# Getting ready to develop HTML5 games with Phaser and TypeScript

## Node.js, npm, webpack, TypeScript and Vite explained in detail to get you started in HTML5 game development

Emanuele Feronato

# Getting ready to develop HTML5 games with Phaser and TypeScript

written by Emanuele Feronato

https://triqui.itch.io
https://twitter.com/triqui
https://triqui.gumroad.com/
https://www.emanueleferonato.com
https://www.facebook.com/emanueleferonato

# The boring but necessary foreword

Often, when I talk to programmers or read about their experiences, I find that many of them would like to develop HTML5 cross platform games, but are held back by an apparent technical difficulty regarding the use of tools such as Node.js, npm, web servers, package builders and so on.

We've all been there, I've been there.

I always wondered why, if in the end HTML5 games are nothing more than simple web pages, I couldn’t just open a **<script>** tag in my web page and start writing my own game.

Actually, no one is stopping you from writing your game that way, but you would miss several opportunities to make your game development faster and more streamlined.

The lack of detailed tutorials taking you by the hand and explain from scratch, step by step, everything you need to do to turn your computer into an HTML5 game creation workstation, is what motivated me to write this guide.

The important thing is that by the end, terms like Node, npm, web server and other devilry, will no longer scare you.

And, believe me, you'll wonder how you've been able to live without these wonderful tools for so long.

I will make sure that this guide will be always up to date, and since I hate boring theory, let's get started right away.

## Some conventions used in this book

In this book I often create files and folders in various paths, and write code using various variable names.

These paths and names chosen by me, and obviously you are free to use your own paths and names; just keep in mind that if you use your own custom names, then you will have to replace mine everywhere in your projects.

For example, if I called a folder **src**, you are free to call it **source** or anything else.

However, afterwards, every time you meet **src**, you will have to remember that you called it your own way and adapt your code, scripts and actions.

Commands to be entered in a Terminal or command line shell are shown this way:

```
I am a command to be entered in a terminal window
```

And source code is shown this way:

```
I am a line of source code
I am another line of source code, and I want to call your attention.
```

Along with this book, you will find a folder called **source code examples** with the source codes listed in these pages.

They are located in subfolders with the same name as chapter name.

Finally, images may be a little different from what you will see on your computer, depending on your operating system and the version of various software installed.

# What is Node.js?

Node.js is an open-source, server-side JavaScript runtime environment that allows developers to build and run JavaScript applications outside of a web browser.

Created by Ryan Dahl in 2009, has become a popular choice for server-side and networking applications.

It uses the V8 JavaScript engine, which is also used in Google Chrome, to execute JavaScript code on the server-side.

Traditionally, JavaScript was primarily used for client-side scripting in web browsers.

Node.js extends the capabilities of JavaScript by enabling it to be used for server-side programming as well.

This means that developers can write server-side applications using JavaScript, leveraging their existing knowledge of the language.

The top three Node.js key characteristics for our purposes are:

**Cross-Platform**: Node.js runs on various operating systems, including Windows, macOS, and Linux. This ensures that applications developed with Node.js are portable and can run consistently across different environments.

**Community and Modules**: Node.js has a vibrant and active community. The npm registry hosts a vast collection of modules, enabling developers to extend the functionality of their applications easily. This community-driven ecosystem contributes to the growth and evolution of Node.js.

**JavaScript Runtime**: Node.js allows the execution of JavaScript code outside of a web browser. It brings JavaScript to the server-side, enabling developers to use a single language for both client-side and server-side development.

## What is npm?

npm, or Node Package Manager, is a central component of the Node.js ecosystem and serves as the default package manager for Node.js.

Created to simplify the management of JavaScript packages and dependencies, npm plays a crucial role in modern web development.

npm allows developers to easily install, manage, and share third-party libraries, tools, and other pieces of code that can be used in their projects.

There are four npm key characteristics useful for our purposes:

**Package Management**: npm facilitates the installation, sharing, and management of JavaScript packages or modules. A package typically includes reusable code, libraries, and tools that developers can easily integrate into their projects.

**Dependency Resolution**: managing dependencies is a critical aspect of software development. npm automatically resolves and installs dependencies required by a project, streamlining the process of ensuring that the correct versions of libraries are used.

**Registry**: the npm registry is a massive collection of public and private packages that developers can leverage in their projects. The public registry is hosted at https://registry.npmjs.org/, and contains a vast array of open-source packages.

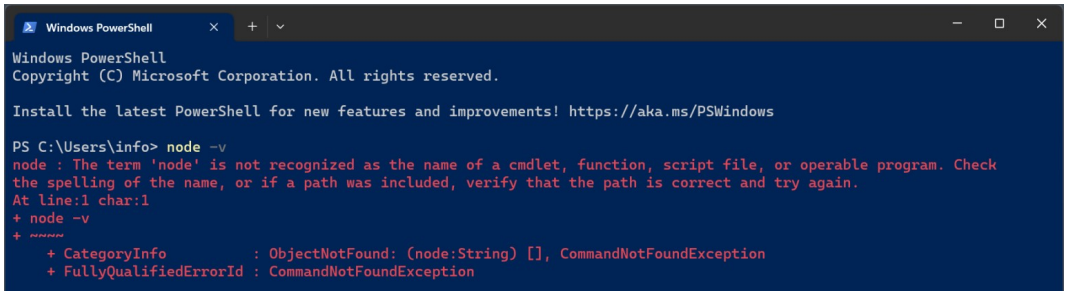
**Scripts**: npm allows developers to define custom scripts to automate various tasks such as testing, building, and deployment, providing a standardized way to execute common project-related operations.

# Do I have Node.js and npm installed?

Just open a command-line shell like PowerShell, available at https://learn.microsoft.com/en-us/powershell/ and write:

```
node -v
```

This way:

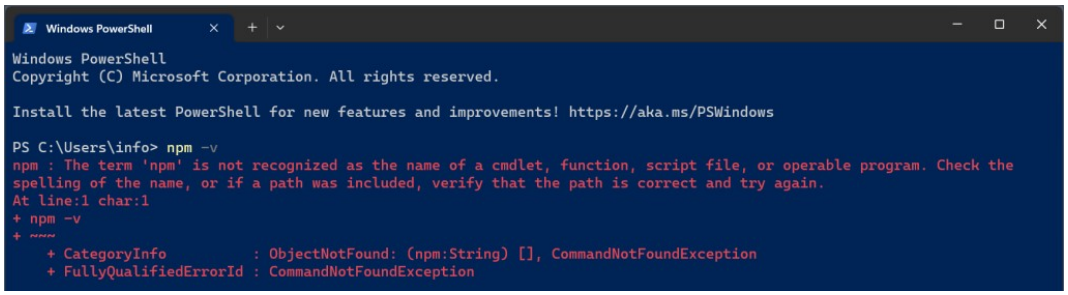



If you get an error like in the picture above, then you don't have Node.js installed.

The same concept applies to npm: just like you did for Node.js, you can check if you have npm installed by writing in your command-line shell:

```
npm -v
```

This way:



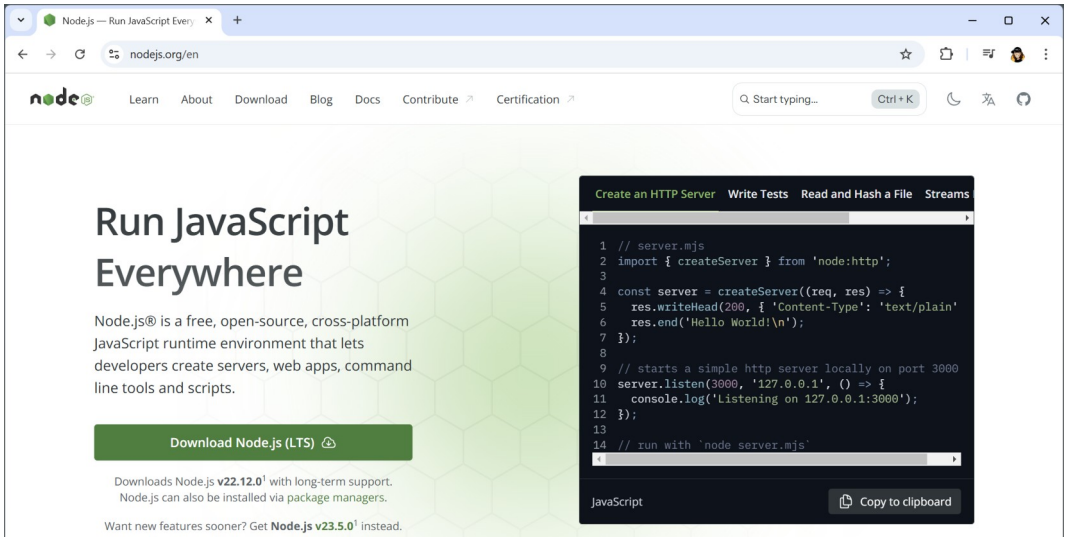

Again, if you get an error, then you don't have npm installed, but we are going to fix this issue right away.

And there is also good news: by installing Node you will also install npm, two birds with one stone.

# How to install Node.js and npm

If you don't have Node.js installed, or if you have an old version, it's time to install or update it.

To install Node.js, go to https://nodejs.org/ and download the version compatible with your operating system.




Once you downloaded and installed Node.js, try once more to write

```
node -v
```

and

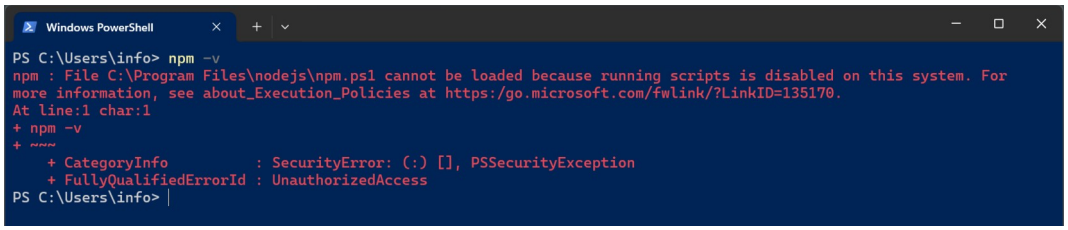
```
npm -v
```

and you will see the versions currently installed.


You should always check your versions of Node.js and npm are up to date.

If you are working in Windows, due to security reasons, in some cases you

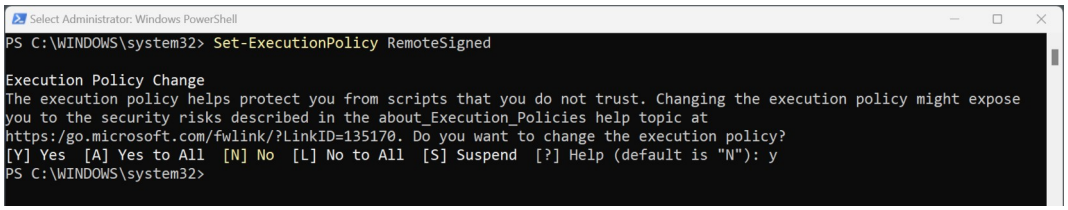
may get a **SecurityError** message when executing **npm -v**, like this one:




To fix it, run PowerShell as administrator and write:

```
Set-ExecutionPolicy RemoteSigned
```

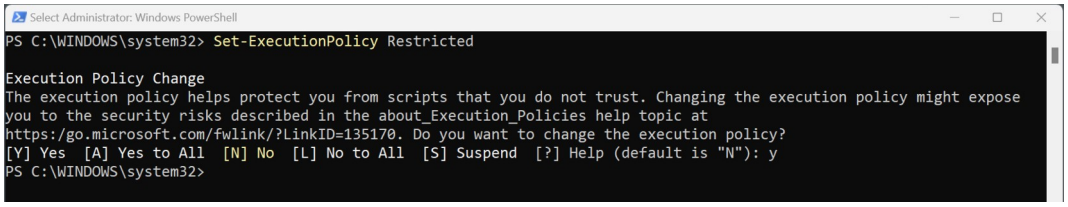
And then press **Y**, this way:




To go back to original settings, just write:

```
Set-ExecutionPolicy Restricted
```

And again press **Y**, this way:




Back to our Node.js and npm, remember to always check then to be up to date.

It' always a good practice to check regularly for updates, for these reasons:

**Security**: updates often include security patches and fixes for vulnerabilities, and running an outdated version of Node.js may expose your application to known security risks.

**Bug Fixes and Performance Improvements**: new releases often come

with bug fixes and performance improvements. Using the latest version helps you benefit from a more stable and optimized runtime environment, which can result in better performance for your applications.

**Community Support**: the Node.js community actively supports and maintains the runtime. If you encounter issues or need assistance, having the latest version can make it easier to seek help from the community, as they are more likely to be familiar with recent releases.

**Ecosystem Compatibility**: other tools, libraries, and packages in the Node.js ecosystem may update their dependencies and features to align with the latest Node.js versions. Keeping Node.js updated helps maintain compatibility with these dependencies.

While keeping Node.js updated is generally a good practice, it's also important to consider the specific needs and constraints of your project.

Before upgrading, it's advisable to check the release notes and documentation for any potential breaking changes or considerations specific to the version you are updating to.

Additionally, testing your application thoroughly after an update is crucial to ensure that it behaves as expected with the new Node.js version.

## What to do once you have Node.js installed

Now that you have Node.js installed, you can write your first application.

Create a file with this JavaScript code:

```
console.log('Hello world, this is JavaScript running!');
```

And save it as **hellonode.js**.

