Building REST APIs in Rust with HTTPS

Table of Contents

Overview

This guide demonstrates building a production-ready REST API in Rust with HTTPS support and a corresponding console client. We'll use modern, well-maintained crates that provide excellent performance and developer experience.

Architecture Overview

- **Server**: Axum-based REST API with HTTPS/TLS
- **Database**: SQLite with SQLx for persistence
- **Serialization**: Serde for JSON handling
- **Client**: Console application using reqwest
- **TLS**: rustls for HTTPS implementation

Framework Comparison

Top Rust Web Frameworks (2024-2025)

Framework	Performance	Developer Experience	Ecosystem	Best For
Axum	High	Excellent	Rich	Balanced choice, great DX
Actix Web	Highest	Good	Mature	Maximum performance
Warp	High	Moderate	Good	Filter-based routing
Rocket	Good	Excellent	Growing	Rapid prototyping

Recommendation: We'll use **Axum** for its excellent balance of performance, developer experience, and seamless Tokio integration.

Project Setup

Server Project Structure

```
rust-api-server/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── models.rs
│   ├── handlers.rs
│   ├── database.rs
│   └── config.rs
├── migrations/
│   └── 001_initial.sql
├── certs/
│   ├── cert.pem
│   └── key.pem
└── README.md
```

Client Project Structure

```
rust-api-client/
├── Cargo.toml
├── src/
│   ├── main.rs
│   ├── client.rs
│   └── models.rs
└── README.md
```

REST API Server Implementation

Cargo.toml Dependencies

```toml
[package]
name = "rust-api-server"
version = "0.1.0"
edition = "2021"

[dependencies]
# Web framework and HTTP
axum = "0.7"
axum-server = { version = "0.7", features = ["tls-rustls"] }
tower = "0.4"
tower-http = { version = "0.5", features = ["cors", "trace"] }
hyper = "1.0"

# Async runtime
tokio = { version = "1.40", features = ["full"] }

# Serialization
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

```
# Database
sqlx = { version = "0.7", features = ["runtime-tokio-rustls", "sqlite", "chrono",
"uuid"] }

# Error handling and utilities
anyhow = "1.0"
thiserror = "1.0"
uuid = { version = "1.0", features = ["v4", "serde"] }
chrono = { version = "0.4", features = ["serde"] }

# Logging
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }

# Environment and configuration
dotenvy = "0.15"
```

Models (src/models.rs)

```
use serde::{Deserialize, Serialize};
use chrono::{DateTime, Utc};
use uuid::Uuid;

#[derive(Debug, Clone, Serialize, Deserialize, sqlx::FromRow)]
pub struct User {
    pub id: Uuid,
    pub username: String,
    pub email: String,
    pub created_at: DateTime<Utc>,
    pub updated_at: DateTime<Utc>,
}

#[derive(Debug, Deserialize)]
pub struct CreateUserRequest {
    pub username: String,
    pub email: String,
}

#[derive(Debug, Deserialize)]
pub struct UpdateUserRequest {
    pub username: Option<String>,
    pub email: Option<String>,
}

#[derive(Debug, Serialize)]
pub struct ApiResponse<T> {
    pub success: bool,
    pub data: Option<T>,
    pub message: String,
}
```

```rust
impl<T> ApiResponse<T> {
    pub fn success(data: T) -> Self {
        Self {
            success: true,
            data: Some(data),
            message: "Operation successful".to_string(),
        }
    }

    pub fn error(message: String) -> Self {
        Self {
            success: false,
            data: None,
            message,
        }
    }
}

