🦀 Complete SeaORM + Axum REST API Guide

A step-by-step guide for building REST APIs with **Axum** and **SeaORM** (PostgreSQL), covering project setup, migrations, entity generation, and full CRUD operations.

> *[!IMPORTANT] This guide uses **SeaORM 1.x** (latest stable). Your current project mixes `sea-orm = "0.12"` with `sea-orm-migration = "1.1"` — you should align them both to `1.1`.*

Table of Contents

---

1. Project Setup & Workspace Structure

The recommended structure is a **Cargo workspace** with 3 crates:

```
rest_api/                    ← workspace root
├── Cargo.toml               ← workspace Cargo.toml
├── .env                     ← DATABASE_URL goes here
├── src/
│   └── main.rs              ← your Axum app
├── entity/                  ← generated entity crate
│   ├── Cargo.toml
│   └── src/
│       ├── lib.rs
│       ├── user.rs
│       └── prelude.rs
└── migration/               ← migration crate
    ├── Cargo.toml
    └── src/
        ├── lib.rs
        ├── main.rs
        └── m20220101_000001_create_table.rs
```

Root `Cargo.toml`

```toml
[package]
name = "rest_api"
version = "0.1.0"
edition = "2021"
```

```toml
[workspace]
members = [".", "entity", "migration"]

[workspace.dependencies]
sea-orm = { version = "1.1", features = ["sqlx-postgres", "runtime-tokio-rustls", "macros"]
}
serde = { version = "1.0", features = ["derive"] }
tokio = { version = "1", features = ["full"] }

[dependencies]
entity = { path = "entity" }
migration = { path = "migration" }
axum = { version = "0.7", features = ["macros"] }
uuid = { version = "1.10", features = ["v4"] }
chrono = "0.4"
tower-http = { version = "0.5", features = ["trace"] }
sea-orm.workspace = true
serde.workspace = true
tokio.workspace = true
serde_json = "1.0"
```

entity/Cargo.toml

```toml
[package]
name = "entity"
version = "0.1.0"
edition = "2021"
publish = false

[lib]
name = "entity"
path = "src/lib.rs"

[dependencies]
serde = { version = "1.0", features = ["derive"] }
sea-orm = { version = "1.1", features = ["sqlx-postgres", "runtime-tokio-rustls"] }
```

migration/Cargo.toml

```toml
[package]
name = "migration"
version = "0.1.0"
edition = "2021"
publish = false

[lib]
name = "migration"
path = "src/lib.rs"
```

```toml
[dependencies]
tokio = { version = "1", features = ["macros", "rt-multi-thread"] }

[dependencies.sea-orm-migration]
version = "1.1"
features = ["runtime-tokio-rustls", "sqlx-postgres"]
```

`.env` file

```
DATABASE_URL=postgres://postgres:YOUR_PASSWORD@localhost:5432/your_database
```

---

2. Installing sea-orm-cli

```
# Install the CLI tool (one-time)
cargo install sea-orm-cli

# Verify installation
sea-orm-cli --version
```

---

3. Creating Migrations

Step 1: Initialize Migration Directory (first time only)

```
# Run from your project root (rest_api/)
sea-orm-cli migrate init
```

This creates the `migration/` folder with a sample migration file.

Step 2: Generate a New Migration File

```
# Create a new migration (auto-timestamped)
sea-orm-cli migrate generate create_user_table
```

This creates a file like `migration/src/m20260216_000001_create_user_table.rs`.

Step 3: Define the Table Schema

Edit the generated migration file:

```rust
// migration/src/m20260216_000001_create_user_table.rs
use sea_orm_migration::{prelude::*, schema::*};

#[derive(DeriveMigrationName)]
pub struct Migration;
```

```rust
#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .create_table(
                Table::create()
                    .table(User::Table)
                    .if_not_exists()
                    .col(pk_auto(User::Id))                         // auto-increment
primary key
                    .col(string(User::Name))                        // VARCHAR NOT NULL
                    .col(string_uniq(User::Email))                  // VARCHAR UNIQUE NOT
NULL
                    .col(string_uniq(User::Username))               // VARCHAR UNIQUE NOT
NULL
                    .col(string(User::Password))                    // VARCHAR NOT NULL
                    .col(string(User::Contact))                     // VARCHAR NOT NULL
                    .col(string_uniq(User::Uuid))                   // VARCHAR UNIQUE NOT
NULL
                    .col(timestamp(User::CreatedAt))                // TIMESTAMP NOT NULL
                    .to_owned(),
            )
            .await
    }

    async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .drop_table(Table::drop().table(User::Table).to_owned())
            .await
    }
}

// Define the column identifiers
#[derive(DeriveIden)]
enum User {
    Table,
    Id,
    Name,
    Email,
    Username,
    Password,
    Contact,
    Uuid,
    CreatedAt,
}
```

