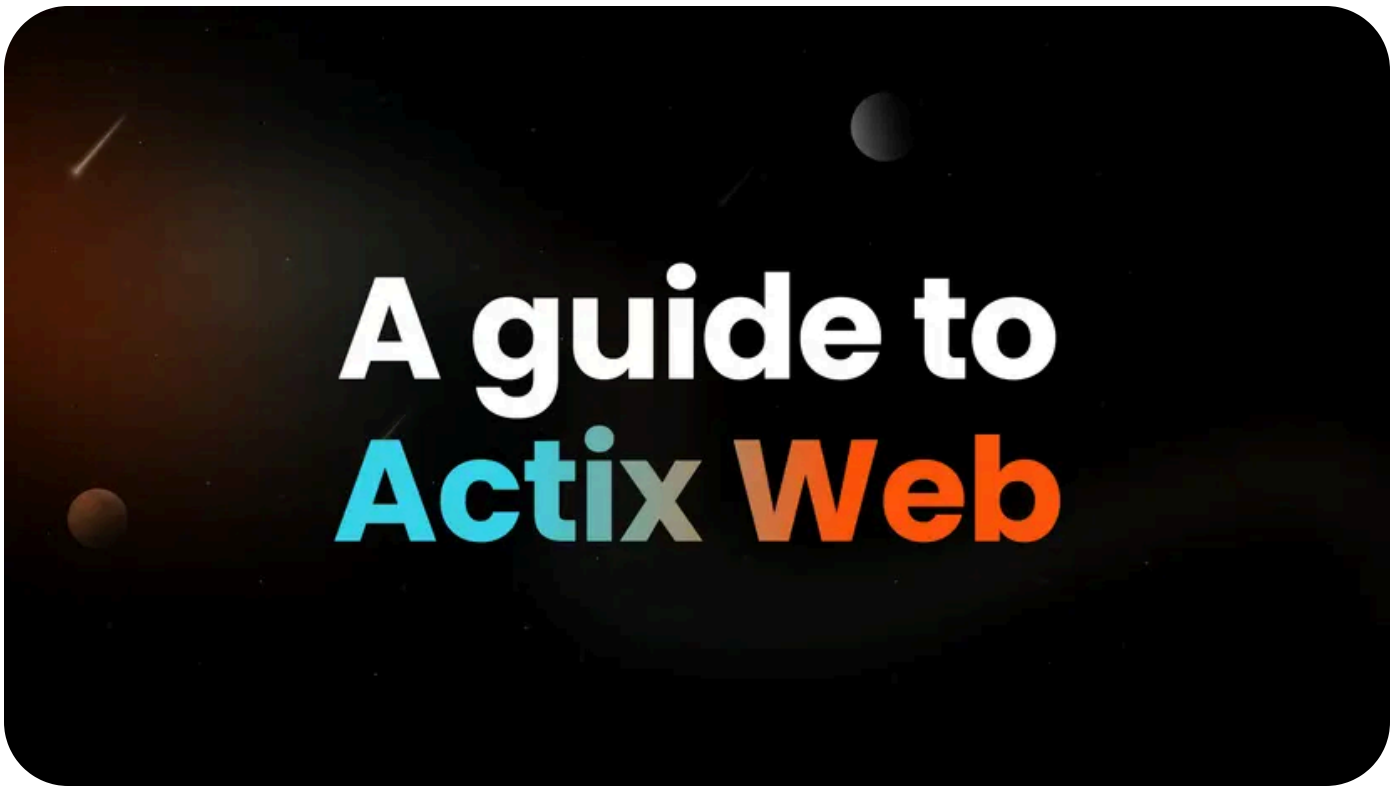
Getting Started with Actix Web in Rust

Joshua Mo • 15 December 2023



To this day, remains an extremely formidable competitor in the Rust web backend ecosystem. It's no wonder that it's lasted so long; despite any impact that previous events may have had on it, it's still going strong and is one of the most recommended web frameworks in Rust. Originally based on the actor framework of the same name (`actix`), it has since moved away, and `actix` is now only really used for websocket endpoints.

This article will primarily be concerning v4.4.

