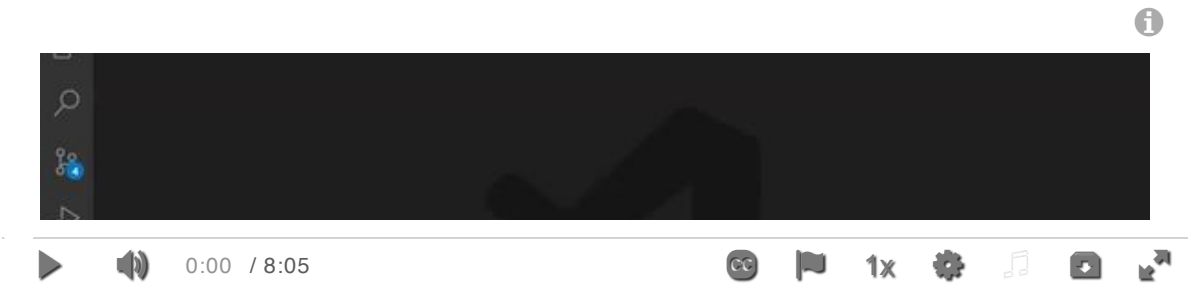
Exploration — Using npm and Express.js to Create Web Applications

Introduction

In this exploration, we will delve into web development by writing our first simple web application using Node.js and Express.js which is a framework that provides a web server and an API for developing web applications. We will also learn about `npm` and how to use it to install packages for our Node programs.

npm: A Package Manager for Node

> **Note:** The following video uses the `touch` command to create a new file. On Windows this command will work only when you have installed certain utilities (which is not required for this course). If `touch` didn't work, on Windows PowerShell you can create a new file, say with name `server.js`, using the command `New-Item -ItemType file server.js`. You can also create new files using VS Code or the file explorer on your OS.



In any substantial program, we don't write everything from scratch. Instead, we look for relevant libraries or packages for the task at hand, and if we find such packages, we use them in our program. Node.js comes with very large number of packages and we will make extensive use of such packages. To install and manage these packages for our Node applications, we will use `npm`. As described on the **Node.js website (https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/)**, `npm` is two things:

- It is online repository for publishing Node.js packages. We can search for `npm` packages on the **Node package manager website (https://www.npmjs.com/)**.
- It is a command-line utility that can install packages from the online repository and manage the dependencies we define for our projects.

Using npm

`npm` is automatically installed with Node, so if you have installed Node you already have `npm`. The complete list of `npm` commands are available **in the npm Docs (https://docs.npmjs.com/cli/v7/commands)**. Here are some common `npm` commands that we will use:

Command	Description
`npm init`	Creates a `package.json` file to initialize a Node.js application.
`npm install <package>`	Installs the specified Node.js package.
`npm start`	Starts a Node.js application.

npm init

This command (https://docs.npmjs.com/cli/v7/commands/npm-init) is used to initialize a new Node.js application. To create and initialize a new Node.js application, do the following:

1. Create a new directory which servers as the **root directory** for the project. This directory is also called the **project directory**, or **app directory**, or **project root**. By default, the name of our project will be the same as the name of this directory.
2. `cd` into the project directory.
3. Run the command `npm init`.

We will be asked a series of questions. For now, we can just hit enter to choose the defaults. At the conclusion of the command, `npm` creates a file `package.json` which defines the properties of our Node.js application.

Note: If we run the command `npm init -y`, `npm` will create the file `package.json` with the default values without prompting us to make any choices.

Example

Create a directory `zip_lookup`, go to this directory, run the command `npm init`, and accept the default values. We see that `npm init` created a `package.json` file, whose contents are shown below:

```
PS C:\Users\nauman\290\code_m3\zip_lookup> cat .\package.json
{
        "name": "zip_lookup",
        "version": "1.0.0",
        "description": "",
        "main": "index.js",
        "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
        },
        "author": "",
        "license": "ISC"
}
```

Note that the property `name` in `package.json` is the same as the name of the project directory. This is because we choose the default values. When `npm init` prompts us for a name, if we specify a different value, `package.json` will set the property `name` to that value instead of using the name of the project directory.

