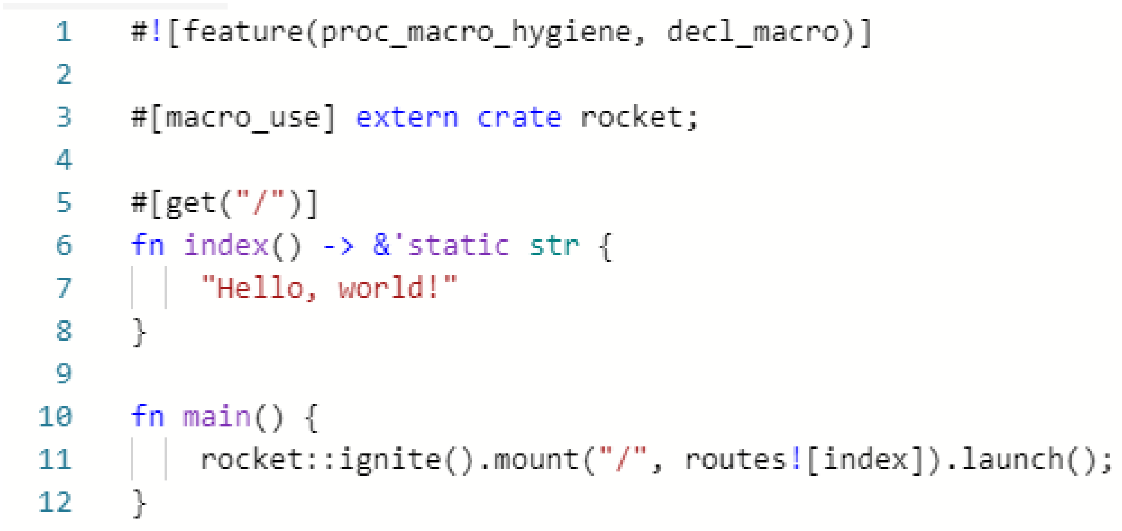
Rocket vs Actix Web Frameworks

Rocket Web Framework

Rocket is a web framework for Rust. Rocket aims to be fast, easy, and flexible and offering guaranteed safety and security.

Rocket routes the incoming requests to the particular request handler using declared routes of an application. Routes can declare by using Rocket's route attributes. The attribute describes the matching route of request. The attribute is placed on top of a function that is the request handler for that route.

The example code is given below



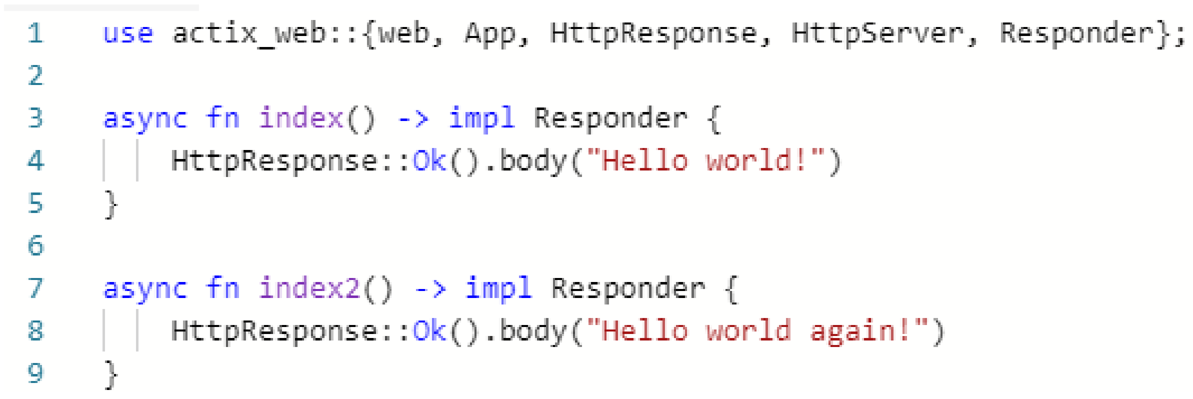
```rust
#![feature(proc_macro_hygiene, decl_macro)]

#[macro_use] extern crate rocket;

#[get("/")]
fn index() -> &'static str {
    "Hello, world!"
}

fn main() {
    rocket::ignite().mount("/", routes![index]).launch();
}
```

Actix Web Framework

Actix is a web framework for Rust. It is a small and pragmatic framework. For Rust programmer it will probably a home quickly, but for another programmer it is easy to pick up.

