RusterAPI: API REST Generadora Dinámica

Proyecto Integrado - Desarrollo de Aplicaciones Multiplataforma

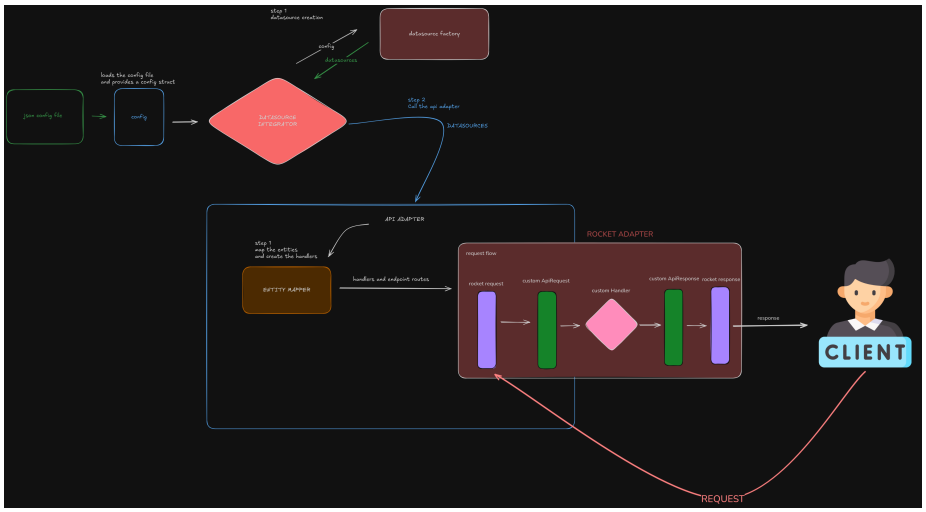

Figure 1: RusterAPI Logo

Autor: Manuel Cervantes **Especialidad:** Desarrollo de Aplicaciones Multiplataforma
Curso académico: 2024-2025
Profesor tutor: Raul Jimenez
Centro educativo: I.E.S. Zaidín-Vergeles (GRANADA)

Índice

12. Despliegue
4. Conclusiones
5. Recursos Bibliográficos
6. Anexos

1. Definición del Proyecto

RusterAPI es una aplicación multiplataforma desarrollada en Rust que permite generar, configurar y ejecutar APIs REST de forma dinámica. La aplicación facilita la creación de APIs basadas en una configuración definida por el usuario, conectándose a diversas fuentes de datos (principalmente bases de datos MariaDB), y generando automáticamente los endpoints CRUD (Create, Read, Update, Delete) para entidades definidas.

El objetivo principal del proyecto es ofrecer una plataforma unificada que simplifique el proceso de desarrollo de APIs, permitiendo a los desarrolladores centrarse en la lógica de negocio en vez de en la implementación técnica de la API. La aplicación combina un backend robusto escrito en Rust con una interfaz de usuario moderna desarrollada en React, todo integrado mediante el framework Tauri para crear una aplicación de escritorio multiplataforma.

Entre las capacidades más destacadas del sistema se incluyen: - Conexión a bases de datos MariaDB y configuración dinámica de entidades - Generación automática de endpoints CRUD para cada entidad - Interfaz gráfica para la configuración y prueba de la API generada - Monitorización del rendimiento y logs del servidor API - Validación y testeo de los endpoints generados

2. Palabras Clave

rust, api, rest, generador, tauri, react, mariadb, multiplataforma, automatización

3. Desarrollo del Proyecto

3.1 Descripción General

RusterAPI emerge como una solución a la repetitividad y complejidad inherente al desarrollo de APIs REST desde cero. El sistema ofrece una alternativa automatizada que abstrae gran parte del trabajo técnico, permitiendo a los desarrolladores configurar visualmente una API y obtener inmediatamente un servidor funcional sin necesidad de escribir código.

La aplicación se divide en dos componentes principales: 1. **Backend en Rust**: Encargado de la generación y ejecución de los servicios API, con un diseño modular y extensible. 2. **Interfaz de usuario en React**: Proporciona una experiencia de usuario intuitiva para la configuración y monitorización del sistema.