npm install

The command **npm install <package> (https://docs.npmjs.com/cli/v7/commands/npm-install)** is used to install a package. It downloads the package from the `npm` online repository and places it in a directory called `node_modules`. `npm install` looks at the transitive dependency information of the package we asked it to install, and downloads all the other packages that the package depends on. By default, the command updates `package.json` and adds the package we asked to install in a property `dependencies` in the file. We can also force this behavior of updating `package.json`, if we run this command with the option `--save`,

Example

Running the command `npm install cities --save` created a directory `node_modules` under `zip_lookup` directory and downloaded the needed Node.js packages into that `node_modules` directory. Note that even though we asked `npm` to install only 1 package, i.e., `cities`, `npm` installed 3 packages. This happened because `npm` transitively looked through the dependency chain for `cities` and the packages `cities` depends on to find all the packages that needed to be downloaded.

```
PS C:\Users\nauman\cs290\code_m3\zip_lookup> npm install cities --save

added 3 packages, and audited 4 packages in 3s

found 0 vulnerabilities
```

Since we specified the `--save` option, `npm` updated the file `package.json`. A new property `dependencies` was added to the file and with the object `"cities": "^2.0.0"` as its value.

```
{
        "name": "zip_lookup",
        "version": "1.0.0",
        "description": "",
        "main": "index.js",
        "scripts": {
                "test": "echo \"Error: no test specified\" && exit 1",
                "start": "node index.js"
        },
        "author": "",
        "license": "ISC",
        "dependencies": {
                "cities": "^2.0.0"
        }
}
```

npm install with no arguments

If we simply run the command **npm install (https://docs.npmjs.com/cli/v7/commands/npm-install)** without any arguments and we are in the directory where `package.json` is located, then `npm` reads `package.json`, downloads the packages listed in `package.json` from the online repository, and places them in the directory `node_modules`.

A very common use of `npm install` without any arguments is when someone has already created a `package.json` file with all the needed dependencies (and the relevant start command that we will discuss later), and provides this `package.json` file to us. In this case, we do not need to run `npm init` or `npm install <package_name>`. We can simply run `npm install` to download the dependencies listed in `package.json`.

package.json and package-lock.json

The file **package.json (https://nodejs.org/en/knowledge/getting-started/npm/what-is-the-file-package-json/)** contains a JSON object and holds meta-data related to our application. We have already discussed **the property name (https://docs.npmjs.com/cli/v7/configuring-npm/package-json#name)**. Let's look at **the property dependencies (https://docs.npmjs.com/cli/v7/configuring-npm/package-json#dependencies)**. We use this property to specify the packages that our application depends on.

Along with the name of the package, we can specify a version we want to use, or even a version range. In our example, the version of the package `cities` is `^2.0.0`. This style of version numbers is called **semantic versioning** and has the format MAJOR_VERSION.MINOR_VERSION.PATCH_VERSION. The caret symbol at the start `^` means that `npm` can download any version `2.x.x`.