Now navigate with your command-line shell to the folder where you saved **hellonode.js** and write:

```
node hellonode.js
```

This way:


And you should see **Hello world, this is JavaScript running** prompted in your command-line shell.

You just executed JavaScript outside a web browser.

Executing JavaScript outside the web browser is advantageous in various scenarios where the capabilities of JavaScript are needed beyond the traditional browser environment.

For instance, you could build web servers using Node.js to handle HTTP requests and responses, or write command-line tools and scripts to automate tasks using JavaScript.

The npm ecosystem further enhances the capabilities by providing a vast collection of open-source libraries and modules.

At the moment, let’s focus on web servers.

# A web page that just won't work locally

Not all web pages are designed to work locally.

Try this example: create a file called **simplepage.html** with this content:

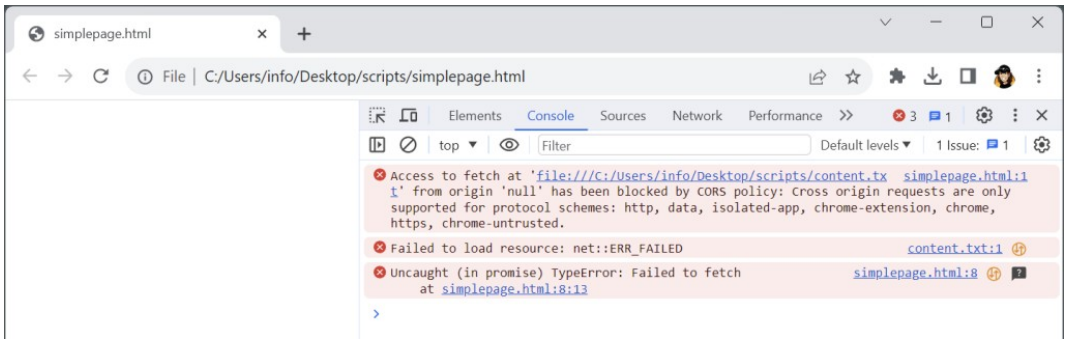
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <meta http-equiv="content-type" content="text/html;
charset=utf-8">
    </head>
    <body>
        <script>
            fetch('./content.txt').then(response =>
response.text()).then(text => document.body.innerHTML = text);
        </script>
    </body>
</html>
```

It just reads the content of **content.txt** file and outputs it to the body.

Create **content.txt** file in the same path with this content:

```
Hello world, this is JavaScript running again!
```

But if you open **simplepage.html** in a browser, you won't see any output:




Even worse, if you open the console, you will see a CORS policy error.

# What is CORS policy?

**Cross-Origin Resource Sharing** (CORS) is a security feature implemented in web browsers that governs how web pages from one domain can request and access resources from another domain.

It is a crucial security mechanism to prevent potential security vulnerabilities that can arise when different websites interact with each other.

One such potential security breach without CORS policy is the **Cross-Site Request Forgery** (CSRF) attack.

A Cross-Site Request Forgery attack involves tricking a user's browser into making an unintended and potentially harmful request to a web application where the user is authenticated.

This attack takes advantage of the fact that the browser automatically includes authentication cookies with requests to a particular domain, potentially leading to unauthorized actions being performed on the user's behalf without their knowledge or consent.

So when a web page on one domain (the “origin”) tries to make a request for resources, such as data, images, or scripts, from another domain, the browser enforces the Same-Origin Policy by default.

This policy restricts web pages from making requests to domains different from the one that served the web page itself.

And no, being on the same folder does not mean being on the same domain, as you just witnessed.

How can we overcome this?

With a local web server.

## What is a web server?

A web server is a computer program or software that serves requests made by clients over the World Wide Web or a private network.

It acts as a mediator between the client, typically a web browser, and the server-side resources such as web pages, images, videos, and other files.

Some popular web server software includes:

**Apache HTTP Server**: one of the most widely used open-source web servers. It is free to use, highly customizable, and supports a wide range of modules for extending functionality.

**Nginx**: a high-performance, open-source web server that is known for its efficiency in handling concurrent connections. It is often used as a reverse proxy and load balancer as well.

**Caddy**: an open-source web server that is known for its simplicity and ease of configuration. It comes with automatic HTTPS by default and is designed to be user-friendly.

**Cherokee**: a free and open-source web server that is lightweight and easy to configure. It supports various features, including virtual hosts, authentication, and more.

But we can also use Node.js to install a web server.

## Your first Node.js web server

We can create a web server with Node.js using the **http module**.

It's a core module that comes bundled with Node.js, allowing developers to create HTTP servers and handle HTTP-related tasks.

It provides the necessary functionality to build web servers and handle incoming HTTP requests.

You can then create an HTTP server using the **http.createServer()** method, providing a callback function that will be invoked for each incoming request.

Let's create a new file called **nodeserver.js** with this content:

```
// include the HTTP module in the application
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

// create a server on your computer:
const server = http.createServer((req, res) => {

    // 200 OK status code means that the request was successful
    res.statusCode = 200;

    // set the content to text
    res.setHeader('Content-Type', 'text/plain');

    // this is what we are going to send to the browser
    res.end('Hello World');
});

// starts the HTTP server listening for connections
server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

Now execute the JavaScript file we just created typing this in your command-line shell, once in the same directory:

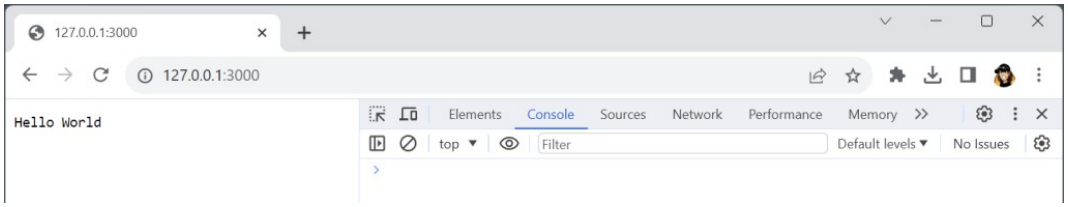
```
node nodeserver.js
```

This way:


As you can see, the script will keep running until you stop it.

Don't stop it at the moment, and open your browser to http://127.0.0.1:3000.

This is what you should see:




Which is a local web server showing the content created on the fly by **nodeserver.js**.

This is something, but still not what we are looking for.

We want our web server to be able to load a page which is already on our disk, rather than creating one on the fly.

## A web server able to launch a page

Now we need another module called **fs module**, where “fs” is “file system”.

It’s another core module that provides functions for interacting with the file system on your computer.

It allows you to perform various file-related operations such as reading from and writing to files, creating and deleting files, and managing directories.

Let’s create another file called **nodepagelauncher.js** with this content:

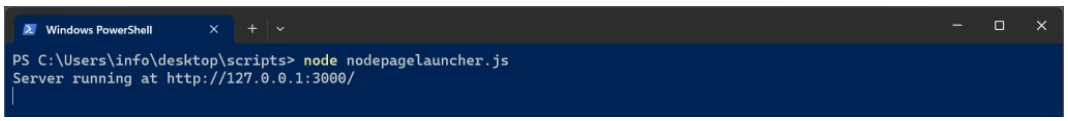
```
const http = require('http');
const fs = require('fs')

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    res.writeHead(200, { 'content-type': 'text/html' })
    fs.createReadStream('simplepage.html').pipe(res)
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

This is trying to build a web server and call **simplepage.html**, which is the page we tried to run before, being blocked by the CORS policy.

And let’s execute it, as usual, with

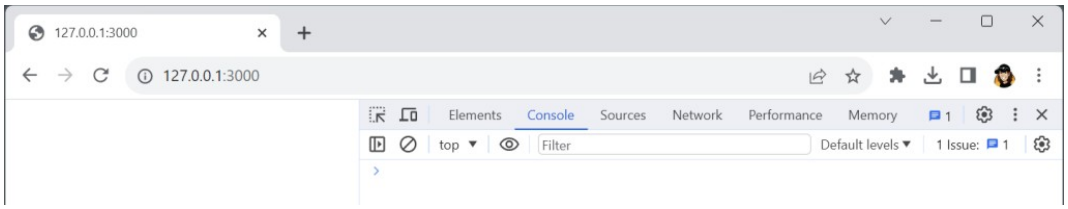
`node nodepagelauncher.js`

This way:




Point the browser to http://127.0.0.1:3000 and you will see, well, nothing.

No errors, but also no content.




But if you look at the elements, or at the web page source, you will see the page has been successfully loaded, it's just it's not fetching **content.txt**.


This happens because our server is only able to read **simplepage.html**.

At each request, it will always deal with **simplepage.html**.

We need our web server to be able to read more pages.

## A web server able to launch more pages

At this point it's time to put all together to create a web server able to launch more pages.

Let's create a new file called **nodemultiplepagelauncher.js** with this content:

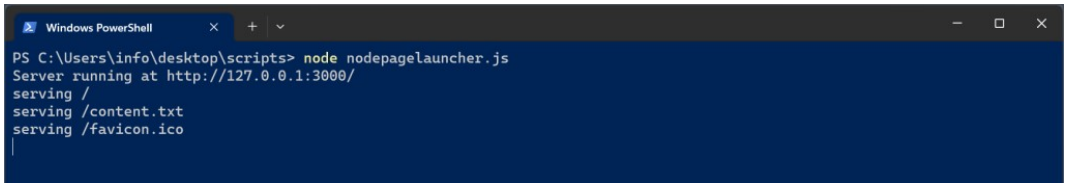
```
const http = require('http');
const fs = require('fs')

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
    console.log('serving ' + req.url);

    // check the URL of the request
    switch(req.url) {

        // is the url content.txt?
        case '/content.txt' :
            res.writeHead(200, { 'content-type': 'text/plain' })
            fs.createReadStream('content.txt').pipe(res);
            break;

        // is the url everything but content.txt?
        default :
            res.writeHead(200, { 'content-type': 'text/html' })
            fs.createReadStream('simplepage.html').pipe(res);
    }
});

server.listen(port, hostname, () => {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

This time we should handle both a default page, our **simplepage.html** file, and **content.txt** file.

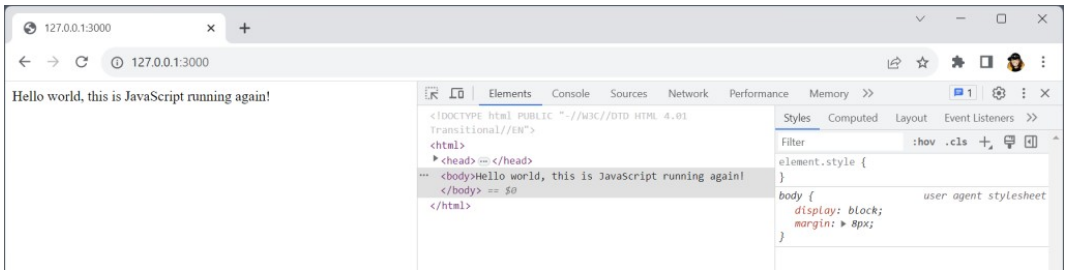
Now, you know what to do: launch it with:

```
node nodemultiplepagelauncher.js
```

This way:

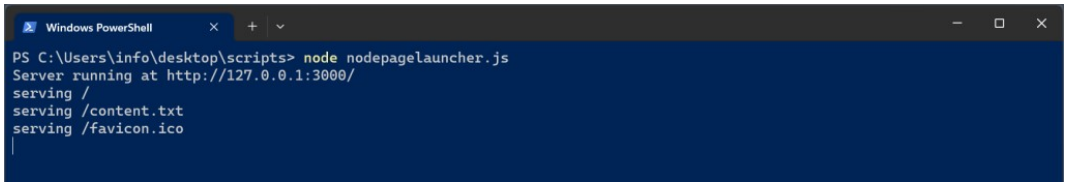



And finally, if you refresh the page at http://127.0.0.1:3000, you should see **simplepage.html** properly rendered.




Thanks to a web server, we were able to render a web page we would never be able to render by just running it in a web browser.

And look at the command-line shell:




It's serving the root page, which we called **simplepage.html**, then **content.txt** and also **favicon.ico**, even though it is not in our folder.

Since HTML5 games make intensive use of external content, such as images and sound files, we need a web server to perform tests locally.

Obviously, it would be impossible to handle all different resources with a **switch** statement, so we need a more universal web server script.

