Learning Objectives

Learners will be able to...

- **Describe each stage of software development and deployment.**
- **Run each stage manually.**
- **Describe the output and result of each stage.**

info

Make Sure You Know

Basic `JavaScript` and `nodejs`.

Limitations

The assignment will use `npm` as a package management tool.

CI/CD workflow

The **CI/CD** software development and deployment process is an important part of software engineering. This process uses the principles of **continuous integration (CI)** and **continuous delivery (CD)** to simplify software development, improve quality, and accelerate time to market.

What is a pipeline?

In development, we set up several **stages** that the code needs to go through before it can be sent to the production system. The goal of these **stages** is to reduce the number of errors that get to the end user by checking for errors at each **stage**.

A **CI/CD pipeline** is made up of a series of **stages** that the code goes through before it gets to production.

Stages of the CI/CD Pipeline

Click on the stages of the pipeline below to learn more about each one.

▼ **Code Checkout**

Every software project begins with **code checkout**, which is the process of retrieving source code from a repository's main (or master) branch, local location, or other external sources.

The goal of this stage is to make sure that developers have access to the most recent version of the code before they start working on it.

▼ **Code Quality Check**

After the software code has been checked out, a **quality check** is done. During this part of the process, automated tests are run to look for problems with the code, such as syntax errors, possible bugs, and code formatting.

▼ **Compile Code**

After the software code passes the quality check, it moves on to the **compilation** stage. Here, all source code is compiled into one executable file that can be run by the software's intended users.

▼ **Unit Tests**

Before software can be deployed to production, it must undergo **unit testing**. During this phase, each software module or component is tested independently for any functionality or performance-related defects.

▼ **Packaging**

After all of the software modules have been tested and found to work well, the software is ready to be **packaged**. This means making a software package with all of the software's resources, like code libraries, configuration files, and other supporting materials.

▼ **Acceptance Tests**

Acceptance testing is the final step before deployment. During this phase, software is tested in a setting that is very similar to how it will be used in production. This helps find any possible problems with the software's compatibility or performance in the environment where it will be used.

▼ **Deployment**

After software has passed acceptance tests, it can be **deployed** into production. The CI/CD software development and deployment process makes sure that software is always up-to-date and error-free when it is deployed.

The CI/CD software development and deployment process streamlines software development for software engineers. It enables software teams to quickly identify and resolve issues, resulting in higher product quality and a shorter time-to-market.

Software developers can use the **CI/CD pipeline** to focus on producing better products rather than worrying about long development cycles or costly mistakes. This ultimately leads to increased software project success, faster product launches, and satisfied consumers.

Checkpoint

Build Stage

The **build stage** includes all necessary steps to compile the software into a working program.

Even some of the languages, like *JavaScript*, do not require compilation step, but more complicated scenarios might include compile for older browsers or older version of `nodejs` engine. Also it might include compiling resources like:
- preprocessing `css` and `html` templates
- optimizing images
- minimizing `.js` files, and
- building source maps.

Not all **compile** steps are always necessary for the build in development process and partial compilation can be used instead. In this case, the **build stage** can come in 3 different flavors.

1. Development build

Development Builds are an important part of the build stage because they provide an environment for developers to create their code. The code can be tested for compilation errors and other issues that may arise while writing code at this stage. **Development builds** are typically created anytime modifications to the source code are made in order to thoroughly test them before incorporating them into a new build.

Development builds should be as quick as possible because compiling and optimizing resources during the debugging or prototyping process can be waste of time. Some languages and tools provide partial compilation, which allows us to monitor file changes and recompile only the altered code rather than the entire project. We don’t need to optimize resources like media files at that stage of development because they are primarily loaded locally.

2. Testing build

Testing Builds come after Development Builds have been established and can often be handled by a separate QA team. A **testing build** primarily focuses on making sure that all parts of the application are functioning correctly. This includes running test scripts and manual tests to ensure that all features of the application work as expected.

