LogRocket
Frontend Analytics

Building web apps with Rust using the Rocket framework

August 26, 2020 · 9 min read

Introduction

Rust is gradually becoming the language of choice for many developers who want to build efficient and reliable applications more quickly while still maintaining flexibility and low-level control. With web frameworks like Rocket, web developers can use Rust to build fast and secure web applications.

In this article, we'll introduce Rust for the web by building a simple web application. While this will be a step-by-step guide on using the Rocket framework to build type-safe, fast, and reliable web applications, this article also assumes that you have a basic understanding of Rust. If you don't, the book *The Rust Programming Language* is a great resource for familiarizing yourself with the language.

Getting started with the Rocket web framework

Before we get started, let's confirm that we have rustup installed on our machine by running the following command on our terminal:

```
rustup --version
```

If the above command results in an error, click here to see rustup installation instructions. Rustup installation sets up Rust and Cargo, Rust's package manager on our local computer.

Setting up our app

With rustup installed, we can use Cargo to create a new Rust project. Let's run the following command on our terminal:

```
cargo new rocket-web --bin
```

This will create a new Rust app named rocket-web. The `--bin` flag tells Cargo to generate this as a binary-based project.

Next, we'll navigate to the new project directory from our terminal and configure Rust nightly as our project toolchain:

```
cd rocket-web
rustup override set nightly
```

Rocket uses unstable features of Rust, like its syntax extensions. This is why we set up the nightly version of Rust as our project toolchain. Let's navigate to the `./cargo.toml` file in our root directory and add `rocket` as a dependency:

```
[dependencies]
rocket = "0.4.5"
```

To use Rocket in our project, let's import it in the `./src/main.rs` file:

```
#[macro_use] extern crate rocket;
```

This imports the macros from the `rocket` crate. Alternatively, we can use the following line:

```
use rocket::*;
```

Next, we'll use the `#![feature()]` flag to enable the unstable `decl_macro` feature for our Rocket project. We'll need it when creating routes for our web app. Just before the `use rocket::*` statement, let's add this line:

```
#![feature(decl_macro)]
```

Creating our first Rocket route

Now that we have this set up, we can go ahead and create our first Rocket route. Le's import the `Json` type from the `rocket::response::content` macro. We'll use this to send a response when our route is called. Our `main.rs` file should look like this after the `Json` import on line 4:

```
#![feature(decl_macro)]
#[macro_use] extern crate rocket;

use rocket::response::content::Json;

fn main() {
    println!("Hello, world!");
}
```

Next, let's paste the following block of code just before the `main` function to create our first route:

```
#[get("/hello")]
fn hello() -> Json<&'static str> {
  Json("{
    'status': 'success',
    'message': 'Hello API!'
  }")
}
```

In the above block of code, we started by using the attribute, `#get("/hello")]` to tell Rocket that our function expects a `GET` request to the `/hello` route. Next, we named our function `hello()` and specified its return type as `Json` with a `<&'static str>` argument.

This means that when a `GET` request is sent to our `/hello` route, it will return a JSON response with body of `'status': 'success'` and `'message': 'Hello API!'` , which we added on lines 4–6.

To test our new route, let's remove the `println!` statement in our `main()` function and paste the following code inside it:

```
rocket::ignite()
  .mount("/api", routes![hello])
  .launch();
```

This uses the `ignite()` method from the `rocket` crate to create a new instance of Rocket, and then mounts our `hello` route with the `mount()` method and base path `/api` . Finally, we used the `launch()` method to start the application server and listen for requests.

We can now run `cargo build` on our terminal to compile our Rocket application. We should get a similar response to this:

```
cargo build
    Finished dev [unoptimized + debuginfo] target(s) in 0.25s
```

Next, let's run the command `cargo run` to start our application. We should get a similar response to this:

```
cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target\debug\rust_rocket.exe`
�👾 Configured for development
    => address: localhost
    => port: 8000
    => log: normal
    => workers: 8
    => secret key: generated
    => limits: forms = 32KiB
    => keep-alive: 5s
    => tls: disabled
🚀   Mounting /api:
    => GET /api/hello (hello)
��� Rocket has launched from http://localhost:8000
```

Finally, we can test our application and route by navigating to `http://localhost:8000/api/hello` on our browser or API client. We should receive the following response:

```
{
    'status': 'success',
    'message': 'Hello API!'
}
```

Handling POST requests with Rocket

We've successfully launched our first Rocket API, but this is just the beginning. Apart from returning JSON responses, Rocket also allows us to return other types like `String` , `Status` , and `Template` . The Rocket documentation does a great job in explaining the different return and response types.