Todo esto se integra mediante Tauri, que permite ejecutar la aplicación como un programa de escritorio nativo en distintos sistemas operativos.

3.2 Tecnologías Utilizadas

El proyecto ha requerido del uso de múltiples tecnologías modernas para su implementación:

Backend: - **Rust**: Lenguaje de programación principal para el backend, elegido por su rendimiento, seguridad y confiabilidad. - **Rocket**: Framework web para Rust, utilizado para gestionar las peticiones HTTP. - **SQLx**: Biblioteca para interactuar con bases de datos SQL desde Rust. - **Serde**: Utilizada para la serialización y deserialización de datos. - **Tokio**: Runtime asíncrono para Rust, permitiendo operaciones concurrentes eficientes.

Frontend: - **React**: Biblioteca de JavaScript para construir interfaces de usuario. - **TypeScript**: Superset tipado de JavaScript para desarrollo más seguro. - **Material-UI**: Biblioteca de componentes para React con estética moderna. - **React Router**: Para gestión de rutas en la aplicación.

Integración y Despliegue: - **Tauri**: Framework para construir aplicaciones de escritorio multiplataforma usando tecnologías web. - **Vite**: Herramienta de construcción para el frontend, proporcionando un entorno de desarrollo rápido. - **JSON Schema**: Para validación y definición de esquemas de configuración.

Base de Datos: - **MariaDB/MySQL**: Sistema de gestión de bases de datos relacional.

3.3 Arquitectura del Sistema

La arquitectura de RusterAPI sigue un diseño modular basado en capas, lo que facilita la extensibilidad y el mantenimiento del código. Las principales capas son:

1. **Capa de Presentación**:
   - Interfaz de usuario React
   - Componentes Tauri para integración con el sistema nativo
2. **Capa de Aplicación**:
   - Gestión de comandos Tauri
   - Procesamiento de configuración
   - Generación dinámica de endpoints
3. **Capa de Dominio**:
   - Lógica de negocio para generación de APIs
   - Adaptadores para diferentes tipos de entidades
   - Gestión de los endpoints CRUD
4. **Capa de Infraestructura**:
   - Conexión con bases de datos
   - Servidor Rocket para API REST
   - Gestión de estado del servidor

El flujo de datos sigue un patrón unidireccional, donde la interfaz de usuario envía comandos al backend a través de la API de Tauri, y el backend procesa estos comandos y devuelve resultados que se muestran en la interfaz.

3.4 Módulos Principales

El sistema se divide en varios módulos principales, cada uno con responsabilidades específicas:

1. **Módulo de Configuración (`config`)**: Gestiona la configuración de la API, incluyendo entidades, bases de datos, y opciones del servidor.
2. **Módulo de Datos (`data`)**: Encargado de la interacción con la fuente de datos, principalmente bases de datos MariaDB.
3. **Módulo API (`api`)**: Contiene la lógica para generar y gestionar los endpoints de la API REST.
    - **Adaptadores**: Conectan la configuración con las fuentes de datos.
    - **Manejadores**: Implementan la lógica CRUD para las entidades.
    - **Rocket**: Gestiona la exposición HTTP de los endpoints.
4. **Módulo de Conexión (`connection`)**: Administra las conexiones a bases de datos y otros servicios externos.
5. **Módulo de Serialización (`serialization`)**: Responsable de convertir datos entre formatos diferentes.
6. **Módulo de Errores (`error`)**: Manejo centralizado de errores y excepciones.
7. **Interfaz Gráfica**: Implementada con React, proporciona las pantallas para:
    - Configuración de conexión a bases de datos
    - Selección y configuración de entidades
    - Monitorización y testeo de la API generada

