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 Yew to Build a Client-side Application and Development with Rust

Introducing Express:

- ExpressJS is a web application framework for Node.js that provides 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:

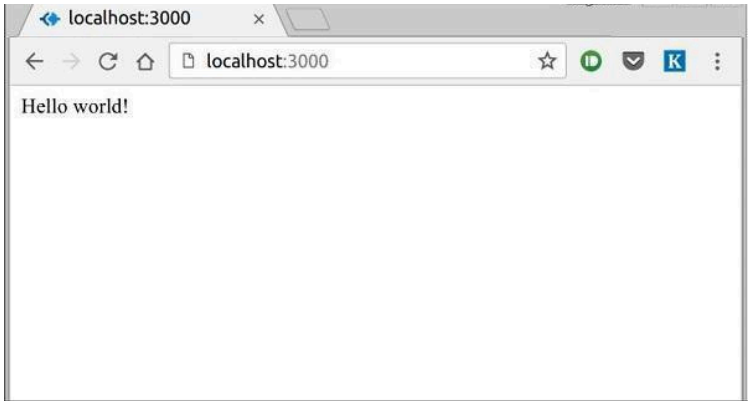
- To start developing with Express, create a new file called **index.js** inside your project directory and type the following in it:

```
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 arrives 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 Yew to Build and Develop a Client-side Application with Rust:

Introduction to Yew:

- 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.

Prerequisites:

- Before we begin our web application using Yew, we must first install Trunk.
- Trunk is a WASM web application bundler for Rust.
- We must also add the wasm build target.
- To install trunk and add the wasm build target, we run the following in the command line:

```
$ cargo install trunk

$ rustup target add wasm32-unknown-unknown
```

- After that we can create a new rust project to start building our web application:

```
$ cargo new yew-app
$ cd yew-app

$ cargo run
```

Getting started:

- First let us add the yew dependency to the cargo.toml file.
- We should add “csr” to specify we want to enable Client-Side Rendering support and Renderer.

```
[package]
name = "yew-app"
version = "0.1.0"
edition = "2021"

[dependencies]
yew = { version = "0.20", features = ["csr"] }
```

- Next we should add the following code to main.rs to create a simple Hello World web application:

```
use yew::prelude::*;

#[function_component(App)]
fn app() -> Html {
    html! {
        <h1>{ "Hello World" }</h1>
    }
}

fn main() {
    yew::Renderer::<App>::new().render();
```

```
}
```

- We also need to create an index.html file in the project directory (in this example: yew-app).
- The following code should be added to that file:

```
<!DOCTYPE html>
<html lang="en">
    <head> </head>
    <body></body>
</html>
```

- Finally we can run our web application using the following command in the command line:

