さうすの Rust 勉強会

lesson02

内容

- `axum` でサーバーを作る
- テストを書く

HTTP の基本

- GET
  - Safe (安全) + Idempotent (冪等)
    - Safe：0 回のとき n 回と同じ
    - Idempotent；1 回とき n 回と同じ
    - Safe であれば Idempotent
  - Body がない (* と思ったほうがいい)
- POST
  - Idempotent でない
  - Body がある

- DELETE
  - Idempotent
  - Body がない (* と思ったほうがいい)
- PUT
  - Idempotent
  - Body がある
- OPTIONS
  - Safe + Idempotent
  - 普通は HTTP サーバーが処理してくれる
- 他のメソッドもあるが、主に使われるのはこのへん

axum とは？

- Rust の HTTP サーバー用のライブラリ
- tokio（async runtime）が開発したライブラリ
- とりあえず使いやすい
  - けど最近の actix-web もほとんど同じ書き方で行けるらしい
- ドキュメント：https://docs.rs/axum/latest/axum/index.html
  - 全部読んだら今日のスライド見なくていい

Hello World

```rust
use axum::{response::IntoResponse, routing::get, Router};

async fn get_hello_world() -> impl IntoResponse {
    "Hello, World!"
}

#[tokio::main]
async fn main() {
    let app = Router::new().route("/", get(get_hello_world));

    axum::Server::bind(&"0.0.0.0:3000".parse().unwrap())
        .serve(app.into_make_service())
        .await
        .unwrap();
}
```

trait（Rust の一般的な概念）

他の言語の interface に近い概念。

```
trait Edible {
  fn eat(&self);
}

struct Apple;

impl Edible for Apple {
  fn eat(&self) {
    println!("Apple is eaten!");
  }
}

fn main() {
  let apple = Apple;
  apple.eat();
}
```

trait extension（よくあるパターン）

他のライブラリで定義された trait を他のライブラリで定義された struct の上に impl することはできないので、他のライブラリで定義された struct を拡張したければ、自前の trait を定義する必要がある。

```
trait StringExt { fn double(&self) -> String; }

// String は std で定義されているので、普通に impl String できない
impl StringExt for String {
  fn double(&self) -> String {
    format!("{self}{self}")
  }
}

fn main() {
  println!("{}", "test".to_string().double());
}
```

handler

HTTP リクエストを処理するもの。ただの関数である。

https://docs.rs/axum/latest/axum/handler/index.html

- Handler では extractor を使ってリクエストから情報を抽出することができる（後述）
- 普通の関数だが、引数は FromResponsePart や FromResponse を impl している必要があり、戻り値は IntoResponse を実装している必要がある

```
// Router::new().route("/:id", get_id) だと仮定する
async fn get_id(Path(id): Path<String>) -> impl IntoResponse {
  return format!("HTTP request tried to get {id}");
}
```

extractor

https://docs.rs/axum/latest/axum/extract/index.html

HTTP リクエストのヘッダー / ボディから何かを「抽出」するもの。

二種類あって、

- `FromRequestParts` はヘッダーしか読めないので、一つの handler で何個も使える
- `FromRequest` はボディも読めるので、一つの handler で一個しか使えない

注：Executor を自作するときは `#[async_trait]` を使うこと！

先と同じ例で説明する

```
// Router::new().route("/:id", get_id) だと仮定する
async fn get_id(
  // ここでは Path という extractor が使われている
  Path(id): Path<String>
) -> impl IntoResponse {
  return format!("HTTP request tried to get {id}");
}
```

state

https://docs.rs/axum/latest/axum/#using-the-state-extractor
https://docs.rs/axum/latest/axum/extract/struct.State.html

extension layer

https://docs.rs/axum/latest/axum/#using-request-extensions
https://docs.rs/axum/latest/axum/struct.Extension.html

State よりは自由に使えるもの。

debug_handler

https://docs.rs/axum-macros/latest/axum_macros/attr.debug_handler.html

もし axum がよくわからないエラーが出たとき handler のほうに付けるとエラーメッセージが読みやすくなるらしい。

```
#[debug_handler]
fn route() -> String {
  "Hello, world".to_string()
}
```

```
error: handlers must be async functions
  --> main.rs:xx:1
   |
xx | fn route() -> String {
   | ^^
```

middleware

https://docs.rs/axum/latest/axum/middleware/index.html

HTTP リクエスト・HTTP レスポンスを編集しうるもの。

実装すると自分で Future を実装する必要があるので、ほとんどの場合は `axum::middleware::from_fn` を使って実装する。

一般的なテストの書き方

少なくとも二つの書き方：

- 普通のファイル (の最後) に

```
#[cfg(test)]
mod tests {
  #[test]
  fn test_1_plus_1_is_2() {
    assert_eq!(1 + 1, 2);
  }
}
```

  と書く

- crate の `tests` ディレクトリに `.rs` ファイルを作る

axum のテストの書き方（公式 example）

`project01` から取ってきた例である。

```
#[tokio::test]
async fn test_nonexistent_key() -> anyhow::Result<()> {
    let router = init_test_router();
    let response = router
        .oneshot(Request::builder().uri("/key/1").body(Body::empty())?)
        .await?;
    assert_eq!(response.status(), StatusCode::NOT_FOUND);
    let body: Bytes = hyper::body::to_bytes(response.into_body()).await.unwrap();
    assert_eq!(&body[..], b"");
    Ok(())
}
```

axum のテストの書き方（axum-test-helper）

`axum-test-helper` というライブラリを使うとより簡潔にテストを書けるはず。しかし正直ある程度複雑なプロジェクトなら自分でヘルパー関数を用意してもいい気がする...

```
#[tokio::test]
async fn test_nonexistent_key() -> anyhow::Result<()> {
    let router = init_test_router();
    let client = axum_test_helper::TestClient::new(router);
    let response = client.get("/uri/1").send().await;
    assert_eq!(response.status(), StatusCode::NOT_FOUND);
    assert_eq!(response.text().await, "");
    Ok(())
}
```

課題

- `project01` を埋めて、テストが通ることを確認しよう
- `project02` を埋めて、テストが通ることを確認しよう
- `project01` と `project02` のテストのコードを理解して、なぜ仕様を確認できていることを理解してみよう
- `project03` を完成する。テストも書こう。