## A web server just working fine

There is no need to reinvent the wheel, so let's try a nice script by The Jared Wilcurt which is perfect for our needs.

```
// npm-Free Server by The Jared Wilcurt
// All you need to run this is an installed copy of Node.JS
// Put this next to the files you want to serve and run: node server.js

// Require in some of the native stuff that comes with Node
var http = require('http');
var url = require('url');
var path = require('path');
var fs = require('fs');
// Port number to use
var port = process.argv[2] || 8000;
// Colors for CLI output
var WHT = '\033[39m';
var RED = '\033[91m';
var GRN = '\033[32m';

// Create the server
http.createServer(function (request, response) {

    // The requested URL, like http://localhost:8000/file.html =>
/file.html
    var uri = url.parse(request.url).pathname;
    // get the /file.html from above and then find it from the current
folder
    var filename = path.join(process.cwd(), uri);

    // Setting up MIME-Type (YOU MAY NEED TO ADD MORE HERE) <--------
    var contentTypesByExtension = {
        '.html': 'text/html',
        '.css':  'text/css',
        '.js':   'text/javascript',
        '.json': 'text/json',
        '.svg':  'image/svg+xml'
    };
```

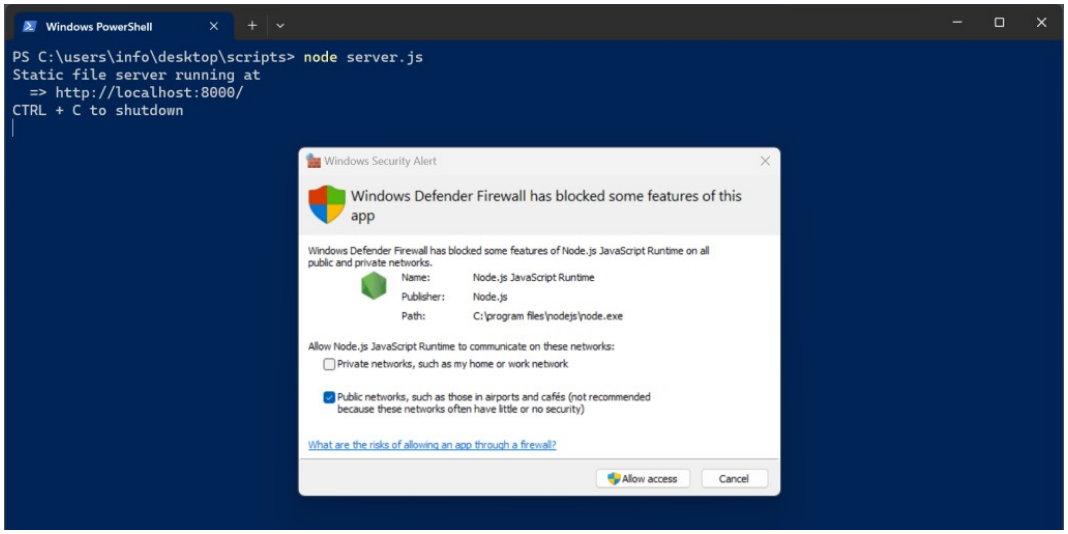
```
    // Check if the requested file exists
    fs.exists(filename, function (exists) {
        // If it doesn't
        if (!exists) {
            // Output a red error pointing to failed request
            console.log(RED + 'FAIL: ' + filename);
            // Redirect the browser to the 404 page
            filename = path.join(process.cwd(), '/404.html');
        // If the requested URL is a folder, like
http://localhost:8000/catpics
        } else if (fs.statSync(filename).isDirectory()) {
            // Output a green line to the console explaining what
folder was requested
            console.log(GRN + 'FLDR: ' + WHT + filename);
            // redirect the user to the index.html in the requested
folder
            filename += '/index.html';
        }

        // Assuming the file exists, read it
        fs.readFile(filename, 'binary', function (err, file) {
            // Output a green line to console explaining the file that
will be loaded in the browser
            console.log(GRN + 'FILE: ' + WHT + filename);
            // If there was an error trying to read the file
            if (err) {
                // Put the error in the browser
                response.writeHead(500, {'Content-Type':
'text/plain'});
                response.write(err + '\n');
                response.end();
                return;
            }

            // Otherwise, declare a headers object and a var for the
MIME-Type
            var headers = {};
            var contentType =
contentTypesByExtension[path.extname(filename)];
            // If the requested file has a matching MIME-Type
```

```
            if (contentType) {
                // Set it in the headers
                headers['Content-Type'] = contentType;
            }

            // Output the read file to the browser for it to load
            response.writeHead(200, headers);
            response.write(file, 'binary');
            response.end();
        });

    });

}).listen(parseInt(port, 10));

// Message to display when server is started
console.log(WHT + 'Static file server running at\n  =>
http://localhost:' + port + '/\nCTRL + C to shutdown');
```

You can save it as **server.js**, and launch it with:

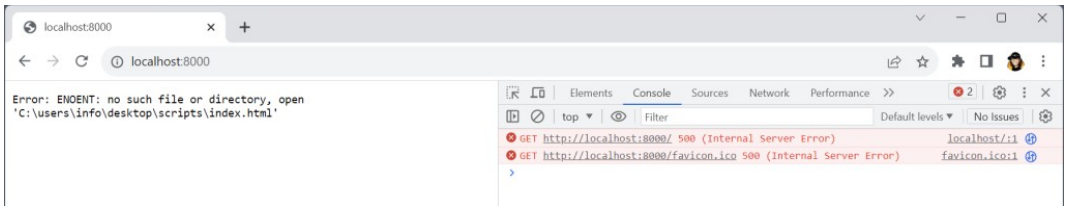
`node server.js`

this way:



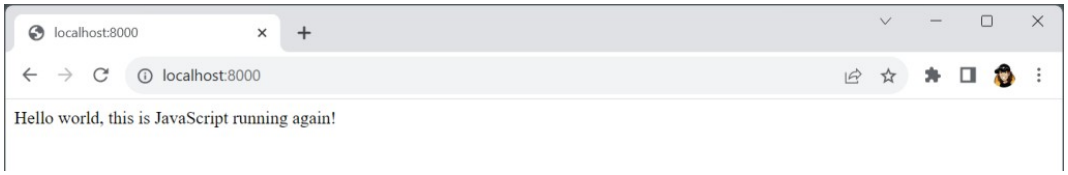

If you get some kind of security alert, like me, just allow access, then point

your browser to http://localhost:8000/ to get, unfortunately, another error.




It's normal, because now we are running something like an actual web server, looking for **index.html** page.

Rename **simplepage.html** to **index.html** and refresh the browser, and you should have your page displayed properly:




Will it work with any page? Sure!

Let's try with a Phaser example.

## Running a Phaser example locally with Node.js web server

Let's try to build the famous Phaser example you can find at https://labs.phaser.io/view.html?src=src\game%20objects\images\image%20rotation.js.

For the sake of semplicity, I am loading Phaser from a CDN and will place the entire script into just one page called **index.html**.

We also need the rotating image, to be saved as **phaser3-logo.png** in the same folder.

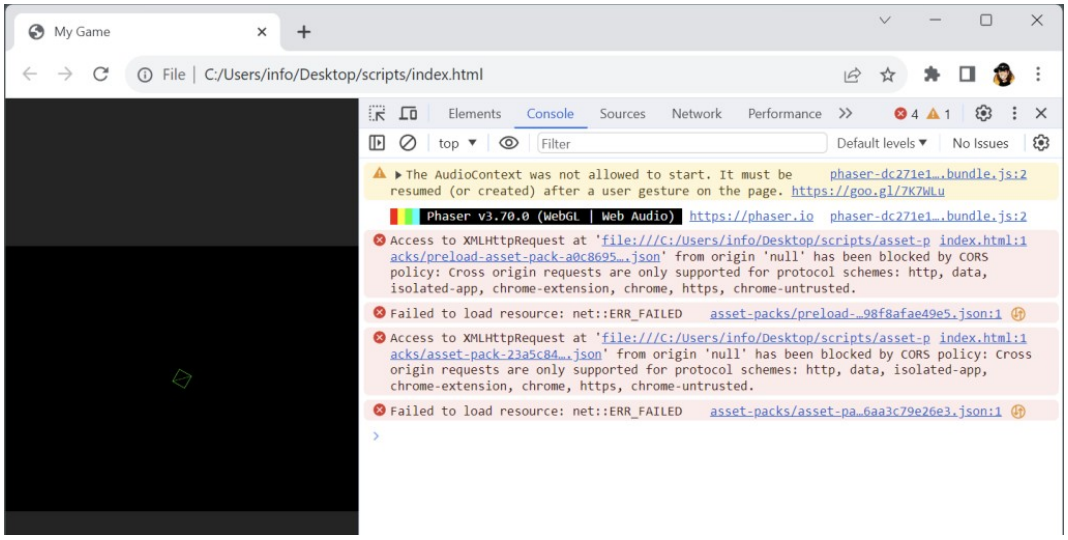
This is the image:

And this is the content of **index.html**:

```
<!DOCTYPE html>
<html>
    <head>
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/phaser/3.70.0/phaser.min.js
"></script>
    </head>
    <body>
        <div id="thegame"></div>
        <script>
            class playGame extends Phaser.Scene {
                constructor() {
                    super('PlayGame');
                }
                preload() {
                    this.load.image('logo', 'phaser3-logo.png');
                }
                create() {
```

```
                this.image = this.add.image(400, 300, 'logo');
            }
            update() {
                this.image.rotation += 0.01;
            }
        }
        let game = new Phaser.Game({
            type : Phaser.AUTO,
            width : 800,
            height : 600,
            parent : 'thegame',
            scene: playGame
        });
    </script>
</body>
</html>
```

First, let's try to open it directly from the web browser, with no web server running:

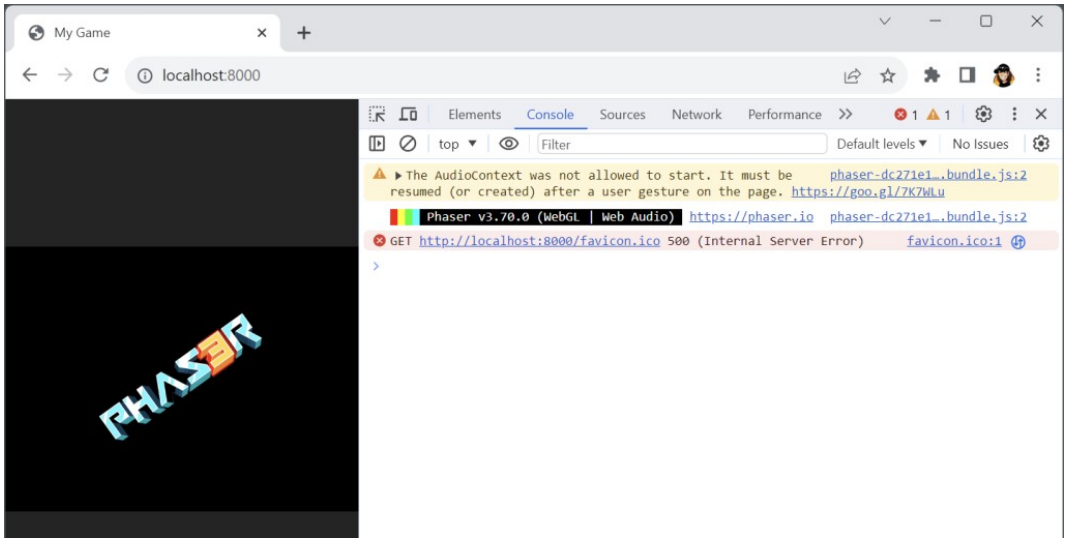



We are getting a CORS error because we aren't allowed to load the Phaser 3 logo, and since we can't load the Phaser logo, all we can see is that wireframe placeholder.

If you copy **server.js** in the same folder as the Phaser project and start the web server as seen in previous chapter with

```
node server.js
```

The page will be properly loaded with all assets, and we can see it working flawlessly.




This is how you should use Node.js to start a web server, and this is also why you should use it.

Now we need a powerful tool to build our Phaser projects.

There are many code editors, but we will choose Visual Studio Code.

## What is Visual Studio Code?

Visual Studio Code (VS Code) is a free, open-source source code editor developed by Microsoft.

It's a lightweight and highly customizable editor that supports a wide range of programming languages.

Unlike the full-fledged Visual Studio IDE, Visual Studio Code is designed to be a lightweight, fast, and cross-platform code editor that can be used for various programming and scripting tasks.

Visual Studio Code has gained widespread popularity among developers due to its speed, flexibility, and the broad range of features it provides.

It's suitable for various development tasks, including web development, mobile app development, and scripting.

Moreover, Visual Studio Code also has these interesting features that will help us in developing HMTL5 games with Phaser:

It’s **cross-platform**: Visual Studio Code is available for Windows, macOS, and Linux, making it a cross-platform code editor.

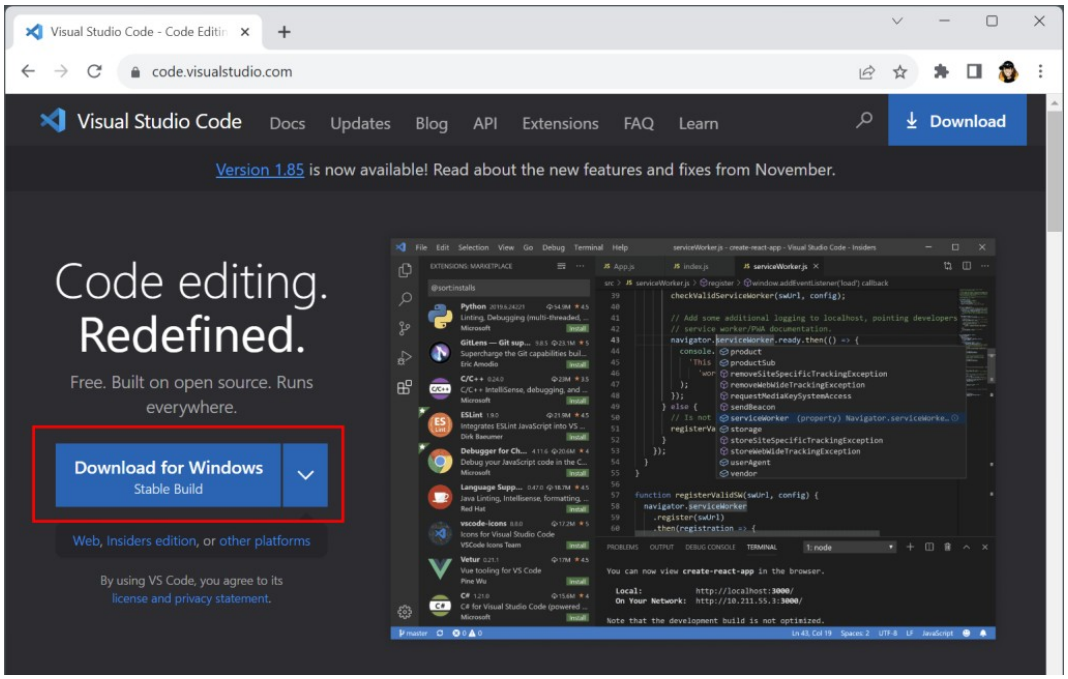
It has a **great language support**: it supports a wide range of programming languages and file types. The editor comes with built-in support for languages like JavaScript, TypeScript, HTML, CSS, which are the languages we will be using, and more, with code completion.