Getting Started with Actix Web

To get started, you want to use `cargo init example-api` to generate your project, `cd` into the folder, and then use the following to add the `actix-web` crate to your project:

```
cargo add actix-web
```

Now you're pretty much ready to start! If you'd like to copy the boilerplate for getting straight to writing your app, here it is:

```
use actix_web::{web, App, HttpServer, Responder};

#[get("/")]
async fn index() -> impl Responder {
    "Hello world!"
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().service(
            // prefixes all resources and routes attached to it...
            web::scope("/")
                // ...so this handles requests for `GET /app/index.html`
                .route("/", web::get().to(index)),
        )
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Routing in Actix Web

When using , mostly any function that returns the `actix_web::Responder` trait can be used as a route. See below for a basic Hello World example:

```
#[get("/")]
async fn index() -> impl Responder {
    "Hello world!"
}
```

This handler function can then be fed into an `actix_web::App` which then gets passed in as a parameter to a `HttpServer`:

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().service(
            // prefixes all resources and routes attached to it...
            web::scope("/")
                // ...so this handles requests for the base route
                .route("/index.html", web::get().to(index)),
        )
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Now whenever you go to `/index.html`, it should return "Hello world!". However, you may find this approach a little bit lacking if you want to create

multiple mini-router types and then merge them all into the app at the end. In this case, you will want the `ServiceConfig` type, which you can also write like this:

```
use actix_web::{web, App, HttpResponse};

// this function could be located in different module
fn config(cfg: &mut web::ServiceConfig) {
    cfg.service(web::resource("/test")
        .route(web::get().to(|| HttpResponse::Ok()))
        .route(web::head().to(|| HttpResponse::MethodNotAllowed()))
    );
}

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

Extractors in Actix Web are exactly that: type-safe request implementations that, when passed into a handler function, will attempt to extract the relevant data from the request for the handler function. For example, the `actix_web::web::Json` extractor will attempt to extract JSON from the request body. To successfully deserialize JSON successfully however, you need to use the `serde` crate - ideally with the `derive` function that adds automatic Deserialize and Serialize derive macros for your structs. You can install serde by executing the following command:

```
cargo add serde -F derive
```

Now you can use it as a derive macro like this:

```rust
use actix_web::web;
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
    username: String,
}

// deserialize `Info` from request's body
#[post("/submit")]
async fn submit(info: web::Json<Info>) -> String {
    format!("Welcome {}!", info.username)
}
```

Actix Web also has support for paths, queries and forms. You'll also need to use `serde` here as well - although with paths, you will additionally want to use the Actix Web routing macro to declare where exactly the path parameters are. We can find examples of all 3 of these in action below:

```rust
#[derive(Deserialize)]
struct Info {
    username: String,
}

// extract path info using serde
#[get("/users/{username}")] // <- define path parameters
async fn index(info: web::Path<Info>) -> String {
  format!("Welcome {}!", info.username)
}

// data is passed in here through query parameters in the URL
// for example, google.com/?hello=world
#[get("/")]
async fn index(info: web::Query<Info>) -> String {
    format!("Welcome {}!", info.username)
}

// data is passed into the Form extractor through a HTML Form element
#[post("/")]
async fn index(form: web::Form<Info>) -> actix_web::Result<String> {
    Ok(format!("Welcome {}!", form.username))
}
```

Interested in writing your own extractor? You can do that! Writing your own extractor simply requires that you implement the `FromRequest` trait. Check out this code for a that shows you exactly how it's done:

```
use actix_web::dev::Payload;
use actix_web::{FromRequest,
  http::header::Header as ParseHeader,
  HttpRequest, error::ParseError
};

#[derive(Clone, PartialEq, Eq, PartialOrd, Ord, Debug)]
pub struct Header<T>(pub T);

impl<T> FromRequest for Header<T>
where
    T: ParseHeader,
{
    type Error = ParseError;
    type Future = Ready<Result<Self, Self::Error>>;

    #[inline]
    fn from_request(req: &HttpRequest, _: &mut Payload) -> Self::Future {
        match T::parse(req) {
            Ok(header) => ok(Header(header)),
            Err(e) => err(e),
        }
    }
}
```

Note that the `T: ParseHeader` trait bound is specific to this trait implementation because in order for a header to be a valid header, it needs to be able to be parsed successfully as a header, with the error implementing `actix_web::error::Error`. Although we also have `extract` as a provided method, `from_request` is the only required method to be implemented here, which returns `Self::Future`. That is to say, we need to return a result that's ready to be awaited - you can find more about the Ready struct . Other extractors, like the JSON extractor, also allow you to change their configuration - you can find out more about it in .

Generally speaking, responses only need to implement the `actix_web::Responder` trait to be able to respond. Although there is a when it comes to actual response types so you should not generally need to implement your own types, there can be specific use cases where this is helpful; for example, being able to document all the types of responses that your application may have.

Adding a Database

Normally when setting up a database, you might need to set up your database connection:

```
use sqlx::PgPoolOptions;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let dbconnection = PgPoolOptions::new()
        .max_connections(5)
        .connect(r#"<db-connection-string-here>"#).await;

    //... rest of your code
}
```

You would then need to provision your own Postgres instance, whether installed locally on your computer, provisioned through Docker or something else. However, with Shuttle we can eliminate this as the runtime provisions the database for you:

```
use actix_web::{get, web::ServiceConfig};
use shuttle_actix_web::ShuttleActixWeb;

#[shuttle_runtime::main]
async fn actixweb(
```

```
    #[shuttle_shared_db::Postgres] pool: PgPool,
) -> ShuttleActixWeb<impl FnOnce(&mut ServiceConfig) + Send + Clone + 'static> {
    let state = AppState { pool };

    // .. the rest of your code
}
```

Locally this is done through Docker, but in deployment there is an overarching process that does this for you! No extra work required. We also have an AWS RDS database offering that requires zero AWS knowledge to set up - visit to find out more.