#[derive(Debug, thiserror::Error)]
pub enum ApiError {
    #[error("Database error: {0}")]
    Database(#[from] sqlx::Error),
    #[error("Not found: {0}")]
    NotFound(String),
    #[error("Validation error: {0}")]
    Validation(String),
    #[error("Internal server error")]
    Internal,
}

impl axum::response::IntoResponse for ApiError {
    fn into_response(self) -> axum::response::Response {
        use axum::http::StatusCode;
        use axum::Json;

        let (status, message) = match self {
            ApiError::Database(_) => (StatusCode::INTERNAL_SERVER_ERROR,
self.to_string()),
            ApiError::NotFound(msg) => (StatusCode::NOT_FOUND, msg),
            ApiError::Validation(msg) => (StatusCode::BAD_REQUEST, msg),
            ApiError::Internal => (StatusCode::INTERNAL_SERVER_ERROR, "Internal
server error".to_string()),
        };

        let response = ApiResponse::<()>::error(message);
        (status, Json(response)).into_response()
    }
}

pub type ApiResult<T> = Result<T, ApiError>;
```

Database Layer (src/database.rs)

```rust
use sqlx::{SqlitePool, Row};
use uuid::Uuid;
use chrono::Utc;
use crate::models::{User, CreateUserRequest, UpdateUserRequest, ApiResult,
ApiError};

pub struct Database {
    pool: SqlitePool,
}

impl Database {
    pub async fn new(database_url: &str) -> anyhow::Result<Self> {
        let pool = SqlitePool::connect(database_url).await?;

        // Run migrations
        sqlx::migrate!("./migrations").run(&pool).await?;

        Ok(Self { pool })
    }

    pub async fn create_user(&self, request: CreateUserRequest) -> ApiResult<User>
{
        let id = Uuid::new_v4();
        let now = Utc::now();

        let user = sqlx::query_as::<_, User>(
            r#"
            INSERT INTO users (id, username, email, created_at, updated_at)
            VALUES (?, ?, ?, ?, ?)
            RETURNING *
            "#
        )
        .bind(id)
        .bind(&request.username)
        .bind(&request.email)
        .bind(now)
        .bind(now)
        .fetch_one(&self.pool)
        .await?;

        Ok(user)
    }

    pub async fn get_user(&self, id: Uuid) -> ApiResult<User> {
        let user = sqlx::query_as::<_, User>("SELECT * FROM users WHERE id = ?")
            .bind(id)
            .fetch_optional(&self.pool)
            .await?
            .ok_or_else(|| ApiError::NotFound("User not found".to_string()))?;
```

```
        Ok(user)
    }

    pub async fn list_users(&self, limit: Option<i64>, offset: Option<i64>) ->
ApiResult<Vec<User>> {
        let limit = limit.unwrap_or(50).min(100); // Cap at 100
        let offset = offset.unwrap_or(0);

        let users = sqlx::query_as::<_, User>(
            "SELECT * FROM users ORDER BY created_at DESC LIMIT ? OFFSET ?"
        )
        .bind(limit)
        .bind(offset)
        .fetch_all(&self.pool)
        .await?;

        Ok(users)
    }

    pub async fn update_user(&self, id: Uuid, request: UpdateUserRequest) ->
ApiResult<User> {
        // First check if user exists
        let existing_user = self.get_user(id).await?;

        let username = request.username.unwrap_or(existing_user.username);
        let email = request.email.unwrap_or(existing_user.email);
        let now = Utc::now();

        let user = sqlx::query_as::<_, User>(
            r#"
            UPDATE users
            SET username = ?, email = ?, updated_at = ?
            WHERE id = ?
            RETURNING *
            "#
        )
        .bind(&username)
        .bind(&email)
        .bind(now)
        .bind(id)
        .fetch_one(&self.pool)
        .await?;

        Ok(user)
    }

    pub async fn delete_user(&self, id: Uuid) -> ApiResult<()> {
        let result = sqlx::query("DELETE FROM users WHERE id = ?")
            .bind(id)
            .execute(&self.pool)
            .await?;

        if result.rows_affected() == 0 {
            return Err(ApiError::NotFound("User not found".to_string()));
```

```
        }

        Ok(())
    }
}
```

Request Handlers (src/handlers.rs)

