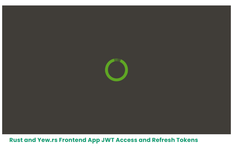
Build a Simple API in Rust

May 7, 2023 0 Comments 8

This tutorial will teach you how to build a simple CRUD API in Rust using the **Warp** web framework and **Tokio**. We'll use Rust and a few well-known libraries to create a RESTful API that runs on a Warp HTTP server and stores data in memory.

Warp is a blazingly fast, rust-based, web server framework built on top of Tokio's **runtime**. It provides a high-level, easy-to-use **filter** system for composing HTTP servers, making it the go-to framework for building web applications in Rust.

The main focus of this tutorial is to create a REST API with endpoints for handling **Create**, **Read**, **Update**, and **Delete** operations against an in-memory database.



Rust and Yew.rs Frontend App JWT Access and Refresh Tokens

More practice:

- Build a CRUD API with Node.js and Sequelize
- Build a CRUD App with FastAPI and SQLAlchemy
- Build a CRUD App with FastAPI and PyMongo
- Build CRUD API with Django REST framework
- Node.js, Express, TypeORM, PostgreSQL: CRUD Rest API
- Build CRUD RESTful API Server with Golang, Gin, and MongoDB

Table of Contents

Prerequisites

I've written this guide to accommodate anyone with a basic understanding of Rust, but these prerequisites are needed to get the most out of the tutorial.

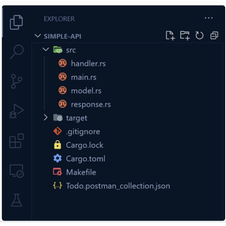
- A basic understanding of Rust will be beneficial
- Basic knowledge of REST API architecture
- Postman or any API testing software

Run the Rust API Project Locally

- Download or clone the Rust API project from **https://github.com/wpcodevo/simple-api** and open the source code in an IDE.
- Run `cargo r -r` in the terminal of the root directory to install the project's dependencies and start the Warp HTTP server.
- Import the **Todo.postman_collection.json** file available in the root directory into Postman or **Thunder Client** VS Code extension to test the API endpoints.

Setup the Rust Project

At the end of this comprehensive guide, your folder structure should look somewhat like this except for the Postman collection and Makefile.



To start, navigate to a convenient location on your machine, and create a new directory where all the files for this project will reside. You can name the folder `simple-api`. Once you have done this, change to the newly-created directory and open it in an IDE or text editor.

```
mkdir simple-api
cd simple-api && code .
```

Next, open the integrated terminal in your IDE or text editor and run this command to initialize the Rust project with Cargo.

```
cargo init
```

Now run these commands to install the dependencies required for the project:

```
cargo add warp
cargo add serde --features derive
cargo add chrono --features serde
cargo add tokio --features full
cargo add pretty_env_logger
cargo add uuid --features v4
```

- `warp` – A web server framework for Rust
- `serde` – A framework for serializing and deserializing Rust data structures to and from various data formats like JSON, XML, etc.
- `chrono` – Has utility functions for manipulating Date and Time in Rust.
- `tokio` – A library that provides a runtime and functions for handling asynchronous I/O in Rust.
- `pretty_env_logger` – A simple logger built on top of **env_logger**.
- `uuid` – A library for generating and parsing UUIDs in Rust.

After the installation, your **Cargo.toml** file should look like this:

Cargo.toml

```
[package]
name = "simple-api"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/refe

[dependencies]
chrono = { version = "0.4.23", features = ["serde"] }
pretty_env_logger = "0.4.0"
serde = { version = "1.0.152", features = ["derive"] }
tokio = { version = "1.24.1", features = ["full"] }
uuid = { version = "1.2.2", features = ["v4"] }
warp = "0.3.3"
```

> You can use the dependency versions provided in the above **Cargo.toml** file in case the latest version of the dependencies breaks your app.