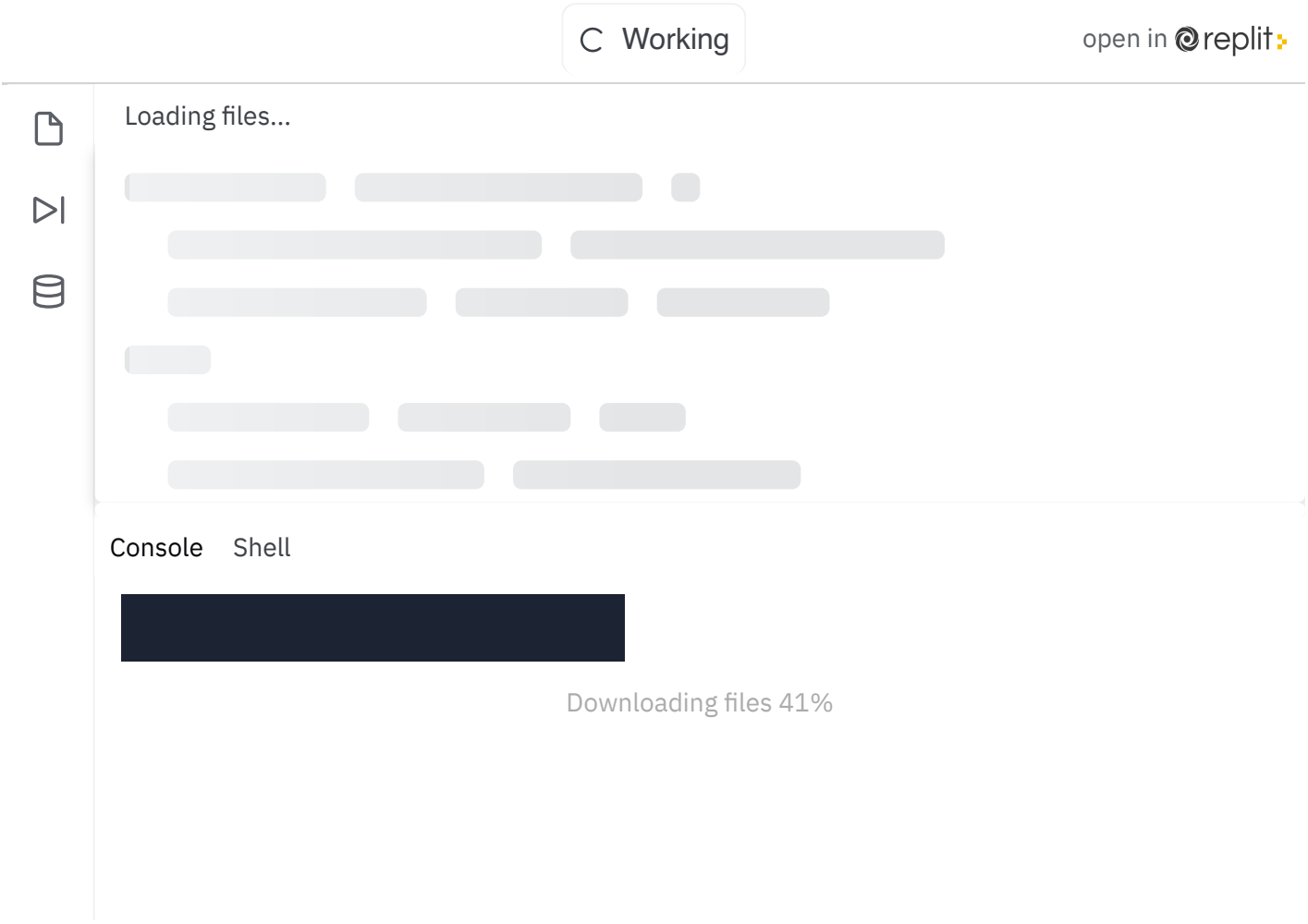
You may have noticed that a file `package-lock.json` has also been created by `npm install`. This file contains information about all the Node.js packages that were installed with the exact version of

each package.

If you are interested in more information about `package.json`, `package-lock.json`, or semantic versions, see the references provided in the Additional Resources section of this exploration. However, reading any of those references is completely optional.

A Simple Node Application

Let us now write some code for a simple Node application. We create a file `index.js` with the following content:

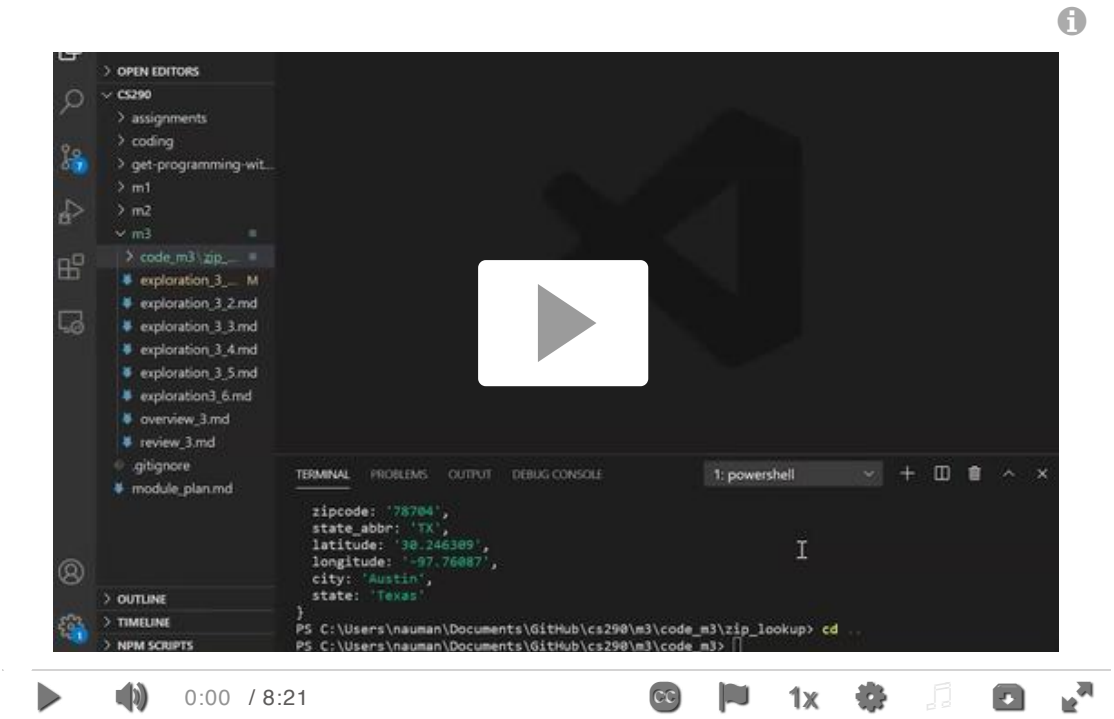


Using require to import packages

In this program, we use **the built-in Node function require() (https://nodejs.org/en/knowledge/getting-started/what-is-require/)** to import **the package `cities` (https://www.npmjs.com/package/cities)** and then call the function `zip_lookup` which is defined in this package. We can run this program using the command `node index.js`.

Using Express.js to Build Web Applications

A number of frameworks have been developed for build web applications using Node.js. In this course, we will use the widely used web framework Express.js, which is also simply referred to as Express. Express.js provides APIs for various common tasks that web applications need to do, e.g., handling requests, generating responses, serving out static files, etc.