An application developed with actix-web will expose an HTTP server contained within a native executable. It can either putted behind another HTTP server like nginx or serve it up as-is. Even in the complete absence of another HTTP server actix-web is powerful enough to provide HTTP 1 and HTTP 2 support as well as SSL/TLS. This makes it useful for building small services ready for distribution.

The example code is given below

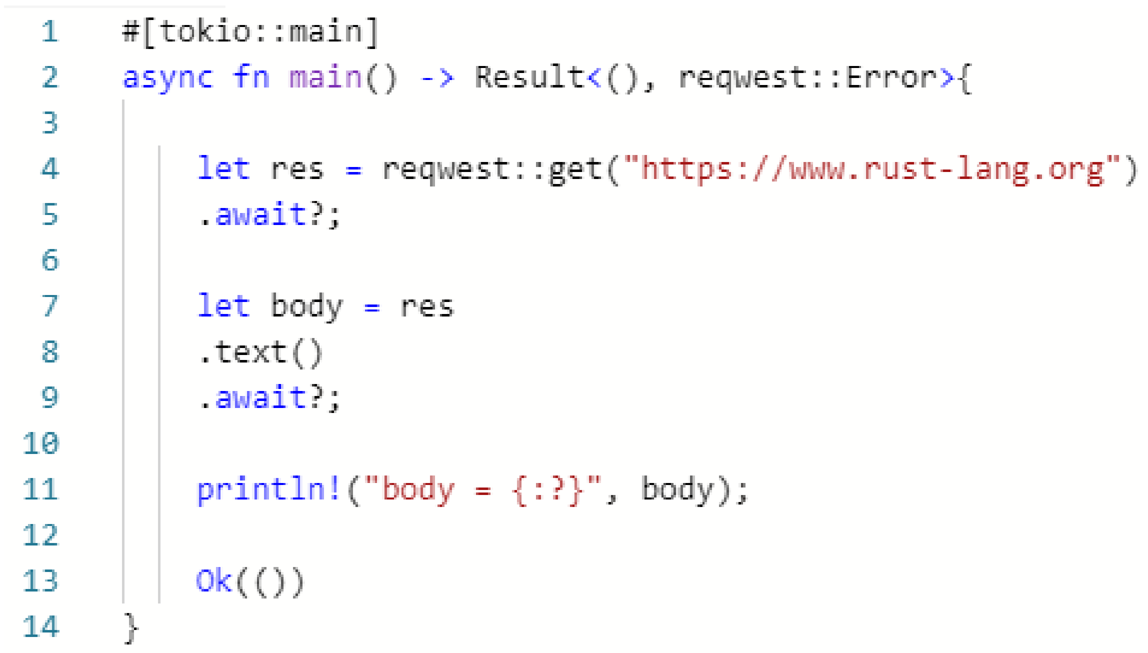


```rust
use actix_web::{web, App, HttpResponse, HttpServer, Responder};

async fn index() -> impl Responder {
    HttpResponse::Ok().body("Hello world!")
}

async fn index2() -> impl Responder {
    HttpResponse::Ok().body("Hello world again!")
}
```