To get our hands dirty, let's create a basic Warp endpoint `/api/healthchecker` to return a simple JSON object. Open the `src/main.rs` file and replace its content with the following code.

src/main.rs

```
use warp::{reply::json, Filter, Rejection, Reply};

type WebResult<T> = std::result::Result<T, Rejection>;

#[derive(Serialize)]
pub struct GenericResponse {
    pub status: String,
    pub message: String,
}

pub async fn health_checker_handler() -> WebResult<impl Reply> {
    const MESSAGE: &str = "Build Simple CRUD API with Rust";

    let response_json = &GenericResponse {
        status: "success".to_string(),
        message: MESSAGE.to_string(),
    });
```

Support Me!
PayPal Donate Now
report this ad

Recent posts
July 10, 2023 Learn Next.js Server Actions and Mutations with Examples
June 15, 2023 Use Flask Framework to Build a RESTful API in Python
June 25, 2023 Integrate Python Flask Framework with Next.js and Deploy
June 25, 2023 Integrate FastAPI Framework with Next.js and Deploy

Categories
C# (2)
C++ (1)
CSS / SCSS (3)
Deno (8)
Golang (28)
JavaScript (4)
NextJs (23)
NodeJS (32)
Programming (19)
Python (17)
React (35)
Rust (23)
Svelte (1)
Vue (7)

report this ad

```
    Ok(ejson(response_json))
}

#[tokio::main]
async fn main() {
    if std::env::var_os("RUST_LOG").is_none() {
        std::env::set_var("RUST_LOG", "apiexinfo");
    }
    pretty_env_logger::init();

    let health_checker = warp::path!("api" / "healthchecker")
        .and(warp::get())
        .and_then(health_checker_handler);

    let routes = health_checker.with(warp::log("api"));

    println!("🚀 Server started successfully");
    warp::serve(routes).run(([0, 0, 0, 0], 8000)).await;
}
```

In the above snippets, we created a **health_checker_handler** route handler that Warp will call when a **GET** request is made to the `/api/healthchecker` endpoint.

Then, we evoked the **pretty_env_logger**'s `init()` function to log the HTTP request information in the terminal.

Then, we used the `warp::path!` filter to create the health checker route and added the **health_checker_handler** function to handle the request. Finally, we passed the `routes` filter to the `warp::serve()` function and called the `.run()` method to bind the specified port and address.

If you have a Node.js background, you may be familiar with Nodemon which restarts the server when required files change. In Rust, we have a similar package called `cargo-watch` which hot-reloads the server upon file changes.

Install the `cargo-watch` binary with this command:

```
$ cargo install cargo-watch
```

Then, we evoked the **pretty_env_logger**'s `init()` function to log the HTTP request information in the terminal.

Then, we used the `warp::path!` filter to create the health checker route and added the **health_checker_handler** function to handle the request. Finally, we passed the `routes` filter to the `warp::serve()` function and called the `.run()` method to bind the specified port and address.

If you have a Node.js background, you may be familiar with Nodemon which restarts the server when required files change. In Rust, we have a similar package called `cargo-watch` which hot-reloads the server upon file changes.

Install the `cargo-watch` binary with this command:

```
$ cargo install cargo-watch
```

Then, we evoked the **pretty_env_logger**'s `init()` function to log the HTTP request information in the terminal.

Then, we used the `warp::path!` filter to create the health checker route and added the **health_checker_handler** function to handle the request. Finally, we passed the `routes` filter to the `warp::serve()` function and called the `.run()` method to bind the specified port and address.

If you have a Node.js background, you may be familiar with Nodemon which restarts the server when required files change. In Rust, we have a similar package called `cargo-watch` which hot-reloads the server upon file changes.

Install the `cargo-watch` binary with this command:

```
$ cargo install cargo-watch
```

The in-memory database will be a vector of a `Todo` model. We'll use Rust's smart pointer called **Arc** along with a **Mutex** to share the in-memory database across the multiple route handlers.

Arc is a type of smart pointer that allows multiple ownership of the same value. It also allows for efficient copying of the pointer to ensure that the value remains allocated as long as any of the shared pointers to it exist. **Mutex** on the other hand will ensure that only one thread can access the in-memory database at a time to prevent concurrent access.

