

Rust et Axum

Il était une fois...

Introduction à la construction de services web avec `rustaxum-starter`.

Préface

Francis Lavoie

Développeur web depuis longtemps (~7335 jours)

Pratique le Typescript au quotidien. Mais passionné de tout le reste



La genèse

Papa!! Fais-moi une histoire inventée!

Finis les classiques.

- il voulait ses amis comme personnages
- il voulait ses propres lieux
- il voulait SA PROPRE aventure différente

Mon plan infaillible

Je n'avais pas besoin d'une architecture "parfaite".

Je voulais:

- livrer la meilleure architecture possible
- Être le plus performant possible
- Un langage pour tout
- Faire quelque chose de robuste et rapide
- Surtout, avoir du fun à le faire

Pourquoi Rust et Axum

- Haute performance par défaut
- Sécurité sans sacrifier la vitesse Empêche les crashes, les bugs mémoire et les conditions de concurrence dès la compilation.
- Conçu pour passer à l'échelle
- Maintenabilité à long terme
- Investissement en compétences fondamentales

Pourquoi s'abstenir

- Développement plus lent au départ
- Bassin de talents plus restreint
- Nécessite davantage de rigueur en ingénierie

La quête du Graal

Langage	Rust stable
Framework HTTP	Axum 0.8
ORM	SeaORM 2.0.x
Templates	Askama 0.15
Front	htmx 2.0 + TailwindCSS 4 + daisyUI 5

L'Objectif

Découvrir Axum, avoir le désir d'en apprendre plus et posséder suffisamment de connaissance pour débuter votre propre aventure sur votre propre projet.

rustling: <https://rustlings.rust-lang.org/>



Chapitre 1

Poser la scène du projet

Structure

```
rustaxum-starter/
├── Cargo.toml
├── src/main.rs
├── src/routes.rs
├── src/handlers/story.rs
├── src/entities/story.rs
├── src/extractors.rs
├── templates/*.html
├── static/themes.css
└── migration/
```

Un projet commence ici

Cargo.toml

```
[package]
name = "rustaxum-starter"
version = "0.1.0"
edition = "2021"

[workspace]
members = [ ".", "migration" ]

[dependencies]
axum = "0.8"
tokio = { version = "1", features = ["full"] }
...
```



Chapitre 2

Demarrer le service

L'entrée

main.rs

```
#[derive(Clone)]
pub struct AppState { pub db: sea_orm::DatabaseConnection, }

#[tokio::main]
async fn main() {
    let db = Database::connect(&database_url).await?;
    let state = AppState { db };

    let mut app = routes::build()
        .layer(TraceLayer::new_for_http())
        .with_state(state);

    ...
}
```

Créer la connexion

main.rs

```
async fn main() {
    ...
    let listener =
        tokio::net::TcpListener::bind("0.0.0.0:3000")
            .await
            .expect("Failed to bind to 0.0.0.0:3000");

    axum::serve(listener, app)
        .await
        .expect("Server error");
}
```

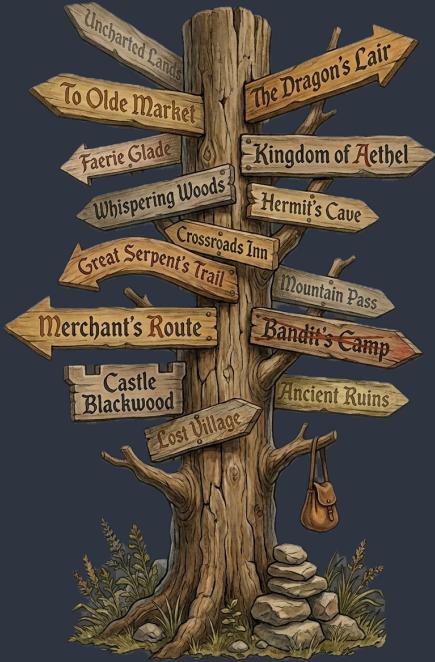
Tips

```
#cfg(debug_assertions)
{
    app = app.layer(
        tower_livereload::LiveReloadLayer::new()
    );
}
```

En debug, le navigateur se rafraîchit automatiquement après recompilation.

Chapitre 3

Faire circuler les requêtes



La route vers le graal

router.rs

```
Router::new()
    .route("/stories",
        get(handlers::story::list_stories)
            .post(handlers::story::create_story))
    .route("/stories/{id}/edit",
        get(handlers::story::edit_story_form))
    .nest_service("/static", ServeDir::new("static"))
```



Chapitre 4

Remplir son baluchon (Entities)

Ce qu'il nous faut pour une histoire

entities/story.rs

```
#[derive(DeriveEntityModel)]
#[sea_orm(table_name = "story")]
pub struct Model {
    #[sea_orm(primary_key)] pub id: i32,
    pub name: String,
    #[sea_orm(column_type = "Text")] pub description: String,
    #[sea_orm(column_type = "Text")] pub content: String,
}
```

