Web Development with Rust and Iron

Danilo Bargen (@dbrgn)

April 16, 2017

Coredump Rapperswil

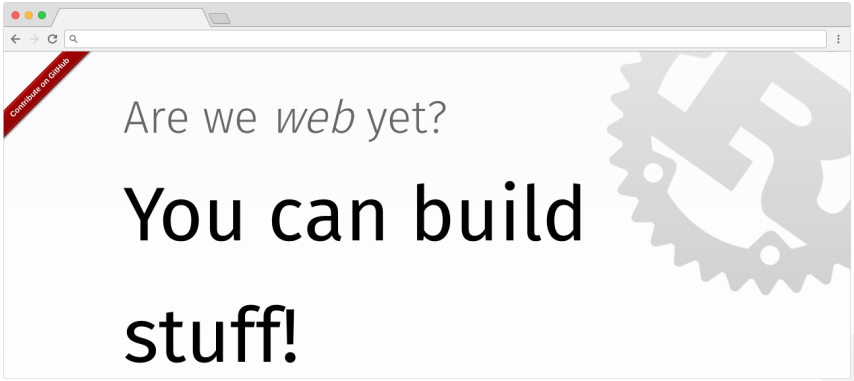
Outline

1. Intro
2. Iron
3. Handlers
4. Modifiers
5. Routing
6. Middleware
7. Error Handling
8. Plugins

Intro

Are We Web Yet?

http://www.arewewebyet.org/



Are We Web Yet?

Current state of affairs:

Rust has a mature HTTP stack and various frameworks enable you to build APIs and backend services quickly. While increasingly more databases drivers become available, ORMs and connections to external services (like search or worker queues) are still scarce. Looking farther, it doesn't necessarily get better. Though there is significant support for base needs (like data compression or logging), a lot more web-specific needs are still unmet and immature.

`http://www.arewewebyet.org/`

In Summary

You can start building stuff, but there are still rough edges and the tooling isn't quite there yet.

Today...

..we'll look at Iron, a web framework built on top of Hyper.

`http://ironframework.io/`

Iron

What is iron?

«Iron is a fast and flexible middleware-oriented server framework that provides a small but robust foundation for creating complex applications and RESTful APIs. No middleware are bundled with Iron - instead, everything is drag-and-drop, allowing for ridiculously modular setups.» — www.ironframework.io

Getting started (1/3)

Cargo.toml:

```
[dependencies]
iron = "^0.5"
```

Getting started (2/3)

src/main.rs:

```
extern crate iron;

use iron::prelude::*;
use iron::status;

fn hello_world(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello World")))
}

fn main() {
    Iron::new(hello_world).http("localhost:3000").unwrap();
}
```

Getting started (3/3)

```
$ cargo run
```

Starting a Server

Let's take a closer look at this snippet:

```
fn main() {
    Iron::new(hello_world)
        .http("localhost:3000")
        .unwrap();
}
```

What happens here?

- We create a new `Iron` from a handler
- We start a HTTP server on `localhost:3000`
- We unwrap the `HttpResult<Listening>`

Handlers

Handlers

But what is a “handler”?

```
[-]fn new(handler: H) -> Iron<H>
```

Instantiate a new instance of `Iron`.

This will create a new `Iron`, the base unit of the server, using the passed in `Handler`.

The Handler Trait

Let's take a look at this `Handler` trait.

Trait iron::middleware::Handler [-] [src]

```
pub trait Handler: Send + Sync + 'static {
    fn handle(&self, _: &mut Request) -> IronResult<Response>;
}
```

[-] `Handler`s are responsible for handling requests by creating Responses from Requests.

That answers our question: A `Handler` takes a `Request` and returns a `Response`.

