Lab 8: Node, Angular, and Server-Side Coding in Rust

Please note: No demos or hand-ins are required for this lab.

Learning Objectives:

- Introducing Node.js – The web server/platform
- Introducing AngularJS – The front-end framework
- Using Rocket to Build a Server-side Application with Rust
- Server-side Development with Actix-web and Rust

Introducing Node.js – The web server/platform:

- Node.js is a software platform that allows you to create your own web server and build web applications on top of it.
- Node.js isn't itself a web server, nor is it a language. It contains a built-in HTTP server library, meaning that you don't need to run a separate web server program such as Apache Tomcat or Internet Information Services (IIS).
- When coded correctly, Node.js is extremely fast and makes very efficient use of system resources. This enables a Node.js application to serve more users on fewer server resources than most of the other mainstream server technologies.
- A Node.js server is *single-threaded* and works differently than the multithreaded systems. Rather than giving each visitor a unique thread and a separate silo of resources, every visitor joins the same thread. A visitor and thread only interact when needed, when the visitor is requesting something, or the thread is responding to a request.
- All APIs of Node.js library are *asynchronous*, that is, they are all non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of events of Node.js helps the server to get a response from the previous API call.
- Node.js applications never buffer any data. These applications simply output the data in chunks.
- *npm* is a package manager for Node.js that gets installed when you install Node.js. npm gives you the ability to download Node.js modules or "packages" to extend the functionality of your application.

Setting up the Environment:

- To install Node.js and npm
    - On a Windows computer: use the Windows Installer
    - On a Linux computer: Open your terminal or press Ctrl + Alt + T and type

```
$ sudo apt install nodejs
$ sudo apt install npm
```

    - On a Mac-book: use the macOS Installer
- To check the installed Node.js and npm versions, type:

```
$ node --version
$ npm --version
```

The Node.js REPL Terminal

- REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.
- Node.js comes bundled with a REPL environment. It performs the following tasks
  - **Read:** Reads a user's input, parses the input into JavaScript data-structure, and stores it in memory.
  - **Eval:** Takes and evaluates the data structure.
  - **Print:** Prints the result.
  - **Loop:** Loops the above command until the user presses ctrl-c twice.
- The REPL feature of Node is very useful in experimenting with Node.js code and to debug JavaScript code.
- REPL can be started by simply running node on terminal or console without any arguments as follows:

```
$ node
```

- With this command you should see the version number along with the Node.js prompt on your terminal as:

```
Welcome to Node.js v13.8.0.
Type ".help" for more information.
>
```

- Here you will be able to test some basic node.js syntax as:

```
> var x = 0
undefined
> do {
   ... x++;
   ... console.log("x: " + x);
   ... }
while ( x < 5 );
x: 1
x: 2
x: 3
x: 4
x: 5
undefined
>
```

- Apart from the basic commands, REPL also allows us to perform basic file management operations such as: **.save filename** to save the current Node REPL session to a file and **.load filename** to load file content in current Node REPL session.

Event-Driven Programming with Node.js

- Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**.
- Every API of Node.js is asynchronous and being single-threaded, they use async function calls to avoid blocking.
- Node uses the observer pattern.

- As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.
- Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern.
- The functions that listen act as Observers. Whenever an event gets called (fired), its listener function starts executing. Node.js has multiple in-built events available through the events module and the EventEmitter class which are used to bind events and event-listeners.
- For example:

```
var events = require('events');

var eventEmitter = new events.EventEmitter();

var connectHandler = function connected() {
   console.log('connection succesful.');

   eventEmitter.emit('data_received');
}

eventEmitter.on('connection', connectHandler);

eventEmitter.on('data_received', function() {
   console.log('data received succesfully.');
});

eventEmitter.emit('connection');

console.log("Program Ended.");
```

Import events module

Create an eventEmitter object

Create an event handler

Fire the data_received event

Bind the connection event with the

Fire the connection event

- To test the above code, create your node project as:

```
$ mkdir testNodeProject
$ cd testNodeProject
```

