .await
    .map_err(|_| {
        error!("Failed to get cats");
        UserError::InternalError
    })?;
    return Ok(HttpResponse::Ok().json(cats_data));
}

// ...
```

```
async fn cat_endpoint(
    pool: web::Data<DbPool>,
    cat_id: web::Path<CatEndpointPath>,
) -> Result<HttpResponse, UserError> {
    cat_id.validate().map_err(|_| {
        warn!("Parameter validation failed");
        UserError::ValidationError
    })?;
    let connection = pool.get().map_err(|_| {
        error!("Failed to get DB connection from pool");
        UserError::InternalError
    })?;

    let query_id = cat_id.id.clone();
    let cat_data = web::block(move || {
        cats.filter(id.eq(query_id)).first::<Cat>(&connection)
    })
    .await
    .map_err(|e| match e {
        error::BlockingError::Error(
            diesel::result::Error::NotFound,
        ) => {
            error!("Cat ID: {} not found in DB", &cat_id.id);
            UserError::NotFoundError
        }
        _ => {
            error!("Unexpected error");
            UserError::InternalError
        }
    })?;
    Ok(HttpResponse::Ok().json(cat_data))
}

// ...

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    env_logger::init();
    let pool = setup_database();

    info!("Listening on port 8080");
    HttpServer::new(move || {
        App::new()
            .wrap(Logger::default())
            .data(pool.clone())
```

```
            .configure(api_config)
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

If you try to trigger a validation error (e.g., by calling `curl localhost:8080/api/cat/-1`), you should see the custom log like the following:

```
[2020-07-21T11:48:04Z INFO  catdex] Listening on port 8080
[2020-07-21T11:48:04Z INFO  actix_server::builder] Starting 4 workers
[2020-07-21T11:48:04Z INFO  actix_server::builder]
    Starting "actix-web-service-127.0.0.1:8080" service on 127.0.0.1:8080
[2020-07-21T11:48:51Z WARN  catdex] Parameter validation failed
[2020-07-21T11:48:51Z DEBUG actix_web::middleware::logger]
    Error in response: ValidationError
[2020-07-21T11:48:51Z INFO  actix_web::middleware::logger]
    127.0.0.1:38362 "GET /api/cat/-1 HTTP/1.1" 400 33 "-
" "curl/7.47.0" 0.002286
```

With carefully planned error handling and logging, you should be able to get good visibility into how your system is behaving in production.

5.9 Enabling HTTPS

Now our API server is ready to serve the users. But we've been testing it with HTTP protocol only. To actually serve this API out on the Internet, it's important to use the HTTPS protocol, which encrypts the communication with TLS (Transport Layer Security)[10].

The first thing you need for HTTPS is a certificate for your domain name. Usually, you obtain a certificate from a Certificate Authority (CA). You can get a free certificate from Let's Encrypt[11], a non-profit CA that tries to create a more secure Web. But for the sake of demonstration we are going to create a self-signed certificate, i.e., we act as our own CA and sign our own certificate.

To generate the certificate (`cert.pem`) and the private key (`key.pem`)[12], you can run this command:

[10]Formerly SSL (Secure Sockets Layer).

[11]https://letsencrypt.org/

[12]How HTTPS works are outside of the scope for this book, you can find many good introductions online by searching "How HTTPS works"

```
sudo apt-get install openssl # You only need to run this once

openssl req -x509 -newkey rsa:4096 \
  -keyout key.pem \
  -out cert.pem \
  -days 365 \
  -sha256 \
  -subj "/CN=localhost"
```

The `openssl` tool will ask you to set a password for the `key.pem` file. If you use this `key.pem`, every time you start the Acitx-web server, you need to enter the password again. To remove the password, you can run

```
openssl rsa -in key.pem -out key-no-password.pem
```

This will generate a new key file `key-no-password.pem`. When deploying this file to the production server, be sure to secure it with file system permissions.

Once we have the certificate and key, there are a few extra steps required for SSL:

- Install the required headers: `sudo apt-get install libssl-dev`
- Add the `openssl` crate to the dependencies.
- Enabled the `openssl` feature on `actix-web` (Listing 5.21).

Listing 5.21: Enabling the `openssl` feature for `actix-web` in `Cargo.toml`

```toml
[package]
name = "catdex"
# ...

[dependencies]
actix-web = { version = "2.0.0", features = ["openssl"] }
# ...
openssl = "0.10.30"
```

Finally, we can change our code so that the `App` builder uses `.bind_openssl()` instead of `.bind()`, shown in Listing 5.22.

Listing 5.22: Enabling SSL

```rust
// ...
use openssl::ssl::{SslAcceptor, SslFiletype, SslMethod};

// ...

#[actix_web::main]
```

```
async fn main() -> std::io::Result<()> {
    env_logger::init();

    let mut builder =
        SslAcceptor::mozilla_intermediate(SslMethod::tls())
            .unwrap();
    builder
        .set_private_key_file(
            "key-no-password.pem",
            SslFiletype::PEM,
        )
        .unwrap();
    builder.set_certificate_chain_file("cert.pem").unwrap();

    let pool = setup_database();

    info!("Listening on port 8080");
    HttpServer::new(move || {
        App::new()
            .wrap(Logger::default())
            .data(pool.clone())
            .configure(api_config)
            .service(
                Files::new("/", "static").show_files_listing(),
            )
    })
    .bind_openssl("127.0.0.1:8080", builder)?
    .run()
    .await
}
```

Now if you start the server with `cargo run`, you should be able to connect the website with `https://localhost:8080` instead of `http://localhost:8080`. Your browser should show a warning because it doesn't trust our self-signed CA.

5.10 Other Alternatives

Since REST APIs can be built with almost any web framework, the frameworks presented in the previous chapter are also relevant here.

Besides REST, there are other protocols you can use to build APIs. For example, gRPC and GraphQL are some of the popular alternatives. For gRPC, there are crates like `tonic`[13] and `grpc`[14]. For GraphQL there is `juniper`[15]. `Juniper` doesn't come with a web server, so it needs to be integrated into a web framework like Actix-web.

Although JSON is one of the most popular data representation formats, you can also use other formats like XML (`serde-xml-rs`[16]) or Protobuf (`protobuf`[17] or `prost` [18]).

Finally, `log` allows us to log in many formats, but they are still for humans to consume. If we log in a machine-readable format (e.g., JSON), many existing log analysis tools can help you index and analyze the log. This is called *structured logging*. Currently, you can use the `slog`[19] ecosystem for structured logging. There are also efforts in introducing structured logging to `log`[20].

[13]https://github.com/hyperium/tonic
[14]https://github.com/stepancheg/grpc-rust
[15]https://github.com/graphql-rust/juniper
[16]https://github.com/RReverser/serde-xml-rs
[17]https://github.com/stepancheg/rust-protobuf/
[18]https://github.com/danburkert/prost
[19]https://github.com/slog-rs/slog
[20]https://github.com/rust-lang/log/issues/149