```
use axum::{
    extract::{Path, Query, State},
    http::StatusCode,
    Json,
};
use serde::Deserialize;
use std::sync::Arc;
use uuid::Uuid;

use crate::database::Database;
use crate::models::{User, CreateUserRequest, UpdateUserRequest, ApiResponse,
ApiResult};

pub type AppState = Arc<Database>;

#[derive(Deserialize)]
pub struct ListQuery {
    limit: Option<i64>,
    offset: Option<i64>,
}

// Health check endpoint
pub async fn health_check() -> Json<ApiResponse<String>> {
    Json(ApiResponse::success("Server is healthy".to_string()))
}

// Create user
pub async fn create_user(
    State(db): State<AppState>,
    Json(request): Json<CreateUserRequest>,
) -> ApiResult<(StatusCode, Json<ApiResponse<User>>)> {
    // Basic validation
    if request.username.is_empty() {
        return Err(crate::models::ApiError::Validation("Username cannot be
empty".to_string()));
    }

    if request.email.is_empty() || !request.email.contains('@') {
        return Err(crate::models::ApiError::Validation("Valid email is
required".to_string()));
    }

    let user = db.create_user(request).await?;
```

```rust
    Ok((StatusCode::CREATED, Json(ApiResponse::success(user))))
}

// Get user by ID
pub async fn get_user(
    State(db): State<AppState>,
    Path(id): Path<Uuid>,
) -> ApiResult<Json<ApiResponse<User>>> {
    let user = db.get_user(id).await?;
    Ok(Json(ApiResponse::success(user)))
}

// List users with pagination
pub async fn list_users(
    State(db): State<AppState>,
    Query(params): Query<ListQuery>,
) -> ApiResult<Json<ApiResponse<Vec<User>>>> {
    let users = db.list_users(params.limit, params.offset).await?;
    Ok(Json(ApiResponse::success(users)))
}

// Update user
pub async fn update_user(
    State(db): State<AppState>,
    Path(id): Path<Uuid>,
    Json(request): Json<UpdateUserRequest>,
) -> ApiResult<Json<ApiResponse<User>>> {
    let user = db.update_user(id, request).await?;
    Ok(Json(ApiResponse::success(user)))
}

// Delete user
pub async fn delete_user(
    State(db): State<AppState>,
    Path(id): Path<Uuid>,
) -> ApiResult<Json<ApiResponse<String>>> {
    db.delete_user(id).await?;
    Ok(Json(ApiResponse::success("User deleted successfully".to_string())))
}
```

Configuration (src/config.rs)

```rust
use std::env;

#[derive(Debug, Clone)]
pub struct Config {
    pub database_url: String,
    pub server_host: String,
    pub server_port: u16,
    pub cert_path: String,
    pub key_path: String,
```

```
    pub log_level: String,
}

impl Config {
    pub fn from_env() -> anyhow::Result<Self> {
        dotenvy::dotenv().ok();

        Ok(Self {
            database_url: env::var("DATABASE_URL")
                .unwrap_or_else(|_| "sqlite:./users.db".to_string()),
            server_host: env::var("SERVER_HOST")
                .unwrap_or_else(|_| "127.0.0.1".to_string()),
            server_port: env::var("SERVER_PORT")
                .unwrap_or_else(|_| "3443".to_string())
                .parse()?,
            cert_path: env::var("CERT_PATH")
                .unwrap_or_else(|_| "./certs/cert.pem".to_string()),
            key_path: env::var("KEY_PATH")
                .unwrap_or_else(|_| "./certs/key.pem".to_string()),
            log_level: env::var("RUST_LOG")
                .unwrap_or_else(|_| "info".to_string()),
        })
    }
}
```

Main Server (src/main.rs)