- To initialize a node project inside the directory and to create the package.json file for your project, in your terminal type:

```
$ npm init
```

- Next create a new file in your project directory and name it main.js and place the example code for event handling with node in the file.
- Now upon checking the output of the code using:

```
$ node main.js
```

- The following result will be presented:

```
connection successful.
data received successfully.
Program Ended.
```

Creating a Web Server using Node.js

- A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients.
- Node.js provides an http module which can be used to create an HTTP client of a server. Following is the bare minimum structure of the HTTP server which listens at 8000 port.
- For example, create a file inside your project call it server.js and type in the following:

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);

         // HTTP Status: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      } else {
         //Page found
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});

         // Write the content of the file to response body
         response.write(data.toString());
      }

      // Send the response body
      response.end();
   });
}).listen(8000);

// Console will print the message
console.log('Server running at http://127.0.0.1:8000/');
```

- Next let's create the following html file named *index.html* in the same directory where you created server.js and type in the following:

```
<html>
```

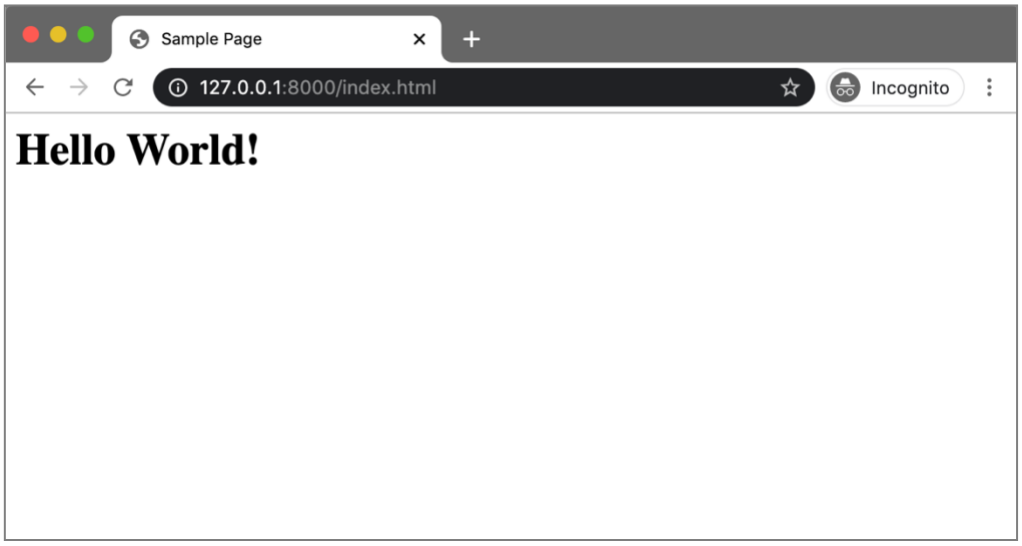
```
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      <h1>Hello World!</h1>
   </body>
</html>
```

- Now, in your terminal type in the following to execute the above code:

```
$ node server.js
```

- Open a browser and direct to http://127.0.0.1:8000/index.html and you should see the following webpage:



- On the server side (i.e., in your terminal) you should see the following output:

```
Server running at http://127.0.0.1:8000/
Request for /index.html received.
```

Node.js Modules

- Like any other programming languages, Node.js modules are libraries that include a set of functions you want to include in your application
- Node.js has a set of built-in modules which you can use without any further installation
- To include a module, use the require() function with the name of the module:

```
var http = require('http');
```

- Now your application has access to the HTTP module, and is able to create a server:

```
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```

Create Your Own Modules

- You can create your own modules, and easily include them in your applications. The following example creates a module that returns a date and time object:

```
exports.myDateTime = function ()                                    {
  return Date();
};
```

- We can save the code above in a file called "dateTest.js" and include and use the module in any of our Node.js files as:

```
var http = require('http');
var dt = require('./dateTest');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("The date and time are currently: " + dt.myDateTime());
  res.end();
}).listen(8080);
```