Comment chercher nos informations

entities/story.rs

```
pub use Entity as Story;

impl Story {
    pub async fn find_all_latest(db: &DatabaseConnection) → Result<
        Self::find()
            .order_by_desc(Column::CreatedAt)
            .all(db)
            .await
    }
}
```

Mettre à jour les données

entities/story.rs

```
impl Story {
    pub async fn update(
        db: &DatabaseConnection, existing: Model,
        name: String, description: String, content: String,
    ) -> Result<Model, DbErr> {
        let mut active: ActiveModel = existing.into();
        active.name = Set(name);
        active.description = Set(description);
        active.content = Set(content);
        active.update(db).await
    }
}
```

Chapitre 5

L'aventure (handlers)



La découverte

handlers/story.rs

```
pub async fn story_detail(  
    HxRequest(is_htmx): HxRequest,  
    State(state): State<AppState>,  
    Path(id): Path<i32>,  
) → Result<Response, StatusCode> {  
    let story = Story::find_one(&state.db, id)  
        .await.map_err(db_err)?;  
    Ok(StoryDetail { is_htmx,  
        story: story.ok_or(StatusCode::NOT_FOUND)?  
    }.into_response())  
}
```

Affronter les données

```
#[derive(Deserialize)]
pub struct StoryForm {
    pub name: String,
    pub description: String,
    pub content: String,
}
```

Le `Form<StoryForm>` devient directement une structure Rust.

La création de notre histoire

```
pub async fn create_story(  
    State(state): State<AppState>,  
    Form(form): Form<StoryForm>,  
) → Result<Response, StatusCode> {  
    Story::create(&state.db, form.name, form.description, form.cor  
        .await  
        .map_err(db_err)?;  
}
```



Chapitre 6

L'extraction

Extracteur personnalisé pour htmx

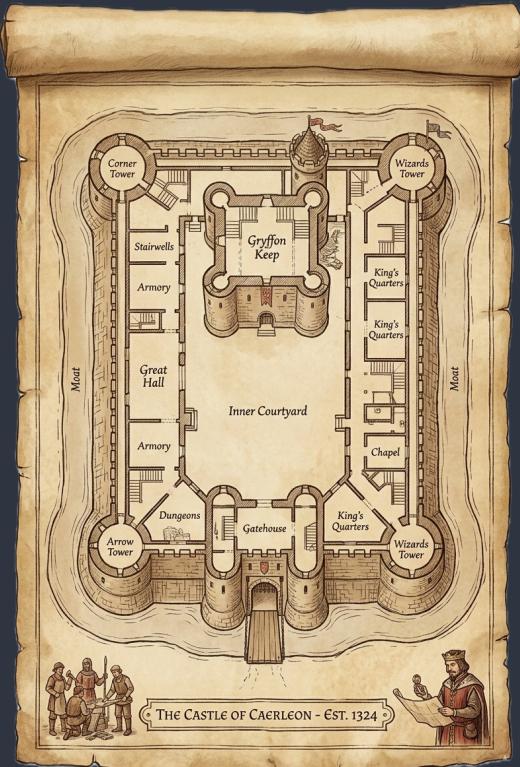
```
pub struct HxRequest(pub bool);

impl<S: Send + Sync> FromRequestParts<S> for HxRequest {
    type Rejection = std::convert::Infallible;
    async fn from_request_parts(parts: &mut Parts, _: &S) -> Result<Ok(HxRequest{parts.headers.contains_key("hx-request")})>
    }
}
```

Un extracteur permet d'extraire d'une requête des informations pertinentes sans dupliquer la logique

Chapitre 7

L'aboutissement (Templates)



La définition

handlers/story.rs

```
#[derive(Template, WebTemplate)]
#[template(path = "stories/list.html")]
pub struct StoryList {
    pub is_htmx: bool,
    pub stories: Vec<story::Model>,
}
```

Le template est vérifié au compile-time.

Les données (render)

handlers/story.rs

```
pub async fn list_stories(  
    HxRequest(is_htmx): HxRequest,  
    State(state): State<AppState>,  
) → Result<Response, StatusCode> {  
    let stories = Story::find_all_latest(&state.db).await.map_err(  
        Ok(StoryList { is_htmx, stories }.into_response())  
}
```

Askama

templates/stories/list.html

```
{% extends "htmx fragment.html" %}  
{% block content %}  
<div>  
    {% if stories.is_empty() %}  
        <div>....</div>  
    {% else %}  
        {% for story in stories %}  
            <h2>{{ story.name }}</h2>  
            <p>{{ story.description }}</p>  
  
            {% endfor %}  
        {% endif %}  
    </div>  
{% endblock %}
```

Htmx à votre service

```
<div class="card card-border shadow-sm"
    hx-get="/stories/{{ story.id }}"
    hx-target="#content"
    hx-push-url="true">
    <h2>{{ story.name }}</h2>
    <p>{{ story.description }}</p>
</div>
```

htmx ici va injecter le html rendu par le serveur dans le bloc au id "content".

base.html: stack front minimale

```
<html data-theme="demotone">
<head>
  <link href="https://cdn.jsdelivr.net/npm/daisyui@5" rel="stylesheet"
  <script src="https://cdn.jsdelivr.net/npm/tailwindcss/browser@4"
  <script src="https://unpkg.com/htm@2.0.4"></script>
</head>
<body class="min-h-screen bg-base-200">
```

Tailwind et htmx restent légères, sans basculer sur un framework SPA.

