Lab: ASP.NET Core SignalR using TypeScript and Webpack

In this lab

1. Prerequisites
2. Create the ASP.NET Core web app
3. Configure the server
4. Configure the client
5. Test the app

This lab demonstrates using Webpack in an ASP.NET Core SignalR web app to bundle and build a client written in TypeScript. Webpack enables developers to bundle and build the client-side resources of a web app.

In this lab, you learn how to:

- Create an ASP.NET Core SignalR app
- Configure the SignalR server
- Configure a build pipeline using Webpack
- Configure the SignalR TypeScript client
- Enable communication between the client and the server

Create the ASP.NET Core web app

Run the following commands in the **Integrated Terminal**:

```
dotnet new web -o SignalRWebpack

cd SignalRWebpack
```

- The `dotnet new` command creates an empty ASP.NET Core web app in a `SignalRWebpack` directory.
- Open the `SignalRWebpack` directory in the code editor.

Run the following .NET CLI command in the **Integrated Terminal**:

```
dotnet add package Microsoft.TypeScript.MSBuild
```

The preceding command adds the Microsoft.TypeScript.MSBuild package, enabling TypeScript compilation in the project.

Configure the server

In this section, you configure the ASP.NET Core web app to send and receive SignalR messages.

1. In `Program.cs`, call [AddSignalR]:

```
```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR();
```
```

2. Again, in `Program.cs`, call UseDefaultFiles and UseStaticFiles:

```
var app = builder.Build();

app.UseDefaultFiles();
app.UseStaticFiles();
```

The preceding code allows the server to locate and serve the `index.html` file. The file is served whether the user enters its full URL or the root URL of the web app.

3. Create a new directory named `Hubs` in the project root, `SignalRWebpack/`, for the SignalR hub class.
4. Create a new file, `Hubs/ChatHub.cs`, with the following code:

```
```
using Microsoft.AspNetCore.SignalR;

namespace SignalRWebpack.Hubs;

public class ChatHub : Hub
{
    public async Task NewMessage(long username, string message) =>
        await Clients.All.SendAsync("messageReceived", username, message);
}
```

The preceding code broadcasts received messages to all connected
users once the server receives them. It\'s unnecessary to have a
generic `on` method to receive all the messages. A method named
after the message name is enough.

In this example:

-   The TypeScript client sends a message identified as
    `newMessage`.
-   The C\# `NewMessage` method expects the data sent by the client.
-   A call is made to
    [SendAsync]
    on
    [Clients.All].
-   The received messages are sent to all clients connected to the
    hub.
```

5. Add the following `using` statement at the top of `Program.cs` to resolve the `ChatHub` reference:

```
```
using SignalRWebpack.Hubs;
```
```

6. In `Program.cs`, map the `/hub` route to the `ChatHub` hub. Replace the code that displays `Hello World!` with the following code:

```
app.MapHub<ChatHub>("/hub");
```

Configure the client

In this section, you create a Node.js project to convert TypeScript to JavaScript and bundle client-side resources, including HTML and CSS, using Webpack.

1. Run the following command in the project root to create a `package.json` file:

```
npm init -y
```

2. Update the content of `package.json` file and save the file changes:

```
{
  "name": "signalrwebpack",
  "version": "1.0.0",
  "private": true,
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

```
Setting the `private` property to `true` prevents package
installation warnings in the next step.
```

3. Install the required npm packages. Run the following command from the project root:

   ```
   npm i -D -E clean-webpack-plugin css-loader html-webpack-plugin mini-css-
   extract-plugin ts-loader typescript webpack webpack-cli
   ```

   The `-E` option disables npm's default behavior of writing semantic versioning range operators to `package.json`. For example, `"webpack": "5.76.1"` is used instead of `"webpack": "^5.76.1"`. This option prevents unintended upgrades to newer package versions.

4. Replace the `scripts` property of `package.json` file with the following code:

```
"scripts": {
  "build": "webpack --mode=development --watch",
  "release": "webpack --mode=production",
  "publish": "npm run release && dotnet publish -c Release"
},
```

```
The following scripts are defined:

-   `build`: Bundles the client-side resources in development mode
    and watches for file changes. The file watcher causes the bundle
    to regenerate each time a project file changes. The `mode`
    option disables production optimizations, such as tree shaking
    and minification. use `build` in development only.
-   `release`: Bundles the client-side resources in production mode.
-   `publish`: Runs the `release` script to bundle the client-side
    resources in production mode. It calls the .NET CLI\'s
    [publish]
    command to publish the app.
```

5. Create a file named `webpack.config.js` in the project root, with the following code:

```
```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
    entry: "./src/index.ts",
    output: {
        path: path.resolve(__dirname, "wwwroot"),
        filename: "[name].[chunkhash].js",
        publicPath: "/",
    },
    resolve: {
        extensions: [".js", ".ts"],
    },
    module: {
        rules: [
            {
                test: /\.ts$/,
                use: "ts-loader",
            },
            {
                test: /\.css$/,
                use: [MiniCssExtractPlugin.loader, "css-loader"],
            },
        ],
    },
    plugins: [
        new CleanWebpackPlugin(),
        new HtmlWebpackPlugin({
            template: "./src/index.html",
        }),
        new MiniCssExtractPlugin({
            filename: "css/[name].[chunkhash].css",
        }),
    ],
```
```

````
};
```

The preceding file configures the Webpack compilation process:

-   The `output` property overrides the default value of `dist`. The
    bundle is instead emitted in the `wwwroot` directory.
-   The `resolve.extensions` array includes `.js` to import the
    SignalR client JavaScript.
````