- To run the above code, save the code in a file named "dateTestModule.js" and type:

```
$ node dateTestModule
```

Exercise 1: Create Your Own Module

- Your module will contain a collection of colors in an array and provide a function to get one at random. You will use the Node.js built-in exports property to make the function and array available to external programs.
- First, you'll begin by deciding what data about colors you will store in your module. Every color will be an object that contains a name property that humans can easily identify, and a code property that is a string containing an HTML color code. HTML color codes are six-digit hexadecimal numbers that allow you to change the color of elements on a web page. You can learn more about HTML color codes by reading this here.
- You will then decide what colors you want to support in your module. Your module will contain an array called allColors that will contain six colors. Your module will also include a function called getRandomColor() that will randomly select a color from your array and return it.
- Using the following Color class, an example of adding a color in the allColors array would be:

```
class Color {
  constructor(name, code) {
    this.name = name;
    this.code = code;
  }
}
```

```
const allColors = [new Color('White', ' #FFFFFF')];
```

- Next add in the getRandomColor function that returns a random color from your all colors array and export the module. You can use the following definition for the getRandomColor() function:

```
exports.getRandomColor = () => {
  <<<<<< add your code here >>>>>>>>>
}

exports.allColors = allColors;
```

- Test your Module with the REPL as follows:

```
$ node
> colors = require('./<Your_module_filename_here>);
> colors.getRandomColor();
```

- Now test your module as a dependency in another javascript file in the same directory.

Introducing AngularJS – The front-end framework:

- AngularJS is a JavaScript framework for working with data directly in the front end.
- AngularJS enables to move some or all of this processing and logic out to the browser, sometimes leaving the server just passing data from the database.
- jQuery versus AngularJS:
    - jQuery is generally added to a page to provide interactivity, after the HTML has been sent to the browser and the Document Object Model (DOM) has completely loaded. AngularJS comes in a step earlier and helps put together the HTML based on the data provided.
    - jQuery is a library, and as such has a collection of features that you can use as you wish. AngularJS is what is known as an opinionated framework. This means that it forces its opinion on you as to how it needs to be used.
- AngularJS lets you extend HTML vocabulary for your application.
- AngularJS is fully extensible and works well with other libraries. Every feature can be modified or replaced to suit your unique development workflow and feature needs.

Installing AngularJS:

- Angular doesn't take much installation because it's really just a library file that you need to download and place in the correct spot in your folder structure. You can download AngularJS from its homepage at http://angularjs.org/.

Two-way data binding with AngularJS: Working with data in a page

- To understand two-way data binding let's start with a look at the traditional approach of one-way data binding. One-way data binding is what you're aiming for when looking at using

Node.js, Express, and MongoDB. Node.js gets the data from MongoDB, and Express then uses a template to compile this data into HTML that's then delivered to the server.

