Lab 6: ExpressJS and Client-Side Programming in Rust

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

Learning Objectives:

- Introducing Express
- Using Stdweb to Build a Client-side Application with Rust
- Client-side Development with Yew and Rust

Introducing Express:

- ExpressJS is a web application framework for Node.js that provides with a simple API to build websites, web apps and back ends.
- Express provides a simple interface for directing incoming URLs to different functions in the code.
- It provides support for a number of different templating engines that make it easier to build HTML pages in an intelligent way by using reusable components from the application.
- Express overcomes the limitations of Node.js in terms of being able to store session state. This helps with providing personalized experiences to individual users.

Setting up the Environment:

- To start developing and using the Express Framework, the Node and the npm (node package manager) should be already installed.
- 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
```

- Prior to installing Express, you need to set up your project directory and especially the "package.json" file.
- Start your terminal and set up your project directory as:

```
$ mkdir myFirstProject
$ cd myFirstProject
```

- To create the package.json file in your terminal type:

```
$ npm init
```

- Now to install Express, switch back to the terminal and type.

```
$ npm install --save express
```

- To make our development process a lot easier, we will install a tool from npm, nodemon. This tool restarts our server as soon as we make a change in any of our files, otherwise we need to restart the server manually after each file modification. To install nodemon, use the following command:

```
$ npm install -g nodemon
```

- To test your Express installation type:

```
$ express --version
```

- Now that we are all set! We can start exploring the utilities of Express.

Writing your first App using Express:

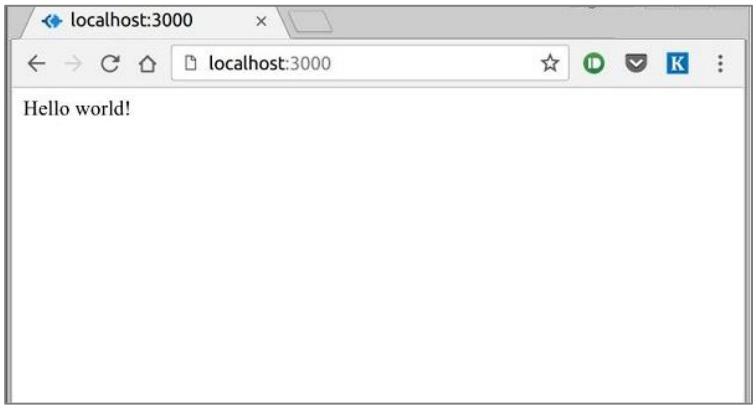
- To start developing with Express, create a new file called **index.js** inside your project directory and type the following in it:

```
var express = require('express');
var app = express();

app.get('/', function(req, res){
   res.send("Hello world!");
});

app.listen(3000);
```

- Save the file, go to your terminal and type the following:

```
$ nodemon index.js
```

- This will start the server. To test this app, open your browser and go to http://localhost:3000



What is happening at the backend?

- var express = require('express');
  - imports Express in our file, we have access to it through the variable Express. We use it to create an application and assign it to var app.
- app.get(route, callback)
  - This function states what to do when a get request at the given route.
  - The callback function has 2 parameters, request(req) and response(res)

- The request object(req) represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.
- the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

Managing Routes with Express:

- Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.
- The following function is used to define routes in an Express application:

```
app.method(path, handler)
```

- This METHOD can be applied to any one of the HTTP verbs (e.g., get, post, delete, etc.). An alternate method also exists, which executes independent of the request type.
- Path is the route at which the request will run.
- Handler is a callback function that executes when a matching request type is found on the relevant route.
- Routes in Express.js simply serve as a mapping service, taking the URL of an incoming request and mapping it to a specific piece of application functionality.
- We can also have multiple different methods at the same route. For example:

```
var express = require('express');
var app = express();

app.get('/hello', function(req, res){
   res.send("Hello World!");
});

app.post('/hello', function(req, res){
   res.send("Post method at '/hello'!\n");
});

app.listen(3000);
```

Different controller files for different collections

- Looking to the future we know that at some point our application will grow, and we don't want to have all the routes and their respective controllers in one file.
- The solution is to have a single route file and one controller file for each logical collection of views.
- As the first step: create a directory called **app-server** inside your project root. Next, inside app-server create a directory **routes**.
- Inside the routes directory now create a file *index.js*
- The next step would be to require the controller files in routes/index.js

```
var express = require('express');
var router = express.Router();
var ctrlView1 = require('../controllers/view1');
```

```
var ctrlView2 = require('../controllers/view2');
```

- Now we have two variables we can reference in the route definitions, which will contain different collections of routes.

