Современное системное программирование на Rust.

Axum crate.

Design features

- Small web framework that only provides router and API for handlers.
- Mostly depends on tokio and tower ecosystem. *Independence is not a goal* - docs.
- Implemented in safe Rust.

Dependencies

- hyper as a low-level HTTP building block.
- tower for middlewares, timeouts, rate limiting, load balancing.
- tracing for advanced logging.
- matchit for effective URL routing.

Tower crate

Glue between application code, libraries providing middleware and libraries that implement servers and/or clients for various network protocols.

tower::Service

```rust
pub trait Service<Request> {
    type Response;
    type Error;
    type Future: Future<Output = Result<Self::Response, Self::Error>>;
    fn poll_ready(&mut self, cx: &mut Context<'_>) -> Poll<Result<(), Self::Error>>;
    fn call(&mut self, req: Request) -> Self::Future;
}
```

Service can be thought as function that takes Request and asynchronously produces Result<Response, Error>.

tower::Layer

```rust
pub trait Layer<S> {
    type Service;
    fn layer(&self, inner: S) -> Self::Service;
}
```

Service decorator allowing to reuse components and combine multiple services together.

Log layer + service showcase

Commonly used middlewares

Handlers under the hood

Your function need to implement Handler trait in order to be valid handler, however axum contains blanket implementations for:

- Async fn's.
- Take <= 16 arguments that implement Send.
  - All except the last argument implement FromRequestParts.
  - The last argument implements FromRequest.
- Returns a Fut: Send.
- Fut::Output: IntoResponse.
- Closure: Send + Clone and be 'static.

Check if given function is valid axum::Handler?

```
async fn root() -> &'static str {
    "Hello, World!"
}
```

- Is it async — Yes.
- 0 arguments — Yes.
- Returns future — Yes.
- Output implements IntoResponse — Yes.

Check if given function is valid axum::Handler?

```rust
async fn create_user(
    Query(params): Query<HashMap<String, String>>,
    Json(payload): Json<CreateUser>,
) -> (StatusCode, Json<User>) {
    // some code
}
```

- Is it async — Yes.
- Query — check (nb: does not impl FromRequest), Json — check.
- Returns future — Yes.
- StatusCode — check, Json — check.

Handler validation

Sometimes it can be hard to debug handler signatures because of bad explanation:

```
async fn create_user(
    Json(payload): Json<CreateUser>, // no marks from lsp about error
    Query(params): Query<HashMap<String, String>>,
) -> (StatusCode, Json<User>) {
    // some code
}
```

Handler validation

You can fix it using debug_handler attribute macro:

```rust
#[debug_handler]

async fn create_user(
    Json(payload): Json<CreateUser>, `Json<_>` consumes the request body and thus must be the last
argument to the handler function
    Query(params): Query<HashMap<String, String>>,
) -> (StatusCode, Json<User>) {
    // some code
}
```

Handler return type. Return type divergence.

Almost always it’s not a bad idea to let the compiler to infer return type for you:

```
async fn create_user(
    Json(payload): Json<CreateUser>,
) -> impl IntoResponse {
    // some code
}
```

Handler return type. Return type divergence.

But what will happen in this case?

```
let rnd = 4;
if rnd == 4 {
    return StatusCode::NOT_FOUND;
}
let user = User {
    id: 1337,
    username: payload.username,
};
(StatusCode::CREATED, Json(user))
```

Handler return type. Return type divergence.

Mismatched types error..

```
    let rnd = 4;
    if rnd == 4 {
        return StatusCode::NOT_FOUND;
    }
    let user = User {
        id: 1337,
        username: payload.username,
    };
    (StatusCode::CREATED, Json(user)) mismatched types expected struct `StatusCode` found tuple
`(StatusCode, Json<User>)`
```

Handler return type. Return type divergence.

Now it’s single return type, but we still diverge internally.

```
async fn create_user(
    Json(payload): Json<CreateUser>,
) -> Response {
    let rnd = 4;
    if rnd == 4 {
        return StatusCode::NOT_FOUND.into_response();
    }
    let user = User {
        id: 1337,
        username: payload.username,
    };
    (StatusCode::CREATED, Json(user)).into_response()
}
```

Extractors

Extractors — handler arguments. See more examples here.

```rust
async fn create_user(
    Query(params): Query<HashMap<String, String>>, // gives the deserialized query parameters
    Json(payload): Json<CreateUser>, // parsing the request body as json
) -> (StatusCode, Json<User>) {
    // some code
}
```

Extractors order

Remember handler’s arguments validation checklist? Now we can answer to the question: why order matters? Since the request body is an asynchronous stream it can only be consumed once, there can be only one once consuming extractor (i.e. the one that implements FromRequest) that should be passed in the end of argument list.

State extractor. Shared mutable state.

Imagine you want to share state between handlers (ex. pool of db connections, mutexed ds). State extractor can help to hold the application state:

```rust
#[derive(Clone)]
pub struct AppState {
    users: Arc<Mutex<Vec<User>>>,
}

impl AppState {
    pub fn add(&self, u: User) {
        let mut data = self.users.lock().expect("poisoned");
        data.push(u)
    }
}
```

State extractor

```rust
// main
let app = Router::new()
        .route("/", get(root))
        .route("/users", post(create_user))
        .layer(LogLayer::with_target("logger"))
        .with_state(state);

async fn create_user(
    Query(params): Query<HashMap<String, String>>,
    State(state): State<AppState>, //  State: FromRequestParts
    Json(payload): Json<CreateUser>,
) -> Response {
    let user = User::new(1337, payload.username);
    state.add(user.clone());
    (StatusCode::CREATED, Json(user)).into_response()
}
```

Additional resources

- source code
- examples: here and here