- Two-way data binding is different. First, the template and data are sent independently to the browser. The browser itself compiles the template into the view and the data into a model. The real difference is that the view is "live." The view is bound to the model, so if the model changes, the view changes instantly. On top of this, if the view changes then the model also changes.
- With Angular you can actually do something like this without coding any JavaScript at all!
- For example, say we have a very simple HTML page, where we have an input box and somewhere to display the input. In the following code snippet we have an input field and an <h1> tag, and we want to take whatever text is entered into the input box and immediately display it after the Hello in the h1 as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
      <title>Angular binding test</title>
</head>
<body>
      Type Something: <input />
      <h1>Hello </h1>
</body>
</html>
```

- Next, assuming we download and reference it locally we need to add it to the page. Also to tell AngularJS that this page is an AngularJS application. To do this, we can add a simple attribute, *ng-app* to the opening <html> tag.
- Now, to take the input from the form and show it in HTML without writing any JavaScript, we just need to bind an AngularJS model to the input and the output, and Angular will do the rest.
- The only requirement is, both bindings will need to reference the same name, so that AngularJS knows that they share the same model. This is done as follows:

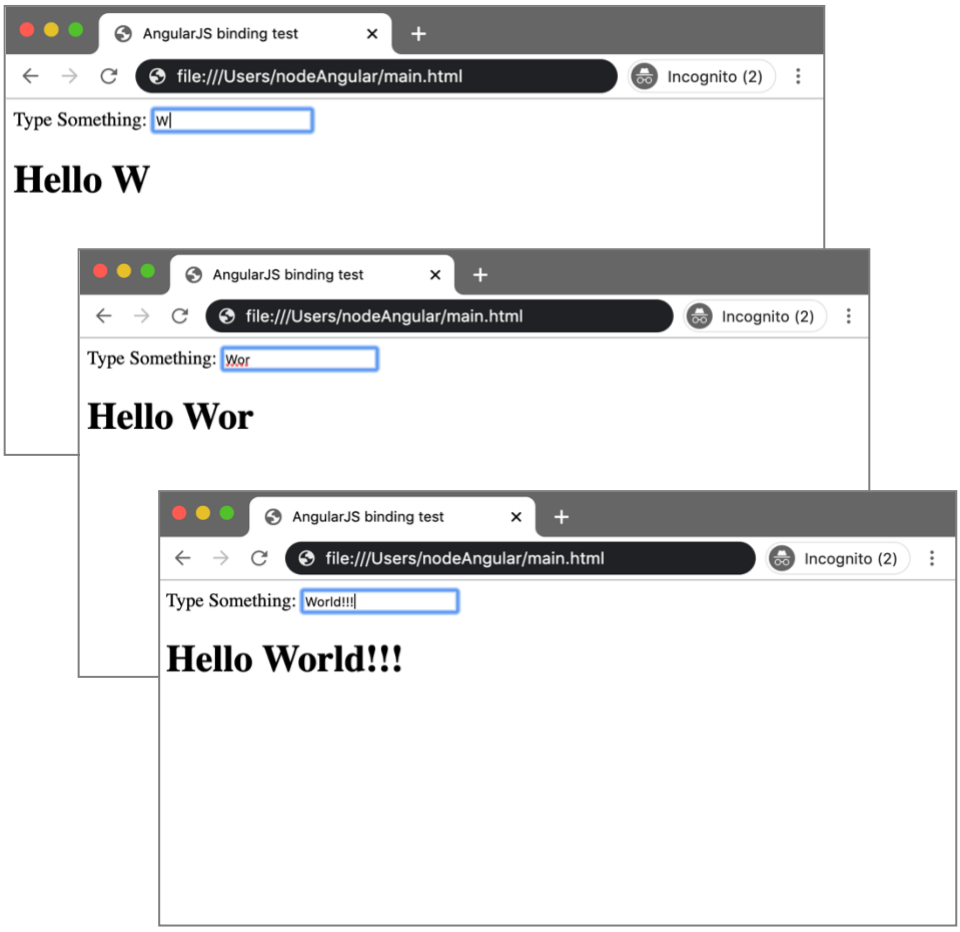
```
<input ng-model="myInput" />
<h1>Hello {{ myInput }}</h1>
```

- Putting it all together, we will end up with the following HTML code:

```
<!DOCTYPE html>
<html ng-app>
<head>
<script src="angular.min.js"></script>
      <meta charset="utf-8">
      <title>AngularJS binding test</title>
</head>
<body>
      Type Something: <input ng-model="myInput" />
      <h1>Hello {{ myInput }}</h1>
</body>
```