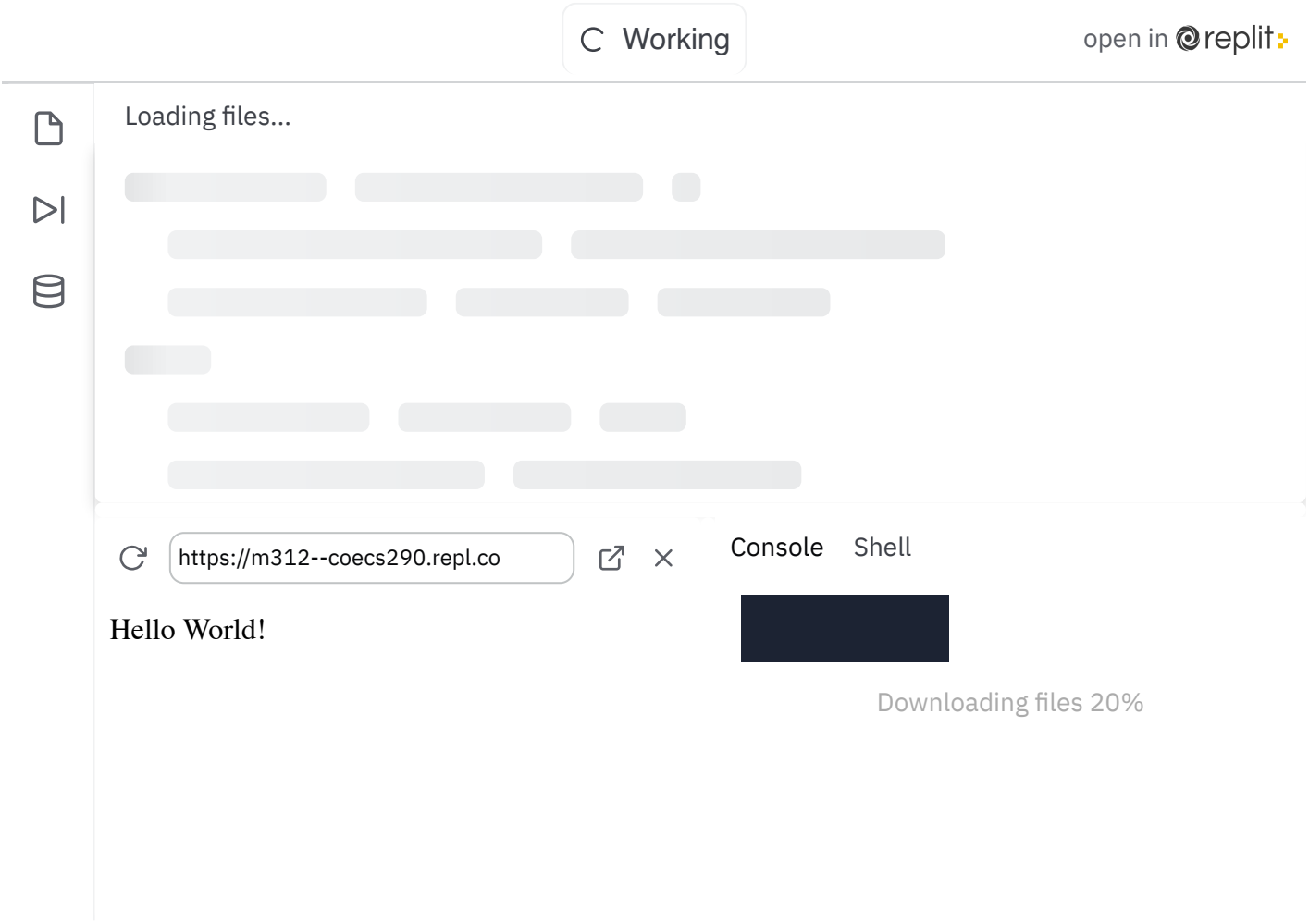
"Hello World" Web Application

> To create and run this app follow along with the video. We recommend that you type the code for `server.js` while following the video. But if you want, you can instead copy the code for `server.js` from the replit shown below (in the replit, the file is named `index.js`, instead of `server.js`, This is because replit website requires that name. However, as you will see, in the video the file will be named `server.js`).

Let's create a new Node.js application in the directory `hello_express`, initialize it using `npm init` and install `express` using `npm install`. We create a file `server.js` which will serve as the main entry point for our program. This requires us to update the property `main` in `package.json` and set its value to `server.js`, i.e.,

```
"main": "server.js",
```

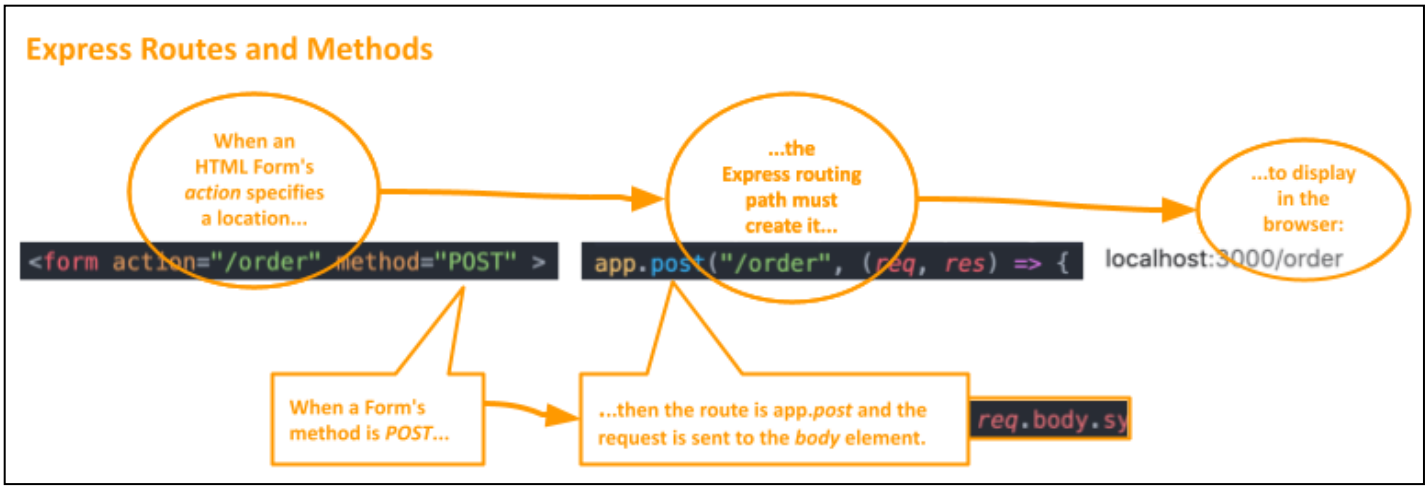
Here is the code of our `server.js` file:



In this program call the **function express() (https://expressjs.com/en/4x/api.html#express)** to create an Express application. Towards the bottom, we call the **function listen (https://expressjs.com/en/4x/api.html#app.listen)** on this application to start our server and start listening for HTTP requests on the specified port.