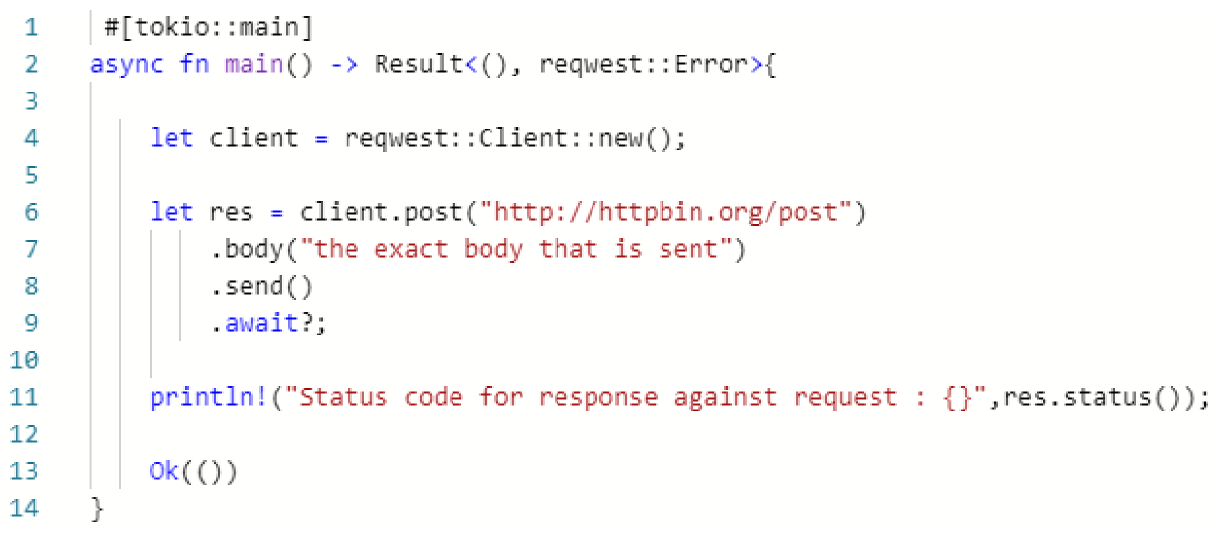
Comparison between Rocket and Actix Web Framework

1. Request:

In Rocket, route attribute can be any one of get, put, post, delete, head, patch, or options, each corresponding to the HTTP method to match against.

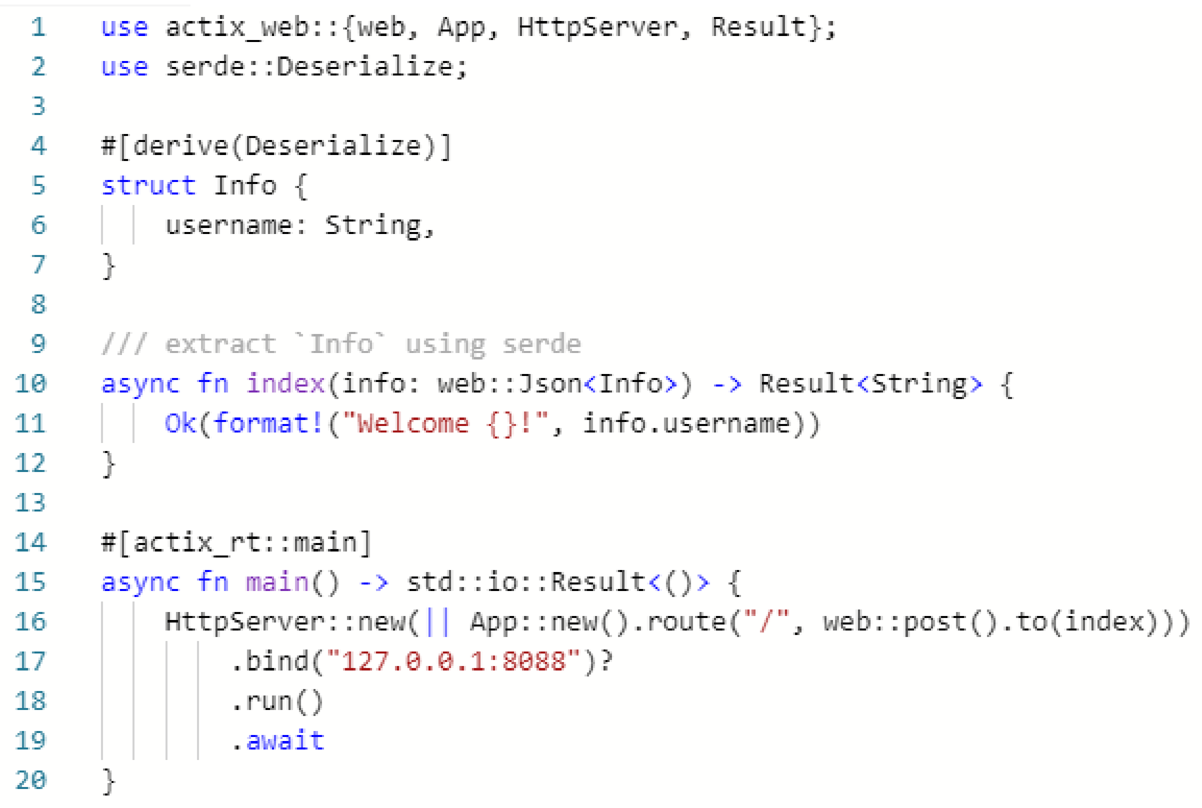


```rust
#[tokio::main]
async fn main() -> Result<(), reqwest::Error>{

    let res = reqwest::get("https://www.rust-lang.org")
    .await?;

    let body = res
    .text()
    .await?;

    println!("body = {:?}", body);

    Ok(())
}
```



```rust
#[tokio::main]
async fn main() -> Result<(), reqwest::Error>{

    let client = reqwest::Client::new();

    let res = client.post("http://httpbin.org/post")
        .body("the exact body that is sent")
        .send()
        .await?;

    println!("Status code for response against request : {}",res.status());

    Ok(())
}
```