```
$ trunk serve --open
```

- Your application should be running on http://localhost:8080/ or http://127.0.0.1:8080/ in a web browser.
- Hello World should be written on the web page.

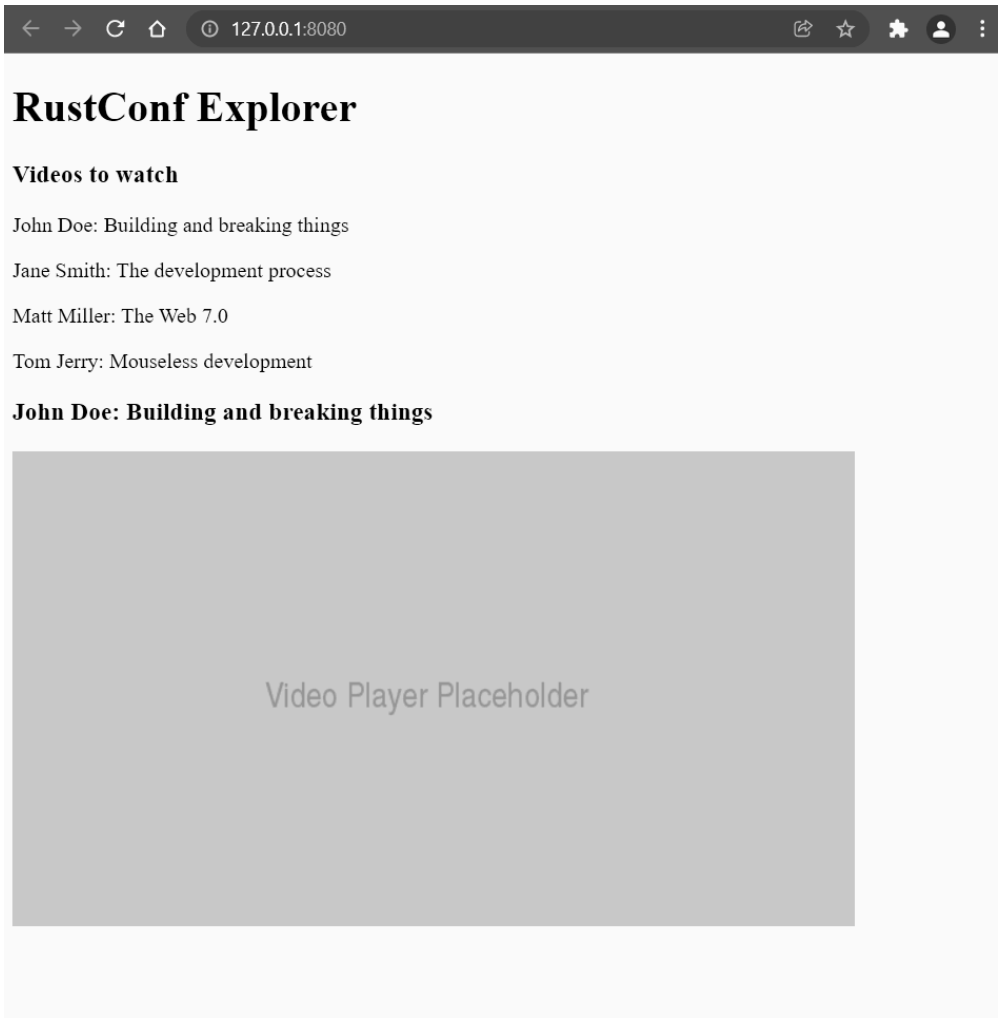
Using HTML:

- Using html is simple. We can simply write our html code in the html!{ … }.
- However there are some differences between normal html and the html!{ … } that we are using here in rust:
    - Expressions must be wrapped in curly braces ({ })
    - There must only be one root node. If you want to have multiple elements without wrapping them in a container, an empty tag/fragment (<> ... </>) is used
    - Elements must be closed properly.
- Here is an example of how to apply html code to our program:

```
use yew::prelude::*;

#[function_component(App)]
fn app() -> Html {
-    html! {
-        <h1>{ "Hello World" }</h1>
-    }
+    html! {
+        <>
+            <h1>{ "RustConf Explorer" }</h1>
+            <div>
+                <h3>{"Videos to watch"}</h3>
+                <p>{ "John Doe: Building and breaking things" }</p>
+                <p>{ "Jane Smith: The development process" }</p>
+                <p>{ "Matt Miller: The Web 7.0" }</p>
+                <p>{ "Tom Jerry: Mouseless development" }</p>
+            </div>
+            <div>
+                <h3>{ "John Doe: Building and breaking things" }</h3>
```

```
+                <img
src="https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
alt="video thumbnail" />
+            </div>
+        </>
+    }
}

fn main() {
    yew::Renderer::<App>::new().render();
}
```

- Running the program should result in the following output on the webpage:



Rust constructs in markup:

- Instead of hardcoding all the code in the html, we can utilize rust constructs in order to make the html code more dynamic.

- In the code sample below, we used a vector to represent the video titles we want to appear on the webpage.
- The vector is then converted to html code and used in the html.
- This lets our html code be dynamic since we can change the size and values within the vector without having to modify the html code.

