Learning Objectives

Learners will be able to...

- **Describe how automation makes the build stage of Software Development.**
- **Create Compile Steps**
- **Automate artifacts creation**

info

Make Sure You Know

Make sure you are familiar with `nodejs` and its basic usage.

Limitations

Familiarity with the content in previous modules in this course required beyond this point.

Introduction

In software development, the **build stage** involves compiling source code into binary code and packaging it for deployment.

This lesson will introduce the concept of **automating the build stage** of software development. Automating this stage makes the process more efficient by removing the need to manually compile source code each time a change is made.

We'll practice our automation using a simple `nodejs` application. Our app is small and doesn't require long compilation times.

Larger projects can take very long time to compile, sometimes hours. To solve this most of the build systems use **partial compilation**, which processes and compiles only the code that has been changed.

A basic `nodejs` project typically contains files including HTML, CSS and JavaScript code. We'll also be using `JSNext`, a javascript standard which is not supported by all browsers.

Webpack

When using `JSNext`, we can use `webpack` to prepare one `.js` file with all these files included and compiled so that it works with modern browsers and is minified for a smaller footprint.

`Webpack` can be configured to **automatically watch the source code** for changes and recompile it when a change is detected. This makes it easier to keep track of code updates while reducing the amount of time needed to manually compile them, thus saving time and effort.

Checkpoint

Development Build

Partial builds are an important tool for streamlining software development builds. Software developers can leverage this capability to save time when building their projects by only compiling files that have been modified during the development process.

To avoid long waiting times, many tools have partial build options with **file watcher**.

File watcher tracks changes made to specified source files and recompiles projects on fly. Tools usually store compiled state for other files in memory that allows us to reduce the amount of resources used during compilation and increase our compilation times.

We can add the following line to our `package.json` file:

```
    "start": "parcel src/index.html"
```

This will allow us to start the development mode using the `npx yarn start` command.

Our application is now running on our localhost at the address `http://localhost:1234/`.

****** to view our deployed application.**

To stop the application, we can press **Control** + **C** `(^+C)`.

Every time a change is made in the source files, a new build will be triggered, and the new version of our project will be quickly and automatically available.

Checkpoint

Production Build

We'll be using the `yarn` package manager to build our project, so we'll need to make sure it's available for use.

Click the button below to install the `yarn` dependency for our project with the `npx yarn` command.

Lets add a `build` section into our `package.json` file. In this file, we have a section labeled `scripts` – this is where we will be adding our new `build` command.

Add the following line to the `scripts` section of our `package.json` file.

```
    "start": "parcel src/index.html",
    "build": "npx parcel build src/index.html"
```

Now we can compile our project by running `npx yarn build` in the terminal.

Click the button below to run the `yarn build` command.

This is an improvement from having to build our project manually. However, after making any changes to our project, we still need to call the compile step manually to do a full project rebuild.

When this step is run, a `dist` folder containing our compiled application will be generated. We can enable it to be viewed using an `html` preview.

****** to preview the application after it has been built.**

From here on out, with each change to any of the source files, the build section in `package.json` will ensure that the project is automatically recompiled without needing manual input.

If we make any change in the file we need to run `npx yarn build` every time and wait for full compilation process. As we mentioned before, for big projects build time can take minutes and sometimes hours.

Checkpoint

Build Github Action

After we have our app's build process working locally, we can create our GitHub Action automation.

- **Navigate to your GitHub repository for this project on GitHub.com**

- **If you haven't already, create a directory in your repository named `.github/workflows/` to hold our `.yaml` configuration file.**

- **In this directory, create a new YAML file named `CI-CD.yml` and copy the following code snippet into the file.**

```
name: CI-CD

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Install dependencies
      run: npx yarn
    - name: Build
      run: npx yarn build
```

In this example, we've defined 3 steps for our GitHub Action to perform when it detects any `push` to our GitHub repository.
1. Checkout the code
- uses: actions/checkout@v3`
2. Install nodejs dependencies and libraries
name: Install dependencies
run: npx yarn
3. Build the code

```
name: Build
run: npx yarn build
```

Now, whenever we push our code to this GitHub repository, GitHub will execute our action and, if successful, create a production version.

Checkpoint

Create artifact

In the previous step, we created a release build, but we didn't save at anywhere. Github actions as most of CI/CD actions are stateless, they do not save the files anywhere unless it is specified as part of steps.

An **artifact** is a deployable resource created as part of a successful build, typically containing compiled objects such as binaries or libraries. By saving our artifacts somewhere, we are able to store a snapshot of your application at any given time, allowing it to be reproduced and deployed if needed.

To create an artifact we need to archive the `dist` folder and upload it to artifactory system.

Using GitHub Actions, we can easily create an artifact by adding an `upload-artifact` step to our workflow file:

Copy and paste the code snippet below into the workflow file.

```
- name: Upload artifact
  uses: actions/upload-artifact@v2
  with:
    name: assets
    path: dist
```

This step archives the contents of the `dist` folder and uploads it to an artifactory system. Now, we can commit and push the updated workflow to GitHub.

After these steps are completed, we can find the compiled project in the Artifacts section of GitHub Action.

Checkpoint