It features an **integrated terminal**, so you won’t have to use any external command-line shell.

Let’s start working with Visual Studio Code.

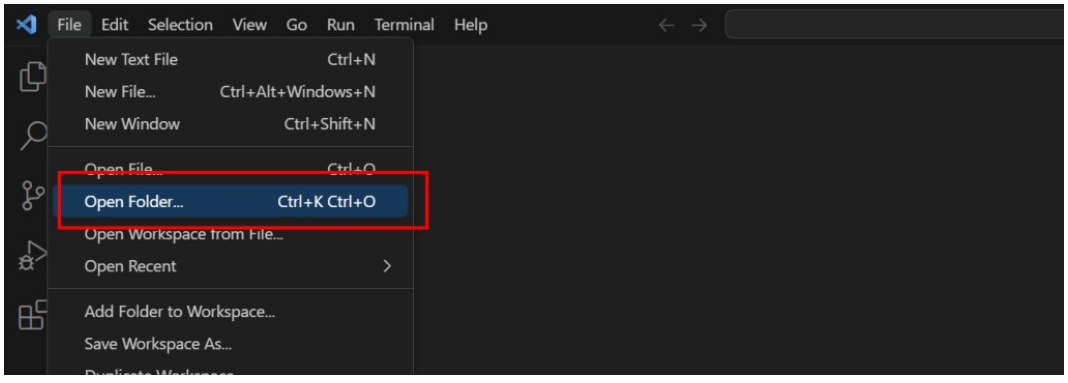
# Configure Visual Studio Code to build HTML5 games with Phaser

Download Visual Studio Code from https://code.visualstudio.com/.



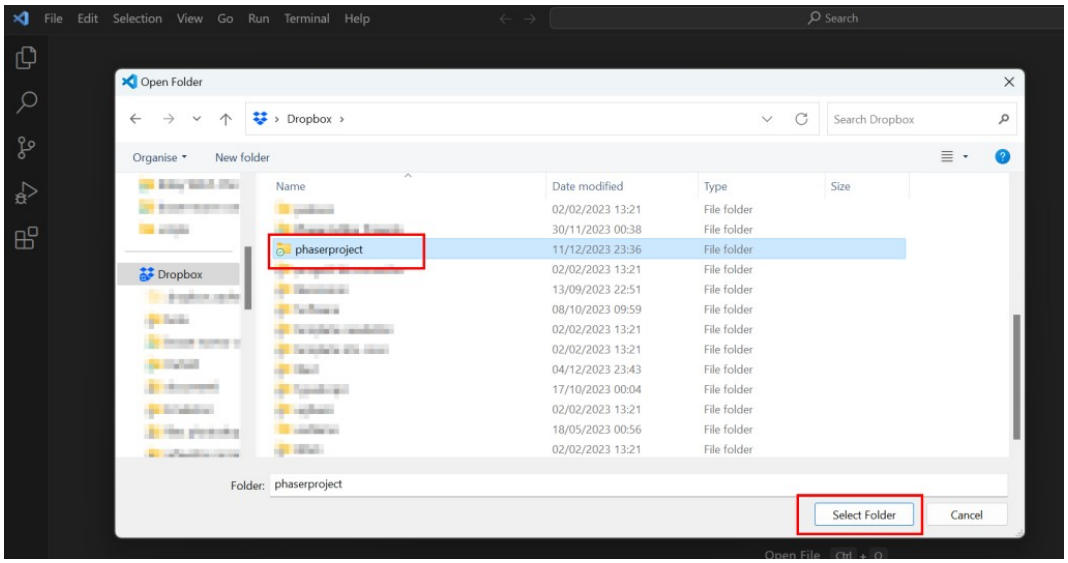

Install it, then create a new empty folder called **phaserproject**.

Launch Visual Studio Code, then select **File > Open Folder**:

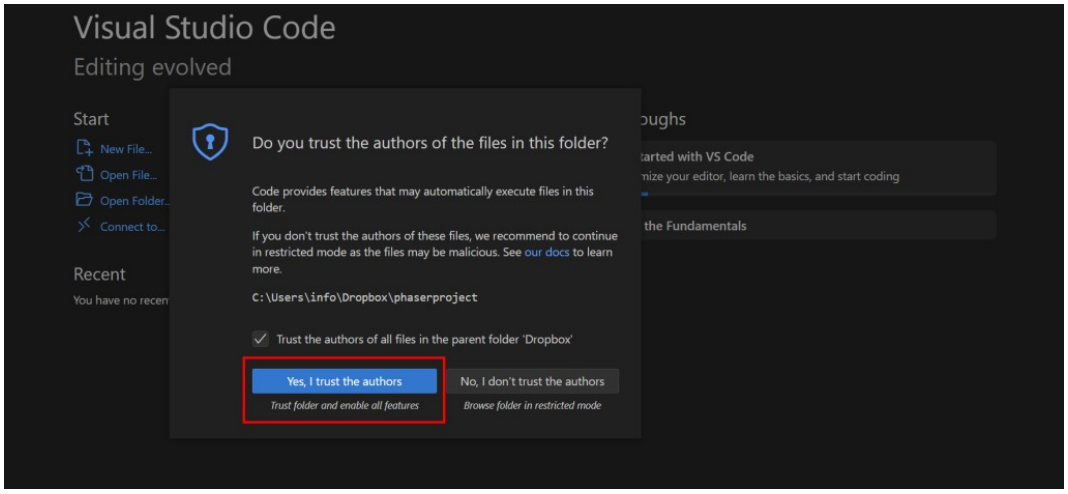



This will be our project folder.

Now select the folder you just created.




If you are asked to trust the authors (you!!), do it.




Until now, we needed a command line shell to perform operations with Node.js.

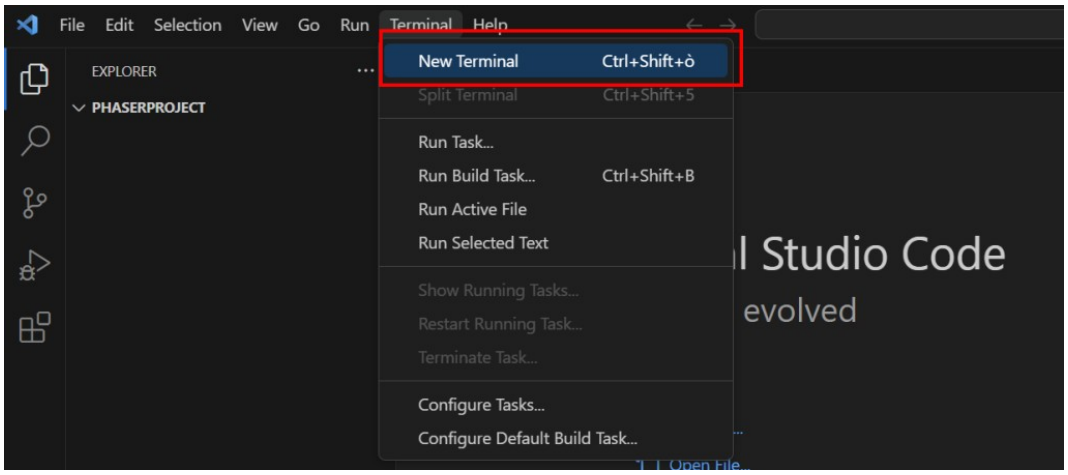
Visual Studio Code has an integrated command-line interface within the VSCode editor.

It will allow you to interact with your project without requiring any external

tool, performing various development tasks directly from the editor.

In **Terminal > New Terminal** you can find the command-line shell.

We are going to use it quite a lot.

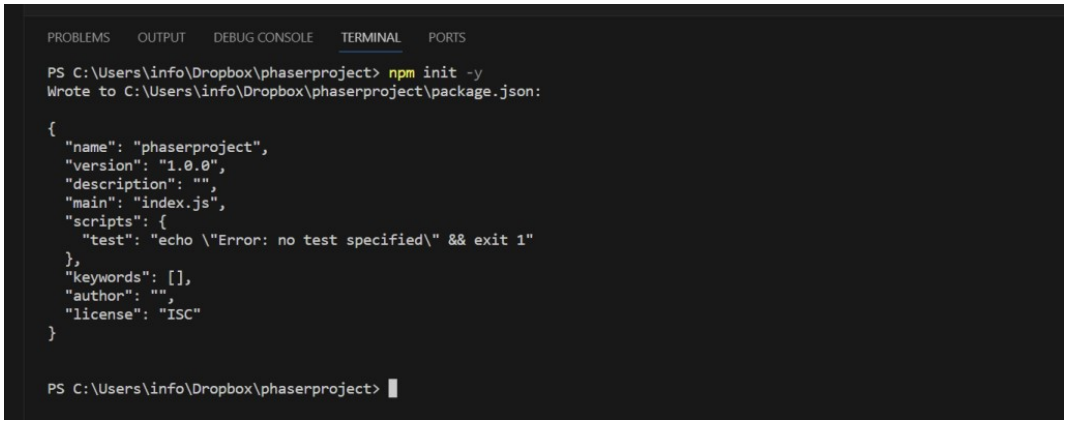



The terminal will open directly in your project directory.

Let's set up a new or existing npm package, by entering in the terminal:

```
npm init -y
```

This way:




A new file, **package.json**, will be created with some default values.

You can enter these values by calling the above command without the final

parameter **-y**, with:

```
npm init
```

But I suggest to always start with the default options which can be changed later.

This is the content of **package.json**:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

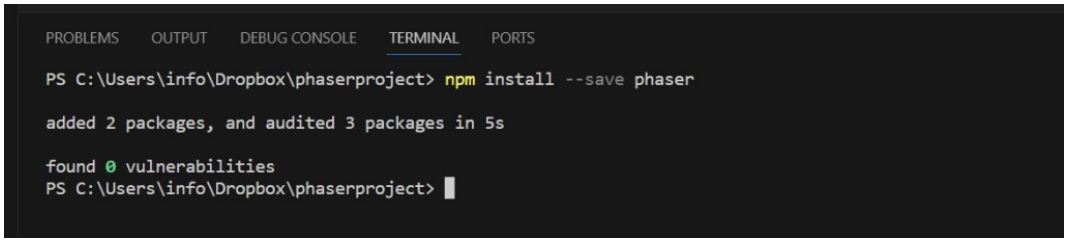
Now it's time to install Phaser.

Which version? The latest.

How to know the latest version to install? We don't mind, thanks to NPM we can always install the latest Phaser version, with:

```
npm install --save phaser
```

This way:




Phaser will be installed in no time.

Now you should see some new lines in **package.json**:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  }
}
```

Highlighted lines tell us there is Phaser 3.70.0, the most recent at the time of writing, as dependency.

Anyway, where did npm install Phaser?

Theoretically we don't care, it is not our problem since we have delegated webpack to take care of that, but for your information it has been installed in **node_modules > phaser** folder.

When you install external packages or libraries for your Node.js project using npm, these packages are typically stored in the **node_modules** folder.

With Phaser installed, it’s time to meet webpack.

## What is webpack?

webpack is a popular open-source JavaScript module bundler widely used in modern web development.

Its primary purpose is to streamline the management, optimization, and bundling of diverse assets, allowing developers to build efficient and performant web applications.

At its core, webpack is adept at handling the modular nature of modern JavaScript applications.

It processes and bundles JavaScript modules, along with other assets like stylesheets, images, and fonts.

This bundling is crucial for optimizing the loading and execution of code in web browsers, promoting a more efficient use of resources.

One of webpack's notable features is its ability to support code splitting, allowing developers to split their code into smaller, manageable chunks.

This facilitates lazy loading, where only the necessary code is loaded as needed, improving the initial load time and overall performance of the application.

The bundler's versatility extends to handling different environments, enabling the configuration of development, production, and other custom settings.

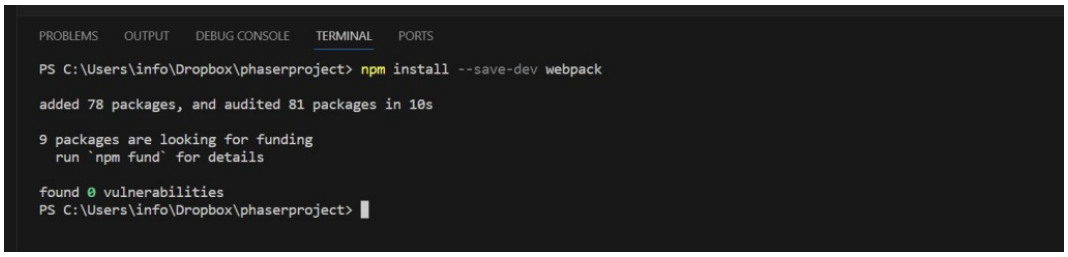
We are going to use webpack to speed up and optimize development and distribution of Phaser games.

# How to install webpack

You can install webpack with:

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

This way:




You will notice **package.json** changed again:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "webpack": "^5.89.0"
  }
}
```

Highlighted lines say there is webpack 5.89.0 as dev dependency.