```
use yew::prelude::*;

+ struct Video {
+    id: usize,
+    title: String,
+    speaker: String,
+    url: String,
+ }

#[function_component(App)]
fn app() -> Html {
+    let videos = vec![
+        Video {
+            id: 1,
+            title: "Building and breaking things".to_string(),
+            speaker: "John Doe".to_string(),
+            url: "https://youtu.be/PsaFVLr8t4E".to_string(),
+        },
+        Video {
+            id: 2,
+            title: "The development process".to_string(),
+            speaker: "Jane Smith".to_string(),
+            url: "https://youtu.be/PsaFVLr8t4E".to_string(),
+        },
+        Video {
+            id: 3,
+            title: "The Web 7.0".to_string(),
+            speaker: "Matt Miller".to_string(),
+            url: "https://youtu.be/PsaFVLr8t4E".to_string(),
+        },
+        Video {
+            id: 4,
+            title: "Mouseless development".to_string(),
+            speaker: "Tom Jerry".to_string(),
+            url: "https://youtu.be/PsaFVLr8t4E".to_string(),
+        },
+    ];

+   let videos = videos.iter().map(|video| html! {
+       <p key={video.id}>{format!("{}: {}", video.speaker, video.title)}</p>
+   }).collect::<Html>();

    html! {
        <>
```

```
            <h1>{ "RustConf Explorer" }</h1>
            <div>
                <h3>{ "Videos to watch" }</h3>
-               <p>{ "John Doe: Building and breaking things" }</p>
-               <p>{ "Jane Smith: The development process" }</p>
-               <p>{ "Matt Miller: The Web 7.0" }</p>
-               <p>{ "Tom Jerry: Mouseless development" }</p>
+               { videos }
            </div>
            // ...
        </>
    }
}

fn main() {
    yew::Renderer::<App>::new().render();
}
```

Components:

- Components are the building blocks that we use to build our applications.
- Components are used to allow our code to be reusable and keep it generic.
- There are two types of components: function and struct components.
- In the following example, we show how to use video lists as its own function component rather than using it in the App function component.
- This makes our code cleaner and reusable.

```
use yew::prelude::*;

+ #[derive(Clone, PartialEq)]
struct Video {
    id: usize,
    title: String,
    speaker: String,
    url: String,
}

+ #[derive(Properties, PartialEq)]
+ struct VideosListProps {
+     videos: Vec<Video>,
+ }

+ #[function_component(VideosList)]
+ fn videos_list(VideosListProps { videos }: &VideosListProps) -> Html {
+     videos.iter().map(|video| html! {
+         <p key={video.id}>{format!("{}: {}", video.speaker,
video.title)}</p>
+     }).collect()
+ }
```

```
#[function_component(App)]
fn app() -> Html {
    // ...
-    let videos = videos.iter().map(|video| html! {
-        <p key={video.id}>{format!("{}: {}", video.speaker,
video.title)}</p>
-    }).collect::<Html>();
-
    html! {
        <>
            <h1>{ "RustConf Explorer" }</h1>
            <div>
                <h3>{"Videos to watch"}</h3>
-               { videos }
+               <VideosList videos={videos} />
            </div>
            // ...
        </>
    }
}

fn main() {
    yew::Renderer::<App>::new().render();
}
```

Making an Interactive Web Application:

- To make an interactive web application, we need to use Callbacks.
- We use Callbacks to pass event handlers between components, which makes the web application interactive.
- In this example, the VideoList component needs to use a Callback to notify when a video is selected.
- VideoList component will “emit” the video to the Callback.
- We created a new component VideoDetails to display the video details when clicked.
- App component is then modified to display the video details when clicked.
- We use use_state in the App component. use_state is called a “hook”. A hook is a function that lets you store state and perform side-effects.

```
// ...