```
mod config;
mod database;
mod handlers;
mod models;

use axum::{
    http::Method,
    routing::{get, post, put, delete},
    Router,
};
use axum_server::tls_rustls::RustlsConfig;
use std::{net::SocketAddr, sync::Arc};
use tower_http::{
    cors::{Any, CorsLayer},
    trace::TraceLayer,
};
use tracing::{info, warn};

use crate::{
    config::Config,
    database::Database,
    handlers::*,
};
```

```rust
#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // Load configuration
    let config = Config::from_env()?;

    // Initialize tracing
    tracing_subscriber::fmt()
        .with_env_filter(&config.log_level)
        .init();

    info!("Starting server with config: {:?}", config);

    // Initialize database
    let database = Arc::new(Database::new(&config.database_url).await?);
    info!("Database connected successfully");

    // Configure CORS
    let cors = CorsLayer::new()
        .allow_methods([Method::GET, Method::POST, Method::PUT, Method::DELETE])
        .allow_headers(Any)
        .allow_origin(Any);

    // Build application with routes
    let app = Router::new()
        .route("/health", get(health_check))
        .route("/api/users", post(create_user))
        .route("/api/users", get(list_users))
        .route("/api/users/:id", get(get_user))
        .route("/api/users/:id", put(update_user))
        .route("/api/users/:id", delete(delete_user))
        .layer(cors)
        .layer(TraceLayer::new_for_http())
        .with_state(database);

    // Try to configure HTTPS
    let addr = SocketAddr::new(config.server_host.parse()?, config.server_port);

    match RustlsConfig::from_pem_file(&config.cert_path, &config.key_path).await {
        Ok(tls_config) => {
            info!("HTTPS server starting on https://{}", addr);
            axum_server::bind_rustls(addr, tls_config)
                .serve(app.into_make_service())
                .await?;
        }
        Err(e) => {
            warn!("Failed to load TLS certificates: {}. Starting HTTP server
instead", e);
            info!("HTTP server starting on http://{}", addr);
            let listener = tokio::net::TcpListener::bind(addr).await?;
            axum::serve(listener, app).await?;
        }
    }
```

```
    Ok(())
}
```

Database Migration (migrations/001_initial.sql)

```
-- Initial migration for users table
CREATE TABLE IF NOT EXISTS users (
    id TEXT PRIMARY KEY NOT NULL,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    created_at TEXT NOT NULL,
    updated_at TEXT NOT NULL
);

-- Create indexes for better performance
CREATE INDEX IF NOT EXISTS idx_users_username ON users(username);
CREATE INDEX IF NOT EXISTS idx_users_email ON users(email);
CREATE INDEX IF NOT EXISTS idx_users_created_at ON users(created_at DESC);
```

HTTPS/TLS Configuration

Generating Self-Signed Certificates

For development, create self-signed certificates:

```
# Create certs directory
mkdir -p certs

# Generate private key
openssl genrsa -out certs/key.pem 2048

# Generate certificate signing request
openssl req -new -key certs/key.pem -out certs/cert.csr -subj
"/C=US/ST=State/L=City/O=Organization/CN=localhost"

# Generate self-signed certificate
openssl x509 -req -in certs/cert.csr -signkey certs/key.pem -out certs/cert.pem -
days 365

# Remove CSR file
rm certs/cert.csr
```

Production TLS with Let's Encrypt

For production, use Let's Encrypt certificates:

```rust
// Add to Cargo.toml
// instant-acme = "0.5"

use instant_acme::{Account, AuthorizationStatus, ChallengeType, Identifier,
LetsEncrypt, NewAccount, NewOrder, OrderStatus};

pub async fn get_letsencrypt_cert(domain: &str) -> anyhow::Result<(String,
String)> {
    // Create Let's Encrypt account
    let account = Account::create(
        &NewAccount {
            contact: &["mailto:admin@example.com"],
            terms_of_service_agreed: true,
            only_return_existing: false,
        },
        LetsEncrypt::Staging.url(), // Use Production for live certs
        None,
    ).await?;