So, create a `model.rs` file in the **src** directory and add the following model definitions.

src/model.rs

```
use chrono::prelude::*;
use serde::{Deserialize, Serialize};
use std::sync::Arc;
use tokio::sync::Mutex;

#[allow(non_snake_case)]
#[derive(Debug, Deserialize, Serialize, Clone)]
pub struct Todo {
    pub id: Option<String>,
    pub title: String,
    pub content: String,
    pub completed: Option<bool>,
    pub createdAt: Option<DateTime<Utc>>,
    pub updatedAt: Option<DateTime<Utc>>,
}

pub type DB = Arc<Mutex<Vec<Todo>>>;

pub fn todo_db() -> DB {
    Arc::new(Mutex::new(Vec::new()))
}

#[derive(Debug, Deserialize)]
pub struct QueryOptions {
    pub page: Option<usize>,
    pub limit: Option<usize>,
}

#[allow(non_snake_case)]
#[derive(Debug, Deserialize, Serialize, Clone)]
pub struct UpdateTodoSchema {
    pub title: Option<String>,
    pub content: Option<String>,
    pub completed: Option<bool>,
}
```

Above, we defined the structure of the in-memory database and assigned it to the `DB` type. Then, we created a `todo_db()` function that will be evoked to initialize the data store with an empty vector.

The **QueryOptions** and **UpdateTodoSchema** structs will be used to extract the query parameters and payload from the request. Both structs are supposed to be in a `schema.rs` file but since they are only two, you can leave them in the **model.rs** file.

Create the API Response Structs

Now let's create structs that implement the `serde::Serialize` trait to help us serialize Structs into JSON formats before sending them to the client in the JSON response.

src/response.rs

```
use crate::model::Todo;
use serde::Serialize;

#[derive(Serialize)]
pub struct GenericResponse {
    pub status: String,
    pub message: String,
}

#[derive(Serialize, Debug)]
pub struct TodoData {
    pub todo: Todo,
}

#[derive(Serialize, Debug)]
pub struct SingleTodoResponse {
    pub status: String,
    pub data: TodoData,
}

#[derive(Serialize, Debug)]
pub struct TodoListResponse {
    pub status: String,
    pub results: usize,
    pub todos: Vec<Todo>,
}
```

src/response.rs

```
use crate::model::Todo;
use serde::Serialize;

#[derive(Serialize)]
pub struct GenericResponse {
    pub status: String,
    pub message: String,
}

#[derive(Serialize, Debug)]
pub struct TodoData {
    pub todo: Todo,
}

#[derive(Serialize, Debug)]
pub struct SingleTodoResponse {
    pub status: String,
    pub data: TodoData,
}

#[derive(Serialize, Debug)]
pub struct TodoListResponse {
    pub status: String,
    pub results: usize,
    pub todos: Vec<Todo>,
}
```

src/response.rs

```
use crate::model::Todo;
use serde::Serialize;

#[derive(Serialize)]
pub struct GenericResponse {
    pub status: String,
    pub message: String,
}

#[derive(Serialize, Debug)]
pub struct TodoData {
    pub todo: Todo,
}

#[derive(Serialize, Debug)]
pub struct SingleTodoResponse {
    pub status: String,
    pub data: TodoData,
}

#[derive(Serialize, Debug)]
pub struct TodoListResponse {
    pub status: String,
    pub results: usize,
    pub todos: Vec<Todo>,
}
```

src/response.rs

```
use crate::model::Todo;
use serde::Serialize;

#[derive(Serialize)]
pub struct GenericResponse {
    pub status: String,
    pub message: String,
}

#[derive(Serialize, Debug)]
pub struct TodoData {
    pub todo: Todo,
}

#[derive(Serialize, Debug)]
pub struct SingleTodoResponse {
    pub status: String,
    pub data: TodoData,
}

#[derive(Serialize, Debug)]
pub struct TodoListResponse {
    pub status: String,
```

Rust	(23)
Svelte	(1)
Vue	(7)

report this ad

```
    pub results: usize,
    pub todos: Vec<Todo>,
}

    });
    OK(json(response_json))
}
```

We created a **response_json** struct from the **GenericResponse** type and passed it to Warp's `reply::json()` method. The `reply::json()` method will convert the struct into a JSON object before the **OK()** method will be evoked to send the JSON object to the client.