Step 4: Register the Migration in `lib.rs`

```rust
// migration/src/lib.rs
pub use sea_orm_migration::prelude::*;
```

```rust
mod m20260216_000001_create_user_table;
// Add more migration modules here as you create them

pub struct Migrator;

#[async_trait::async_trait]
impl MigratorTrait for Migrator {
    fn migrations() -> Vec<Box<dyn MigrationTrait>> {
        vec![
            Box::new(m20260216_000001_create_user_table::Migration),
            // Add more migrations here in chronological order
        ]
    }
}
```

Common Column Helper Functions

Helper Function	SQL Result
`pk_auto(col)`	`INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL`
`string(col)`	`VARCHAR NOT NULL`
`string_null(col)`	`VARCHAR` (nullable)
`string_uniq(col)`	`VARCHAR UNIQUE NOT NULL`
`integer(col)`	`INTEGER NOT NULL`
`big_integer(col)`	`BIGINT NOT NULL`
`boolean(col)`	`BOOLEAN NOT NULL`
`timestamp(col)`	`TIMESTAMP NOT NULL`
`timestamp_null(col)`	`TIMESTAMP` (nullable)
`text(col)`	`TEXT NOT NULL`
`text_null(col)`	`TEXT` (nullable)
`float(col)`	`FLOAT NOT NULL`
`double(col)`	`DOUBLE NOT NULL`
`uuid(col)`	`UUID NOT NULL`

---

4. Running Migrations

```bash
# Run all pending migrations (creates tables in database)
sea-orm-cli migrate up

# Or with explicit database URL
```

```
sea-orm-cli migrate up -u postgres://postgres:YOUR_PASSWORD@localhost:5432/your_database

# Rollback last migration
sea-orm-cli migrate down

# Rollback all migrations
sea-orm-cli migrate fresh   # drops all tables and re-runs all migrations

# Check migration status
sea-orm-cli migrate status
```

> *[!TIP] If you have `DATABASE_URL` set in your `.env` file, you don't need to pass `-u` every time.*

---

5. Generating Entities

After running migrations (tables exist in the database), generate entity files:

```
# Generate entities as a separate crate (RECOMMENDED)
sea-orm-cli generate entity \
    -u postgres://postgres:YOUR_PASSWORD@localhost:5432/your_database \
    -o entity/src \
    -l \
    --with-serde both \
    --expanded-format
```

What Each Flag Does

Flag	Purpose
`-u` or `--database-url`	Your database connection string
`-o` or `--output-dir`	Where to put generated files (use `entity/src`)
`-l` or `--lib`	Generate `lib.rs` instead of `mod.rs` (needed for crate)
`--with-serde both`	Add `Serialize`/`Deserialize` derives to models
`--expanded-format`	Verbose entity format (easier to read and customize)
`--compact-format`	Shorter format (default, uses derive macros)

For Your Project (copy-paste ready)

```
sea-orm-cli generate entity -u postgres://postgres:9819375722Aditya@localhost:5432/test -o entity/src -l --with-serde both --expanded-format
```

> *[!CAUTION] **Output directory matters!** Use `-o entity/src` (NOT `-o ./src/entity`). The entity files must go into the `entity` crate's `src/` directory, not into your app's `src/` directory.*

Generated Files

After running the command, you'll get:

```
entity/src/
├── lib.rs       ← re-exports all entity modules
├── prelude.rs   ← convenient re-exports (Entity as TableName)
├── user.rs      ← one file per database table
└── post.rs      ← (if you have a posts table)
```

---

6. Updating Entities (Adding/Modifying Tables)

Adding a New Column to Existing Table

Step 1: Create a new migration:

```
sea-orm-cli migrate generate add_bio_to_user
```

Step 2: Write the migration:

```rust
// migration/src/m20260217_000001_add_bio_to_user.rs
use sea_orm_migration::{prelude::*, schema::*};

#[derive(DeriveMigrationName)]
pub struct Migration;

#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .alter_table(
                Table::alter()
                    .table(User::Table)
                    .add_column(text_null(User::Bio))   // nullable TEXT column
                    .to_owned(),
            )
            .await
    }

    async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .alter_table(
                Table::alter()
                    .table(User::Table)
                    .drop_column(User::Bio)
                    .to_owned(),
            )
            .await
    }
}

#[derive(DeriveIden)]
```

```
enum User {
    Table,
    Bio,
}
```