    // Create new order
    let identifier = Identifier::Dns(domain.to_string());
    let (mut order, order_url) = account
        .new_order(&NewOrder {
            identifiers: &[identifier],
        })
        .await?;

    // Handle HTTP-01 challenge
    let authorizations = order.authorizations().await?;
    for authz in &authorizations {
        match authz.status {
            AuthorizationStatus::Pending => {
                let challenge = authz
                    .challenges
                    .iter()
                    .find(|c| c.r#type == ChallengeType::Http01)
                    .ok_or_else(|| anyhow::anyhow!("No HTTP-01 challenge
found"))?;

                let key_auth = order.key_authorization(challenge);

                // TODO: Serve key_auth at http://{domain}/.well-known/acme-
challenge/{challenge.token}
                // This requires setting up an HTTP server to serve the challenge

                order.set_challenge_ready(&challenge.url).await?;
            }
            AuthorizationStatus::Valid => {}
            _ => anyhow::bail!("Authorization failed"),
        }
    }

    // Wait for order to be ready
```

```
    let mut tries = 1u8;
    let mut delay = std::time::Duration::from_millis(250);
    loop {
        match order.refresh().await?.status {
            OrderStatus::Ready => break,
            OrderStatus::Processing => {
                if tries < 5 {
                    tries += 1;
                    tokio::time::sleep(delay).await;
                    delay *= 2;
                } else {
                    anyhow::bail!("Order processing timeout");
                }
            }
            other => anyhow::bail!("Order failed with status: {:?}", other),
        }
    }

    // Generate private key and CSR
    let private_key = rcgen::KeyPair::generate(&rcgen::PKCS_ECDSA_P256_SHA256)?;
    let mut params = rcgen::CertificateParams::new(vec![domain.to_string()]);
    params.key_pair = Some(private_key);
    let cert = rcgen::Certificate::from_params(params)?;
    let csr = cert.serialize_request_der()?;

    // Finalize order
    order.finalize(&csr).await?;

    // Download certificate
    let cert_chain_pem = loop {
        match order.certificate().await? {
            Some(cert_chain_pem) => break cert_chain_pem,
            None => tokio::time::sleep(std::time::Duration::from_secs(1)).await,
        }
    };

    let private_key_pem = cert.serialize_private_key_pem();

    Ok((cert_chain_pem, private_key_pem))
}
```

Environment Configuration (.env)

```
DATABASE_URL=sqlite:./users.db
SERVER_HOST=127.0.0.1
SERVER_PORT=3443
CERT_PATH=./certs/cert.pem
KEY_PATH=./certs/key.pem
RUST_LOG=info
```

Console Client Implementation

Client Dependencies (Cargo.toml)

```toml
[package]
name = "rust-api-client"
version = "0.1.0"
edition = "2021"

[dependencies]
# HTTP client
reqwest = { version = "0.12", features = ["json", "rustls-tls"] }

# Async runtime
tokio = { version = "1.40", features = ["full"] }

# Serialization
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

# CLI and utilities
clap = { version = "4.0", features = ["derive"] }
uuid = { version = "1.0", features = ["v4"] }
anyhow = "1.0"
colored = "2.0"

# TLS
rustls = "0.23"
webpki-roots = "0.26"
```

Client Models (src/models.rs)

```rust
use serde::{Deserialize, Serialize};
use uuid::Uuid;

// Re-use the same models as the server
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct User {
    pub id: Uuid,
    pub username: String,
    pub email: String,
    pub created_at: String,
    pub updated_at: String,
}

#[derive(Debug, Serialize)]
pub struct CreateUserRequest {
    pub username: String,
    pub email: String,
}
```

```rust
#[derive(Debug, Serialize)]
pub struct UpdateUserRequest {
    pub username: Option<String>,
    pub email: Option<String>,
}

#[derive(Debug, Deserialize)]
pub struct ApiResponse<T> {
    pub success: bool,
    pub data: Option<T>,
    pub message: String,
}
```

HTTP Client (src/client.rs)