You probably noticed we installed Phaser using **--save**, and webpack using **--save-dev**.

For this reason, Phaser is now listed in **dependencies**, while webpack is listed in **devDependencies**.

What's the difference?

Under **dependencies** you'll find the libraries which needs to be included in the final distributable bundle.

In this case, a Phaser game obviously requires Phaser.

Under **devDependencies** you'll find the libraries we use during development, but which you don't need anymore once you publish the final bundle.

This means the distributable package won't include webpack.

But webpack alone would not have this much use, that's why we are going to install some plugins.

Plugins allow you to perform a wide range of tasks during different stages of the build process, such as optimizing code, generating assets, manipulating the output, and more.

They play a crucial role in customizing and enhancing the webpack build pipeline.

Let's see the plugins we need to install.

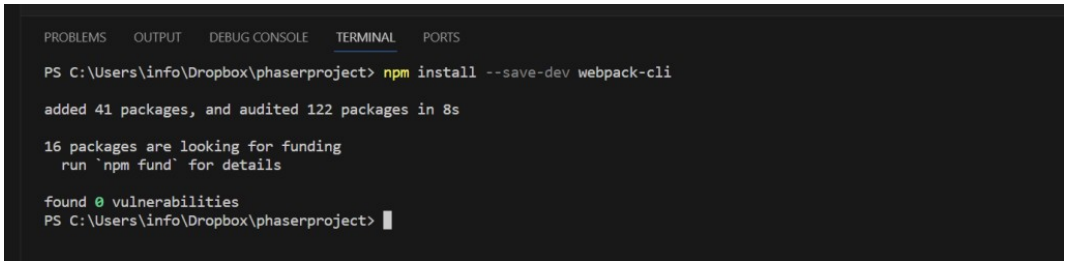
## Installing useful webpack plugins

Installing webpack plugins is easy and can be done directly from the terminal.

**webpack-cli** provides a flexible set of commands for developers to increase speed when setting up a custom webpack project, and can be installed with:

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

This way:




And look at the highlighted line of **package.json**:

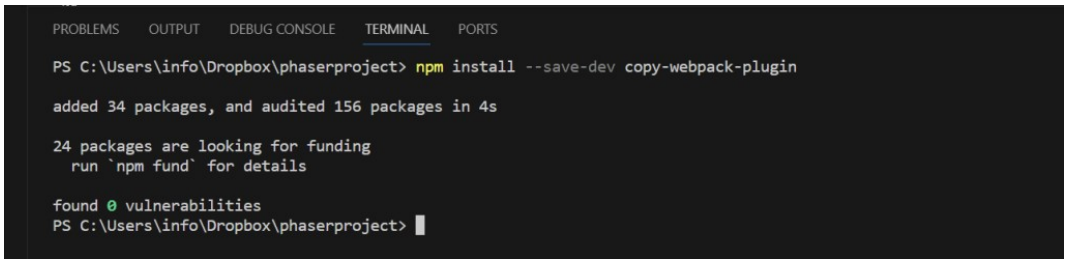
```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4"
  }
}
```

The second plugin we need to install is **copy-webpack-plugin**, to copy individual files or entire directories, which already exist, to the build directory.

It can be installed with:

```
npm install --save-dev copy-webpack-plugin
```

This way:




And here it is a new line in **package.json**:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "copy-webpack-plugin": "^11.0.0",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4"
  }
}
```

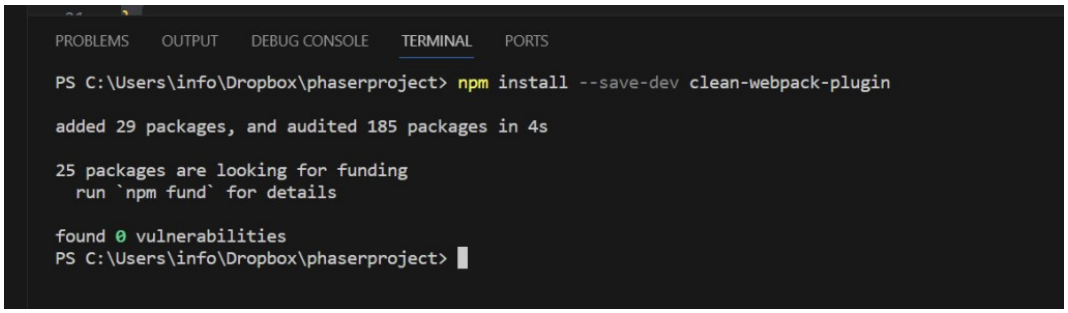
Now we have to install **clean-webpack-plugin**, a webpack plugin to

remove or clean your build folders.

We can do it with the command:

```
npm install --save-dev clean-webpack-plugin
```

This way:




**package.json** content will change once more, look at the highlighted line:

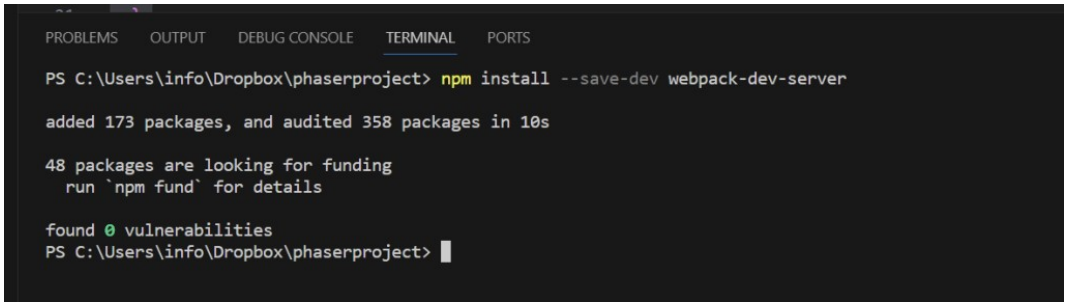
```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "clean-webpack-plugin": "^4.0.0",
    "copy-webpack-plugin": "^11.0.0",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4"
  }
}
```

Finally we'll install **webpack-dev-server** plugin, which uses webpack with

a development server that provides live reloading, with this command:

```
npm install --save-dev webpack-dev-server
```

This way:




And here it is another change to **package.json**:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "clean-webpack-plugin": "^4.0.0",
    "copy-webpack-plugin": "^11.0.0",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4",
    "webpack-dev-server": "^4.15.1"
  }
}
```

Why do we need live reloading?

## What is live reloading?

Live reloading is a development feature that automatically refreshes a web application or webpage in real-time whenever changes are made to the source code or related files.

It aims to enhance the development workflow by providing immediate feedback to developers as they modify and save their code.

When live reloading is enabled, a monitoring system watches for changes in the project files.

As soon as a change is detected, the system triggers an automatic refresh of the application in the web browser, ensuring that the latest code changes take effect instantly.

This eliminates the need for manual page refreshes, streamlining the development process and improving productivity.

Live reloading is particularly valuable in web development, where developers frequently tweak HTML, CSS, and JavaScript code to see the impact of changes.

It enhances the developer experience by providing a more interactive and efficient workflow, allowing developers to focus on coding without the interruption of manual browser refreshes.

It basically works following three steps:

**File Monitoring**: live reloading tools monitor the project's source code, assets, and configuration files for any modifications.

**Change Detection**: when a file is edited and saved, the live reloading system detects the changes through file watchers or event listeners.

**Automatic Refresh**: upon detecting changes, the system triggers an automatic refresh of the web application in the connected browser.

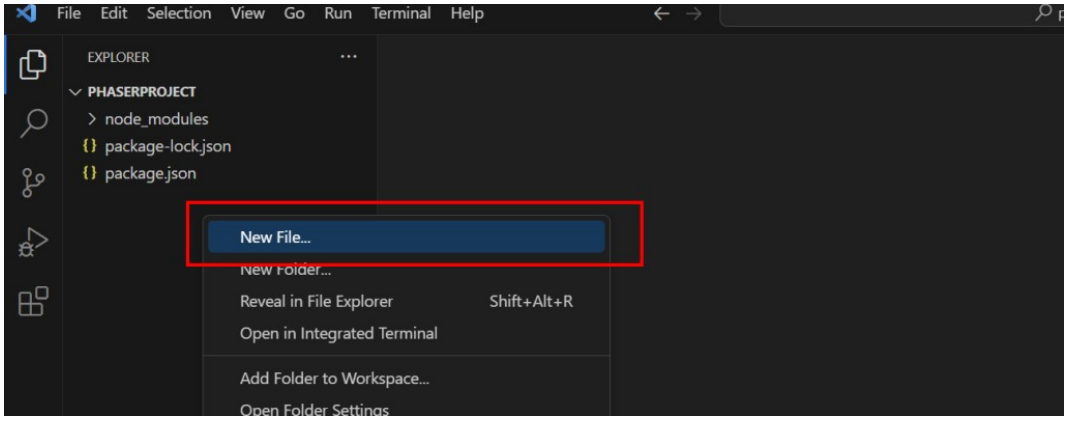
## Create a webpack configuration file for development

What to do with webpack and its plugins?

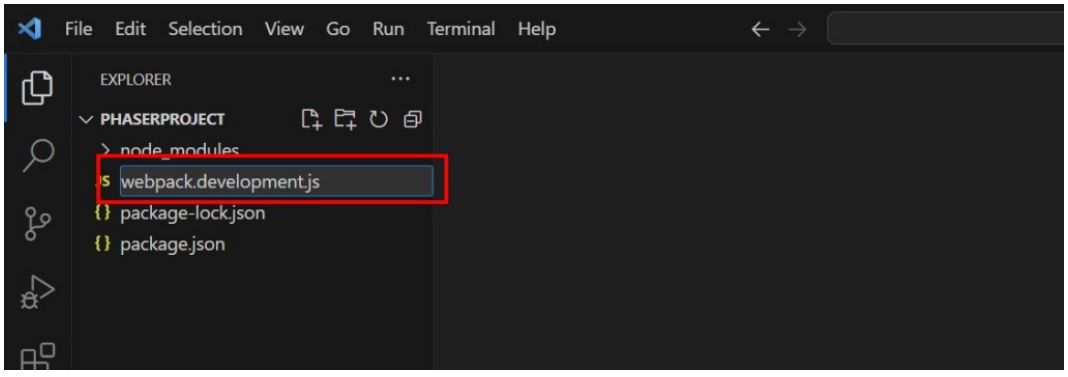
They will be used to automate the development and distribution processes of our Phaser projects, to go from script to web server test featuring live reloading with one command from Terminal, and with another command we'll create a folder with all files needed for distribution.

First, we need to create a configuration file for development.

In **Explorer** panel, right click and select **New File**.




Call the new file **webpack.development.js**.




As you are about to see, these configuration files are just JavaScript scripts.

This is the content of **webpack.development.js**:

```
const path = require('path');

module.exports = {
    entry : {

        // this is our entry point, the main JavaScript file
        app : './src/main.js',
    },
    output : {

        // this is our output file, the one which bundles all libraries
        filename : 'main.js',

        // and this is the path of the output bundle, "dist" folder
        path : path.resolve(__dirname, 'dist'),
    },

    // we are in development mode
    mode : 'development',

    // we need a source map
    devtool : 'inline-source-map',

    // development server root is "src" folder
    devServer : {
        static : './src'
    }
};
```

It’s pretty straightforward and mainly describes folders and files to be used, the only curious thing here is the source map.

A **source map** is a file that maps the source code of an application back to its original, unminified, and untranspiled form.

It is particularly useful in the context of web development where code written in high-level languages (such as TypeScript or ES6 JavaScript) is often transpiled and minified before being deployed to production.

The use of source maps significantly improves the debugging experience in production, as developers can work with the more readable and understandable original source code rather than the minified and transpiled version.

Many build tools and bundlers, such as webpack, automatically generate source maps as part of their build process.

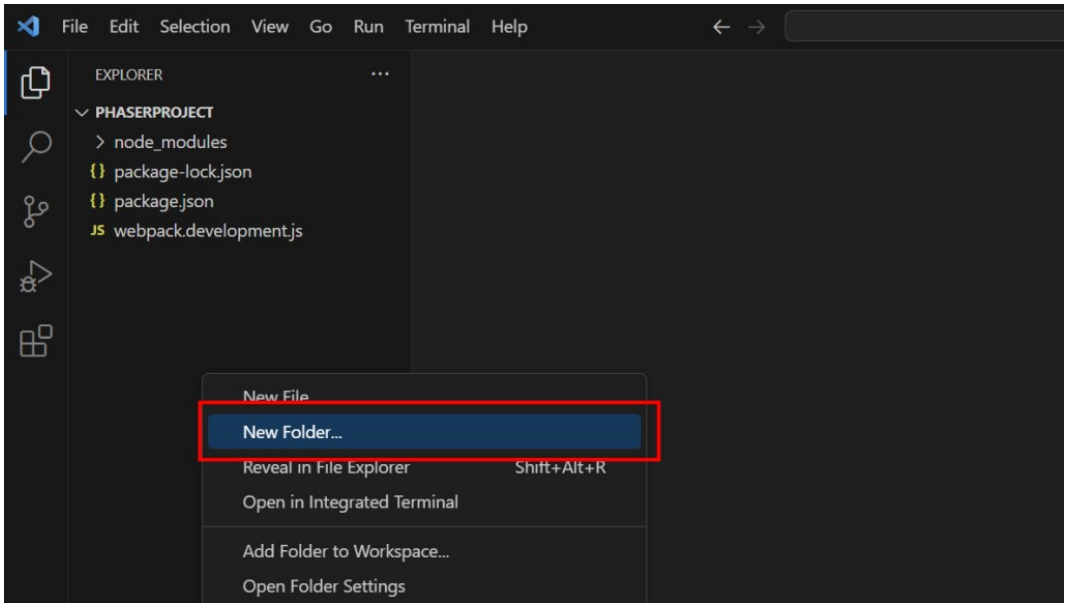
Developers can enable or disable source map generation based on their needs and preferences during development and production builds.