Step 3: Register in `lib.rs` :

```
mod m20260217_000001_add_bio_to_user;

// Add to migrations() vec:
Box::new(m20260217_000001_add_bio_to_user::Migration),
```

Step 4: Run migration & regenerate entities:

```
sea-orm-cli migrate up
sea-orm-cli generate entity -u YOUR_DB_URL -o entity/src -l --with-serde both --expanded-format
```

> *[!IMPORTANT]* ***Always regenerate entities after changing the schema.*** *The entity files are auto-generated from your actual database — running the generate command will update them to match.*

---

7. Multiple Entities with Relations

Example: Adding a `Post` Table (User has many Posts)

Step 1: Create migration:

```
sea-orm-cli migrate generate create_post_table
```

Step 2: Define the migration with a foreign key:

```
use sea_orm_migration::{prelude::*, schema::*};

#[derive(DeriveMigrationName)]
pub struct Migration;

#[async_trait::async_trait]
impl MigrationTrait for Migration {
    async fn up(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .create_table(
                Table::create()
                    .table(Post::Table)
                    .if_not_exists()
                    .col(pk_auto(Post::Id))
                    .col(string(Post::Title))
                    .col(text(Post::Content))
                    .col(big_integer(Post::UserId))              // foreign key column
                    .col(timestamp(Post::CreatedAt))
```

```
                    .foreign_key(
                        ForeignKey::create()
                            .name("fk_post_user_id")
                            .from(Post::Table, Post::UserId)
                            .to(User::Table, User::Id)
                            .on_delete(ForeignKeyAction::Cascade)
                            .on_update(ForeignKeyAction::Cascade),
                    )
                    .to_owned(),
            )
            .await
    }

    async fn down(&self, manager: &SchemaManager) -> Result<(), DbErr> {
        manager
            .drop_table(Table::drop().table(Post::Table).to_owned())
            .await
    }
}

#[derive(DeriveIden)]
enum Post {
    Table,
    Id,
    Title,
    Content,
    UserId,
    CreatedAt,
}

#[derive(DeriveIden)]
enum User {
    Table,
    Id,
}
```

Step 3: Run migration & regenerate entities:

```
sea-orm-cli migrate up
sea-orm-cli generate entity -u YOUR_DB_URL -o entity/src -l --with-serde both --expanded-format
```

SeaORM will **automatically detect foreign keys** and generate the `Relation` enum and `Related` impl in the entity files.

---

8. CRUD Operations

Below are all the CRUD operations. These assume you have `db: &DatabaseConnection` available.

🟢 CREATE (Insert)

Insert One

```
use entity::user;
use sea_orm::{ActiveModelTrait, Set};

let new_user = user::ActiveModel {
    name: Set("Aditya".to_owned()),
    email: Set("aditya@example.com".to_owned()),
    username: Set("aditya1".to_owned()),
    password: Set("hashed_password".to_owned()),
    contact: Set("8591059220".to_owned()),
    uuid: Set(uuid::Uuid::new_v4().to_string()),
    created_at: Set(chrono::Utc::now().naive_utc()),
    ..Default::default()  // id will be auto-generated
};

// Insert and get back the full Model (with auto-generated id)
let created_user: user::Model = new_user.insert(db).await?;
println!("Created user with id: {}", created_user.id);
```

Insert Many

```
use entity::user;
use sea_orm::{Set, EntityTrait};
use entity::prelude::User;

let user1 = user::ActiveModel {
    name: Set("Alice".to_owned()),
    email: Set("alice@example.com".to_owned()),
    // ... other fields
    ..Default::default()
};

let user2 = user::ActiveModel {
    name: Set("Bob".to_owned()),
    email: Set("bob@example.com".to_owned()),
    // ... other fields
    ..Default::default()
};

let result = User::insert_many([user1, user2])
    .exec(db)
    .await?;
```

🔵 READ (Select)

Find by Primary Key

```rust
use entity::prelude::User;
use entity::user;
use sea_orm::EntityTrait;

// Find one by ID
let user: Option<user::Model> = User::find_by_id(1).one(db).await?;

if let Some(user) = user {
    println!("Found: {}", user.name);
}
```

Find All

```rust
let all_users: Vec<user::Model> = User::find().all(db).await?;
```