6. Create the `src` directory in the project root. The `src` directory should contain the following files:

    - `index.html`, which defines the homepage's boilerplate markup:

````
    ```
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <meta charset="utf-8" />
        <title>ASP.NET Core SignalR with TypeScript and Webpack</title>
      </head>
      <body>
        <div id="divMessages" class="messages"></div>
        <div class="input-zone">
          <label id="lblMessage" for="tbMessage">Message:</label>
          <input id="tbMessage" class="input-zone-input" type="text" />
          <button id="btnSend">Send</button>
        </div>
      </body>
    </html>
    ```

-   `css/main.css`, which provides CSS styles for the homepage:

    ```
    *,
    *::before,
    *::after {
      box-sizing: border-box;
    }

    html,
    body {
      margin: 0;
      padding: 0;
    }

    .input-zone {
      align-items: center;
      display: flex;
      flex-direction: row;
      margin: 10px;
````

```
    }

    .input-zone-input {
      flex: 1;
      margin-right: 10px;
    }

    .message-author {
      font-weight: bold;
    }

    .messages {
      border: 1px solid #000;
      margin: 10px;
      max-height: 300px;
      min-height: 300px;
      overflow-y: auto;
      padding: 5px;
    }
    ```

-   `tsconfig.json`, which configures the TypeScript compiler to produce ECMAScriptt:

    ```
    {
      "compilerOptions": {
        "target": "es5"
      }
    }
    ```

-   `index.ts`:

    ```
    import * as signalR from "@microsoft/signalr";
    import "./css/main.css";

    const divMessages: HTMLDivElement = document.querySelector("#divMessages");
    const tbMessage: HTMLInputElement = document.querySelector("#tbMessage");
    const btnSend: HTMLButtonElement = document.querySelector("#btnSend");
    const username = new Date().getTime();

    const connection = new signalR.HubConnectionBuilder()
        .withUrl("/hub")
        .build();

    connection.on("messageReceived", (username: string, message: string) => {
      const m = document.createElement("div");

      m.innerHTML = `<div class="message-author">${username}</div><div>${message}
```

```
</div>`;

      divMessages.appendChild(m);
      divMessages.scrollTop = divMessages.scrollHeight;
    });

    connection.start().catch((err) => document.write(err));

    tbMessage.addEventListener("keyup", (e: KeyboardEvent) => {
      if (e.key === "Enter") {
        send();
      }
    });

    btnSend.addEventListener("click", send);

    function send() {
      connection.send("newMessage", username, tbMessage.value)
        .then(() => (tbMessage.value = ""));
    }
    ```

    The preceding code retrieves references to DOM elements and
    attaches two event handlers:

- `keyup`: Fires when the user types in the `tbMessage`
        textbox and calls the `send` function when the user presses
        the **Enter** key.
- `click`: Fires when the user selects the **Send** button and
        calls `send` function is called.

    The `HubConnectionBuilder` class creates a new builder for
    configuring the server connection. The `withUrl` function
    configures the hub URL.

    SignalR enables the exchange of messages between a client and a
    server. Each message has a specific name. For example, messages
    with the name `messageReceived` can run the logic responsible
    for displaying the new message in the messages zone. Listening
    to a specific message can be done via the `on` function. Any
    number of message names can be listened to. It\'s also possible
    to pass parameters to the message, such as the author\'s name
    and the content of the message received. Once the client
    receives a message, a new `div` element is created with the
    author\'s name and the message content in its `innerHTML`
    attribute. It\'s added to the main `div` element displaying the
    messages.

    Sending a message through the WebSockets connection requires
    calling the `send` method. The method\'s first parameter is the
    message name. The message data inhabits the other parameters. In
    this example, a message identified as `newMessage` is sent to
```
```

```
    the server. The message consists of the username and the user
    input from a text box. If the send works, the text box value is
    cleared.
```

7. Run the following command at the project root:

```
```
npm i @microsoft/signalr @types/node
```

The preceding command installs:

-   The [SignalR TypeScript
    client](https://www.npmjs.com/package/@microsoft/signalr), which
    allows the client to send messages to the server.
-   The TypeScript type definitions for Node.js, which enables
    compile-time checking of Node.js types.
```

Test the app

1. Run Webpack in `release` mode by executing the following command in the project root:

```
```
npm run release
```

This command generates the client-side assets to be served when
running the app. The assets are placed in the `wwwroot` folder.

Webpack completed the following tasks:

-   Purged the contents of the `wwwroot` directory.
-   Converted the TypeScript to JavaScript in a process known as
    *transpilation*.
-   Mangled the generated JavaScript to reduce file size in a
    process known as *minification*.
-   Copied the processed JavaScript, CSS, and HTML files from `src`
    to the `wwwroot` directory.
-   Injected the following elements into the `wwwroot/index.html`
    file:
    -   A `<link>` tag, referencing the `wwwroot/main.<hash>.css`
        file. This tag is placed immediately before the closing
        `</head>` tag.
    -   A `<script>` tag, referencing the minified
        `wwwroot/main.<hash>.js` file. This tag is placed
        immediately before the closing `</body>` tag.
```

2. Build and run the app by executing the following command in the project root:

```
```
dotnet run
```
```

```
The web server starts the app and makes it available on localhost.
```

3. Open a browser to `https://PORT-YOUR_GITPOD_URL.gitpod.io`. The `wwwroot/index.html` file is served. Copy the URL from the address bar.
4. Open another browser instance (any browser). Paste the URL in the address bar.
5. Choose either browser, type something in the **Message** text box, and select the **Send** button. The unique user name and message are displayed on both pages instantly.