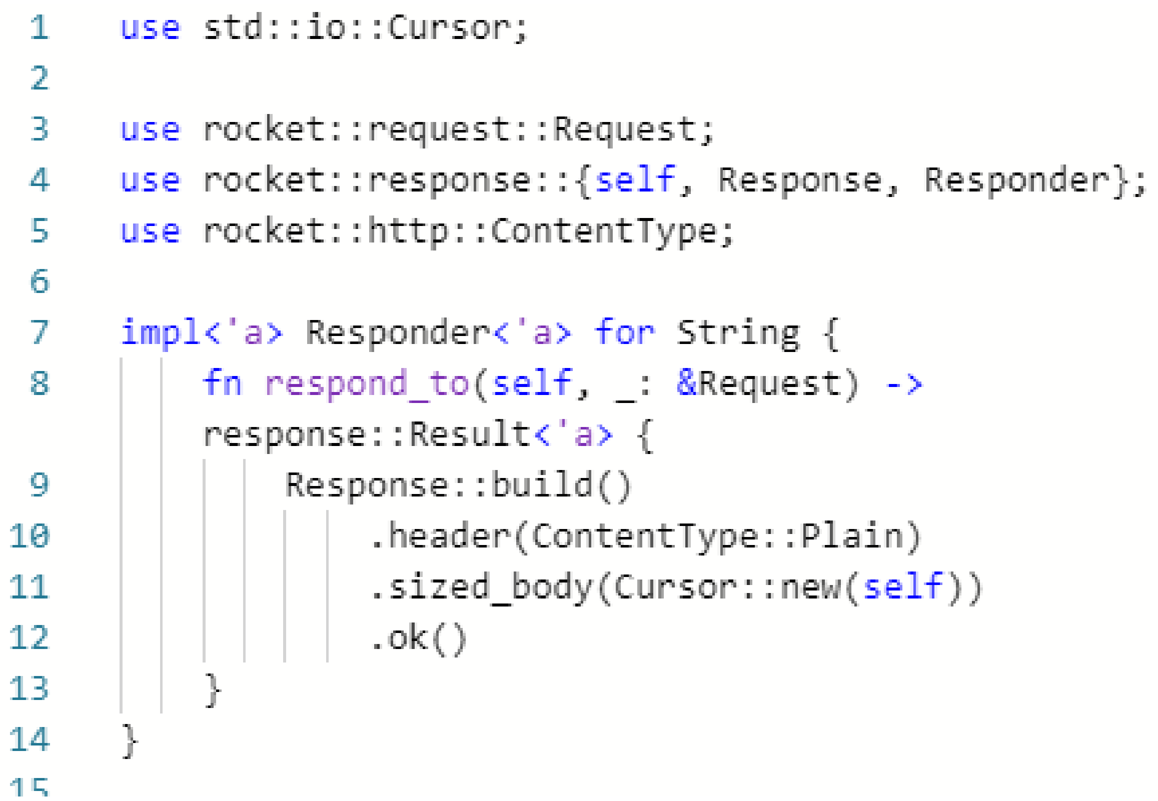
Whereas, In Actix to use Json extractor. First, you define a handler function that accepts Json<T> as a parameter, then, you use the .to() method for registering this handler. It is also possible to accept arbitrary valid json object by using serde_json::Value as a type T.



```rust
use actix_web::{web, App, HttpServer, Result};
use serde::Deserialize;

#[derive(Deserialize)]
struct Info {
    username: String,
}

/// extract `Info` using serde
async fn index(info: web::Json<Info>) -> Result<String> {
    Ok(format!("Welcome {}!", info.username))
}

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().route("/", web::post().to(index)))
        .bind("127.0.0.1:8088")?
        .run()
        .await
}
```