#[derive(Properties, PartialEq)]
struct VideosListProps {
    videos: Vec<Video>,
+    on_click: Callback<Video>
}

#[function_component(VideosList)]
- fn videos_list(VideosListProps { videos }: &VideosListProps) -> Html {
```

```
+ fn videos_list(VideosListProps { videos, on_click }: &VideosListProps) ->
Html {
+    let on_click = on_click.clone();
    videos.iter().map(|video| {
+        let on_video_select = {
+            let on_click = on_click.clone();
+            let video = video.clone();
+            Callback::from(move |_| {
+                on_click.emit(video.clone())
+            })
+        };

        html! {
-            <p key={video.id}>{format!("{}: {}", video.speaker,
video.title)}</p>
+            <p key={video.id} onclick={on_video_select}>{format!("{}: {}",
video.speaker, video.title)}</p>
        }
    }).collect()
}

+ #[derive(Properties, PartialEq)]
+ struct VideosDetailsProps {
+    video: Video,
+ }

+ #[function_component(VideoDetails)]
+ fn video_details(VideosDetailsProps { video }: &VideosDetailsProps) -> Html
+ {
+    html! {
+        <div>
+            <h3>{ video.title.clone() }</h3>
+            <img
src="https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
alt="video thumbnail" />
+        </div>
+    }
+ }

#[function_component(App)]
fn app() -> Html {
    // ...
+    let selected_video = use_state(|| None);

+    let on_video_select = {
+        let selected_video = selected_video.clone();
+        Callback::from(move |video: Video| {
+            selected_video.set(Some(video))
+        })
+    };
```

```
+    let details = selected_video.as_ref().map(|video| html! {
+        <VideoDetails video={video.clone()} />
+    });

    html! {
        <>
            <h1>{ "RustConf Explorer" }</h1>
            <div>
                <h3>{"Videos to watch"}</h3>
-               <VideosList videos={videos} />
+               <VideosList videos={videos}
on_click={on_video_select.clone()} />
            </div>
+            { for details }
-            <div>
-                <h3>{ "John Doe: Building and breaking things" }</h3>
-                <img
src="https://via.placeholder.com/640x360.png?text=Video+Player+Placeholder"
alt="video thumbnail" />
-            </div>
        </>
    }
}

fn main() {
    yew::Renderer::<App>::new().render();
}
```

Fetching data using external REST API:

- Usually data comes from an external API.
- In order to fetch the data from these external API, we will need to using the following crates:
    - gloo-net: For making the fetch call.
    - serde: Has derive features for de-serializing the JSON response.
    - wasm-bindgen-futures: For executing Rust Future as a Promise.

```
[dependencies]
yew = { version = "0.20", features = ["csr"] }
+ gloo-net = "0.2"
+ serde = { version = "1.0", features = ["derive"] }
+ wasm-bindgen-futures = "0.4"
```

- Finally we will modify the code to make the fetch request instead of using hardcoded data.

```
use yew::prelude::*;
+ use serde::Deserialize;
+ use gloo_net::http::Request;
```

```
- #[derive(Clone, PartialEq)]
+ #[derive(Clone, PartialEq, Deserialize)]
struct Video {
    id: usize,
    title: String,
    speaker: String,
    url: String,
}

// ...

#[function_component(App)]
fn app() -> Html {
-    let videos = vec![
-        // ...
-    ]
+    let videos = use_state(|| vec![]);
+    {
+        let videos = videos.clone();
+        use_effect_with_deps(move |_| {
+            let videos = videos.clone();
+            wasm_bindgen_futures::spawn_local(async move {
+                let fetched_videos: Vec<Video> =
Request::get("/tutorial/data.json")
+                    .send()
+                    .await
+                    .unwrap()
+                    .json()
+                    .await
+                    .unwrap();
+                videos.set(fetched_videos);
+            });
+            || ()
+        }, ());
+    }

    // ...

    html! {
        <>
            <h1>{ "RustConf Explorer" }</h1>
            <div>
                <h3>{"Videos to watch"}</h3>
-                <VideosList videos={videos}
on_click={on_video_select.clone()} />
+                <VideosList videos={(*videos).clone()}
on_click={on_video_select.clone()} />
            </div>
            { for details }
        </>
```

```
    }
}
```

- In order to avoid problems with CORS (Cross-origin resource sharing), we will use a proxy server.

```
$ trunk serve --proxy-backend=https://yew.rs/tutorial
```

Exercise2: Explore the Yew API

- More examples of Yew applications that you can test and learn from can be found here.
- Currently, the example above has no styles and is not very appealing. However Trunk can be used to add style sheets to your application.
- The yew example above was referenced from https://yew.rs/docs/tutorial .