Testing builds are heavier because they include debug information, which allows you to see more specific information about any errors, including the file and line number. Because we don't conduct as many speed optimizations, the compilation may be faster than the release. Any issues identified during this build need to be fixed before a Release Build can be created.

3. Release build

Release Builds involve collecting all parts of the code, such as compiled binaries, configuration files, and any other resources needed by the application together into a deployable package to be delivered to production system. The package is then optimized for size, usability, and performance on its target platform before eventually being pushed out to production environments. This also may include media files optimization to reduce size or support better client-side caching.

The **Build Stage** of CI/CD software development is critical for ensuring that apps work properly while also allowing for faster releases with less effort. Understanding these builds is essential for effective and efficient software development.

Checkpoint

Manual Build

Webpack is a powerful tool for building and bundling Node.js applications by allowing us to bundle all of our source code and dependencies into a single file that can be added to your website or app. This makes it easier to share, install, and keep up with the application.

Let’s explore how to manually build a Node.js app using webpack.

- **Check the working path in the terminal using the `pwd` command to ensure we’re located in the `/buildProject` folder.**

```
/home/codio/workspace/buildProject
```

First, we’ll setup our `/buildProject` directory as an `npm` project

- **Run the command below to initialize the current project directory, `/buildProject`, as an npm project.**

```
npm init -y
```

NPM uses a file named `package.json` to manage dependencies and tasks.

- **Take a look at the `package.json` file to the left.**

Notice that it holds all of the important information about our application including:
- Version number
- Name
- Dependencies

- **Run the command below to install `webpack` locally into our project and install the `webpack-cli`.**

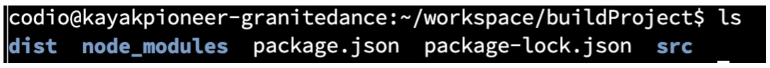
```
npm install webpack webpack-cli --save-dev
```

It’s common for software development projects to have different types of builds for development, testing, and release.

- **Use the `ls` command in the terminal to list the contents of the**

`/buildProject` folder.

Notice this project includes an `/src` and a `/dist` directory to separate the **development** and **release** builds, respectively.



```
codio@kayakpioneer-granitedance:~/workspace/buildProject$ ls
dist  node_modules  package.json  package-lock.json  src
```

Screenshot of terminal output showing the contents of the /buildProject directory

The build step is where all of our dependencies and scripts are bundled together into a runnable application.

- **Click the button below to bundle all necessities for our project by installing the `lodash` library locally.**

At this point, we can run the `npx webpack` command to manually build our project. This command will use ‘src/index.js’ as the starting point and generate ‘dist/main.js’ as the resulting output.

- **Click the button below to run the build process using `webpack`.**

The destination of our build is the `dist/index.html` inside our `buildProject`. We should be able to view our build to `index.html` in the browser by ****.

Checkpoint

Quality Check

Code Quality Standards

Quality is key when it comes to creating successful software. **Quality check**, or **Quality Assurance (QA)** is a crucial step in the CI/CD pipeline where we can use **linting** to help us keep our code clean and error-free by spotting unused variables, unreachable parts of the code, and preventing usage of non-memory safe commands.

Linting involves running automated checks on the source code of programs and calling out any potential issues or errors that could arise during runtime. **Linting rules** are like standards for how you should write your code – making sure it's consistent, with correct syntax and few bugs.

If you're working in a team or a company, it's very common to have set of code rules to make the code more readable with fewer mistakes. Even so, **linting** is a great habit, even if you're the only one working on a project.

Linting helps us:

- Keep the code clean by spotting unused variables or unreachable parts of the code.
- Keep the code safe by preventing usage of non memory safe commands.
- Maintain the same code style through the project with the same code formatting rules.