App State in Actix Web

Routing is great and all (as well as adding databases being pretty easy!) but when you need to store variables, you may be wanting to look for something that lets you store and use them across your application. This is where shared mutable state comes in: you declare it while building your service across your whole application, then you can use it as an extractor in your handler functions. For example, you might need to share a database pool, a counter or a shared hashmap of websocket subscribers. You can declare and use state like this:

```
use sqlx::PgPool;

#[derive(Clone)]
struct AppState {
    db: PgPool
}

#[get("/")]
async fn index(data: web::Data<AppState>) -> String {
    let res = sqlx::query("SELECT 'Hello World!'").fetch_all(&data.db).await.unwr
```

```
        format!("{res}")
    }
```

You can then implement it like this:

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let db = connect_to_db();
    let state = web::Data::new(AppState { db });

    HttpServer::new(move || {
        // move app state into the closure
        App::new()
            .app_data(state.clone()) // <- register the created data
            .route("/", web::get().to(index))
    })
    .bind(("127.0.0.1", 8080))?
    .run()
    .await
}
```

Middleware in Actix Web

Within , middleware is used as a medium for being able to add general functionality to a (set of) route(s) by taking the request before the handler function runs, carrying out some operations, running the actual handler function itself and then the middleware does additional processing (if required). By default, Actix Web has several default middlewares that we can use, including logging, path normalisation, access external services and modifying application state (through the `ServiceRequest` type).

See below for an example of how to implement a default Logger middleware:

```rust
use actix_web::{middleware::Logger, App};

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    // access logs are printed with the INFO level so ensure it is enabled by de
    env_logger::init_from_env(env_logger::Env::new().default_filter_or("info"));

    let app = App::new()
        .wrap(Logger::default());

    // ... rest of your code
}
```

Additionally, you can also write your own middleware in Actix Web! For many use cases, we can use the handy `middleware::from_fn` helper from the sister crate `actix-web-lab` (which will be promoted to `actix-web` itself in an upcoming release). For example, printing a messages at different parts of the request handling flow like this:

```rust
use actix_web::{body::MessageBody, dev::{ServiceRequest, ServiceResponse}};
use actix_web_lab::middleware::{from_fn, Next};

async fn print_before_and_after_handler(
    req: ServiceRequest,
    next: Next<impl MessageBody>,
) -> Result<ServiceResponse<impl MessageBody>, Error> {
    println!("Hi from start. You requested: {}", req.path());
    let res = next.call(req).await?;
    println!("Hi from response");
    Ok(res)
}

let app = App::new()
    .wrap(from_fn(print_before_and_after_handler))
    .route(
        "/",
        web::get().to(|| async { "Hello from handler!" }),
);
```

To be able to write more complex middleware, we actually need to implement two traits - `Service<Req>` which is for implementing the actual middleware itself as well as `Transform<S, Req>` which is the required for the builder for the actual Service that handles requests (in terms of when we're building our service, we will actually use the builder, not the middleware! The outer service will automatically call the middleware upon detecting a HTTP request).

For an actual middleware implementation, let's have a look at writing a middleware that simply prints messages. We can create the builder for the middleware by implementing the `Transform` trait:

```
use std::{future::{ready, Ready, Future}, pin::Pin};

use actix_web::{
    dev::{forward_ready, Service, ServiceRequest, ServiceResponse, Transform},
    web, Error,
};

pub struct SayHi;

// `S` - type of the next service
// `B` - type of response's body
impl<S, B> Transform<S, ServiceRequest> for SayHi
where
    S: Service<ServiceRequest, Response = ServiceResponse<B>, Error = Error>,
    S::Future: 'static,
    B: 'static,
{
    // setting up the types for the middleware to work
    type Response = ServiceResponse<B>;
    type Error = Error;
    type InitError = ();
    type Transform = SayHiMiddleware<S>;
    type Future = Ready<Result<Self::Transform, Self::InitError>>;

    // this immediately returns the middleware
    fn new_transform(&self, service: S) -> Self::Future {
        ready(Ok(SayHiMiddleware { service }))
    }
}
```

Now we can write the middleware itself! Internally the middleware must implement a generic type - which then gets declared in the `Service` trait. Note that we manually re-implement a type from `futures_util` called `LocalBoxFuture` - that is to say, a future that doesn't require the `Send` trait and is safe to use because it implements `Unpin` on dereferencing, which automatically cancels any previous thread-safety guarantees.