```
</html>
```

- Now when we execute the html and test the outcome of the code, it looks as follows:



Exercise 2: Two-way Data Binding with AngularJS

- Using the above example, create a simple HTML page that shows two way binding.
- Improvise the example by adding multiple text boxes for entering the first and last names of individuals. Upon typing in the names the page should show a message “Hello <first-name> <last-name>”.
- Check out this Github repository with 50 AngularJS examples. Execute and understand some of the examples (No. 8, 22, and 25) that perform data binding, and searching, and sorting using AngularJS.

Using Rocket to Build a Server-side Application with Rust:

- Rocket is a web framework for Rust (nightly) with a focus on ease-of-use and speed.
- Rocket's design is centered around three core philosophies:
    - *Security, correctness, and developer experience are paramount*: Rocket is easy to use while taking great measures to ensure that your application is secure and correct without cognitive overhead.
    - *All request handling information should be typed and self-contained*: Rocket does the convertion of strings to native types for you with zero programming overhead. What's more, Rocket's request handling is self-contained with zero global state: handlers are regular functions with regular arguments.
    - *Decisions should not be forced*: Templates, serialization, sessions, and just about everything else are all pluggable, optional components. While Rocket has official support and libraries for each of these, they are completely optional and swappable.

Anatomy of a Rocket Application:

- Rocket's main task is to route incoming requests to the appropriate request handler using your application's declared routes. Routes are declared using Rocket's route attributes. The attribute describes the requests that match the route. The attribute is placed on top of a function that is the request handler for that route.
- A simple example of routing using Rocket:

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

- Rocket allows you to interpret segments of a request path dynamically. To illustrate, let's use the following route:

```
#[get("/hello/<name>/<age>")]
fn hello(name: String, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}
```

- The hello route above matches two dynamic path segments declared inside brackets in the path: <name> and <age>. Dynamic means that the segment can have any value.
- Each dynamic parameter (name and age) must have a type, here &str and u8, respectively. Rocket will attempt to parse the string in the parameter's position in the path into that type. The route will only be called if parsing succeeds. To parse the string, Rocket uses the FromParam trait, which you can implement for your own types!
- Requested (body) data is handled in a special way in Rocket: via the FromData trait. Any type that implements FromData can be derived from incoming body data. To tell Rocket that you're expecting requested body data, the data route argument is used with the name of the parameter in the request handler:

```
#[post("/login", data = "<user_form>")]
fn login(user_form: Form<UserLogin>) -> String {
```

```
    format!("Hello, {}!", user_form.name)
}
```

- The login route above says that it expects data of type Form<UserLogin> in the user_form parameter. The Form type is a built-in Rocket type that knows how to parse web forms into structures. Rocket will automatically attempt to parse the request body into the Form and call the login handler if parsing succeeds.
- In addition to dynamic path and data parameters, request handlers can also contain a third type of parameter: request guards. Request guards aren't declared in the route attribute, and any number of them can appear in the request handler signature.
- Request guards protect the handler from running unless some set of conditions are met by the incoming request metadata. For instance, if you are writing an API that requires sensitive calls to be accompanied by an API key in the request header, Rocket can protect those calls via a custom ApiKey request guard:

```
#[get("/sensitive")]
fn sensitive(key: ApiKey) { ... }
```

- ApiKey protects the sensitive handler from running incorrectly. For Rocket to call the sensitive handler, the ApiKey type needs to be derived through a FromRequest implementation, which in this case, validates the API key header. Request guards are a powerful and unique Rocket concept; they centralize application policy and invariants through types.

Getting Started with Rocket:

- Before you can start writing a Rocket application, you'll need a nightly version of Rust installed. We recommend you use rustup to install or configure such a version by running the command:

```
$ rustup default nightly
```

- If you prefer, once we setup a project directory in the following section, you can use directory overrides to use the nightly version only for your Rocket project by running the following command in the directory:

```
$ rustup override set nightly
```

- Let's write our first Rocket application! Start by creating a new binary-based Cargo project and changing into the new directory:

```
$ cargo new hello-rocket --bin
$ cd hello-rocket
```

- Now, add Rocket as a dependency in your Cargo.toml file as:

```
[dependencies]
rocket = "0.4.4"
```

- Modify src/main.rs so that it contains the code for the Rocket Hello, world! program, reproduced below:

```
#![feature(proc_macro_hygiene)]

#[macro_use] extern crate rocket;

#[cfg(test)] mod tests;

#[get("/hello/<name>/<age>")]
fn hello(name: String, age: u8) -> String {
    format!("Hello, {} year old named {}!", age, name)
}