We'll see how we can use Rocket to render HTML soon, but before that, let's see what creating a `POST` route in Rocket is like. We'll use our `POST` route to add book information to a dummy database.

Rocket has support for databases like MySQL, Postgres, SQLite, Redis, and MongoDB. You can read more about the database options here. We'll use a vector to create our dummy database for this demo.

Firstly, let's define what type of information we'll be expecting from our user when they send a request to our book route. We'll use a struct named `Book` for this. Just before the `hello()` route we created, let's define our new struct:

```rust
struct Book {
  title: String,
  author: String,
  isbn: String
}
```

Next, let's create our `POST` route with the following lines of code:

```rust
#[post("/book", data = "<book_form>")]
fn new_book(book_form: Book) -> String {
}
```

This time around, we added the type of data Rocket should expect when watching for requests as the second argument of the route attribute `#[post()]`. We went ahead and supplied the type for `book_form` in our `new_book()` function argument and defined our function return type as `String`. If we try to compile this, we should get an error message similar to the following:

```
the trait bound `Book: rocket::data::FromDataSimple` is not satisfied
```

To fix this, let's add the following line to our import statements at the top of our file to import the `Form` type:

```
use rocket::request::Form;
```

Next, we'll add the `#[derive(FromForm)]` attribute to our `Book` struct. Our struct declaration should now look like this:

```
#[derive(FromForm)]
struct Book {
  title: String,
  author: String,
  isbn: String
}
```

Now, we can implement the `[FromData]` trait by wrapping the `book_form` type we supplied as our `new_book()` function argument with the `Form` type we just imported:

```
#[post("/book", data = "<book_form>")]
fn new_book(book_form: Form<Book>) -> String {
}
```

Next, we'll tell our route what to do whenever it is called. Right inside our `new_book()` function, let's paste the following code:

```
let book: Book = book_form.into_inner();
let mut dummy_db: Vec<Book> = Vec::new();
dummy_db.push(book);
format!("Book added successfully: {:?}", dummy_db)
```

In the above block, we used the `book_form.into_inner()` method to get the request body from our user, then we defined our dummy database as a vector with type `Book` and pushed the data we received from our user to it using the `dummy_db.push(book)` expression.

Finally, we returned the string `"Book added successfully: {:?}", dummy_db)`. We used the `format!` method for this because we added the vector `dummy_db` to our string response. Let's also add the `Debug` flag in our `Book` struct attribute to make this possible:

```
#[derive(FromForm, Debug)]
struct Book {
  ...
}
```

Next, we'll add our new `POST` route to the `/api` path in our `main()` function:

```
rocket::ignite()
  .mount("/api", routes![hello])
  .launch();
```

Now, we can rebuild our app using `cargo build` and run it with `cargo run` to test our `POST` route. We can use `cargo-watch` to compile and run our application so that we don't have to rebuild every time we make changes to our app. Let's install and use `cargo-watch` by running the following commands on our terminal:

```
cargo install cargo-watch
cargo watch -x run
```

Handling 404 routes with Rocket

Let's create a new route for handling 404 responses for nonexistent routes. We'll name our route `not_found` and call it whenever a user requests for a route that does not exist. Let's paste the following block of code before the `main` function:

```
#[catch(404)]
fn not_found(req: &Request) -> String {
    format!("Oh no! We couldn't find the requested path '{}'",
req.uri())
}
```

In the above block, we started by using the `#[catch(404)]` attribute to tell Rocket to return a 404 error when this route is called. We then defined our `not_found()` function and supplied it a `req` parameter with type `Request` and specified `String` for its return type. Finally, we returned our error message and included the requested path using the `req.uri()` method.