Find All Records Route Handler

The **todos_list_handler** will be evoked by Warp when a **GET** request is made to the `/api/todos?page=1&limit=10` endpoint. This function will take a struct that will be populated with query parameters by the `warp::query::<T>()` filter and a reference to the data store.

src/handler.rs

```
pub async fn todos_list_handler(opts: QueryOptions, db: DB) -> WebResult<impl
    let todos = db.lock().await;

    let limit = opts.limit.unwrap_or(10);
    let offset = (opts.page.unwrap_or(1) - 1) * limit;

    let todos: Vec<Todo> = todos.clone().into_iter().skip(offset).take(limit)

    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    OK(json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    Ok(Json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method

```
    let json_response = TodoListResponse {
        status: "success".to_string(),
        results: todos.len(),
        todos,
    };
    Ok(Json(&json_response))
```

To access the vector of Todos in the data store, we called the `.lock()` method on the Mutex which returns a `LockGuard` and used the `.await` method to wait for the lock to be available before acquiring it.

Next, we created two statements to calculate the **limit** and **skip** values and assigned them to the **limit** and **offset** variables respectively. Then, we called the `.into_iter()` method on the vector to return an iterator that takes ownership of the vector, yielding each of the Todo structs in order.

After that, the `.skip()` method will be called with the **offset** value to return an iterator that skips the specified number of elements before the `.take()` method At this point, we've created all the CRUD route handlers and the in-memory database. Now let's create Warp filters to evoke the route handlers and add a CORS filter middleware that will configure the server to accept cross-origin requests.

src/main.rs

```
mod handler;
mod model;
mod response;

use model::{QueryOptions, DB};
use warp::{http::Method, Filter, Rejection};

type WebResult<T> = std::result::Result<T, Rejection>;

#[tokio::main]
async fn main() {
    if std::env::var_os("RUST_LOG").is_none() {
        std::env::set_var("RUST_LOG", "api=info");
    }
    pretty_env_logger::init();

    let db = model::todo_db();

    let todo_router = warp::path("api" / "todos");
    let todo_router_id = warp::path("api" / "todos" / String);

    let health_checker = warp::path("api" / "healthchecker")
        .and(warp::get())
        .and_then(handler::health_checker_handler);

    let cors = warp::cors()
        .allow_methods(&[Method::GET, Method::POST, Method::PATCH, Method::
        .allow_origins(vec!["http://localhost:3000/", "http://localhost:800
        .allow_headers(vec!["content-type"])
        .allow_credentials(true);

    let todo_routes = todo_router
        .and(warp::post())
        .and(warp::body::json())
        .and(with_db(db.clone()))
```

In the above snippet, we defined a **todo_router** filter to match the `/api/todos` path and a **todo_router_id** filter to match the `/api/todos/{id}` path.

Then, we created a **todo_routes** filter from the **todo_router** filter to handle **POST** and **GET** requests. Also, we defined a **todo_routes_id** filter from the **todo_router_id** filter to handle **GET**, **PATCH**, and **DELETE** requests.

Finally, we created a **routes** filter from the **CORS**, **request logger**, **todo_routes_id**, **todo_routes**, and **health_checker** filters.

We are now ready to start the Warp HTTP server again. Run the command below to start the HTTP server on port **8000**.