Our Hello World Handler

Ok, let's go back to the handler in our example.

```
fn hello_world(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((status::Ok, "Hello World")))
}
```

It's a bare function, not a struct. This works, because of this generic trait implementation provided by Iron:

```
impl<F> Handler for F
  where F: Send + Sync + 'static +
           Fn(&mut Request) -> IronResult<Response>
```

The Response Object

So what's this `Response`?

```
Response::with((status::Ok, "Hello World"))
```

The definition looks like this:

```
pub struct Response {
    pub status: Option<Status>,
    pub headers: Headers,
    pub extensions: TypeMap,
    pub body: Option<Box<WriteBody>>,
}
```

Modifiers

Modifiers

This is the `Response::with` signature:

```
fn with<M: Modifier<Response>>(m: M) -> Response
```

It takes a single parameter of type `M: Modifier<Response>` and generates a `Response` from that. So what is a `Modifier`?

Modifiers

The documentation of the `modifier` crate define it as follows:

> *«Overloadable modification through both owned and mutable references to a type with minimal code duplication.»*

Let's take a look at the signature:

```
pub trait Modifier<F: ?Sized> {
    fn modify(self, &mut F);
}
```

So a `Modifier<Response>` simply modifies a `Response`.

Modifiers

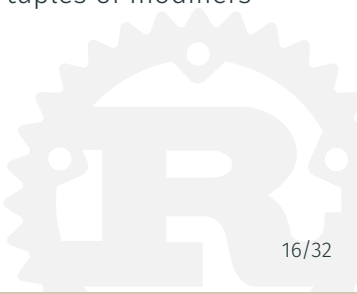
Going back to our example, we can now understand it:

```
Response::with((status::Ok, "Hello World"))
```

The parameter values in that tuple are both modifiers:

- The `status::Ok` modifier modifies the `status` attribute
- The string `"Hello World"` modifies the `body` attribute

Furthermore, the `Modifier` trait is also implemented for n-tuples of modifiers (up to length 6 currently).

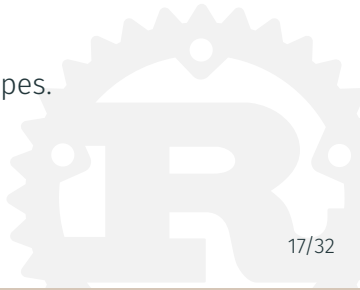


More Modifiers

That system of modifiers is quite smart, since it allows for nice composition. Here are some other modifier impls:

- An `&[u8]` modifies raw response body bytes.
- A `File` or `&Path` or `PathBuf` sets the response body to the contents of the file at this path. Furthermore, it also sets the content type based on the file mime type.
- A `Header` sets a response header.
- A `Redirect` creates a redirect response.

The `Modifier` trait can also be implemented for custom types.



Back to Handlers

Now that we know how modifiers work, let's go back to the handler.

```
fn main() {
    Iron::new(hello_world).http("localhost:3000").unwrap();
}
```

As you can imagine, a single handler is usually not enough in a web application. We want to compose handlers. Luckily, we can do that.

Routing

Routing

There's a crate called `router` that provides a `Router` struct which is also a `Handler`.

```
extern crate router;

let mut router = router::Router::new();
router.get("/", index, "index");
router.get("/:query", queryHandler, "query");
router.post("/", postHandler, "post");

Iron::new(router).http("localhost:3000").unwrap();
```

Middleware

Middleware

A `Modifier` only allows to change a single response. But it cannot intercept the entire request-response cycle. If we need to do that (e.g. for caching, authentication, logging, etc) we need a middleware.

Iron comes with only basic modifiers for setting the status, body, and various headers, and the infrastructure for creating modifiers, plugins, and middleware. No plugins or middleware are bundled with Iron.

Middleware Types

There are three types of middleware:

- A `BeforeMiddleware` can do pre-processing of a request.
- An `AfterMiddleware` can do post-processing of a response.
- An `AroundMiddleware` wraps a handler and can access both the request and the response.

Middleware is registered together with a handler in a `Chain`.

Let's take a look at a concrete example!