```
var express = require('express');
var router = express.Router();
var ctrlView1 = require('../controllers/view1');
var ctrlView2 = require('../controllers/view2');

/* View1 pages */
router.get('/', ctrlView1.task1);
router.get('/view1/task2, ctrlView1.task2);
router.get('/view1/task3, ctrlView1.task3);

/* View2 pages */
router.get('/view2, ctrlView2.task1);

module.exports = router;
```

Building basic controllers

- To create controllers for different views of your web-application, create a directory called **controllers** inside your app-server directory
- Inside this directory add two controller files for your two views namely: **view1.js** and **view2.js**
- Inside the view1.js add the following code:

```
/* GET html page for task1 */
module.exports.task1 = function(req, res){
      res.render('index', { title: 'task1' });
};

/* GET html page for task2 */
module.exports.task2 = function(req, res){
      res.render('index', { title: 'task2' });
};

/* GET html page for task3 */
module.exports.task3 = function(req, res){
      res.render('index', { title: 'task3’ });
};
```

- Inside the view2.js add the following code:

```
/* GET html page for task1 in view2 */
module.exports.task1 = function(req, res){
      res.render('index', { title: 'View2-task1' });
};
```

Moving data from the view to the controllers

- Since the controllers will do most of the processing in the application, it is important to accurately pass data from the views to the controllers.
- If you note the previous listing, we are already sending a piece of data to the view. The second parameter in the *render* function is a JavaScript object that contains the data to send to the view.
- We can send additional data to the view in the same way in JSON format as:

```
module.exports.task1 = function(req, res){
      res.render('index', {
           title: 'My first application with Express',
           pageHeader: {
                    title: 'My application',
                    description: 'An application to explore Express!'
              }
   });
};
```

- The passed data can now be accessed from the template index.html;

Adding Middleware to Express routes

- Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle.
- In Express, middleware gets in the middle of the route and the controller. So, once a route is called, the middleware is activated before the controller, and can prevent the controller from running or change the data being sent.
- Here is a simple example of a middleware function in action:

```
var express = require('express');
var app = express();

app.use(function(req, res, next){
   console.log("A new request received at " + Date.now());
   next();
});

app.get('/', (req, res, next) => {
  res.send('Welcome to the Home Page');
});
```

Exercise1: A Simple RESTful Application with Express and NodeJS

- Try out this Github repository for a simple Weather Website using Node.js, Express, and OpenWeatherMap's API
    - Change the output temperature from Fahrenheit Scale to Celsius Scale
- For a more detailed example, check out this Github repository with an example of a RESTful Web App with Node.js, Express, and MongoDB

Using Stdweb to Build a Client-side Application with Rust:

- The stdweb library forms the basis for many of the web assembly frameworks and libraries that exist in Rust.
- Stdweb exposes a full native set of Web based APIs for interfacing with web browsers whereas with wasm_bindgen the developer needs to specify which parts of the web API they want to use.
- Stdweb has many nice features built directly on top of it; this includes the ability to write JavaScript code inside of the Rust application using the js! macro and the ability to access various top-level objects using the console! macro. For example:

```
let message = "Welcome to Rust";
let result = js! {
     console.log( @{message} );
     return 5 + 5;
};

println!( "The outcome is = {:?}", result );
```

- Building applications using the stdweb library is very familiar for developers who are well versed in JavaScript and front-end development without giving up the advantages of using Rust as a language.

Characteristics of the Stdweb library

- Exposes a full suite of Web APIs as exposed by web browsers.
- Tries to follow the original JavaScript conventions and structure as much as possible, except in cases where doing otherwise results in a clearly superior design.
- Is a building block from which higher-level frameworks and libraries can be built.
- Makes it convenient and easy to embed JavaScript code directly into Rust and to marshal data between the two.
- Integrates with the wider Rust ecosystem, e.g. support marshalling (serialization) of structs which implement serde's Serializable.
- Puts Rust in the driver's seat where a non-trivial Web application can be written without touching JavaScript at all.
- Allows Rust to take part in the upcoming WebAssembly (re)volution.
- Makes it possible to trivially create standalone libraries which are easily callable from JavaScript.

Getting Started – Creating a Web Application with Stdweb

- The first step is to install cargo-web. For this, open the Terminal and type:

```
$ cargo install -f cargo-web
```

- Next let us create our project:

```
$ cargo new my_first_stdweb_proj
```

- Next open the cargo.toml define the stdweb dependency

```
[dependencies]
stdweb = “0.4.7”
```

- Next open the main.rs file and bring in the external crate for stdweb and write a simple few lines of code as:

```
#[macro_use]
extern crate stdweb;

fn main() {
      let message = “Welcome to Stdweb!”;
      js! {
            console.log( @{message} );
      };
}
```

- Now to compile and execute your code, in the terminal type:

```
$ cargo web build --target=wasm32-unknown-unknown
```

- Once you finish the compilation you would see that it will create a javascript (.js) file and a web assembly (.wasm) file with same the name as your project (i.e., my_first_stdweb_proj).
- Now create a directory under the root directory of your project and name it static.
- Copy the .js and .wasm files and paste them into the static directory.
- Create an html file inside the static directory and name it index.html
- Inside the index.html file, reference the javascript module as:

```
<html>
      <head>
      <title>Stdweb Example</title>
      </head>

      <body>
      <h1>Welcome to Stdweb</h1>
      <script src=” my_first_stdweb_proj.js”></script>
      </body>
</html>
```

- To see the outcome of the project, open terminal and type:

```
$ cargo web start
```

- Open a browser and visit http://localhost:8000 to see the output

Making Event Listeners for the Mouse and Keyboard

- The stdweb library also exposes many event-based types which we can use to create listeners. This includes IEvent, KeyDownEvent, KeyUpEvent, MouseDownEvent, MouseMoveEvent, MouseUpEvent as well as many others
- For example :

```
use
stdweb::web::event::{
      IEvent,  IKeyboardEvent,  KeyDownEvent,  KeyUpEvent,  KeyboardLocation,
MouseButton, MouseDownEvent, MouseMoveEvent, MouseUpEvent, }

web::window().add_event_listener(move |event: MouseMoveEvent| {
        if on_mouse_move((event.client_x() as f64, event.client_y() as f64)) {
            draw_box(
                &ctx,
                "orange",
                (event.client_x() as f64, event.client_y() as f64),
                (10.0, 10.0),
            );
        }
    });
```

- In this example, we add event listeners to the window object.
- We are able to gain access to this window object through the web module from inside of the stdweb library.
- We add the event listeners by using closures as normal.

Accessing HTML Canvas from Rust

- Along with the ability to create event listeners and use macros to access various parts of the web API; stdweb gives developers the ability to gain access to the canvas API.
- This includes the ability to access the canvas context with a CanvasRenderingContext2d type and a CanvasRenderingContext3d type.
- The canvas itself can be expressed using the CanvasElement type.
- For example:

```
use stdweb::web::{self, CanvasRenderingContext2d, IEventTarget,
INonElementParentNode};

let ctx: CanvasRenderingContext2d = canvas.get_context().unwrap();

draw_box(&ctx, "red", (20.0, 20.0), (150.0, 100.0));
draw_box(&ctx, "green", (200.0, 20.0), (150.0, 150.0));
draw_box(&ctx, "blue", (100.0, 20.0), (150.0, 150.0));
```

Exposing Rust functions to JavaScript

- Stdweb exports a procedural attribute macro called *js_export* which can be used to mark arbitrary functions for export. For example:

```
#[js_export]
fn hash( string: String ) -> String {
    let mut hasher = Sha1::new();
    hasher.update( string.as_bytes() );
    hasher.digest().to_string()
}
```

- This supports almost every type you can pass through the js! macro, which includes objects, arrays, arbitrary DOM types, etc.
- A current limitation of the #[js_export] is that all of the functions you want to export must be defined in your lib.rs, or alternatively they can be defined in another file, but you'll have to import them with pub use another_module::* into your lib.rs.
- Now, If you compile this code using:

```
$ cargo web build --target=wasm32-unknown-unknown
```

- Then two files with .js and .wasm extensions will be generated.
- You can copy them into your JavaScript project and load like any other JavaScript file as:

```
<script src="hasher.js"></script>
```

- After it's loaded you can access Rust.hasher, which is a *Promise* that will be resolved once the WebAssembly module is loaded. Inside that promise you'll find everything which you've marked with #[js_export]:

```
<script src="hasher.js"></script>
<script>
    Rust.hasher.then( function( hasher ) {
        console.log( hasher.hash( "Hello world!" ) );
    });
</script>
```

- You can also use the very same hasher.js from Nodejs:

```
var hasher = require( "hasher.js" ); // `hasher.js` is exported from Rust code
console.log( hasher.hash( "Hello world!" ) );
```

- For the Nodejs environment the WebAssembly is compiled synchronously.

Exercise2: Create a Simple Web Application with Stdweb

- Create the web-application discussed in the example and add a few elements to the canvas
- Check out this Github repository for more information on the Stdweb library

Client-side Development with Yew and Rust:

- Yew is a modern Rust framework for creating multi-threaded front-end web apps with WebAssembly.
- It features a component-based framework which makes it easy to create interactive user interfaces.
- It has great performance by minimizing DOM API calls and by helping developers easily offload processing to background web workers.
- It supports JavaScript interoperability, allowing developers to leverage NPM packages and integrate with existing JavaScript applications.
- Yew is a framework that takes many of the concepts of the Elm programming language and implements them into the Rust programming language.
- All Elm programs follow a powerful and opinionated MVC style and Yew adopts this style to make it possible to do Reactive Functional Programming in Rust.
- The Yew Framework also has a powerful macro system which allows the user to build HTML interfaces directly inside of Rust in much the same way that you would in a framework like React or Elm.

Getting started

- Now that you have already installed cargo web, starting a project with Yew will be simple.
- To begin with, create your first Yew project in the terminal as:

```
$ cargo new my-first-yew-app
```

- Move to your project directory and update the cargo.toml file with the dependencies as:

```
[dependencies]
yew = “0.11.0”
```

- In the main.rs file type the following:

```
use yew::{html, Component, ComponentLink, Html, InputData, ShouldRender};

pub struct Model {
    link: ComponentLink<Self>,
    value: String,
}

pub enum Msg {
    GotInput(String),
    Clicked,
}

impl Component for Model {
    type Message = Msg;
    type Properties = ();

fn create(_: Self::Properties, link: ComponentLink<Self>) -> Self {
        Model {
```

```
            link,
            value: "".into(),
        }
    }

    fn update(&mut self, msg: Self::Message) -> ShouldRender {
        match msg {
            Msg::GotInput(new_value) => {
                self.value = new_value;
            }
            Msg::Clicked => {
                self.value = "Changed Value".to_string();
            }
        }
        true
    }

    fn view(&self) -> Html {
        html! {
            <div>
                <div>
                    <textarea rows=5
                        value=&self.value
                        oninput=self.link.callback(|e: InputData|
Msg::GotInput(e.value))
                        placeholder="placeholder">
                    </textarea>
                    <button   onclick=self.link.callback(|_|   Msg::Clicked)>{
"change value" }</button>
                </div>
                <div>
                    {&self.value}
                </div>
            </div>
        }
    }
}

fn main() {
    yew::start_app::<Model>();
}
```

- Execute the project using the command below followed by directing to http://localhost:8000

```
$ cargo web start
```

Exercise3: Explore the Yew API

- Extend the above example and create an HTML form containing a text-box and radio button.
- Check out this Github repository for the Yew library and look into the examples