#[get("/hello/<name>")]
fn hi(name: String) -> String {
    name
}

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

- The above code creates an index route, mounts the route at the / path, and launches the application.
- Now, executing the program you should see the following:

```
$ cargo run
  Configured for development.
    => address: localhost
    => port: 8000
    => log: normal
    => workers: [logical cores * 2]
    => secret key: generated
    => limits: forms = 32KiB
    => keep-alive: 5s
    => tls: disabled
  Mounting '/':
    => GET / (index)
  Rocket has launched from http://localhost:8000
```

- To see the application in action, open any browser and direct to http://localhost:8000

Exercise 3: Create a Simple Application with Rocket

- Create the application discussed in the example and add a few elements to the canvas
- Check out this Github repository for more examples on creating applications with Rocket.

Server-side Development with Actix-web and Rust:

- Actix web is a small, pragmatic, and extremely fast web framework for Rust.
- An application developed with actix-web will expose an HTTP server contained within a native executable. You can either put this 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.
- Actix-web runs on Rust 1.39 or later and it works with stable releases.

Getting started with Actix-web:

- To start with Actix-web, creating a new binary-based Cargo project "hello-world" and change into the new directory.

```
$ cargo new hello-world
$ cd hello-world
```

- Now, in order to work with Actix-web, add actix-web as a dependency of your project by ensuring your Cargo.toml contains the following:

```
[dependencies]
actix-web = "2.0"
```

- Since in our example, we will be using #[actix_rt::main] macro, we also need to add *actix-rt* to our dependencies. At this point, the Cargo.toml should look like following:

```
[dependencies]
actix-web = "2.0"
actix-rt = "1.0"
```

- Next, in order to implement a web server, we first need to create a request handler. Request handlers refer to the callable function that contains the application logic to handle an incoming request to the server.
- With Actix-web, a request handler is an async function that accepts zero or more parameters that can be extracted from a request (i.e., impl FromRequest) and returns a type that can be converted into an HttpResponse (i.e., impl Responder). For example:

```
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!")
}
```

- Request handling with Actix-web happens in two stages. First the handler object is called, returning any object that implements the Responder trait. Then, respond_to() is called on the returned object, converting itself to a HttpResponse or Error.
- In the following example, we bind the routes to their respective handlers. Here, we create an App instance and register the request handler with the application's route on a path and with a particular HTTP method. After that, the application instance can be used with HttpServer to listen for incoming connections. The server accepts a function that should return an application factory:

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(index))
            .route("/again", web::get().to(index2))
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

- Now you can compile and run the program with *cargo run* and then open a browser and head over to http://localhost:8000/ to see the results.
- In the above example, we create a simple HTTPServer with Actix-web. The HttpServer type is responsible for serving http requests.
- HttpServer accepts an application factory as a parameter, and the application factory must implement Send + Sync traits. More about that in the multi-threading section.
- To bind to a specific socket address, bind() must be used, and it may be called multiple times. To bind a ssl socket, bind_openssl() or bind_rustls() should be used. To run the http server, use the HttpServer::run() method.
- Different methods of the HTTPServer:
    - run() - returns instance of Server type.
    - pause() - Pause accepting incoming connections
    - resume() - Resume accepting incoming connections
    - stop() - Stop incoming connection processing, stop all workers and exit
- HttpServer supports graceful shutdown. After receiving a stop signal, workers have a specific amount of time to finish serving requests. Any workers still alive after the timeout are dropped by force. By default, the shutdown timeout is set to 30 seconds. You can change this parameter with the HttpServer::shutdown_timeout() method.
- You can send a stop message to the server with the server address and specify if you want graceful shutdown or not. The start() method returns the address of the server.