Setting up routes

The term **routing** means how an application is set to map HTTP requests to resources, or more precisely to map HTTP requests to **endpoints**. The term endpoint refers to the combination of a URL with an HTTP method. For example, a request to the URL `/foo` with HTTP method `GET` is one endpoint and a request to the URL `/foo` with HTTP method `POST` is a different endpoint.



In Express, we define routes using the `app` API. We define routes with the `app` API using the following structure:

```
app.method(URL, handler function)
```

Note that there is no particular function named `app.method`. Instead, the API provides functions for various HTTP methods, such as `app.get`, `app.post` and so on. We define a route for an HTTP request with a `GET` method by using `app.get`, for an HTTP request with `POST` method by using `app.post`, and so on. The URL for the route is specified as the parameter `URL` and the function that should be invoked for this route is provided as the 2nd argument to `app.method`.

As a concrete example, we define a route for handling `GET` requests to the URL `/` as follows:

```
app.get("/", (req, res) => {
        res.send("Hello World!");
      });
```

In this case, our handler function is an anonymous function defined using the arrow syntax. This function takes two parameters, a `req` object which corresponds to the HTTP request and a `res` object which corresponds to the HTTP response. In the above example, we call **the method send (https://expressjs.com/en/4x/api.html#res.send)** on the response object `res` with the string value `Hello World`. Whenever the argument to `send` is a string, Express automatically sets the `Content-Type` header to `text/html` in the HTTP response. If we start the program using `node server.js`, then in a browser enter the URL `http://localhost:3000/`, our program will send back in a HTML document with the string `Hello World!` in the body of the document.

If we wanted to call the same handler function for `POST` requests to the URL `/`, we would add the following code to our program:

```
app.post("/", (req, res) => {
        res.send("Hello World!");
      });
```

npm start

The **command npm start (https://docs.npmjs.com/cli/v7/commands/npm-start)** is a standard way to start a Node.js application. This command looks at the `start` property within the `scripts` property in `package.json`. It runs whichever command is provided as the value of the `start` property. For example, if we update the `scripts` property in `package.json` as follows, then `npm start` will run the command `node server.js`.

```
"scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "node server.js"
      },
```

`npm` also provides a **stop command (https://docs.npmjs.com/cli/v7/commands/npm-stop)** that requires a bit of configuration. For our purposes, we can stop an application by using `Ctrl-C`.

Using nodemon for automatic restart of application

Let's look at one more tool to improve our development process. We can use a package `nodemon` which automatically restarts a node application when it detects file changes in the directory. To use `nodemon` install it using `npm install` and then change the `start` command in `package.json` to use `nodemon` instead of `node`. Here is our complete `package.json` file after the changes for using `nodemon` have been done.

```
{
        "name": "hello_express",
        "version": "1.0.0",
        "description": "",
        "main": "server.js",
        "scripts": {
                "test": "echo \"Error: no test specified\" && exit 1",
                "start": "nodemon server.js"
        },
        "author": "",
        "license": "ISC",
        "dependencies": {
                "express": "^4.17.1",
                "nodemon": "^2.0.7"
        }
}
```

Modules vs Packages

You might have noticed that we have been using the term packages for the dependencies we installed from the `npm` online repository, e.g., `express`, `cities`. But `npm install` downloads these dependencies to a directory named `node_modules`. There is a difference between **the terms module and package (https://docs.npmjs.com/about-packages-and-modules)**. Mostly a module consists of just one file, and a package is used to bundle one or modules together. The two terms

are many times used interchangeably and we don't need to worry about distinguishing between them.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Homepage of **npm** (https://www.npmjs.com/). We can search for packages on this page.
- The complete list of `npm` commands is available in the **npm Docs** (https://docs.npmjs.com/cli/v7/commands).
- Detail info on **package.json** (https://docs.npmjs.com/cli/v7/configuring-npm/package-json) and on **package-lock.json** (https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json).
- A general reference on **semantic versioning** (https://semver.org/).
- Homepage of **Express.js** (https://expressjs.com/). The API docs are particularly useful for us.
- A comprehensive list of Node frameworks is available from the **Node Frameworks site** (http://nodeframework.com/). In addition to Express.js, other popular Node.js frameworks include Koa.js, Hapi.js, Sails.js, and Total.js.