Eclipse Scout JS
Beginner's Guide

Scout Team

Version 22.0

Table of Contents

Chapter 1. *Hello World* with Plain Scout JS

In this chapter we will create a *plain* Eclipse Scout JS application, that is, an application that does not require Maven or Java, only Node.js.

When the user opens this application in the browser, there is a text field and a button. Once the user enters some text and presses the button, the application displays a message box including that text.

The goal of this chapter is to provide a first impression of the Scout JS framework. We will start by getting the application running and then take a look at the code.

1.1. Prerequisites

Make sure you have Node.js 12 or higher (with npm 6.9 or higher) installed.

1.2. Get the Code

For now, you have to manually recreate the helloworld folder with all its files or clone the whole docs repository, checkout branch `releases/22.0` and use or copy that folder from your working tree.

Listing 1. Files and folders of the application

```
.
 |   package.json
 |   webpack.config.js
 |
 ├──────res
 |       index.html
 |       texts.json
 |
 └──────src
     |   helloworld.js
     |   helloworld.less
     |
     ├──────desktop
     |       Desktop.js
     |       DesktopModel.js
     |
     └──────greeting
             HelloForm.js
             HelloFormModel.js
```

1.3. Build the Application

In the main folder, where the file `package.json` is located, open a terminal and execute `npm install`.

This creates a folder `node_modules`, containing all (direct and transitive) dependencies, as well as a

file `package-lock.json`, listing all the specific versions of these dependencies.

If the dependencies defined in `package.json` change, run `npm install` again to update the `node_modules` folder.

Now execute **`npm run build:dev`**. This creates a `dist` folder that contains the transpiled and bundled files to be served to the browser.

Use `npm run build:dev:watch` to have these files automatically updated when the corresponding source files change.

1.4. Run the Application

Use the same or start a new terminal in the main folder and execute **`npm run serve:dev:watch`**.

This starts a little development server and opens the URL `http://127.0.0.1:8080/` in your default browser. The server has live reload capability, that is, as soon as files in the `dist` folder change, the browser tab will reload automatically.

Type some text in the field and press the button to test the application. Also check out how the layout changes when you narrow the browser window (or e.g. use Google Chrome's DevTools to emulate a smaller device).

1.5. Understand the Code

Let's now have a closer look at the files that were needed to build this application.

In the main folder there are files containing information for the build, e.g. dependencies and entry points. In the subfolder `res/` there are static resources that are just copied to `dist/res/` in the build. And in the subfolder `src/` you find the source files that are transformed and bundled by webpack.

1.5.1. Build Information

npm

For `npm` commands like `npm install` or `npm run <script>`, the file `package.json` provides the necessary information.

Listing 2. package.json

```
{
  "scripts": {
    "build:dev": "scout-scripts build:dev",
    "build:dev:watch": "scout-scripts build:dev:watch",
    "serve:dev:watch": "live-server --mount=/:dist/res --mount=/:dist/dev"
  },
  "devDependencies": {
    "@eclipse-scout/cli": "^11.0.9",
    "live-server": "^1.2.1"
  },
  "dependencies": {
    "@eclipse-scout/core": "^11.0.9"
  }
}
```

The `scripts` define what `npm run` should execute. They work a bit like aliases in Bash. To have all needed files available at `http://127.0.0.1:8080/`, we need to mount the folders `dist/res` and `dist/dev` to the root path `/` when starting the development server.

Modules defined in `devDependencies` and `dependencies` are downloaded to the `node_modules` folder on `npm install`. The dependency versions are prefixed with a `^` (caret), which means *compatible version*. That is, when running `npm install`, the newest version with the same major-level will be downloaded, unless another compatible version already exists in the `node_modules` folder or is already defined in the `package-lock.json` file.

For more detailed and general information about `package.json` and `package-lock.json`, see the official documentation on Node.js: The package.json guide and The package-lock.json file.

webpack

As defined in `package.json`, the script `build:dev` executes `scout-scripts build:dev`. `scout-scripts` is a command provided by the `@eclipse-scout/cli` module. With the `build:dev` argument, this command uses webpack to transform and bundle the source files and write the results to the `dist/dev` folder.