Find with Conditions (WHERE clause)

```rust
use sea_orm::{EntityTrait, QueryFilter, ColumnTrait};
use entity::user;
use entity::prelude::User;

// Find by email
let user = User::find()
    .filter(user::Column::Email.eq("aditya@example.com"))
    .one(db)
    .await?;

// Find multiple with LIKE
let users = User::find()
    .filter(user::Column::Name.contains("Aditya"))
    .all(db)
    .await?;

// Find with multiple conditions (AND)
let users = User::find()
    .filter(user::Column::Name.contains("Aditya"))
    .filter(user::Column::Email.ends_with("@gmail.com"))
    .all(db)
    .await?;
```

Find with Ordering

```rust
use sea_orm::{EntityTrait, QueryOrder};

let users = User::find()
    .order_by_asc(user::Column::Name)
    .all(db)
    .await?;
```

```rust
let users = User::find()
    .order_by_desc(user::Column::CreatedAt)
    .all(db)
    .await?;
```

Pagination

```rust
use sea_orm::{EntityTrait, PaginatorTrait};

// Get page 1, 10 items per page
let paginator = User::find()
    .paginate(db, 10);   // 10 items per page

let total_pages = paginator.num_pages().await?;
let page_1: Vec<user::Model> = paginator.fetch_page(0).await?;  // 0-indexed
```

🟡 UPDATE

Update One (find then update)

```rust
use entity::prelude::User;
use entity::user;
use sea_orm::{EntityTrait, ActiveModelTrait, Set};

// Step 1: Find the user
let user: Option<user::Model> = User::find_by_id(1).one(db).await?;

// Step 2: Convert to ActiveModel and modify
let mut user: user::ActiveModel = user.unwrap().into();
user.name = Set("New Name".to_owned());
user.email = Set("newemail@example.com".to_owned());

// Step 3: Save — only changed (Set) fields are updated
let updated_user: user::Model = user.update(db).await?;
```

Update Many (bulk update)

```rust
use entity::prelude::User;
use entity::user;
use sea_orm::{EntityTrait, QueryFilter, ColumnTrait, Set};
use sea_orm::sea_query::Expr;

// Update all users matching a condition
User::update_many()
    .col_expr(user::Column::Name, Expr::value("Updated Name"))
    .filter(user::Column::Email.contains("@old-domain.com"))
```

```
    .exec(db)
    .await?;
```

🔴 DELETE

Delete One (find then delete)

```
use entity::prelude::User;
use entity::user;
use sea_orm::{EntityTrait, ModelTrait};

let user: Option<user::Model> = User::find_by_id(1).one(db).await?;
let user: user::Model = user.unwrap();

let result = user.delete(db).await?;
assert_eq!(result.rows_affected, 1);
```

Delete by Primary Key (without fetching)

```
use entity::prelude::User;
use sea_orm::EntityTrait;

let result = User::delete_by_id(1).exec(db).await?;
```

Delete Many

```
use entity::user;
use sea_orm::{EntityTrait, QueryFilter, ColumnTrait};

let result = user::Entity::delete_many()
    .filter(user::Column::Name.contains("test"))
    .exec(db)
    .await?;
println!("Deleted {} rows", result.rows_affected);
```