```rust
use reqwest::{Client, ClientBuilder};
use serde::de::DeserializeOwned;
use std::time::Duration;
use uuid::Uuid;

use crate::models::{User, CreateUserRequest, UpdateUserRequest, ApiResponse};

pub struct ApiClient {
    client: Client,
    base_url: String,
}

impl ApiClient {
    pub fn new(base_url: String, accept_invalid_certs: bool) ->
anyhow::Result<Self> {
        let mut builder = ClientBuilder::new()
            .timeout(Duration::from_secs(30))
            .user_agent("rust-api-client/0.1.0");

        if accept_invalid_certs {
            builder = builder.danger_accept_invalid_certs(true);
        }

        let client = builder.build()?;

        Ok(Self { client, base_url })
    }

    async fn request<T: DeserializeOwned>(&self, method: reqwest::Method, path:
&str, body: Option<impl serde::Serialize>) -> anyhow::Result<ApiResponse<T>> {
        let url = format!("{}{}", self.base_url, path);
        let mut request = self.client.request(method, &url);

        if let Some(body) = body {
            request = request.json(&body);
```

```
        }

        let response = request.send().await?;
        let status = response.status();
        let text = response.text().await?;

        if !status.is_success() {
            anyhow::bail!("HTTP {} - {}", status, text);
        }

        let api_response: ApiResponse<T> = serde_json::from_str(&text)
            .map_err(|e| anyhow::anyhow!("Failed to parse response: {} - Body:
{}", e, text))?;

        Ok(api_response)
    }

    pub async fn health_check(&self) -> anyhow::Result<String> {
        let response: ApiResponse<String> = self.request(reqwest::Method::GET,
"/health", None::<()>).await?;
        Ok(response.data.unwrap_or(response.message))
    }

    pub async fn create_user(&self, username: String, email: String) ->
anyhow::Result<User> {
        let request = CreateUserRequest { username, email };
        let response: ApiResponse<User> = self.request(reqwest::Method::POST,
"/api/users", Some(request)).await?;
        response.data.ok_or_else(|| anyhow::anyhow!("No user data in response"))
    }

    pub async fn get_user(&self, id: Uuid) -> anyhow::Result<User> {
        let path = format!("/api/users/{}", id);
        let response: ApiResponse<User> = self.request(reqwest::Method::GET,
&path, None::<()>).await?;
        response.data.ok_or_else(|| anyhow::anyhow!("No user data in response"))
    }

    pub async fn list_users(&self, limit: Option<u32>, offset: Option<u32>) ->
anyhow::Result<Vec<User>> {
        let mut path = "/api/users".to_string();
        let mut params = Vec::new();

        if let Some(limit) = limit {
            params.push(format!("limit={}", limit));
        }
        if let Some(offset) = offset {
            params.push(format!("offset={}", offset));
        }

        if !params.is_empty() {
            path.push_str(&format!("?{}", params.join("&")));
        }
```

```
        let response: ApiResponse<Vec<User>> = self.request(reqwest::Method::GET,
&path, None::<()>).await?;
        Ok(response.data.unwrap_or_default())
    }

    pub async fn update_user(&self, id: Uuid, username: Option<String>, email:
Option<String>) -> anyhow::Result<User> {
        let request = UpdateUserRequest { username, email };
        let path = format!("/api/users/{}", id);
        let response: ApiResponse<User> = self.request(reqwest::Method::PUT,
&path, Some(request)).await?;
        response.data.ok_or_else(|| anyhow::anyhow!("No user data in response"))
    }

    pub async fn delete_user(&self, id: Uuid) -> anyhow::Result<String> {
        let path = format!("/api/users/{}", id);
        let response: ApiResponse<String> = self.request(reqwest::Method::DELETE,
&path, None::<()>).await?;
        Ok(response.message)
    }
}
```

CLI Application (src/main.rs)