3.5 Estructura del Proyecto

El proyecto sigue una estructura clara para organizar los diferentes componentes:

```
RusterAPI/
├── Cargo.toml              # Configuración del proyecto Rust
├── README.md               # Documentación principal
├── frontend/               # Código fuente de la interfaz de usuario
│   ├── package.json        # Dependencias de NPM
│   ├── src/                # Código fuente React
│   │   ├── components/     # Componentes de UI
│   │   ├── hooks/          # Hooks personalizados
│   │   ├── utils/          # Utilidades
```

```
│   │   └── App.tsx         # Componente principal
├── src/                   # Código fuente Rust para la biblioteca central
│   ├── api/               # Módulos de API REST
│   ├── config/            # Gestión de configuración
│   ├── data/              # Acceso a datos
│   ├── connection/        # Gestión de conexiones
│   ├── serialization/     # Conversión de datos
│   ├── error.rs           # Manejo de errores
│   └── lib.rs             # Punto de entrada de la biblioteca
└── src-tauri/             # Integración con Tauri
    ├── Cargo.toml         # Dependencias Rust para Tauri
    ├── tauri.conf.json    # Configuración Tauri
    ├── config/            # Archivos de configuración
    └── src/               # Código fuente Rust para Tauri
        └── main.rs        # Punto de entrada
```

3.6 Interfaz de Usuario

La interfaz de usuario de RusterAPI está diseñada para ser moderna, intuitiva y responsive. Sigue un enfoque de diseño basado en componentes y utiliza Material-UI para proporcionar una experiencia coherente.

Las principales pantallas de la aplicación incluyen:

1. **Conexión a Base de Datos**: Permite configurar la conexión a la fuente de datos.
2. **Selección de Entidades**: Muestra las tablas disponibles y permite seleccionar cuáles exponer como API.
3. **Configuración de Entidades**: Permite personalizar cada entidad, definiendo campos, relaciones y validaciones.
4. **Monitor del Servidor**: Muestra estadísticas en tiempo real del rendimiento de la API.
5. **Testing de API**: Herramienta para probar los endpoints generados directamente desde la interfaz.

La navegación entre pantallas sigue un flujo de trabajo lógico, guiando al usuario a través del proceso de configuración y despliegue de la API.

3.7 API y Servicios

El corazón del sistema es el módulo de API, que implementa un enfoque flexible para generar servicios REST. Los principales componentes incluyen:

1. **ApiAdapter**: Actúa como punto de entrada para las solicitudes HTTP y las dirige a los manejadores correspondientes.
2. **EntityApi**: Representa una entidad expuesta como API, con sus correspondientes endpoints.

3. **EndpointHandler**: Funciones que manejan las operaciones CRUD para cada entidad.

Para cada entidad, se generan automáticamente los siguientes endpoints:

- `GET /{entidad}`: Lista todos los elementos de la entidad
- `GET /{entidad}/:id`: Obtiene un elemento específico por su ID
- `POST /{entidad}`: Crea un nuevo elemento
- `PUT /{entidad}/:id`: Actualiza un elemento existente
- `DELETE /{entidad}/:id`: Elimina un elemento

La API utiliza JSON como formato de intercambio de datos y sigue principios REST para su diseño.

3.8 Base de Datos

RusterAPI está diseñado principalmente para trabajar con MariaDB/MySQL, pero su arquitectura permite extender el soporte a otros sistemas de bases de datos en el futuro.

La interacción con la base de datos se realiza a través de:

1. **DataSource**: Interfaz abstracta para acceder a los datos.
2. **DbConfig**: Configuración de conexión a la base de datos.
3. **ConnectionManager**: Gestiona el pool de conexiones para optimizar el rendimiento.

El sistema valida la conexión a la base de datos antes de iniciar el servidor API, proporcionando mensajes de error detallados en caso de fallos.

3.9 Seguridad

Aunque el foco principal del proyecto es la generación dinámica de APIs, se han considerado aspectos de seguridad básicos:

1. **Validación de entradas**: Los datos recibidos se validan antes de ser procesados.
2. **Control de errores**: Los errores se manejan de forma segura, sin exponer información sensible.
3. **Seguridad en conexiones**: Las contraseñas y datos sensibles se protegen adecuadamente en la configuración.

El sistema está preparado para implementar mecanismos de autenticación y autorización más avanzados en futuras versiones.