```
cargo watch -q -c -w src/ -x run
```

Test the API Endpoints

Now that we've completed the CRUD API and the Warp HTTP server is listening on port **8000**, let's open an API testing software like Postman, Insomnia, or Thunder Client VS Code extension to test the endpoints.

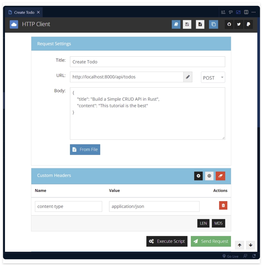
In my case, I'll be using the **HTTP Client** VS Code extension because it has a simple and intuitive interface.

Add New Record

To add a new record to the in-memory database, add the JSON object below to the request body and make a **POST** request to the `/api/todos` endpoint.

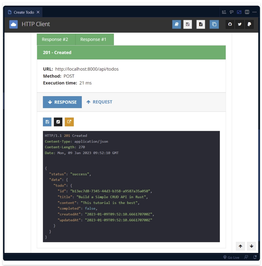
```
{
  "title": "Build a Simple CRUD API in Rust",
  "content": "This tutorial is the best"
}
```

If you intend to use the HTTP Client extension, add a content-type of `application/json` to the "**Custom Headers**" section before making the request to the Rust API.



Once the Warp HTTP server receives the request, the Serde crate will be called to deserialize the JSON payload into a Rust data structure before the `create_todo_handler` function will be evoked to add the new record to the in-memory database.

The **create_todo_handler** will then return the newly-created record in the JSON response to the client.

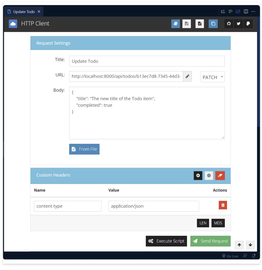


Update a Record

To update single and multiple fields of a record, add the edited JSON object to the request body and make a **PATCH** request to the `/api/todos/{id}` endpoint.

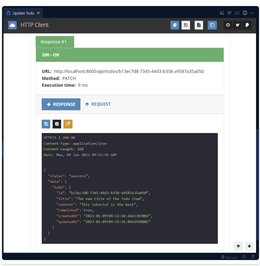
```
{
  "title": "The new title of the Todo item",
  "completed": true
}
```

Below is an image illustrating the information that must be provided before making the request to the Rust API.



The Rust API will then use the Serde crate to deserialize the JSON object and call the `edit_todo_handler` function to update the fields of the record that matches the ID provided in the URL parameter.

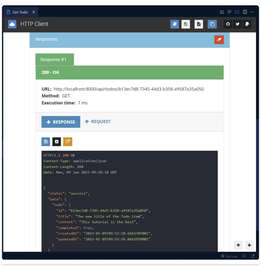
After the update is complete, the **edit_todo_handler** function will return the newly-updated record to the client in the JSON response.



Get a Record

To retrieve a single record from the in-memory database, add the ID of the record to the URL parameter and make a **GET** request to the `/api/todos/{id}` endpoint.

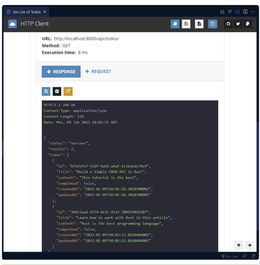
Warp will then call the `get_todo_handler` function to find the record that matches the ID in the in-memory database and return the found record to the client.



Get All Records

To retrieve a selected or paginated list of records from the Rust API, add the **limit** and **page** query parameters to the request URL and make a **GET** request to the `/api/todos?page=1&limit=10` endpoint.

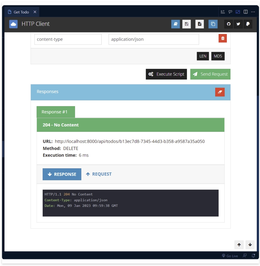
If the **page** and **limit** query parameters are absent in the URL, the Warp server will only return **10** records in the JSON response.



Delete a Record

To delete a record from the in-memory database, append the ID of the record to the URL parameter and make a **DELETE** request to the `/api/todos/{id}` endpoint.

Warp will then call the `delete_todo_handler` function to find and delete the record that matches the ID from the in-memory database.



Conclusion

What a ride! In this article, you learned how to create a simple API using Warp and Tokio. We covered how to build endpoints to handle the basic CRUD functionalities and store data in an in-memory database.

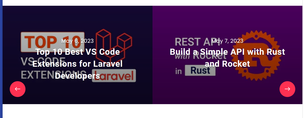
You can find the complete Rust CRUD API source code on **GitHub**.

Download Source Code

Share Article:

https://codeveweb.com/build-a-simple-api-in-rust/

Tags: Rust



Leave a Reply

Comment

Name

Email

Save my name, email, and website in this browser for the next time I comment.

This site is protected by reCAPTCHA and the Google **Privacy Policy** and **Terms of Service** apply.

Post Comment

This site uses Akismet to reduce spam. **Learn how your comment data is processed.**

report this ad

Tag Cloud

.NET C# create website Deno developers Django DOM DOM methods fastapi Golang Golang API graphql gRPC API html template JavaScript JavaScript DOM landing page Material-UI Material UI MUI NestJs nodejs api Prisma programmers programming language programming terms pure nodejs api python python development React React error React Hook Form ReactJs React Query

STK Query | Rust | Svelte | TypeScript | vscode | vscode extension | vs code extesions | vscode tips and tricks | VueJs | website | Yewrs

Copyright @ 2023 CodevoWeb

WordPress Theme by EstudioPatagon

report this ad