Web Servers with Rust and Actix

Rust Meetup 2021/05/20

Introduction

Agenda

- Motivation
- Basic project setup
- Return data
- Receive data
- State management
- App structure

Me

Michael Gerhäuser

Software Engineer @ Method Park

Mobile Apps & Web Apps

Rust Meetup & Rust Workshops

Why use Rust?

- Static types
- Rust brings compile time memory safety
- Smaller memory foot print
- High performance *[1]*

[1] https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=fortune

Why use Actix?

Basics

Project setup & crates

Create a new binary project

```
cargo new --bin <awesome project>
```

Add the `actix-web` crate to it

```
1 [dependencies]
2 actix-web = "3"
```

Declare routes

```
1 use actix_web::{get, HttpResponse, Responder};
2
3 #[get("/")]
4 async fn hello() -> impl Responder {
5     HttpResponse::Ok().body("Hello world!")
6 }
```

Instantiate the server

```rust
use actix_web::{HttpServer, App};

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .service(hello)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Routes in App factory

```
 1 async fn hello() -> impl Responder {
 2     HttpResponse::Ok().body("Hello world!")
 3 }
 4
 5 #[actix_web::main]
 6 async fn main() -> std::io::Result<()> {
 7     HttpServer::new(|| {
 8         App::new()
 9             .route("/", web::get().to(hello))
10             .route("/hello", web::get().to(hello))
11     })
12     .bind("127.0.0.1:8080")?
13     .run()
14     .await
15 }
```

Modify return codes

```
1 #[get("/topsecret")]
2 async fn unauthorized() -> impl Responder {
3     HttpResponse::Unauthorized()
4 }
```

More: *https://docs.rs/actix-web/3.3.2/actix_web/struct.HttpResponse.html*

Format responses

```rust
use serde::{Serialize};

#[derive(Serialize)]
struct Response {
    name: String,
}

#[get("/json")]
async fn json() -> impl Responder {
    let response = Response { name: "Rust".to_string() };
    HttpResponse::Ok().json(response)
}
```

Simplify returns

Instead of returning `HttpResponse` objects you can simply return strings as well.

```
1 #[get("/")]
2 async fn index() -> String {
3     "Hello".to_owned()
4 }
```

This response will always have the content type `text/plain`.

Input processing

Extractors

Until now, our handlers didn't have any input parameters - because they didn't process any input yet.

To handle all kinds of handler input, e.g. path and query parameters, payload, app state, Actix uses extractors.

The first extractor we take a look at is the Path extractor.

Path extractor

```
#[get("/user/{id}")]
async fn get_user(path: web::Path<(u32)>) -> impl Responder {
    let response = format!("Searching for user {}", path.0.0);
    HttpResponse::Ok().body(response)
}
```

Path extractor

```rust
#[derive(Deserialize)]
struct ArticlePath {
    id: u32,
    name: String,
}

#[get("/article/{id}/{name}")]
async fn get_article(article: web::Path<ArticlePath>) -> impl
    format!("id {}, name {}", article.id, article.name)
}
```

Query extractor

```rust
#[derive(Deserialize)]
struct OrderQuery {
    order_id: String,
}

#[get("/order")]
async fn get_order(info: web::Query<OrderQuery>) -> String {
    format!("Requesting order {}", info.order_id)
}
```

JSON extractor

```rust
#[derive(Deserialize, Debug)]
struct DataSample {
    id: u32,
    name: String,
}

#[post("/payload")]
async fn post_payload(sample: web::Json<DataSample>) -> String
    format!("Payload: {:?}", sample)
}
```

Application State

Application State

First we take a look on non-persistent application state, e.g. handle session data or other ephemeral data.

Define application state

```rust
struct AppState {
    counter: Mutex<u32>,
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let state = web::Data::new(AppState {
        counter: Mutex::new(0),
    });

    HttpServer::new(move || {
        App::new()
            .app_data(state.clone())
            .service(get_count)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Access the app state

Access is granted via the `Data` extractor

```
#[get("/count")]
async fn get_count(state: web::Data<AppState>) -> String {
    let mut counter = state.counter.lock().unwrap();
    *counter += 1;

    format!("Number of times requested: {}", counter)
}
```

Persistent state

The connection to a persistent data layer, e.g. database, is handled similarly.

```
let pool = /* Connect to database */

HttpServer::new(move || {
    App::new()
        .data(pool.clone())
        …
})
.bind("127.0.0.1:8088")?
.run()
.await
```

data() vs app_data()

`app_data()` accepts `web::Data()` and is shared across all application threads.

`data()` accepts arbitrary data that it wraps in `web::Data`. A new instance of the data is potentially created for each thread.

Thread Local State

```
struct LocalState {
    counter: std::cell::Cell<u32>,
}

#[get("/count")]
async fn get_count(local_state: web::Data<LocalState>) -> String
    local_state.counter.set(local_state.counter.get() + 1);

    format!("Requested {} times", local_state.counter.get())
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(move || {
        App::new()
            .data(LocalState {
                counter: std::cell::Cell::new(0)
            })
    })
    …
}
```

Synchronization

Be careful with synchronization primitives like `Mutex` or `RwLock`. The actix-web framework handles requests asynchronously. By blocking thread execution, all concurrent request handling processes would block. If you need to share or update some state from multiple threads, consider using the tokio synchronization primitives.

https://actix.rs/docs/extractors/

Synchronization

As a rule of thumb, using a synchronous mutex from within asynchronous code is fine as long as contention remains low and the lock is not held across calls to `.await`.

https://tokio.rs/tokio/tutorial/shared-state

Synchronization

The std mutex is fine to use in async code as long as locks aren't held across await points;

https://github.com/actix/actix-website/issues/201

Application structure

Extract handler definition

We can put the route definition into a separate module

```
pub fn user_routes(config: &mut web::ServiceConfig) {
    config
        .route("", web::get().to(get_users))
        .route("", web::post().to(post_user))
        .route("/{id}", web::get().to(get_user))
        …;
}
```

Apply routes to App

Tell the App about the handler config

```
1       HttpServer::new(move || {
2           App::new()
3               .service(web::scope("/users").configure(user_routes
4       })
5       .bind("127.0.0.1:8088")?
6       .run()
7       .await
```

Usage

POSTing to `/users` runs the `post_user()` handler.

GETting `/users/12` invokes the `get_user()` function.

Questions?

Links

- https://actix.rs
- https://tokio.rs

Thank you!

Exercise

Create a guestbook with in-memory storage.