

Learning Objectives

Learners will be able to...

- Explain task automation
- Automate tasks using npm
- Understand tasks outputs

info

Make Sure You Know

Basic JavaScript and nodejs.

Limitations

The assignment will use npm as a package management tool.

Introduction

In the previous assignment we learned about the main steps used in software development process. In this assignment we will go a bit further trying to automate some operations.

Task automation can play an important role in streamlining the development of software applications by eliminating tedious manual tasks, making them automatic, and reducing errors caused by human error. By automating certain tasks, developers can save time on routine tasks and ensure that code quality remains consistent.

NPM (Node Package Manager) is an open-source package manager that simplifies the installation and management of application dependencies. NPM allows developers to quickly install packages from their registry, ensuring all dependencies are up-to-date and required libraries are available whenever needed. Using npm, developers can also create custom scripts or tasks for automation purposes.

For these lessons, we'll use npm to create tasks in our package manager, perform the same operation using package manager, and create dependencies.

Automation

Task automation includes **using tools or scripts to manage specific portions of the software development lifecycle** so that developers do not have to conduct them manually. **Task Automation** is an excellent way to save time, eliminate errors during the development process, and promote team collaboration.

When considering when to use automation, think about:

- How much effort would be required to do the task manually against how long it will take to automate it.
- If there are tasks that need to be performed over and over again during a process.

If automating something makes your process **faster** and more **efficient**, it's usually worth the effort.

A few examples of tasks we can automate include:

1. **Testing** - Detecting mistakes early in the development cycle saves developers time and improves product quality before it goes live. Software testing frameworks such as **JUnit** or **Selenium** can assist you in quickly and efficiently creating test cases for your codebase.
2. **Code Check** - Software analysis tools can be used to inspect code for security flaws, coding practices, and performance problems. By automating these tests, you can verify that the code you're producing is of good quality before releasing it to the public.
3. **Dependency Management** - To reduce development time, software developers frequently use third-party libraries or frameworks. Package managers (e.g., **NPM**) can help you keep track of the versions of these dependencies you have installed and ensure they are updated on a regular basis for maximum compatibility with your application.
4. **Deployment** - Automatic deployment methods enable developers to swiftly release code changes into production without having to manually configure servers with each update. By automating this process, teams can focus on producing code, rather than dealing with the difficulties of setting up production environments.

Automation is a great consideration for any project because it is a simple approach to ensure that your software is well-tested and up-to-date before going live.

Checkpoint

Create a task

We can automate many tasks for our application by using `npm`, allowing us to run tasks quickly and without having to type out the code by hand every time.

NPM uses a file named `package.json` to manage dependencies and tasks. To automate a task with `npm`, we need to open the `package.json` file for our project. This is where we will write all of our **tasks**, or commands we want the package manager to execute.

Let's define a simple style check tasks

In the `scripts` section of our `package.json` file, we'll add a command to let `npm` know how to execute what we request.

Add a task named `style` that runs the `eslint` program to perform a style check on our `src/simpleCalc.js` file.

```
"style": "eslint src/simpleCalc.js"
```

Once the command is written into our `package.json` file and saved, we can run the automation task for your project in an open terminal using the following syntax.

```
npm run <name_of_task>
```

When running this command, we'd replace `<name_of_task>` with the name we defined for the task when writing out the command, in this case `style`. **NPM** will then run the command we gave it, automating the task.

```
npm run style
```

Checkpoint

Output

NPM provides us with **output** that indicates whether a task has failed or succeeded. Understanding this output is key to finding mistakes in our code, so it's important to understand what `npm` is telling us.

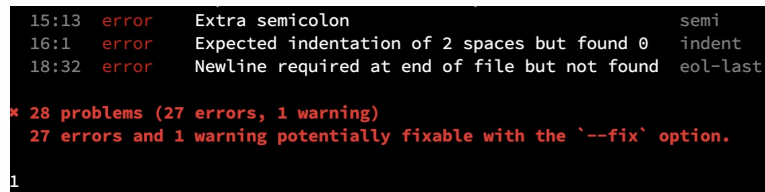
The **exit code** of the task defines success or error. If the result of the script is 0 then the task was completed successfully. Otherwise, `npm` will return an exit code of 1, indicating an error which can prevent the execution of next tasks.

Let's take a look at an `npm` task failure.

Run the following command in the terminal.

```
npm run style; echo $?
```

We expect this task to fail. Our output should look similar to the output below.



```
15:13 error Extra semicolon semi
16:1 error Expected indentation of 2 spaces but found 0 indent
18:32 error Newline required at end of file but not found eol-last

* 28 problems (27 errors, 1 warning)
  27 errors and 1 warning potentially fixable with the '--fix' option.

1
```

`.guides/img/styleFail`

In this example, `npm` has returned an exit code of 1, indicating that the task has failed as well as additional information to locate the problem, such as:

- **Line Number & Position** 15:13
- **Error Message:** error
- **Additional Information:** Extra Semicolon

To fix more substantial errors, `npm` provides guidance on how to resolve them or where to look for help in its documentation guide. We're also provided with a debugging log, `npm-debug.log`, with more detailed information on the task error.

Once we have fixed all issues, `npm` will indicate success by returning an exit code of 0.

By understanding `npm` outputs, we can accurately detect when errors occur and quickly resolve them to ensure that your tasks are properly automated.

NPM provides a wealth of information about its commands and how to use them, so make sure to check their **documentation** for any additional help or guidance needed.

Checkpoint

Dependencies

Using `npm`, it is possible to create dependencies between tasks. A **dependency** expresses a relationship between two tasks, defining that one cannot be performed until the other has been completed.

In `npm`, this is often expressed as “*run X after Y*”. For example, running `npm style` might require that `npm build` must be run before it in order to ensure successful completion of the overall task.

We can define task **dependencies** in the `script` section of our `package.json` file. If we wanted to run `npm build` before `npm style`, we would add the following code to your `package.json`:

```
"release": "npm build && npm style"
```

This code specifies that `npm build` be executed first, followed by `npm style`. This ensures that the `npm style` task will not begin until the `npm build` has been properly completed.

Using this approach, `npm` can assist prevent bad releases from reaching production by ensuring that all tasks have been performed correctly prior to executing the release or deploy steps.

You'll often find that in many projects, style and unit tests should be pass successfully before release and deploy stages are executed.

Checkpoint