At this time, we are ready to write our first Phaser project in this new environment.

The simple official example is a good example to build with webpack, as it also features an image.

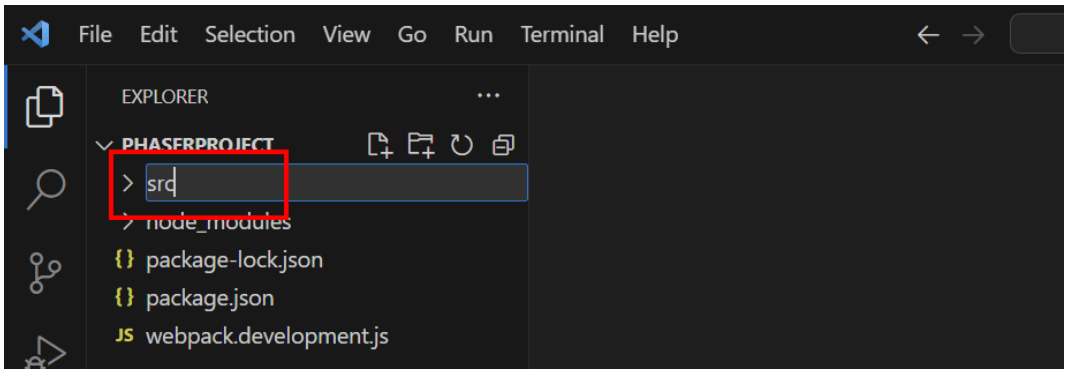
It’s the one you already built to make it run with the Node.js server, but this time it will be written in a more modern JavaScript.

Right click in the **Explorer** panel and select **New Folder**.

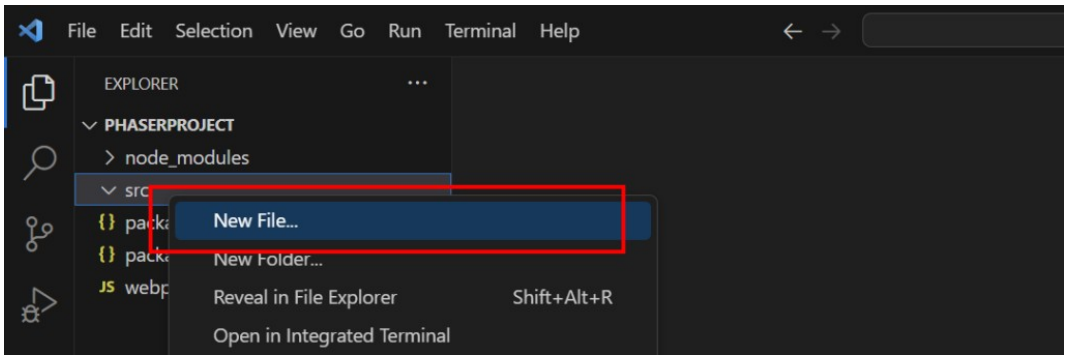



This is the folder that will contain scripts and files of our Phaser project.

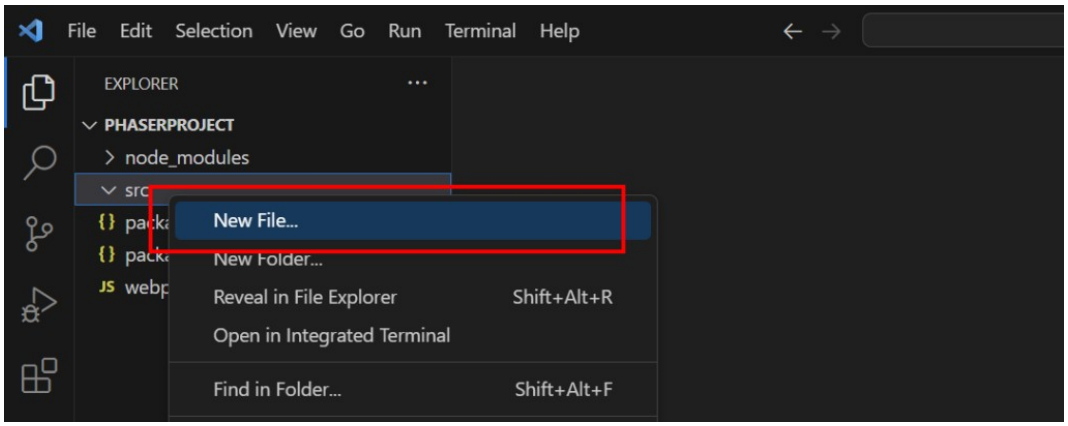
Call the new folder **src** as defined in **webpack.development.js**.




Then right click on **src** folder and select **New File**.




Call the new file **index.html**.




This time we won't write the entire script in one file, and **index.html** will only contain the HTML of the web page which will host the game.

This is the content of **index.html**:

```
<!DOCTYPE html>
<html>
    <head>
        <script src="main.js"></script>
    </head>
<body>
    <div id="thegame"></div>
</body>
</html>
```

Look in the highlighted lines how we include **main.js** which is the entry point as specified in **webpack.development.js**, and the **thegame** element where we want our Phaser game to run in.

Now, create in the same folder a file called **main.js** and write this content, which is roughly the same as the one you can see in the official example.

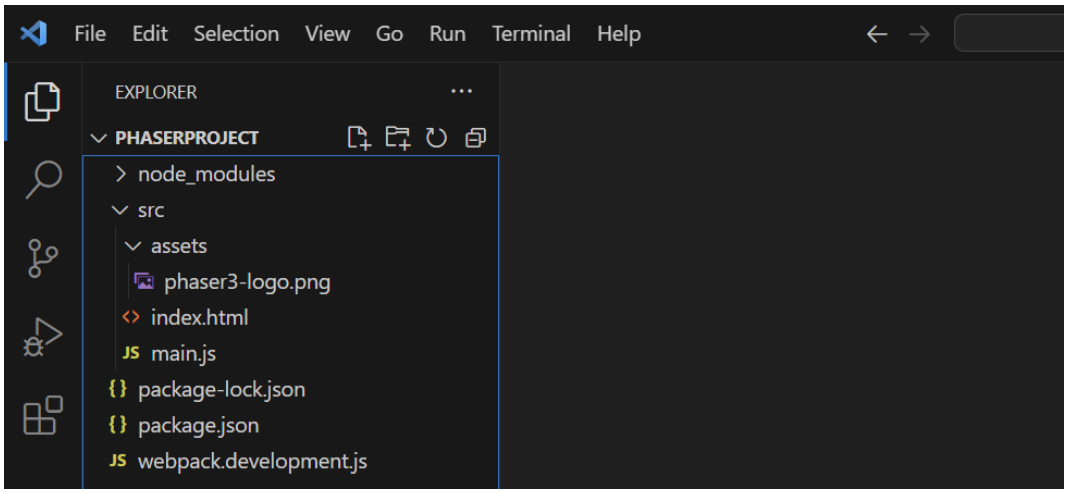
```
import 'phaser';

class PlayGame extends Phaser.Scene {
    constructor() {
        super("PlayGame");
    }
    preload() {
        this.load.image('logo', 'assets/phaser3-logo.png');
    }
    create() {
        this.image = this.add.image(400, 300, 'logo');
    }
    update() {
        this.image.rotation += 0.01;
    }
}

let config = {
    width: 800,
    height: 600,
    parent: 'thegame',
```

```
    scene: PlayGame
};

new Phaser.Game(config);
```

Remember to include the Phaser logo image.

Create a new folder in **src** folder and call it **assets**, then copy the Phaser logo inside assets folder.

This is how your project structure should look like, in **Explorer** panel:




Now rewrite the **scripts** content of **package.json** this way:

```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "development": "webpack serve --open --config
webpack.development.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
```

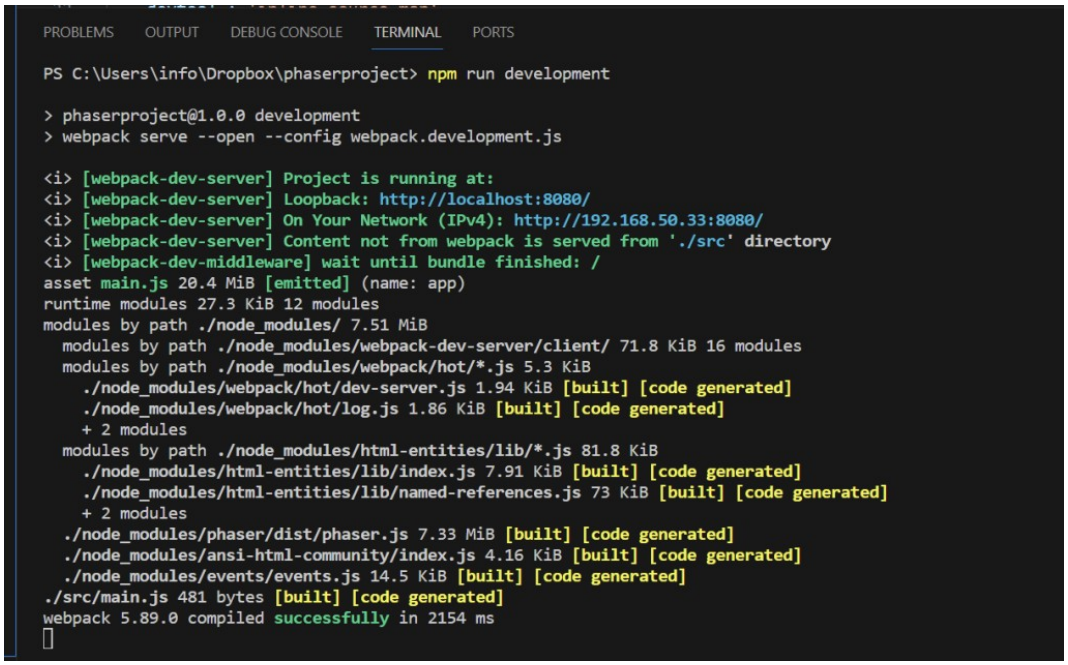
```
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "clean-webpack-plugin": "^4.0.0",
    "copy-webpack-plugin": "^11.0.0",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4",
    "webpack-dev-server": "^4.15.1"
  }
}
```

Basically we created a new script which starts a server using the configuration specified in **webpack.development.js**.

Now let's run the project with

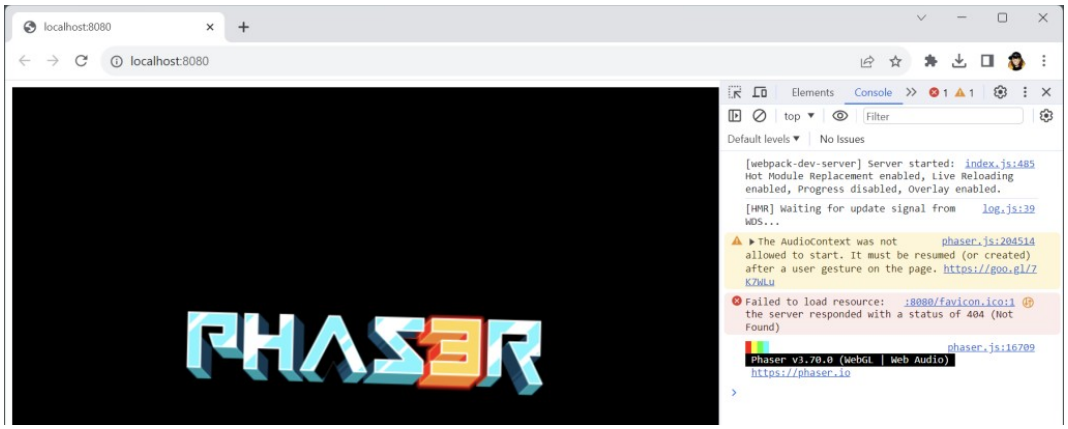
`npm run development`

This way:




And finally a browser page should open with your first Phaser script

running, thanks to webpack.