3.10 Patrones de Diseño

RusterAPI implementa varios patrones de diseño para mejorar la calidad y mantenibilidad del código:

1. **Patrón Adaptador**: Utilizado para integrar diferentes fuentes de datos con el sistema de API.
2. **Patrón Fábrica**: Implementado en `DataSourceFactory` para crear instancias de fuentes de datos.
3. **Inyección de Dependencias**: Los componentes reciben sus dependencias en vez de crearlas internamente.
4. **Patrón Observador**: Utilizado para monitorizar el estado del servidor API.
5. **Modelo-Vista-Controlador (MVC)**: Separación clara entre la lógica de negocio, la presentación y el control.

3.11 Testing

El proyecto incluye varios niveles de pruebas para garantizar la calidad del código:

1. **Pruebas Unitarias**: Verifican el funcionamiento correcto de componentes individuales.
2. **Pruebas de Integración**: Comprueban la correcta interacción entre diferentes módulos.
3. **Herramienta de Testing**: La interfaz incluye una herramienta para probar manualmente los endpoints generados.

Las pruebas se ejecutan automáticamente durante el desarrollo y antes de las liberaciones.

3.12 Despliegue

La aplicación se distribuye como un ejecutable nativo gracias a Tauri, lo que permite su instalación directa en Windows, macOS y Linux sin necesidad de instalar dependencias adicionales.

El proceso de construcción y empaquetado se realiza mediante:

1. Compilación del backend en Rust
2. Compilación del frontend React
3. Integración mediante Tauri
4. Generación de instaladores específicos para cada plataforma

4. Conclusiones

El desarrollo de RusterAPI ha representado un desafío técnico significativo, pero también una oportunidad para aplicar y consolidar conocimientos adquiridos durante la formación en Desarrollo de Aplicaciones Multiplataforma. Entre las principales conclusiones extraídas del proyecto destacan:

1. **Importancia de la arquitectura modular**: La decisión de implementar una arquitectura por capas y modular ha facilitado enormemente el

desarrollo y mantenimiento del sistema, permitiendo realizar cambios en componentes específicos sin afectar al resto.

2. **Ventajas de Rust**: La elección de Rust como lenguaje principal para el backend ha demostrado ser acertada, proporcionando rendimiento, seguridad y un sistema de tipos robusto que ha evitado numerosos problemas potenciales.

3. **Integración frontend-backend**: El uso de Tauri ha permitido crear una aplicación de escritorio completa sin sacrificar la potencia de las tecnologías web modernas para la interfaz de usuario.

4. **Desafíos enfrentados**: Durante el desarrollo surgieron varios obstáculos, especialmente en la implementación de la generación dinámica de endpoints y la gestión asíncrona de las conexiones a bases de datos. La resolución de estos problemas ha requerido investigación y experimentación constante.

5. **Aprendizaje adquirido**: El proyecto ha permitido profundizar en tecnologías emergentes como Rust y Tauri, así como consolidar conocimientos en desarrollo de APIs REST, programación asíncrona y arquitectura de software.

6. **Futuras mejoras**: Como todo proyecto de software, RusterAPI tiene potencial para continuar evolucionando. Entre las posibles mejoras futuras se incluyen el soporte para más tipos de bases de datos, implementación de autenticación OAuth, generación de documentación automática de la API y mejoras en el rendimiento para grandes volúmenes de datos.

En resumen, RusterAPI representa una aplicación práctica de los conocimientos adquiridos durante la formación, demostrando capacidades para diseñar, implementar y desplegar soluciones software completas y funcionales.