Usage

Compared to the **Development build** we wouldn't need to run a linting and code check with every codebase change. This stage is used mostly to check that our pull requests are compliant with the standards, helping to flag any potential problems before they become a larger issue.

Checkpoint

Linting

JSLint is a great tool for helping us issues with your code quickly and easily, ensuring that it meets coding standards and best practices. JSLint can be installed globally through npm by running the following command in the terminal:

- **Use the command below to install `jslint`.**

```
sudo npm i -g jslint
```

Now, we can run `jslint` on a file of our choosing. Let’s check `src/simpleCalc.js`.

```
jslint src/simpleCalc.js
```

JSLint gives us feedback for every line where it detects a syntax or stylistic error and gives us suggestions on how to fix the problem. This step will not pass successfully until all of the linting suggestions have been addressed.

Checkpoint

Testing

Testing frameworks

Software testing is an essential part of software development and ensures that a product meets quality standards. Before the code reaches the customer, **testing helps detect any errors, bugs, or security issues**.

Software testing frameworks are classified into two types: **unit testing** and **integration testing**.

Unit testing

Unit testing is a type of software testing that **looks at individual parts or units of a software system**, isolating them from other parts of the application, to make sure they work as they should.

Unit Testing focuses on the internal structure and behavior of the program. It usually involves writing test cases that exercise different parts of the code and then checking whether these tests pass or fail.

Unit tests are automated and **run each time code is changed** to ensure that new code does not break existing functionality, allowing us to minimize execution time and isolate the testing use-cases from external factors.

Integration testing

Integration Testing involves **combining different parts of a software application or system**, and assessing how they interact with each other. Testing is performed on interfaces between components, as well as data exchange between components.

Integration testing should include both functional and non-functional tests, to ensure that the application meets all requirements in terms of performance, usability, security, etc.

A comprehensive software testing process should include both unit and integration testing.

Unit testing should be done first because it is the most detailed type of testing and allows developers to identify errors within individual code components.

Then, **integration testing** should be performed to catch any issues that may arise when components interact with one another.

This combination of unit and integration testing contributes to the end product being bug-free and of high quality.

Checkpoint

Manual Testing

Jest is a JavaScript testing framework used to test React applications and other JavaScript codebases. Jest is fast, reliable, and easy to use, with a large collection of built-in tools that help developers write robust test cases quickly and efficiently.

NPM gives us an easy way to install for use in local development or as part of the CI/CD process with the `npm install -g jest` command.

- **Click the button below to install `jest` for our `testProject`.**

Let’s take a look at our sample project to the left.

We have a test file, `simpleCalc.test.js`, that defines some tests for our app, `simpleCalc.js`.

At the top of our test file, we **require** the file we want to test, `/src/simpleCalc`.

```
const mathOperations = require('../src/simpleCalc');
```

This gives our test file access to the functions present in our application so we can use the functions for our various tests.

Let’s look at one of the tests we’ve defined.

```
test("Addition of 2 numbers", () => {  //defines a test
"Addition...

var result = mathOperations.sum(1,2)  // defines a variable,
result, and plugs 1 and 2 the sum function from simpleCalc

// assert
expect(result).toBe(3);  // we assume if we put 1 and 2 into the
sum function, the result will be 3.
});
```

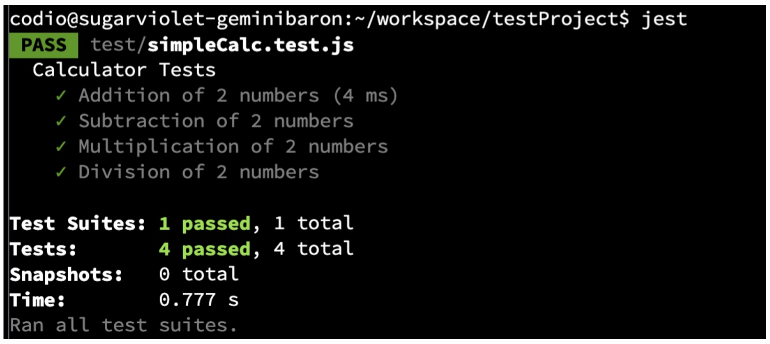
This test does a few things:
- Defines a test named `"Addition of 2 numbers"`
- Defines a variable named `result` and plugs 1 and 2 into the `sum` function from our `simpleCalc.js` app.
- Makes an **assertion**, or assumption, that if we call the `sum` function with