Using Attribute Macros to Define Routes:

- Alternatively, routes can also be defined using macro attributes which allow you to specify the routes above your functions:

```
use actix_web::get;

#[get("/hello")]
async fn index3() -> impl Responder {
    HttpResponse::Ok().body("Hey there!")
}
```

- Then the route can be registered using service() in the main.rs file as:

```
App::new().service(index3)
```

Auto-Reloading Development Server:

- During development it can be very handy to have cargo automatically recompile the code on change. This can be accomplished by using *cargo-watch*.
- An actix app will typically bind to a port listening for incoming HTTP requests, it makes sense to combine this with the listenfd crate and the systemfd utility to ensure the socket is kept open while the app is compiling and reloading.
- *systemfd* will open a socket and pass it to *cargo-watch* which will watch for changes and then invoke the compiler and run your actix app. The actix app will then use listenfd to pick up the socket that *systemfd* opened.
- In order to implement automatic reloading, you need to install *cargo-watch* and *systemfd* as:

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

- Additionally, you need to slightly modify your cargo.toml file as follows to let your actix-web app know that it can pick up an external socket opened by systemfd:

```
[dependencies]
actix-web = "2.0"
actix-rt = "1.0"
listenfd = "0.3"
```

- Then modify the server code, we have written in page 14 to only invoke bind as a fallback:

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    let mut listenfd = ListenFd::from_env();
    let mut server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(index))
            .route("/again", web::get().to(index2))
    })
    server = if let Some(l) = listenfd.take_tcp_listener(0).unwrap() {
        server.listen(l)?
    } else {
        server.bind("127.0.0.1:8000")?
    };

    server.run().await
```

```
}
```

- To execute the code, open the terminal and type:

```
$ systemfd --no-pid -s http::3000 -- cargo watch -x run
```

Writing an Application with Actix-web:

- Actix-web provides various primitives to build web applications with Rust. It provides routing, middleware, pre-processing of requests, post-processing of responses, etc.
- All actix-web servers are built around the App instance. It is used for registering routes for resources. It also stores application state shared across all handlers within same scope.
- An application's scope acts as a namespace for all routes, i.e. all routes for a specific application scope have the same url path prefix.
- For example, an application with scope /app, any request with the paths /app, /app/, or /app/test would match; however, the path /application would not match. The following listing uses our server example from page 14 and shows an example for the scope /app.

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

async fn index() -> impl Responder {
    "Hello world!"
}

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

- In this example, an application with the /app prefix and an index.html resource are created. This resource is available through the /app/index.html url.

State of Actix-web Applications:

- Application state is shared with all routes and resources within the same scope. State can be accessed with the web::Data<T> extractor where T is type of state. .
- The following example shows an application that stores the application name in the state:

```
use actix_web::{web, App, HttpServer};
use std::sync::Mutex;

// This struct represents state
struct AppState {
```

```
    app_name: String,
}

async fn index(data: web::Data<AppState>) -> String {
    let app_name = &data.app_name; // <- get app_name

    format!("Hello {}!", app_name) // <- response with app_name
}

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

- In the above example, we pass in the state when initializing the App and then start the application. Any number of state types could be registered within application.

Shared Mutable State:

- HttpServer accepts an application factory rather than an application instance. The http server constructs an application instance for each thread; thus, the application data must be constructed multiple times.
- In the following example, we will write an application with mutable, shared state. First, we define our state and create our handler then we register the data in an App:

```
struct AppStateWithCounter {
    counter: Mutex<i32>, // <- Mutex is necessary to mutate safely across
threads
}

async fn _index(data: web::Data<AppStateWithCounter>) -> String {
    let mut counter = data.counter.lock().unwrap(); // <- get counter's
MutexGuard
    *counter += 1; // <- access counter inside MutexGuard

    format!("Request number: {}", counter) // <- response with count
}

#[actix_rt::main]
async fn _main() -> std::io::Result<()> {
    let counter = web::Data::new(AppStateWithCounter {
```

```
        counter: Mutex::new(0),
    });

    HttpServer::new(move || {
        // move counter into the closure
        App::new()
            .app_data(counter.clone()) // <- register the created data
            .route("/", web::get().to(_index))
    })
    .bind("127.0.0.1:8088")?
    .run()
    .await
}
```

- The web::scope() method allows us to set a specific application prefix. This scope represents a resource prefix that will be prepended to all resource patterns added by the resource configuration. This can be used to help mount a set of routes at a different location while still maintaining the same resource names. You can learn more about using an application scope to compose applications here.