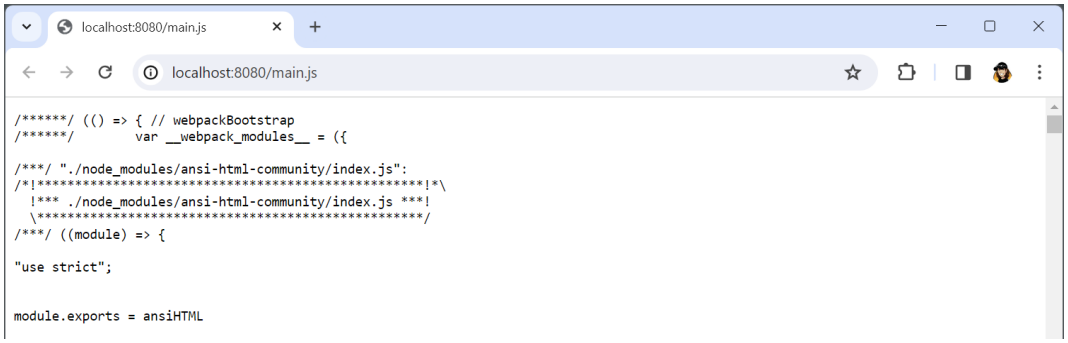

It's worth noting that this script won't work if executed directly in the browser, even if launch it through a web server.

Did you notice that **import 'phaser'** in the first line of **main.js**?

The **import** keyword in JavaScript is used for importing functionality from other modules or scripts.

It is part of the ECMAScript 2015 (ES6) module system, which provides a way to organize and structure code in a more modular fashion.

If you look at the source code of **main.js** in the web browser, you will see it looks way different than the one you wrote in Visual Studio Code:




It's a single JavaScript file merging our script, Phaser and any other resource specified in **dependencies** in **package.json** file.

# Create a webpack configuration file for distribution

In the same way you created **webpack.development.js**, create a new file called **webpack.distribution.ts**, with this content:

```
const path = require('path');

// here we use the plugins to clear folders and copy folder content
const CopyPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
    entry: {

        // this is our entry point, the main JavaScript file
        app: './src/main.js',
    },
    output: {

        // this is our output file, the one which bundles all libraries
        filename: 'main.js',

        // and this is the path of the output bundle, "dist" folder
        path: path.resolve(__dirname, 'dist'),
    },

    // we are in production mode
    mode: 'production',
    plugins: [

        // here we clean the destination folder
        new CleanWebpackPlugin({
            cleanStaleWebpackAssets: false
        }),

        // here we copy some files to destination folder.
        // which files?
        new CopyPlugin({
            patterns: [
                {
```

```
                    // src/index.html
                    from: 'index.html',
                    context: 'src/'
                },
                {
                    // every file inside src/assets folder
                    from: 'assets/*',
                    context: 'src/'
                }
            ]
        })
    ]
}
```

Here we will be using most of the webpack plugins installed in previous post.

Now we need to add a line to **package.json**:

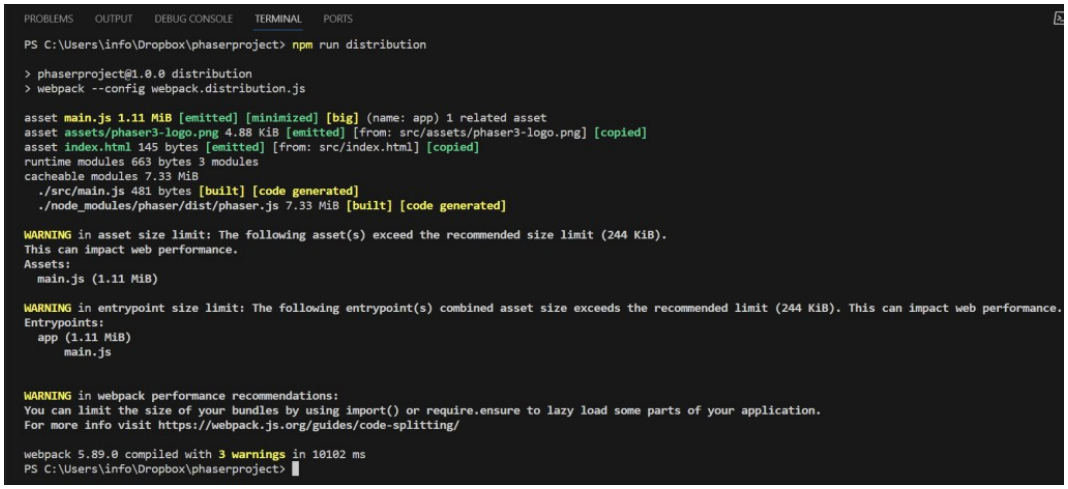
```
{
  "name": "phaserproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "development": "webpack serve --open --config
webpack.development.js",
    "distribution": "webpack --config webpack.distribution.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "phaser": "^3.70.0"
  },
  "devDependencies": {
    "clean-webpack-plugin": "^4.0.0",
    "copy-webpack-plugin": "^11.0.0",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4",
```

```
    "webpack-dev-server": "^4.15.1"
  }
}
```

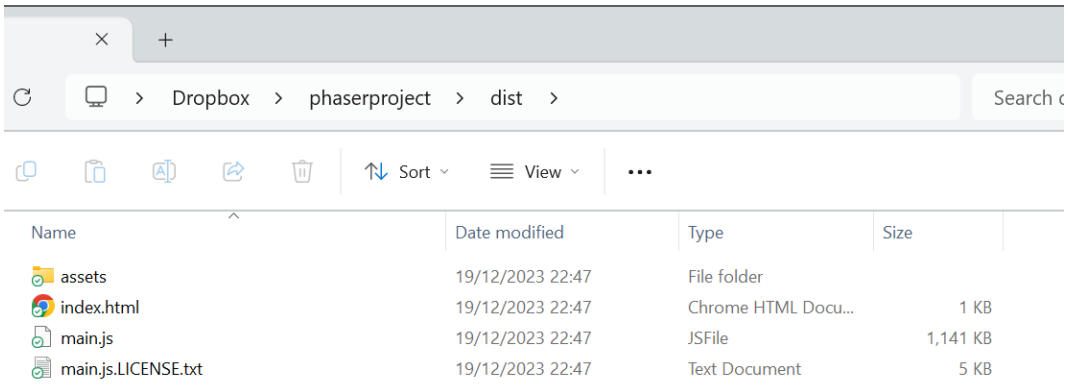
Now, since our basic project is finished and ready for distribution, just enter in the terminal:

**`npm run distribution`**

This way:




After some seconds, your game will be exported in **dist** folder, which should look this way:

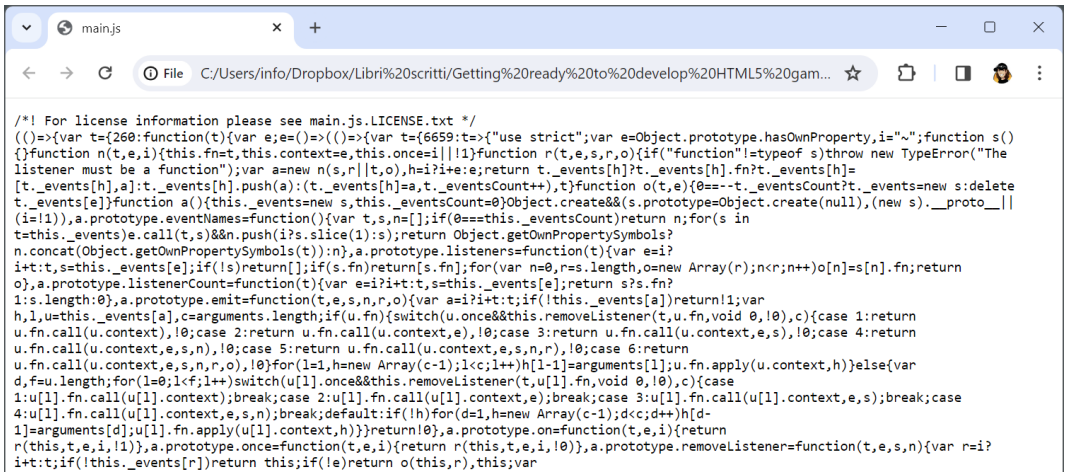



As you can see, we have only one big JavaScript file called **main.js**, which includes our scripts, Phaser and any other external library listed in

**package.json** under dependencies.

This is the folder you will upload to your website or give to portals that distribute content, and it can be also opened with a web server.

In this case the content of **main.js** is even more complex, because it has also been minified.




Minification involves the removal of unnecessary characters from the source code without affecting its functionality.

The primary goal of minifying scripts is to reduce their file size, which can lead to faster downloads and improved website or application performance.

And what about live reloading?

If while you are developing in the production mode you change the contents of any file within the **src** folder, for example the logo image, the background color of the page, or the rotation speed, the page hosting the result will refresh itself showing the new result.

We are done with Phaser and JavaScript, so let’s move to TypeScript.

# What is TypeScript?

TypeScript is a programming language developed by Microsoft.

It is a superset of JavaScript, which means that all valid JavaScript code is also valid TypeScript code, but TypeScript extends JavaScript by adding some interesting features, such as:

**Static Typing**, allowing developers to declare types for variables and catch type-related errors during the development process, leading to more robust and maintainable code.

**Object-Oriented Programming** (OOP) including classes, interfaces, and inheritance. This makes it easier to structure and organize code in a more modular and reusable way.

**Interfaces**, which allow developers to define contracts for object shapes. This helps in creating more predictable and maintainable code.

Remember you cannot run TypeScript directly in the browser because browsers understand and execute JavaScript, not TypeScript.

TypeScript needs to be transpiled into JavaScript before it can be executed in a browser.

And let’s not forget that any TypeScript programmer can write JavaScript, but not every JavaScript programmer can write TypeScript, so why not move up a level?

# How to install TypeScript

To add TypeScript to our project, just enter the terminal:

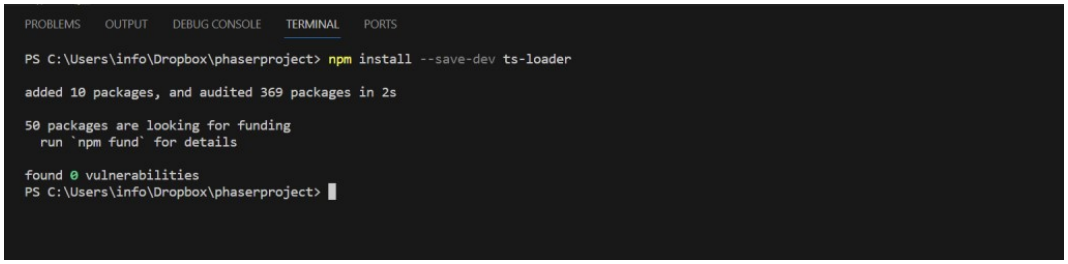
```
npm install --save-dev typescript
```

This way:


Then we need to install the loader with

```
npm install --save-dev ts-loader
```

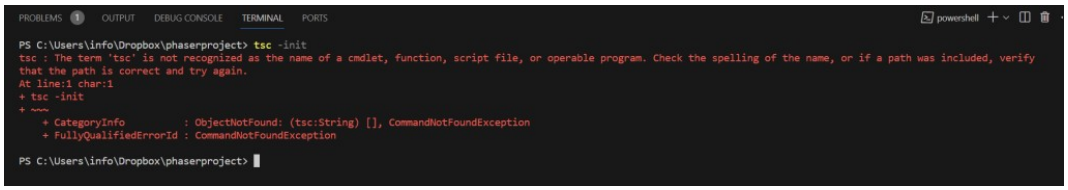
This way:




Now it's time to init TypeScript with:

```
tsc -init
```

This way:

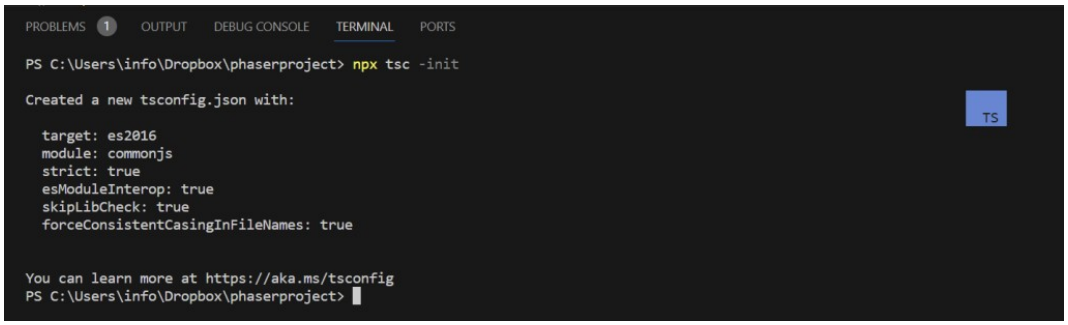



Everything should already work like this on the first shot, but in some cases you could get an error like the one shown above, saying the term 'tsc' is not recognized.

Sometimes you just need to restart Visual Studio Code and launch again the init command, but in a few cases you will have to use npx:

```
npx tsc init
```

This way:




This time it will work without any problem.

But wait, what happened?

In theory, the above error could have been avoided by installing TypeScript globally.

The point is you should never install packages globally, as you may want each project to use a different version of the same package.

This is where npx comes to help us.

npx is a package runner tool that comes with Node.js, providing a convenient way to execute Node.js packages or binaries without the need for global installations.

It stands for "Node Package eXecute".