```
mod client;
mod models;

use clap::{Parser, Subcommand};
use colored::*;
use uuid::Uuid;

use crate::client::ApiClient;

#[derive(Parser)]
#[command(name = "api-client")]
#[command(about = "A CLI client for the Rust REST API")]
struct Cli {
    #[arg(long, default_value = "https://127.0.0.1:3443")]
    url: String,

    #[arg(long)]
    insecure: bool,

    #[command(subcommand)]
    command: Commands,
}

#[derive(Subcommand)]
enum Commands {
    /// Check API health
    Health,
```

```
    /// Create a new user
    CreateUser {
        username: String,
        email: String,
    },
    /// Get user by ID
    GetUser {
        id: String,
    },
    /// List all users
    ListUsers {
        #[arg(long)]
        limit: Option<u32>,
        #[arg(long)]
        offset: Option<u32>,
    },
    /// Update user
    UpdateUser {
        id: String,
        #[arg(long)]
        username: Option<String>,
        #[arg(long)]
        email: Option<String>,
    },
    /// Delete user
    DeleteUser {
        id: String,
    },
}

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let cli = Cli::parse();
    let client = ApiClient::new(cli.url, cli.insecure)?;

    match cli.command {
        Commands::Health => {
            println!("Checking API health...");
            match client.health_check().await {
                Ok(message) => println!("{} {}", "✓".green(), message),
                Err(e) => println!("{} Health check failed: {}", "✗".red(), e),
            }
        }

        Commands::CreateUser { username, email } => {
            println!("Creating user...");
            match client.create_user(username, email).await {
                Ok(user) => {
                    println!("{} User created successfully:", "✓".green());
                    print_user(&user);
                }
                Err(e) => println!("{} Failed to create user: {}", "✗".red(), e),
            }
        }
```

```
        Commands::GetUser { id } => {
            let user_id = Uuid::parse_str(&id)?;
            println!("Fetching user {}...", id);
            match client.get_user(user_id).await {
                Ok(user) => {
                    println!("{} User found:", "✓".green());
                    print_user(&user);
                }
                Err(e) => println!("{} Failed to get user: {}", "✗".red(), e),
            }
        }

        Commands::ListUsers { limit, offset } => {
            println!("Fetching users...");
            match client.list_users(limit, offset).await {
                Ok(users) => {
                    println!("{} Found {} users:", "✓".green(), users.len());
                    for user in users {
                        print_user(&user);
                        println!("---");
                    }
                }
                Err(e) => println!("{} Failed to list users: {}", "✗".red(), e),
            }
        }

        Commands::UpdateUser { id, username, email } => {
            let user_id = Uuid::parse_str(&id)?;
            println!("Updating user {}...", id);
            match client.update_user(user_id, username, email).await {
                Ok(user) => {
                    println!("{} User updated successfully:", "✓".green());
                    print_user(&user);
                }
                Err(e) => println!("{} Failed to update user: {}", "✗".red(), e),
            }
        }

        Commands::DeleteUser { id } => {
            let user_id = Uuid::parse_str(&id)?;
            println!("Deleting user {}...", id);
            match client.delete_user(user_id).await {
                Ok(message) => println!("{} {}", "✓".green(), message),
                Err(e) => println!("{} Failed to delete user: {}", "✗".red(), e),
            }
        }
    }

    Ok(())
}

fn print_user(user: &crate::models::User) {
    println!("ID: {}", user.id.to_string().cyan());
```

```
    println!("Username: {}", user.username.bright_white());
    println!("Email: {}", user.email.bright_white());
    println!("Created: {}", user.created_at.dim());
    println!("Updated: {}", user.updated_at.dim());
}
```

Testing and Deployment

Integration Tests

Create `tests/integration_test.rs`:

```
use std::sync::Arc;
use uuid::Uuid;
use rust_api_server::{
    database::Database,
    models::{CreateUserRequest, UpdateUserRequest},
};