Chapitre 8

Pièges et réflexes



Piège 1: ordre des extracteurs

```
// ✗ Erreur de compilation: Form consomme le body, doit être en
async fn create(Form(form): Form<StoryForm>, State(state): State<A
// ✓ State (FromRequestParts) d'abord, Form (FromRequest) en dern
async fn create(State(state): State<AppState>, Form(form): Form<St
```

Le body HTTP est un flux asynchrone **consommé** une seule fois. Les extracteurs qui lisent le body (`Form`, `Json`, `String`, `Bytes`) doivent être le **dernier** argument. On ne peut pas en utiliser deux (ex. `Form` et `Json`) dans le même handler.

Piège 2: syntaxe des routes et état partagé

Routes: En Axum 0.8 les captures utilisent `{id}`. L'ancien format `:id` ou `*path` fait paniquer au runtime sauf si on appelle `.without_v07_checks()`.

État: Préférer `State + .with_state(state)` à `Extension . Extension` donne une **500** si le type n'est pas dans les extensions (middleware oublié); `State` est vérifié par le compilateur. Votre type d'état doit être `Clone` (ou enveloppé dans `Arc`).

Piège 3: erreur HTTP propre

```
let story = Story::find_one(&state.db, id)
    .await
    .map_err(db_err)?
    .ok_or(HttpStatusCode::NOT_FOUND)?;
```

Dans un handler, `unwrap()` ou `expect()` font planter tout le thread: l'utilisateur reçoit une déconnection brute. Retourner `Result<T, E>` avec `T` et `E` qui implémentent `IntoResponse` (ex. `StatusCode`) permet d'envoyer une vraie réponse HTTP. Les échecs d'extracteurs (ex. JSON invalide) n'appellent pas votre handler—enveloppez dans `Result<Json<T>, JsonRejection>` si vous voulez gérer l'erreur dans le handler.

Piège 4: limite du body (2 Mo par défaut)

`Json`, `Form` et `String` utilisent `Bytes` en interne. Pour des raisons de sécurité, la taille du body est plafonnée à 2 Mo par défaut. Un body plus gros est rejeté.

Pour des uploads ou gros payloads, ajoutez une limite explicite (ou désactivez la limite par défaut puis utilisez la vôtre):

```
use axum::extract::DefaultBodyLimit;

let app = Router::new()
    .route("/upload", post(upload_handler))
    .layer(DefaultBodyLimit::max(10 * 1024 * 1024)); // 10 Mo
```

Piège 5: middleware et services qui échouent

Axum exige que les services aient le type d'erreur **Infallible**: toute erreur doit être convertie en réponse. Si vous branchez un middleware qui peut échouer (ex. `tower :: timeout`), ou un `route_service` qui retourne une `Error`, il faut envelopper avec `HandleErrorLayer` / `HandleError` pour transformer l'erreur en réponse (ex. 504 Request Timeout).

Piège 6: ordre des layers et des routes

Le `.layer()` s'applique **uniquement aux routes déjà déclarées** avant l'appel. Les routes ajoutées après n'ont pas ce middleware. Pour n'avoir un layer que sur certaines routes, construisez des sous-routers avec leur layer puis `.merge()`.

State: Appelez `.with_state(state)` après avoir défini toutes les routes qui en ont besoin. `Router<S>` signifie « il manque un state de type S »; seul `Router<()>` peut être passé à `axum::serve()`.

Conclusion



Les hauts et les bas

Rust

- **Pour** : Robustesse du typage, performances, utilisation mémoire minimale, écosystème (Cargo, crates)
- **Contre** : courbe d'apprentissage, moins de librairies que JS/Python

Axum

- **Pour** : léger, extracteurs typés, évolutif (Router, layers), template typés et compilés avec Askama
- **Contre** : écosystème plus jeune, doc et exemples moins nombreux qu'Express/Actix, erreurs parfois cryptiques

Ce que vous savez maintenant

```
Requete → Router → Handler  
Handler → SeaORM → PostgreSQL  
Handler → Askama → HTML  
HTML → htmx → Magie
```

Vous pouvez lire [rustaxum-starter](#), comprendre le flux, puis l'étendre en confiance.

Prochaine etape du projet

Le starter peut évoluer vers:

- génération d'histoires par IA
- authentification utilisateur
- recherche et filtres
- tests d'intégration Axum
- observabilité (tracing, metrics)

Merci

Questions

Projet de démarrage



https://gitlab.com/francis_l_projects/rustaxum-starter

Feedback