Respond with a Custom Type from a Request Handler Function:

- To return a custom type directly from a handler function, the type needs to implement the Responder trait.
- The following example shows a response for a custom type that serializes to an application/json response:

```
use actix_web::{Error, HttpRequest, HttpResponse, Responder};
use serde::Serialize;
use futures::future::{ready, Ready};

#[derive(Serialize)]
struct MyObj {
    name: &'static str,
}

// Responder
impl Responder for MyObj {
    type Error = Error;
    type Future = Ready<Result<HttpResponse, Error>>;

    fn respond_to(self, _req: &HttpRequest) -> Self::Future {
        let body = serde_json::to_string(&self).unwrap();

        // Create response and set content type
        ready(Ok(HttpResponse::Ok()
            .content_type("application/json")
            .body(body)))
    }
}
```

```
async fn index() -> impl Responder {
    MyObj { name: "user" }
}
```

- Sometimes, you need to return different types of responses. For example, you can error check and return errors or any result that requires different types.
- For this case, the *Either* type can be used. *Either* allows combining two different responder types into a single type. Here you can learn more about the *Either* type.

Multi-threading with Actix-web:

- HttpServer automatically starts a number of http workers, by default this number is equal to number of logical CPUs in the system. This number can be overridden with the HttpServer::workers() method. For example:

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

#[actix_rt::main]
async fn main() {
    HttpServer::new(|| {
        App::new().route("/", web::get().to(|| HttpResponse::Ok()))
    })
    .workers(4); // <- Start 4 workers
}
```

- Once the workers are created, they each receive a separate application instance to handle requests. Application state is not shared between the threads, and handlers are free to manipulate their copy of the state with no concurrency concerns.
- Application state does not need to be Send or Sync, but application factory must. To share state between worker threads, use an Arc<…>.
- Special care should be taken once sharing and synchronization are introduced. In many cases, performance costs are inadvertently introduced as a result of locking the shared state for modifications.
- Since each worker thread processes its requests sequentially, handlers which block the current thread will cause the current worker to stop processing new requests. For this reason, any long, non-cpu-bound operation (e.g. I/O, database operations, etc.) should be expressed as futures or asynchronous functions. Async handlers get executed concurrently by worker threads and thus don't block execution. For example:

```
async fn my_handler() -> impl Responder {
    tokio::time::delay_for(Duration::from_secs(5)).await; // <-- Ok. Worker
thread will handle other requests here
    "response"
}
```

Type-safe information extraction with Actix-web:

- Actix-web provides a facility for type-safe request information access called extractors (i.e., impl FromRequest). By default, actix-web provides several extractor implementations.
- An extractor can be accessed as an argument to a handler function. Actix-web supports up to 10 extractors per handler function. Argument position does not matter. For example:

```
async fn index(
    path: web::Path<(String, String)>,
    json: web::Json<MyInfo>,
) -> impl Responder {
    format!("{} {} {} {}", path.0, path.1, json.id, json.username)
}
```

- In the above example, *path* provides information that can be extracted from the Request's path. You can deserialize any variable segment from the path. Also, in the example, *json* allows to deserialize a request body into a struct.
- Here you can learn more about different extractor implementations provided by actix-web.

ExpressJS vs Actix-Web:

- Whilst Actix provides 85% running cost saving in heavily loaded environments, ExpressJS is fast, minimalist and most popular Node.js web framework. Hence, when to use Rust and Actix-web rather than Node and Express can be an interesting question to ponder upon.
- A detailed analysis of their similarities and differences using a simple test scenario can be found here.

Exercise 4: Create a simple HTTPServer with Actix-web & Explore the API

- Create the basic HTTPServer with Actix-web discussed in the example in page 14 and execute the code. Modify your code to respond with a custom type from a request handler function.

- Explore the actix_web crate for more details.

- Check out this Github repository for several usage examples of the Actix-web API. To run any specific example from the repository, open the terminal and type:

```
$ git clone https://github.com/actix/actix
$ cargo run --example <name of example>
```