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 Tower-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 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 way. 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, 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 user's input, parses the input into JavaScript data-structure, and stores 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 codes and to debug JavaScript codes.
- REPL can be started by simply running node on terminal or console without any arguments as follows:

```
$ node
```

- With this command you sould 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 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 maintain concurrency.
- Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

- 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 to events act as Observers. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and 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 handler

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 8081 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(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

- 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>
```

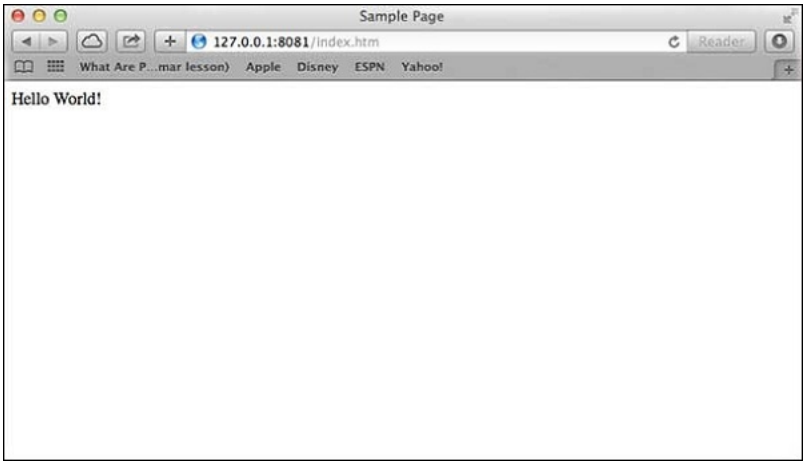
```
   <head>
      <title>Sample Page</title>
   </head>

   <body>
      Hello World!
   </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://127.0.0.1:8081/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://127.0.0.1:8081/
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
```

Exercise1: 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 Angular 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>
```

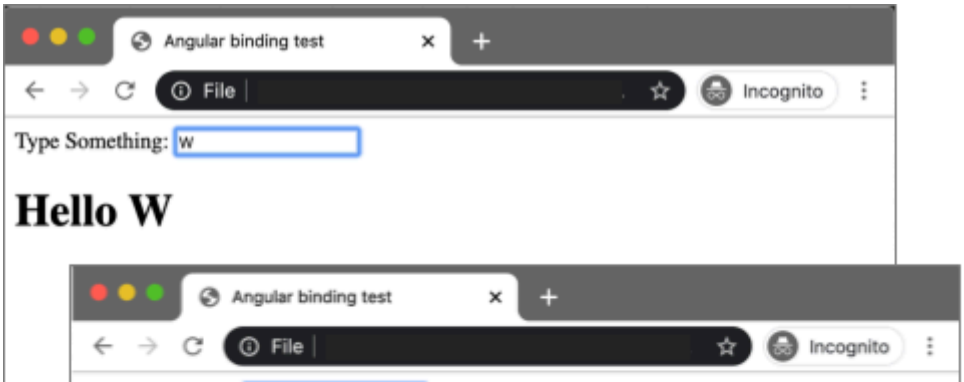
- Next, assuming we download and reference it locally we need to add it to the page. Also to tell Angular that this page is an Angular application. To do this we can add a simple attribute, *ng-app* to the opening <html> tag.
- Now, to take the input from the form and show it in the HTML without writing any JavaScript we just need to bind an Angular 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 Angular 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>Angular 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:



Exercise2: Two-way Data Binding with AngularJS

- Using the above example and create a simple HTML page that shows two way binding.
- Improvise the example by adding multiple text boxes for entering 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 the examples that perform data binding, sorting, and searching 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, expressibility, and speed.
- Rocket's design is centered around three core philosophies:
    - *Security, correctness, and developer experience are paramount*: The path of least resistance should lead you to the most secure, correct web application, though security and correctness should not come at the cost of a degraded developer experience. 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*: Because the web and HTTP are themselves untyped (or stringly typed, as some call it), this means that something or someone has to convert strings to native types. Rocket does this 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!
- Request 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 request 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 per-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(decl_macro)]
#![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

Exercise3: 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 Tower-web and Rust:

- Tower Web is a fast web framework that aims to remove boilerplate.
- The goal is to decouple all HTTP concepts from the application logic. You implement your application using "plain old Rust types" and Tower Web uses a macro to generate the necessary glue to serve the application as an HTTP service.
- The bulk of Tower Web lies in the impl_web macro. Tower web also provides #[derive(Extract)] (for extracting data out of the HTTP request) and #[derive(Response)] (for converting a struct to an HTTP response).
- Towerweb is:
    - **Fast**: Fully asynchronous, built on Tokio and Hyper.
    - **Ergonomic**: Tower-web decouples HTTP from your application logic, removing all boilerplate.
    - **Works on Rust stable**: You can use it today.
- The impl_web! macro in tower_web wraps one or more impl blocks and generates Resource implementations. These structs may then be passed to ServiceBuilder.
- impl_web! looks for methods that have a routing attribute. These methods will be exposed from the web service. All other methods will be ignored.

Getting started with Tower-web:

- To start with Tower-web add the following dependencies in the cargo.toml file of your project:

```
[dependencies]
tower-web = "0.3.7"
tokio = "0.2.13"
```

- A simple “Hello World” example with Tower Web:

```
#[macro_use]
extern crate tower_web;
extern crate tokio;

use tower_web::ServiceBuilder;
use tokio::prelude::*;

/// This type will be part of the web service as a resource.
#[derive(Clone, Debug)]
struct HelloWorld;

/// This will be the JSON response
#[derive(Response)]
struct HelloResponse {
    message: &'static str,
}

impl_web! {
    impl HelloWorld {
        #[get("/")]
```

```
        #[content_type("json")]
        fn hello_world(&self) -> Result<HelloResponse, ()> {
            Ok(HelloResponse {
                message: "hello world",
            })
        }
    }
}

pub fn main() {
    let addr = "127.0.0.1:8080".parse().expect("Invalid address");
    println!("Listening on http://{}", addr);

    ServiceBuilder::new()
        .resource(HelloWorld)
        .run(&addr)
        .unwrap();
}
```

- Now, you can execute the code with the cargo run command.

Exercise4: Explore the TowerWeb API

- Create the Hello World project with Tower-web discussed the above and execute the code.
- Explore the tower_web crate for more details.
- Check out this Github repository for several usage examples of the tower-web API.