2. **Response**:

In Rocket, Types that implement Responder know how to generate a Response from their values. A Response includes an HTTP status, headers, and body. The body may either be fixed-sized or streaming. The given Responder implementation decides which to use. For instance, String uses a fixed-sized body, while File uses a streamed response. Responders may dynamically adjust their responses according to the incoming Request they are responding to.

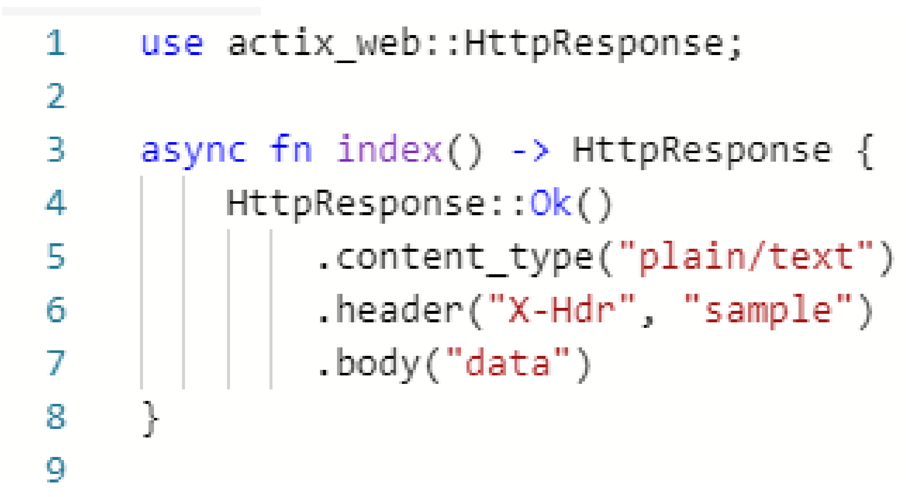


```rust
use std::io::Cursor;

use rocket::request::Request;
use rocket::response::{self, Response, Responder};
use rocket::http::ContentType;

impl<'a> Responder<'a> for String {
    fn respond_to(self, _: &Request) -> response::Result<'a> {
        Response::build()
            .header(ContentType::Plain)
            .sized_body(Cursor::new(self))
            .ok()
    }
}
```

On the other hand, In Actix A builder-like pattern is used to construct an instance of HttpResponse. HttpResponse provides several methods that return a HttpResponseBuilder instance, which implements various convenience methods for building responses.