#[tokio::test]
async fn test_user_crud_operations() {
    // Use in-memory SQLite for testing
    let db = Arc::new(Database::new("sqlite::memory:").await.unwrap());

    // Test create user
    let create_request = CreateUserRequest {
        username: "testuser".to_string(),
        email: "test@example.com".to_string(),
    };

    let user = db.create_user(create_request).await.unwrap();
    assert_eq!(user.username, "testuser");
    assert_eq!(user.email, "test@example.com");

    // Test get user
    let fetched_user = db.get_user(user.id).await.unwrap();
    assert_eq!(fetched_user.id, user.id);

    // Test update user
    let update_request = UpdateUserRequest {
        username: Some("updateduser".to_string()),
        email: None,
    };

    let updated_user = db.update_user(user.id, update_request).await.unwrap();
    assert_eq!(updated_user.username, "updateduser");
    assert_eq!(updated_user.email, "test@example.com"); // Should remain unchanged

    // Test list users
    let users = db.list_users(Some(10), Some(0)).await.unwrap();
    assert_eq!(users.len(), 1);
```

```
    // Test delete user
    db.delete_user(user.id).await.unwrap();

    // Verify user is deleted
    let result = db.get_user(user.id).await;
    assert!(result.is_err());
}
```

Docker Deployment

Create `Dockerfile`:

```
# Build stage
FROM rust:1.75 as builder

WORKDIR /app
COPY Cargo.toml Cargo.lock ./
COPY src ./src
COPY migrations ./migrations

RUN cargo build --release

# Runtime stage
FROM debian:bookworm-slim

RUN apt-get update && apt-get install -y \
    ca-certificates \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY --from=builder /app/target/release/rust-api-server .
COPY --from=builder /app/migrations ./migrations

EXPOSE 3443

CMD ["./rust-api-server"]
```

Create `docker-compose.yml`:

```
version: '3.8'

services:
  api:
    build: .
    ports:
      - "3443:3443"
    environment:
```

```
      - DATABASE_URL=sqlite:./data/users.db
      - SERVER_HOST=0.0.0.0
      - SERVER_PORT=3443
      - RUST_LOG=info
    volumes:
      - ./data:/app/data
      - ./certs:/app/certs
    restart: unless-stopped
```

Usage Examples

Start the server:

```
cd rust-api-server
cargo run
```

Using the client:

```
cd rust-api-client

# Check health
cargo run -- health

# Create user
cargo run -- create-user john john@example.com

# List users
cargo run -- list-users

# Get specific user
cargo run -- get-user <user-id>

# Update user
cargo run -- update-user <user-id> --username johnny --email johnny@example.com

# Delete user
cargo run -- delete-user <user-id>

# Use with custom URL (for different environments)
cargo run -- --url https://api.example.com list-users

# Accept invalid certificates (for development)
cargo run -- --insecure health
```

Best Practices

Security

1. **Always use HTTPS in production**
2. **Validate all input data**
3. **Implement rate limiting** with `tower-governor`
4. **Use secure headers** with `tower-http`
5. **Sanitize error messages** to avoid information leakage

Performance

1. **Use connection pooling** for database connections
2. **Implement caching** with Redis for frequently accessed data
3. **Add compression** with `CompressionLayer`
4. **Use async throughout** the application stack
5. **Profile and benchmark** critical paths

Monitoring

1. **Structured logging** with `tracing`
2. **Metrics collection** with Prometheus
3. **Health checks** for load balancers
4. **Error tracking** with Sentry integration
5. **Performance monitoring** with OpenTelemetry

Code Quality

1. **Comprehensive testing** (unit, integration, end-to-end)
2. **Error handling** with proper error types
3. **Documentation** with `cargo doc`
4. **Linting** with `clippy`
5. **Formatting** with `rustfmt`

This guide provides a solid foundation for building production-ready REST APIs in Rust with modern tooling and best practices.