Logging Middleware

Let's create a simple middleware that logs every request to the terminal.

```
fn request_logger(req: &mut Request) -> IronResult<()> {
    println!("{} {}", req.method, req.url);
    Ok(())
}
```

This works, because – similar to handlers – the `BeforeMiddleware` is implemented for certain functions:

```
impl<F> BeforeMiddleware for F
  where F: Send + Sync + 'static +
           Fn(&mut Request) -> IronResult<()>
```

Now we create two handlers and a router:

```
fn hello(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((iron::status::Ok, "Hello")))
}

fn world(_: &mut Request) -> IronResult<Response> {
    Ok(Response::with((iron::status::Ok, "World")))
}

fn main() {
    let mut router = Router::new();
    router.get("/hello", hello, "hello");
    router.get("/world", world, "world");
    ...
```

Logging Middleware

Finally, we wrap the router into a `Chain` and link in our middlware. Since the `Chain` is also a `Handler`, so we can start the HTTP server with it.

```
let mut chain = Chain::new(router);
chain.link_before(request_logger);
println!("Starting server on :3000");
Iron::new(chain).http("localhost:3000").unwrap();
```

This is the log output after two requests:

```
Starting server on :3000
GET http://localhost:3000/hello
GET http://localhost:3000/world
```

Error Handling

Quick disgression: Error handling

I haven't totally figured out error handling yet, but I think this is the gist of it:

- A handler returns an `IronResult<Response>`.
- If you want to do things like returning a 400 error if request validation fails, simply return `Ok(Response::with(status::BadRequest))`.
- You only return an `Err(IronError)` if you want the error to be handled by middleware.
- A middleware can catch the error and turn it into an `Ok(Response)` (error recovery). If it doesn't want to handle that error, it simply returns it again.
- To simplify returning an `IronError`, you can use the `itry!` macro. It works like `try!`, but wraps the error value in `IronError`.

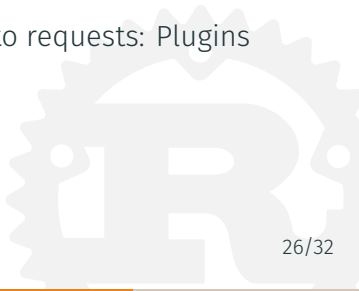
Plugins

Plugins

There's one last missing piece of the puzzle: How do we handle the following problems?

- We want a global database connection pool that every handler should be able to access
- We want to implement session handling, handlers should be able to access the session
- Inside a handler, we want to know whether a user is authenticated

There's a generic mechanism for attaching additional data to requests: Plugins (aka extensions).



User Middleware

Let's start with writing a middleware that checks whether a user has provided his username in the `Authorization` header.

```
use iron::headers::Authorization;

#[derive(Debug)]
struct User { username: String }

fn user_middleware(req: &mut Request) -> IronResult<()> {
    let user = req.headers.get::<Authorization<String>>()
        .map(|header| User { username: header.0.clone() });
    println!("User is {:?}", user);
    Ok(())
}
```

Then we set up that middleware with our hello world handler and send two HTTP requests:

```
$ http GET :3000/
$ http GET :3000/ Authorization:danilo
```

The log output looks like this:

```
Starting server on :3000
User is None
User is Some(User { username: "danilo" })
```

User Middleware

Fine. But now how do we pass that information to the handlers?

One of the fields on a `Request` instance is this one:

```
pub extensions: TypeMap,
```

A `TypeMap` is a map that contains types as keys. If we want to be able to put a value into that map, it needs to implement the `Key` trait.

```
use iron::typemap::Key;
impl Key for User {
    type Value = User;
}
```

User Middleware

Now we can write the user info into the `extensions` field...

```
fn user_middleware(req: &mut Request) -> IronResult<()> {
    let auth = req.headers.get::<Authorization<String>>();
    if let Some(header) = auth {
        let user = User { username: header.0.clone() };
        req.extensions.insert::<User>(user);
    };
    Ok(())
}
```

User Middleware

...and retrieve it from our handler.

```
fn hello_world(req: &mut Request) -> IronResult<Response> {
    let text = match req.extensions.get::<User>() {
        Some(user) => format!("Hello, {}", user.username),
        None => format!("Hello, world"),
    };
    Ok(Response::with((iron::status::Ok, text)))
}
```

User Middleware

Testing that it works:

```
$ http -b GET :3000/
Hello, world
$ http -b GET :3000/ Authorization:Ferris
Hello, Ferris
```

Thank you! Questions?

`www.coredump.ch`

Slides: `URL~will~follow`

Examples: `URL~will~follow`