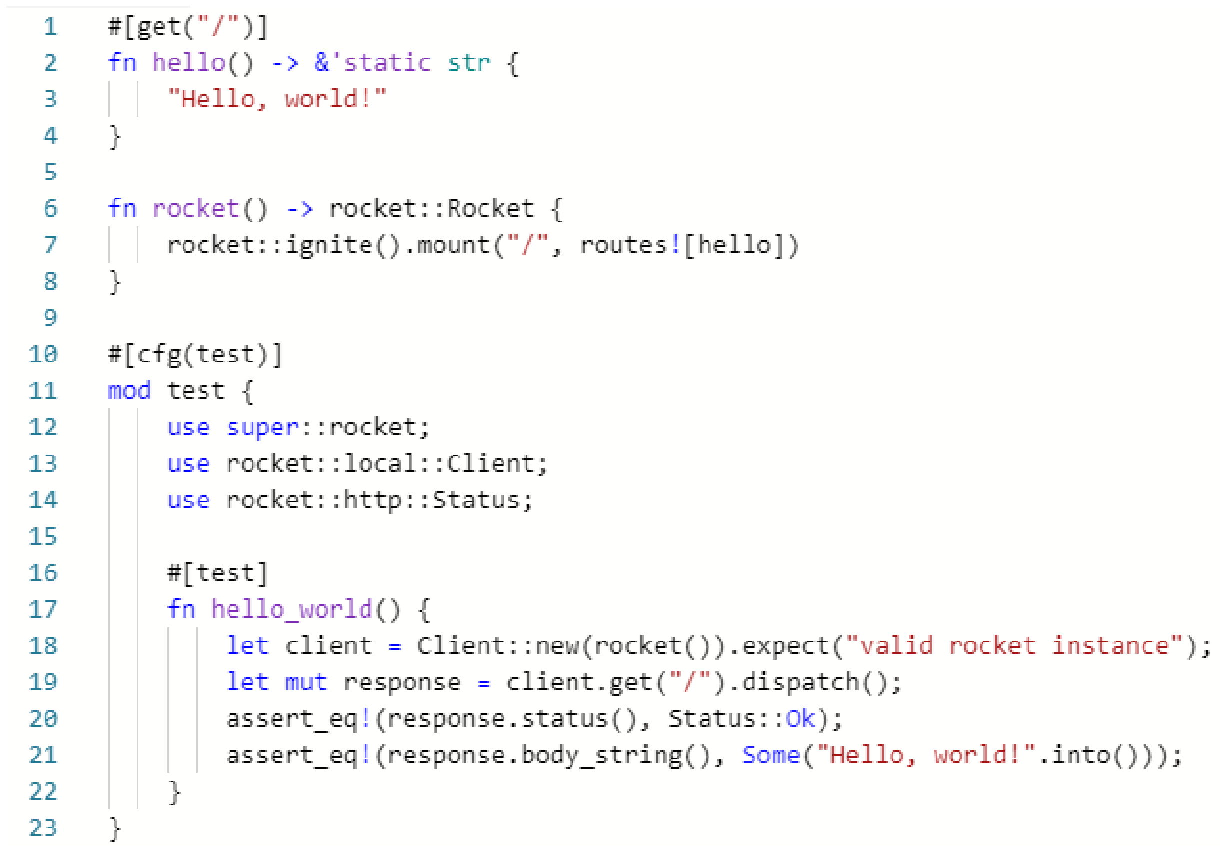
Check the documentation for type descriptions.

The methods .body, .finish, and .json finalize response creation and return a constructed HttpResponse instance. If this methods is called on the same builder instance multiple times, the builder will panic.