```
pub struct SayHiMiddleware<S> {
    /// The next service to call
    service: S,
}

// This future doesn't have the requirement of being `Send`.
// See: futures_util::future::LocalBoxFuture
type LocalBoxFuture<T> = Pin<Box<dyn Future<Output = T> + 'static>>;

// `S`: type of the wrapped service
// `B`: type of the body - try to be generic over the body where possible
impl<S, B> Service<ServiceRequest> for SayHiMiddleware<S>
where
    S: Service<ServiceRequest, Response = ServiceResponse<B>, Error = Error>,
    S::Future: 'static,
    B: 'static,
{
    type Response = ServiceResponse<B>;
    type Error = Error;
    type Future = LocalBoxFuture<Result<Self::Response, Self::Error>>;

    // This service is ready when its next service is ready
    forward_ready!(service);

    fn call(&self, req: ServiceRequest) -> Self::Future {
        println!("Hi from start. You requested: {}", req.path());

        // A more complex middleware, could return an error or an early response

        // we do not immediately await this, which means nothing happens
        // this future gets moved into a Box
        let fut = self.service.call(req);

        Box::pin(async move {
            // this future gets awaited now
            let res = fut.await?;
```

```
            // we can now do any work we need to after the request
            println!("Hi from response");
            Ok(res)
        })
    }
}
```

Now that we've written our middleware, we can now add it to our App:

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    let app = App::new()
         .wrap(SayHi);

    // ... rest of your code
}
```

Static Files in Actix Web

Plain, no-frills static file serving in is done through the `actix_files` crate - to add it, you simply need to add it through Cargo like so:

```
cargo add actix-files
```

Setting up a route for static file serving would look like this:

```
use actix_files::NamedFile;
use actix_web::{HttpRequest, Result};
use std::path::PathBuf;
```

```
#[get("/")]
async fn index(req: HttpRequest) -> Result<NamedFile> {
    let path: PathBuf = req.match_info().query("filename").parse().unwrap();
    Ok(NamedFile::open(path)?)
}
```

This route allows us to serve any file that can be found and matches the filename - for example, if we have a base route that serves this route, if we then run our app and go to `/index.html`, the route will try to look for a file named `index.html` in the project root.

Note that you should *not* under any circumstances try and use a path tail with `.*` to return a `NamedFile` - this has serious security implications and will open your web service up to path traversal! This is documented in the Actix Web docs , and you can find more about path traversal attacks . As a guard against this, you can attempt to validate the path file is correct or is not attempting to traverse outside of the intended folder by using `std::fs::canoncalize`).

However, this is a bit clumsy when you need to serve multiple files - particularly if you need to serve a folder of HTML files, for instance. To be able to serve a folder of files from your web service, the best way to do this would be to use `actix_files::Files` and attach it to your `App`:

```
use actix_files as fs;
use actix_web::{App, HttpServer};

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new().service(
            fs::Files::new("/static", ".")
                .use_last_modified(true),
        )
    })
    .bind(("127.0.0.1", 8080))?
    .run()
```

```
        .await
}
```

Note that you can also augment the `Files` struct with several options like showing the file directory itself at the base route (for the file service) and allowing hidden files, which you can find more about .

Additionally, we can also use the power of HTML templating with to supercharge our HTML file serving! We can get started like so:

```
cargo add askama askama-actix-web -F askama/with-actix-web
```

This adds itself as well as the `Responder` trait implementation for the `askama::Template` type. expects your files to be in a subfolder of the project root called `templates` by default, so let's create the folder and then create an `index.html` file with the following HTML code in:

```
Hello, {{name}}!
```

To use Askama in our app, we need to declare a struct that uses the `Template` derive macro and use askama's `template` macro to point the struct to the file that we want it to use:

```
#[derive(Template)]
#[template(path = "index.html")]
struct IndexTemplate<'a> {
    name: &'a str
}

#[get("/")]
async fn index_route() -> impl Responder {
    IndexTemplate { name: "Shuttle" }
}
```

Then we can add it as a regular handler function in our Actix Web service and we're good to go! When you go to a path that returns `index_route`, you should see "Hello, Shuttle!" as the HTML response.

Interested in learning more about Askama? We have a Shuttle Launchpad newsletter issue that talks about this! You can find it .

Deploying

Deployment with Rust backend programs, in general, can be less than ideal due to having to use Dockerfiles, although if you are experienced with Docker already this may not be such an issue for you - particularly if you are using `cargo-chef`. However, if you're using Shuttle you can just use `cargo shuttle deploy` and you're done already. No setup is required.

Finishing Up

Thanks for reading! Actix Web is a strong framework that you can use to boost your Rust portfolio and is a great framework to do a deep dive into Rust with if you're looking to build your first Rust API.

Interested in more?

- Check out this .
- Compare how Actix Web measures up against other frameworks in .