Scout provides a default webpack configuration which we use and adjust as follows.

Listing 3. webpack.config.js

```
const baseConfig = require('@eclipse-scout/cli/scripts/webpack-defaults');
module.exports = (env, args) => {
  args.resDirArray = ['./res', './node_modules/@eclipse-scout/core/res'];
  const config = baseConfig(env, args);
  config.entry = {
    'helloworld': './src/helloworld.js',
    'helloworld-theme': './src/helloworld.less'
  };
  return config;
};
```

The `args.resDirArray` defines the folders with static resources to be copied to `dist/res`. In addition to the static resources of our application, we also need Scout's static resources in `node_modules/@eclipse-scout/core/res`, mainly for the icon font `scoutIcons.woff`.

In `config.entry`, the entry points for bundling JavaScript and CSS files are defined. For our application, the target files `helloworld.js` and `helloworld-theme.css` (defined without the file extension) are generated from the source files `src/helloworld.js` and `src/helloworld.less`, respectively.

The `-theme` **suffix** of the target CSS file is important for Scout's post-processing to work properly. Also, make sure that you don't use exactly the same name as for the target *JS* file. Other than that, you can name the target files whatever you want, just make sure you also adjust the **references in** `index.html` accordingly (see next section).

For more details on the build, see the *Build Stack* chapter in the technical guide.

1.5.2. Static Resources

For an HTML file to be valid (see The W3C Markup Validation Service), it has to define a `DOCTYPE`, a default language and a title. Furthermore, to allow for responsive web design, we include the `<meta>` viewport element.

Listing 4. res/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello World</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="helloworld-theme.css" />
  </head>
  <body>
    <div class="scout"></div>
    <script src="vendors~helloworld.js"></script>
    <script src="helloworld.js"></script>
  </body>
</html>
```

The `<link>` and `<script>` elements include the CSS and JavaScript files generated by the build. The order of these elements is important. In particular, the `<div>` element with the class "scout" has to be placed before the inclusion of the scripts, since it is used to build the final DOM for our application.

The other file in our `res/` folder, **`texts.json`**, contains all texts used in the Scout core module. In the future, this file should be part of `@eclipse-scout/core` and contain texts for multiple languages, but for now we have to place it next to our own static resources. You can get this file e.g. using Google Chrome's DevTools when loading the Scout JS Widgets application.

1.5.3. Source Files

Entry Points

Listing 5. src/helloworld.js

```
import { scout, App as ScoutApp } from '@eclipse-scout/core';
import { Desktop } from './desktop/Desktop'
import { HelloForm } from './greeting/HelloForm'

scout.addObjectFactories({
    'Desktop': () => new Desktop()
});

window.helloworld = Object.assign(window.helloworld || {}, {
    HelloForm
});

new ScoutApp().init({
    bootstrap: {
        textsUrl: 'texts.json'
    }
});
```

In our main JavaScript file, we import the `scout` namespace object as well as the class `App`. We import `App as ScoutApp` here to make it easier to distinguish it from our own classes, e.g. `Desktop`.

Before we initialize an instance of the Scout application, providing the location of the texts file, we do two other things:

1. Use `scout.addObjectFactories` to register a function (identified by 'Desktop') that provides an instance of our `Desktop` class. The desktop is the main widget of a Scout application and the root of every other widget. On application initialization, Scout is using that factory to create the desktop of our application.
2. Define our own namespace object, `helloworld`, and put our `HelloForm` class in it, so Scout can use it to build modular widgets at runtime (see `DesktopModel.js`).

Listing 6. src/helloworld.less

```
@import "~@eclipse-scout/core/src/index";
```

Since we don't need any custom styling for our application, we just import Scout's LESS module as is in our LESS file.

To try out Scout's dark theme, just import `index-dark` instead of `index`.

Widgets

We follow the best practice of separating model (layout, structure) and behavior code. This also makes it easier to e.g. reuse a form that should look similar elsewhere but behave differently.

A typical model definition for a Scout widget defines an `objectType`. This is specified as a string containing the corresponding class, prefixed with the namespace in which it is accessible. Without a namespace prefix, the namespace `scout` is used as default.

Other object properties are used to configure the widget based on the specified `objectType`.

Listing 7. src/greeting/DesktopModel.js