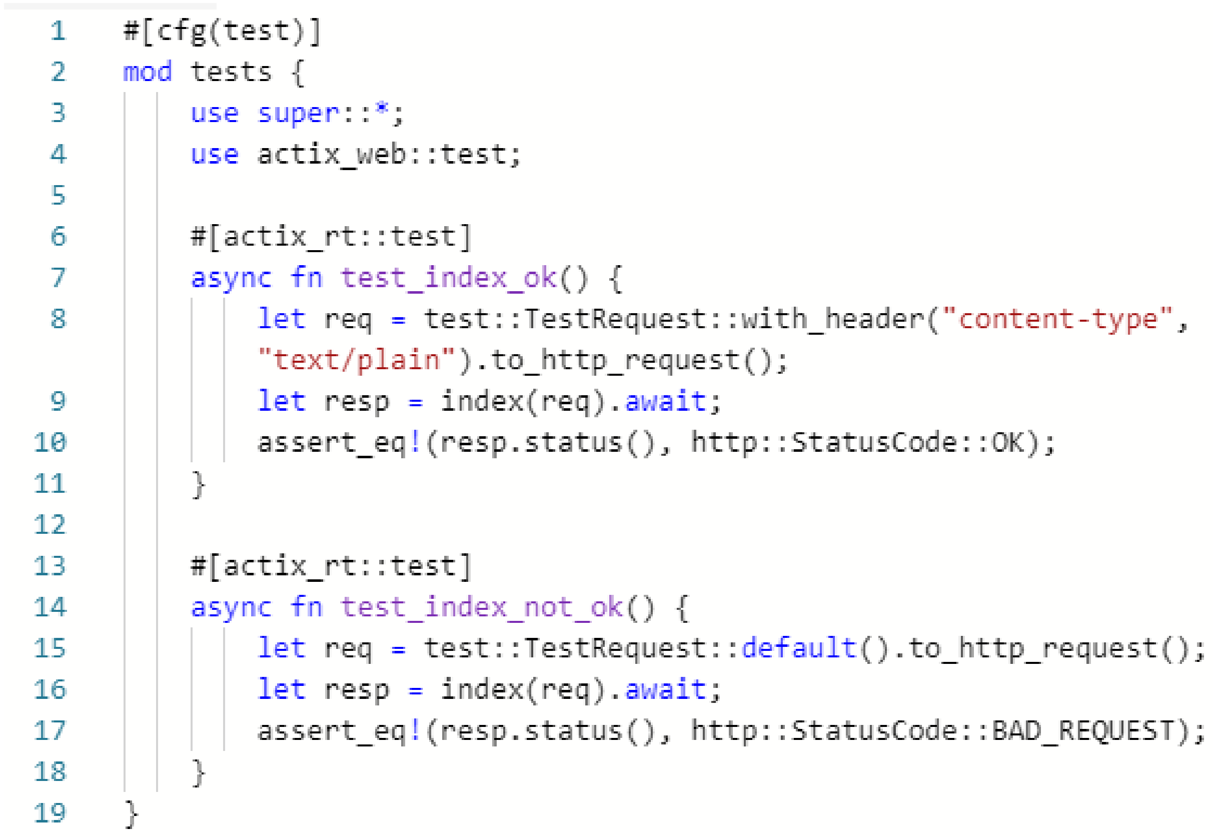

```rust
use actix_web::HttpResponse;

async fn index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("plain/text")
        .header("X-Hdr", "sample")
        .body("data")
}

```

3. Testing

In Rocket, applications are tested by dispatching requests to a local instance of Rocket. The local module contains all of the structures necessary to do so. In particular, it contains a Client structure that is used to create LocalRequest structures that can be dispatched against a given Rocket instance


```rust
#[get("/")]
fn hello() -> &'static str {
    "Hello, world!"
}

fn rocket() -> rocket::Rocket {
    rocket::ignite().mount("/", routes![hello])
}

#[cfg(test)]
mod test {
    use super::rocket;
    use rocket::local::Client;
    use rocket::http::Status;

    #[test]
    fn hello_world() {
        let client = Client::new(rocket()).expect("valid rocket instance");
        let mut response = client.get("/").dispatch();
        assert_eq!(response.status(), Status::Ok);
        assert_eq!(response.body_string(), Some("Hello, world!".into()));
    }
}
```

While in Actix, For unit testing, actix-web provides a request builder type. TestRequest implements a builder-like pattern. You can generate a HttpRequest instance with to_http_request() and call your handler with it.



```rust
#[cfg(test)]
mod tests {
    use super::*;
    use actix_web::test;

    #[actix_rt::test]
    async fn test_index_ok() {
        let req = test::TestRequest::with_header("content-type", "text/plain").to_http_request();
        let resp = index(req).await;
        assert_eq!(resp.status(), http::StatusCode::OK);
    }

    #[actix_rt::test]
    async fn test_index_not_ok() {
        let req = test::TestRequest::default().to_http_request();
        let resp = index(req).await;
        assert_eq!(resp.status(), http::StatusCode::BAD_REQUEST);
    }
}
```