Before we continue, let's import the `Request` type we just used in our `not_found` route by adding the following line to our import statements:

```
use rocket::Request;
```

Our imports should now look like this:

```
use rocket::Request;
use rocket::response::content::Json;
use rocket::request::Form;
```

Next, let's modify our `Rocket` instance in the `main()` function to this:

```
rocket::ignite()
  .register(catchers![not_found])
  .mount("/api", routes![hello])
  .launch();
```

We called the `register()` method in our Rocket instance with our `not_found` route via the `catchers!` macro. To test our `not_found` route, let's navigate to a path that does not exist from our browser or API client. For example, when we navigate to `localhost:8000/api/nothingness` , we should get the following response:

```
Oh no! We couldn't find the requested path '/api/nothingness'
```

Using Rocket to render HTML templates

Let's explore rendering HTML templates with Rocket. We'll start by creating a new `GET` route with attribute `#[get("/")]` just before the `hello` route we created earlier:

```
#[get("/")]
fn index() -> Template {
}
```

We'll use this new route for our application's landing page. Notice that we named our function `index` , and this time, our function's return type is `Template` . Let's import the `Template` type by adding the following line to our import statements:

```
use rocket_contrib::templates::Template;
```

Next, we'll include the following code in our `./cargo.toml` file, right after the dependencies section:

```
[dependencies.rocket_contrib]
version = "0.4.5"
features = ["handlebars_templates"]
```

This adds support for rendering the `handlebars_templates` engine in our app. Rocket also has support for the Tera template engine. We can include by either replacing the `handlebars_templates` in features, or add it like this:

```
features = ["handlebars_templates", "tera_templates"]
```

Now, we can create the `./templates` folder in our project's root directory. This is where Rocket will look for our template files by default. Let's create a new file `home.hbs` in the `./templates` directory and paste the following code inside it:

```
<head>
  <style type="text/css">
    body {
      background: #1a6875;
      font-family: Arial, Helvetica, sans-serif;
    }
    .left-side {
      max-width: 45%;
      text-align: right;
      position: absolute;
      top: 15%;
      left: 0;
    }
    .title {
      color: #fff;
      font-size: 5em;
    }
    .sub-title {
      color: #fff;
```

If you're not familiar with the Handlebars templating language, you can check here to go through their documentation. It's very similar to regular HTML code. If you notice, inside our `<body>` tag and in the `left-side` div, we have a paragraph with class of `sub-title` that contains `Hello {{first_name}} {{last_name}}😊` . We'll supply these variables when rendering our `home.hbs` file.

Let's go back to our index route in the `./src/main.rs` file. Inside the function, we'll create a struct named `Context` and use Serde to implement Serialize on our struct. Our struct will define the type for the variables our template file is expecting:

```
#[derive(Serialize)]
struct Context {
  first_name: String,
  last_name: String
}
```

We'll also import the `Serialize` type that we just used for our `Context` struct:

```
use serde::Serialize;
```

Our import statements should now look like this:

```
use rocket::Request;
use rocket::response::content::Json;
use rocket::request::Form;
use rocket_contrib::templates::Template;
use serde::Serialize;
```

Next, let's add it to our dependencies in the `cargo.toml` file:

```
[dependencies]
rocket = "0.4.5"
serde = { version = "1.0", features = ["derive"] }
```

Back to our `./src/main.rs` file — in the `index()` function, we'll declare a new variable named `context` and use the `Context` type we created earlier to provide it its values:

```
let context = Context {
  first_name: String::from("Jane"),
  last_name: String::from("Doe")
};
```

Now that we've added the values we used in the `home.hbs` file, let's return our template with the data we just created:

```
Template::render("home", context)
```

After this is done, our index route should look like this:

```
#[get("/")]
fn index() -> Template {
  #[derive(Serialize)]
  struct Context {
    first_name: String,
    last_name: String
  }
  let context = Context {
    first_name: String::from("Ebenezer"),
    last_name: String::from("Don")
  };
  Template::render("home", context)
}
```