the values we've defined (1 and 2), that the value of the `result` variable will be 3.
- If this is not the case, our test fails and we can assume there's something wrong with our function.

Now, we can run these tests with `jest` to see how our functions are working in our app.

Click the button below to run our unit tests for our project with the `jest` command.

Jest gives us feedback on how many tests we've run and whether each test has passed or failed.



```
codio@sugarviolet-geminibaron:~/workspace/testProject$ jest
 PASS  test/simpleCalc.test.js
  Calculator Tests
    ✓ Addition of 2 numbers (4 ms)
    ✓ Subtraction of 2 numbers
    ✓ Multiplication of 2 numbers
    ✓ Division of 2 numbers

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        0.777 s
Ran all test suites.
```

Terminal output of jest command results

Using this framework, we can easily run tests on individual parts of our application to make sure they work well together, as well as alone.

Checkpoint

Packaging & Deployment

The final steps in the CI/CD pipeline are the **packaging** and **deployment** of the code to either a production or testing system. The goal of packaging and deploying code is to ensure that applications are always secure and up to date, allowing users to access the most recent versions of an application.

Packaging

Packaging is the process of converting installed files into a single file format that can be distributed, executed, and deployed across different systems. This allows software developers to store, or archive, their applications in an artifactory or repository while making them accessible to multiple users.

Packaging also helps us keep applications current and easily accessible for rollback or subsequent deployment, saving us time spent on manual updates. Packages can be distributed in a variety of formats, including `tar`, `zip`, `rpm`, and `deb` files.

Deployment

Deployment is the process of moving an application from its development environment to a production environment. This includes storing the packaged file in an appropriate repository and ensuring all required components are installed correctly. **Deployment** also typically includes running automated tests to ensure the code meets certain quality standards before being deployed into production.

Deployment differs depending on the delivery system; tools such as Kubernetes containers enable automated deployment with features such as version tracking, knowing which version is currently running, and performing rollbacks. After deployment, we can keep an eye on performance metrics and fix any problems that may arise.

Packaging and **deployment** are essential parts of the CI/CD pipeline because they ensure that code is thoroughly tested before making it available to end users. Developers can rest assured that their applications are safe and secure in production systems by using CI/CD tools such as Kubernetes containers or automated tests.

Checkpoint

Manual Deployment

Let's explore what it looks like to deploy an application manually.

Navigate to the `deployProject` directory using the `cd` command in the terminal.

```
cd deployProject
```

Now, we'll setup our `/deployProject` directory as an `npm` project

- **Run the command below to initialize the current project directory, `/deployProject`, as an `npm` project.**

```
npm init -y
```

We can make sure we're in the correct directory by using the `pwd` command.

Express

Express is a powerful Node.js web application framework that lets developers build and deploy static and dynamic web apps and APIs with little code. Express middleware responds to client requests, making routing and authentication easy.

Now, we can install Express in our working directory.

Copy and run the command below in the terminal to install express.

```
npm install express
```

Let's take a closer look at our simple application.

We can see in our `index.html` file, to the left, that our application simply displays a success message.

There is also an `app.js` file in our project that sets up a localhost server for our application and displays our `index.html` file to the server.

We can use the `node` command to launch our server and deploy our application.

Run the command below in the terminal to launch our server.

```
node app.js
```

Now, we should be able to view our deployed application at the location of our localhost.

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

To end the deployment, you can press **Control(^) + C**

Checkpoint