---

9. Sharing Database Connection with Axum State

Instead of creating a new DB connection per request, share it via Axum's `State`:

```
// src/main.rs
use axum::{Router, routing::{get, post, put, delete}, extract::State};
use sea_orm::{Database, DatabaseConnection};
use std::sync::Arc;

mod users;

#[derive(Clone)]
```

```rust
pub struct AppState {
    pub db: DatabaseConnection,
}

#[tokio::main]
async fn main() {
    let database_url = "postgres://postgres:YOUR_PASSWORD@localhost:5432/test";
    let db = Database::connect(database_url).await.unwrap();

    let state = AppState { db };

    let router = Router::new()
        .route("/users",       get(users::get_all_users))
        .route("/users/:id",   get(users::get_user_by_id))
        .route("/users",       post(users::create_user))
        .route("/users/:id",   put(users::update_user))
        .route("/users/:id",   delete(users::delete_user))
        .with_state(state);

    let listener = tokio::net::TcpListener::bind("0.0.0.0:8000").await.unwrap();
    println!("🚀 Server running on http://localhost:8000");
    axum::serve(listener, router).await.unwrap();
}
```

```rust
// src/users.rs
use axum::{extract::{State, Path}, Json};
use sea_orm::*;
use entity::{user, prelude::User};
use serde::{Deserialize, Serialize};
use crate::AppState;

#[derive(Serialize, Deserialize)]
pub struct CreateUserRequest {
    pub name: String,
    pub email: String,
    pub username: String,
    pub password: String,
    pub contact: String,
}

// GET /users
pub async fn get_all_users(
    State(state): State<AppState>,
) -> Json<Vec<user::Model>> {
    let users = User::find().all(&state.db).await.unwrap();
    Json(users)
}

// GET /users/:id
pub async fn get_user_by_id(
    State(state): State<AppState>,
```

```rust
    Path(id): Path<i64>,
) -> Json<Option<user::Model>> {
    let user = User::find_by_id(id).one(&state.db).await.unwrap();
    Json(user)
}

// POST /users
pub async fn create_user(
    State(state): State<AppState>,
    Json(req): Json<CreateUserRequest>,
) -> Json<user::Model> {
    let new_user = user::ActiveModel {
        name: Set(req.name),
        email: Set(req.email),
        username: Set(req.username),
        password: Set(req.password),
        contact: Set(req.contact),
        uuid: Set(uuid::Uuid::new_v4().to_string()),
        created_at: Set(chrono::Utc::now().naive_utc()),
        ..Default::default()
    };
    let created = new_user.insert(&state.db).await.unwrap();
    Json(created)
}

// PUT /users/:id
pub async fn update_user(
    State(state): State<AppState>,
    Path(id): Path<i64>,
    Json(req): Json<CreateUserRequest>,
) -> Json<user::Model> {
    let user = User::find_by_id(id).one(&state.db).await.unwrap().unwrap();
    let mut user: user::ActiveModel = user.into();
    user.name = Set(req.name);
    user.email = Set(req.email);
    user.username = Set(req.username);
    user.password = Set(req.password);
    user.contact = Set(req.contact);
    let updated = user.update(&state.db).await.unwrap();
    Json(updated)
}

// DELETE /users/:id
pub async fn delete_user(
    State(state): State<AppState>,
    Path(id): Path<i64>,
) -> Json<String> {
    let result = User::delete_by_id(id).exec(&state.db).await.unwrap();
    Json(format!("Deleted {} row(s)", result.rows_affected))
}
```

10. Quick Reference Cheat Sheet

CLI Commands Summary

Task	Command
Install CLI	`cargo install sea-orm-cli`
Init migrations	`sea-orm-cli migrate init`
Create new migration	`sea-orm-cli migrate generate NAME`
Run migrations	`sea-orm-cli migrate up`
Rollback last migration	`sea-orm-cli migrate down`
Drop & re-run all	`sea-orm-cli migrate fresh`
Check status	`sea-orm-cli migrate status`
Generate entities	`sea-orm-cli generate entity -u DB_URL -o entity/src -l --with-serde both`

Workflow: Adding a New Table

```
1. sea-orm-cli migrate generate create_TABLE_NAME
2. Edit the generated migration file (define columns)
3. Register migration in migration/src/lib.rs
4. sea-orm-cli migrate up
5. sea-orm-cli generate entity -u DB_URL -o entity/src -l --with-serde both
6. Use the new entity in your Axum handlers
```

Workflow: Modifying an Existing Table

```
1. sea-orm-cli migrate generate add_COLUMN_to_TABLE
2. Edit migration: use Table::alter() instead of Table::create()
3. Register migration in migration/src/lib.rs
4. sea-orm-cli migrate up
5. sea-orm-cli generate entity -u DB_URL -o entity/src -l --with-serde both
```

Common Imports

```rust
// For CRUD operations
use sea_orm::{
    Database, DatabaseConnection,   // connecting
    EntityTrait,                    // find, insert, update, delete
    ActiveModelTrait,               // .insert(), .update() on ActiveModel
    ModelTrait,                     // .delete() on Model
    Set,                            // Set(value) for ActiveModel fields
    QueryFilter,                    // .filter()
    QueryOrder,                     // .order_by_asc/desc()
    ColumnTrait,                    // .eq(), .contains(), .is_in()
    PaginatorTrait,                 // .paginate()
```

```
};

// For entity types
use entity::prelude::User;      // Entity alias
use entity::user;               // Module with Model, ActiveModel, Column, etc.
```

> *[!TIP]* ***Official Docs:*** *sea-ql.org/SeaORM/docs* ***Examples:*** *github.com/SeaQL/sea-orm/tree/master/examples*