To make our template renderable, we'll need to register it. For this, we'll attach the `Template::fairing()` method on our Rocket instance with `.attach(Template::fairing())`, and then mount our `index` route and use `"/"` as its base. Our `main()` function should look like this after we're done:

```
fn main() {
  rocket::ignite()
    .register(catchers![not_found])
    .mount("/", routes![index])
    .mount("/api", routes![hello, new_book])
    .attach(Template::fairing())
    .launch();
}
```

Notice that we mounted the `index` separately from the `hello` and `new_book` routes. Since this is our landing page, we're using a different `base` path `"/"` so that we only need to navigate to `localhost:8000` to see our rendered template.

Our `./src/main.rs` file should now look like this:

```
#![feature(decl_macro)]
#[macro_use] extern crate rocket;
use rocket::Request;
use rocket::response::content::Json;
use rocket::request::Form;
use rocket_contrib::templates::Template;
use serde::Serialize;

#[derive(FromForm, Debug)]
struct Book {
  title: String,
  author: String,
  isbn: String
}

#[get("/")]
fn index() -> Template {
  #[derive(Serialize)]
  struct Context {
```

Now, when we run our application and navigate to `localhost:8000` , we should see a page similar to this on our browser:

Conclusion

In this article, we've introduced Rust for the web through the Rocket framework. We covered the basics of Rocket, how to set up up web APIs, response types, error handling, and rendering HTML through the Handlebars template engine.

While Rocket is a good fit for building web APIs, it might not be the best choice for handling frontend rendering, like we did in the last part of this article. However, Rust shines in this area through the Yew framework, which was built for creating multi-threaded frontend web apps with WebAssembly.

Overall, Rocket makes writing web applications relatively fast compared to other web frameworks, and it does this with very little boilerplate code. Here's a link to the GitHub repo for our demo app.

LogRocket: Full visibility into web frontends for Rust apps

Debugging Rust applications can be difficult, especially when users experience issues that are difficult to reproduce. If you're interested in monitoring and tracking performance of your Rust apps, automatically surfacing errors, and tracking slow network requests and load time, try LogRocket.

LogRocket is like a DVR for web and mobile apps, recording literally everything that happens on your Rust app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting metrics like client CPU load, client memory usage, and more.

Modernize how you debug your Rust apps — start monitoring for free.

Ebenezer Don Follow

Full-stack software engineer with a passion for building meaningful products that ease the lives of users.

#rust

7 Replies to “Building web apps with Rust using the Rocket framework”

TitanEric Says:
November 5, 2020 at 8:43 am

Reply

Hi, it’s a great post!!!
Just a little typo after defining POST API.

It should be
“`rust
rocket::ignite()
.mount(“/api”, routes![hello, new_book])
.launch();
“`

Ebenezer Don Says:
November 10, 2020 at 11:35 am

Reply

Thanks, TitanEric! You're right, we'll make the correction.

Marit Says:
December 7, 2020 at 5:40 am

Reply

Thanks for this post! Being totally new to rust, also to command line stuff etc; how would I test the post to /book? I tried some things in the address bar, as well as some curl commands but can't seem to understand.. Thanks!

Ebenezer Don Says:
December 7, 2020 at 9:59 am

Reply

Hi Marit, I'm glad you found the article helpful! To easily make the POST requests, you'll need an API client like Postman (https://www.postman.com/) or Insomnia (https://insomnia.rest).

Please let me know if this helps.

Marit Says:
December 8, 2020 at 12:19 pm

Reply

Thanks, I will look into that!

Ze Says:
December 12, 2020 at 5:00 pm

Reply

just in the begginning I see a flaw in your code: the json response is invalid since it can't understand single quotes, so you should change it all in the response...
it goes a little like this:
Json("{

\”status\”: \”success\”,
\”message\”: \”Hello API!\”
}”)

Ebenezer Don Says:
December 14, 2020 at 8:38 am

Reply

Hi Ze, it’s actually correct. Here’s a link to to the Rocket documentation: https://rocket.rs/v0.4/guide/responses/

Leave a Reply

Enter your comment here...