```
export default () => ({
    objectType: 'Desktop',
    navigationHandleVisible: false,
    navigationVisible: false,
    headerVisible: false,
    views: [
        {
            objectType: 'helloworld.HelloForm'
        }
    ]
});
```

The default desktop consists of a navigation, a header and a bench. We only need the bench for our application, so we hide the other parts, including the handle to toggle the navigation.

A desktop can contain outlines and/or views. We provide an instance of our HelloForm as a view on our desktop.

Listing 8. src/greeting/Desktop.js

```
import { Desktop as ScoutDesktop, models } from '@eclipse-scout/core';
import DesktopModel from './DesktopModel';

export class Desktop extends ScoutDesktop {

    constructor() {
        super();
    }

    _jsonModel() {
        return models.get(DesktopModel);
    }
}
```

Our desktop doesn't have any custom behavior, so we only import the `DesktopModel` here, in the `_jsonModel()` function.

Listing 9. src/greeting/HelloFormModel.js

```
import { GroupBox } from '@eclipse-scout/core';

export default () => ({
    objectType: 'Form',
    displayHint: 'view',
    modal: false,
    rootGroupBox: {
        objectType: 'GroupBox',
        borderDecoration: GroupBox.BorderDecoration.EMPTY,
        fields: [
            {
                id: 'NameField',
                objectType: 'StringField',
                label: 'Name'
            },
            {
                id: 'GreetButton',
                objectType: 'Button',
                label: 'Say Hello',
                keyStroke: 'enter',
                processButton: false
            }
        ]
    }
})
```

Our form is defined to be non-modal and displayed as a view (rather than a dialog). It consists of a string field and a button. These are in a group box inside the form. We define an empty border decoration around this group box to have a little padding.

The *Enter* key is defined as the keyboard shortcut for our button and we set `processButton: false` to place the button next to our field instead of above it.

Listing 10. src/greeting/HelloForm.js

```
import { Form, models, MessageBoxes } from '@eclipse-scout/core';
import HelloFormModel from './HelloFormModel';

export class HelloForm extends Form {

    constructor() {
        super();
    }

    _jsonModel() {
        return models.get(HelloFormModel)
    }

    _init(model) {
        super._init(model);
        this.widget('GreetButton').on('click', event => {
            let name = this.widget('NameField').value || 'stranger';
            MessageBoxes.openOk(this.session.desktop, `Hello ${name}!`);
        });
    }
}
```

As in `Desktop.js`, we import the model but additionally add an event handler in the `_init(model)` function to implement the desired behavior when the button is clicked.

To accomplish this, we can access our button and field by their respective `id` (see `HelloFormModel.js`). An OK message box with the desired text is displayed using the convenience class `MessageBoxes` from Scout.

1.5.4. Git configuration

If you want to add the created application to a Git repository, it is recommended to exclude some files from the SCM.

As a starting point, put a file named `.gitignore` with the following content in the main folder of the application.

Listing 11. .gitignore

```
# Git
*.orig

# Node.js
node_modules/
dist/
test-results/

# Do not check in any log files
*.log
```

See the gitignore Documentation for details.

1.6. What's Next?

Now that you have successfully created your first Scout JS application, you might want to learn more about Scout JS.

To see more example code of Scout JS, we recommend looking at the Scout JS Widgets application and its source code.

If you are interested in Scout's concepts, architecture and features you probably want to have a look a the Scout JS Technical Guide

In case you should get stuck somewhere and need help, try to get answers by searching the web. And if despite reasonable efforts this approach does not help, contact us on the Scout forum. Should you have solved issues on your own, please consider sharing your findings in the forum as this can help other folks too.

We wish you all the best on your journey with Scout.

Do you want to improve this document? Have a look at the sources on GitHub.