5. Recursos Bibliográficos

- Klabnik, S., & Nichols, C. (2023). *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/
- Rocket Team (2023). *Rocket Web Framework Documentation*. https://rocket.rs/v0.5/guide/
- Tauri Team (2023). *Tauri Documentation*. https://tauri.app/v1/guides/
- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm (Consultado: 15/02/2023)
- SQLx Team (2023). *SQLx Documentation*. https://github.com/launchbadge/sqlx/tree/main/sqlx-docs

- React Team (2023). *React Documentation*. https://reactjs.org/docs/getting-started.html
- MariaDB Documentation (2023). *MariaDB Server Documentation*. https://mariadb.com/kb/en/documentation/
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
- Richardson, L., & Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media.
- Material-UI Team (2023). *Material-UI Documentation*. https://mui.com/getting-started/installation/

6. Anexos

Anexo I: Ejemplos de Código

Ejemplo 1: Definición de la estructura ApiAdapter

```rust
/// ApiAdapter serves as the main interface for handling API operations.
/// It acts as a bridge between the configuration and the data sources for each entity.
pub struct ApiAdapter<T> {
    pub config: Config,
    pub entities: HashMap<String, EntityApi<T>>,
}

impl<T: ApiEntity> ApiAdapter<T> {
    /// Creates a new ApiAdapter with the provided configuration and data sources
    pub fn new(config: Config, datasources: HashMap<String, Box<dyn DataSource<T>>>) -> Self {
        let mut entities = HashMap::new();
        entity_mapper(&config, datasources, &mut entities);
        Self { config, entities }
    }

    /// Starts the API server based on the configuration
    pub async fn start_server(&self) -> Result<()> {
        // Use the Rocket adapter for server implementation
        rocket_adapter::start_server(self.clone()).await
    }
}
```

Ejemplo 2: Implementación de un endpoint CRUD

```rust
/// Registers a create endpoint for an entity
pub fn register_create_endpoint<T>(
    datasource: Box<dyn DataSource<T>>,
```

```
    entity: &Entity,
    endpoints: &mut HashMap<String, EndpointHandler<T>>,
)
where
    T: ApiEntity,
{
    let base_path = entity.name.clone();
    let endpoint_key = format!("POST:{}", base_path);

    // Create a thread-safe clone of the datasource for the handler
    let ds = datasource.box_clone();
    let entity_name = entity.name.clone();

    // Handler for the create endpoint
    let handler = Arc::new(move |request: ApiRequest| -> Result<ApiResponse<T>> {
        // Validate that we have a request body
        let body = match &request.body {
            Some(b) if !b.is_empty() => b,
            _ => return Err(RusterApiError::BadRequest("Request body is required".to_string())),
        };

        // Deserialize the request body into the entity type
        let new_item: T = serde_json::from_str(body).map_err(|e| {
            RusterApiError::BadRequest(format!("Invalid request format: {}", e))
        })?;

        // Attempt to create the item in the datasource
        match ds.create(new_item, Some(&entity_name)) {
            Ok(created_item) => {
                Ok(ApiResponse {
                    status: 201,
                    headers: default_headers(),
                    body: Some(ApiResponseBody::Single(created_item)),
                })
            },
            Err(e) => {
                Err(RusterApiError::ServerError(format!("Failed to create item: {}", e)))
            }
        }
    });

    // Register the handler for this endpoint
    if endpoints.insert(endpoint_key.clone(), handler.clone()).is_some() {
        eprintln!("Warning: Overwriting existing handler for endpoint key: {}", endpoint_key);
    }
}
```

Anexo II: Diagramas

[Insertar diagramas relevantes como diagramas de clases, diagramas de secuencia, diagramas de arquitectura, etc.]

Anexo III: Manual de Usuario

[Insertar manual de usuario básico con instrucciones para instalar y utilizar la aplicación]

Anexo IV: Glosario de Términos

- **API REST**: Interfaz de programación de aplicaciones que sigue los principios de arquitectura REST (Representational State Transfer).
- **CRUD**: Create, Read, Update, Delete. Operaciones básicas para la manipulación de datos.
- **Endpoint**: Punto de acceso en una API que representa un recurso específico.
- **MariaDB**: Sistema de gestión de bases de datos relacional, fork de MySQL.
- **Rust**: Lenguaje de programación de sistemas enfocado en seguridad, especialmente seguridad de memoria.
- **React**: Biblioteca de JavaScript para construir interfaces de usuario.
- **Tauri**: Framework para construir aplicaciones de escritorio multiplataforma usando tecnologías web.
- **JWT**: JSON Web Token, estándar para la creación de tokens de acceso.