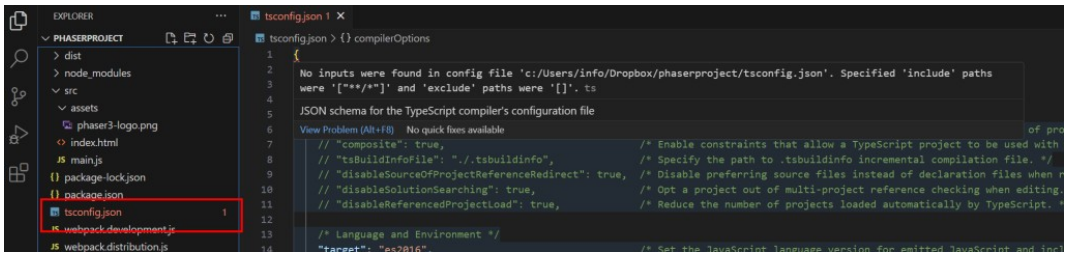
npx was introduced to address the issue of having to install and manage global npm packages for command-line tools, especially when you need a tool infrequently or only for a specific project.

Were you able to install TypeScript even without npx?

Good for you, in any case now you have one more piece of information.

# From JavaScript script to TypeScript

Problems may not be over anyway, because a new file called **tsconfig.json** has been created, but it could display an error like “No Inputs were found in config file”.




Never mind, we’ll fix it later, at the moment let’s have a look at **tsconfig.json** and change the line with **strictPropertyInitialization** property by removing the comment and setting it to **false**:

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig to read more about this file */

    // rest of the file

    /* Type Checking */

    // rest of the file

    "strictPropertyInitialization": false,

    // rest of the file

  }
}
```

It’s under the section called **Type Checking**.

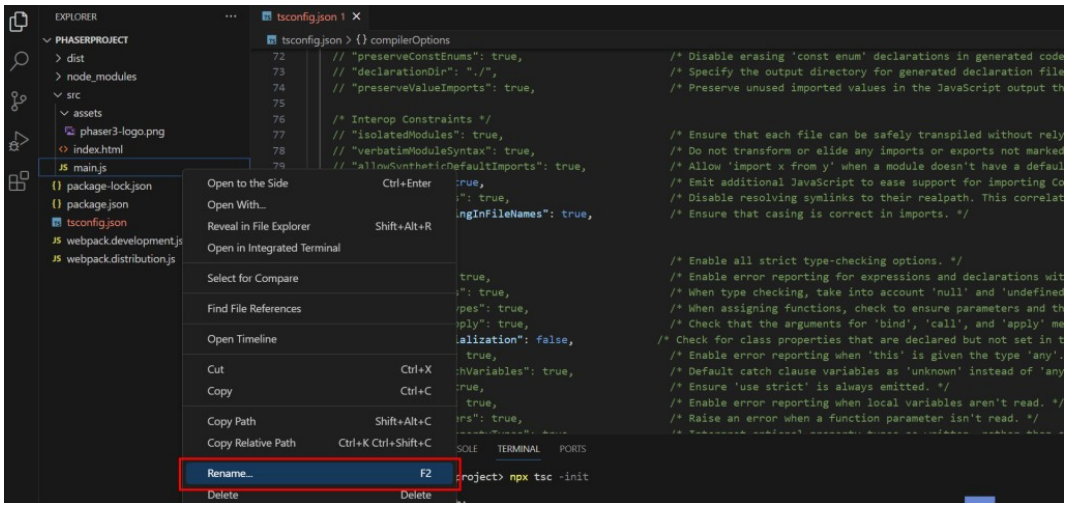
When **strictPropertyInitialization** property is set to **true**, TypeScript will raise an error when a class property was declared but not set in the

constructor.

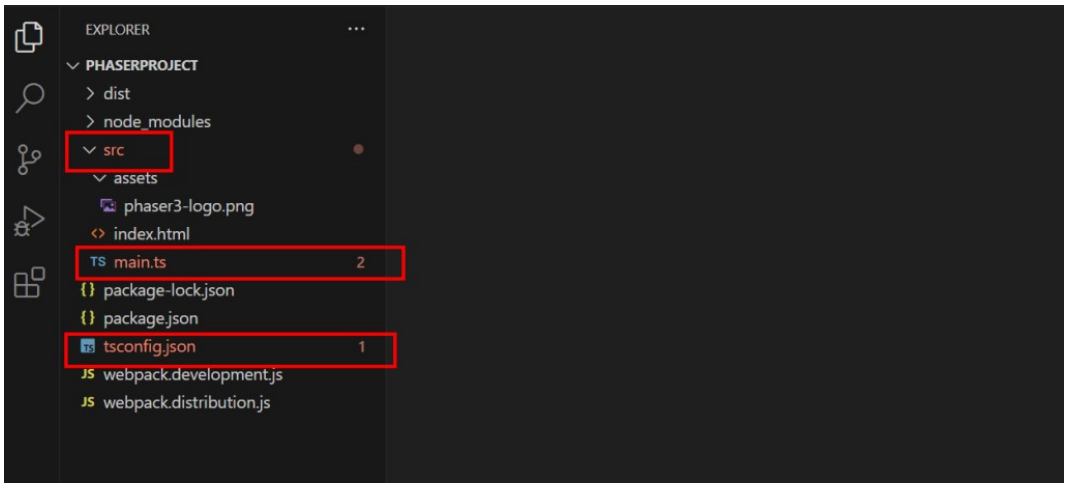
It could be quicker, sometimes, to have class properties which aren't necessarily set in the constructor, so let's set it to false and move on.

Now we have to convert our JavaScript file into TypeScript, and the first thing to do is to rename **main.js**.

In Explorer panel, right click on **main.js** and select **Rename**.




Call it **main.ts**, where **ts** is the extension used by TypeScript files:




Cool, now you have more errors, but we are going to fix them all, for good.

First, this is the new source code of **main.ts**, which is basically the official Phaser example rewritten with TypeScript:

```
import 'phaser';

class PlayGame extends Phaser.Scene {
    image: Phaser.GameObjects.Image;
    constructor() {
        super("PlayGame");
    }
    preload(): void {
        this.load.image('logo', 'assets/phaser3-logo.png');
    }
    create(): void {
        this.image = this.add.image(400, 300, 'logo');
    }
    update(): void {
        this.image.rotation += 0.01;
    }
}

let configObject: Phaser.Types.Core.GameConfig = {
    scale: {
        mode: Phaser.Scale.FIT,
        autoCenter: Phaser.Scale.CENTER_BOTH,
        parent: 'thegame',
        width: 800,
        height: 600
    },
    scene: PlayGame
};

new Phaser.Game(configObject);
```

Then, we need to change a bit **webpack.development.js** and **webpack.distribution.ts** to tell them now we are working with TypeScript.

Basically we have to tell webpack to convert our TypeScript files into vanilla JavaScript so that they can be run in a web browser.

Let's start from **webpack.development.js**:

```
const path = require('path');

module.exports = {
    entry : {

        // this is our entry point, the main TypeScript file
        app: './src/main.ts',
    },
    output : {

        // this is our output file, the one which bundles all libraries
        filename : 'main.js',

        // and this is the path of the output bundle, "dist" folder
        path : path.resolve(__dirname, 'dist'),
    },

    // we are in development mode
    mode : 'development',

    // we need a source map
    devtool : 'inline-source-map',

    // development server root is "src" folder
    devServer : {
        static : './src'
    },

    // list of extensions to resolve, in resolve order
    resolve: {
        extensions: [ '.ts', '.tsx', '.js' ]
    },

    // loader to handle TypeScript file type
    module: {
        rules: [{
            test: /\.tsx?$/,
            use: 'ts-loader',
```

```
            exclude: /node_modules/
        }]
    }
};
```

As you can see, the entry point has been changed to main.ts since we are working with TypeScript, but the output file remains a JavaScript file because browsers aren't able to run TypeScript.

Then, **webpack.distribution.js**:

```
const path = require('path');

// here we use the plugins to clear folders and copy folder content
const CopyPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
    entry: {

        // this is our entry point, the main TypeScript file
        app: './src/main.ts',
    },
    output: {

        // this is our output file, the one which bundles all libraries
        filename: 'main.js',

        // and this is the path of the output bundle, "dist" folder
        path: path.resolve(__dirname, 'dist'),
    },

    // we are in production mode
    mode: 'production',
    plugins: [

        // here we clean the destination folder
        new CleanWebpackPlugin({
            cleanStaleWebpackAssets: false
        }),
```

```
        // here we copy some files to destination folder.
        // which files?
        new CopyPlugin({
            patterns: [
                {
                    // src/index.html
                    from: 'index.html',
                    context: 'src/'
                },
                {
                    // every file inside src/assets folder
                    from: 'assets/*',
                    context: 'src/'
                }
            ]
        })
    ],

    // list of extensions to resolve, in resolve order
    resolve: {
        extensions: [ '.ts', '.tsx', '.js' ]
    },

    // loader to handle TypeScript file type
    module: {
        rules: [{
            test: /\.tsx?$/,
            use: 'ts-loader',
            exclude: /node_modules/
        }]
    }
};
```

At this time, you are ready to build HTML5 games with TypeScript.

You will work with the development server, and once you are satisfied with the result, publish a distributable JavaScript version of your work along with all required assets.

Just like in previous steps, you can start your development server with:

```
npm run development
```

And create your distributable game with:

```
npm run distribution
```

What if you still have an error in **tsconfig.json**?

Try to close and reopen the project folder in Visual Studio Code, and if the error persists, you can get rid of it by adding these lines to **tsconfig.json**:

```
{
  "compilerOptions": {

  // this part remains unchanged

  },
  "include": [
    "./src/**/*.ts"
  ]
}
```

Now you are ready to start building HTML5 games with Phaser and TypeScript the proper way, but there's something more to see, just in case your project is bigger than one file.

## Organizing your scripts and folders

The Phaser project we just built is a very simple example and its code fits comfortably in one single file.

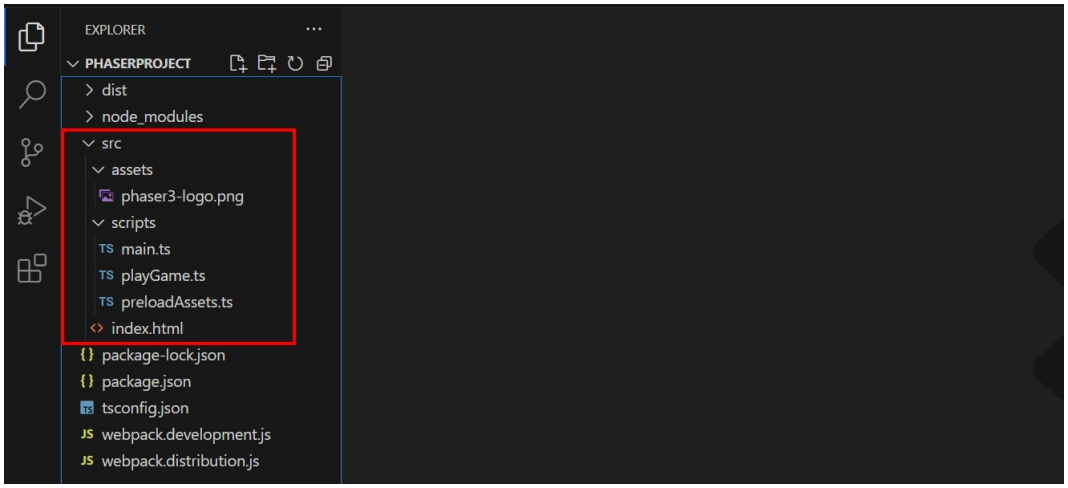
I hope, however, that you will manage far more complex projects that need many more files, and even if not, it is always a good practice to break up the code to make it more easily reusable.

So the idea is to divide the code into chunks of code in multiple files.

Create a folder inside **src** folder and call it **scripts**.

Move main.ts inside scripts folder, and create two more TypeScript files: **preloadAssets.ts** and **playGame.ts**.

The project folder now should look this way:




In **main.ts** we'll only create the configuration object and start the game:

```
import 'phaser';
import { PreloadAssets } from './preloadAssets';
import { PlayGame } from './playGame';

let configObject : Phaser.Types.Core.GameConfig = {
    scale : {
        mode : Phaser.Scale.FIT,
```

```
        autoCenter : Phaser.Scale.CENTER_BOTH,
        parent : 'thegame',
        width : 800,
        height : 600
    },
    scene: [PreloadAssets, PlayGame]
};

new Phaser.Game(configObject);
```

**preloadassets.js** will take care of preloading all the necessary assets, and then run the main game.

```
export class PreloadAssets extends Phaser.Scene {

    constructor() {
        super({
            key : 'PreloadAssets'
        });
    }

    preload() : void {
        this.load.image('logo', 'assets/phaser3-logo.png');
    }

    create() : void {
        this.scene.start('PlayGame');
    }
}
```

Finally, **playGame.ts** contains the game itself, if we can call "game" a rotating image.

```
export class PlayGame extends Phaser.Scene {

    image : Phaser.GameObjects.Image;

    constructor() {
        super({
```

```
            key : 'PlayGame'
        });
    }

    create(): void {
        this.image = this.add.image(400, 300, 'logo');
    }

    update(): void {
        this.image.rotation += 0.01;
    }
}
```

Since the path of main.ts changed, we need to update **webpack.development.js** this way:

```
const path = require('path');

module.exports = {
    entry : {

        // this is our entry point, the main TypeScript file
        app: './src/scripts/main.ts',
    },

    // ...
    // same as before
    // ...
};
```

And **webpack.distribution.js** needs to be updated too:

```
const path = require('path');

// here we use the plugins to clear folders and copy folder content
const CopyPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
    entry: {
```

```
        // this is our entry point, the main TypeScript file
        app: './src/scripts/main.ts',
    },

    // ...
    // same as before
    // ...

};
```

Now you can keep working in a more organized way.

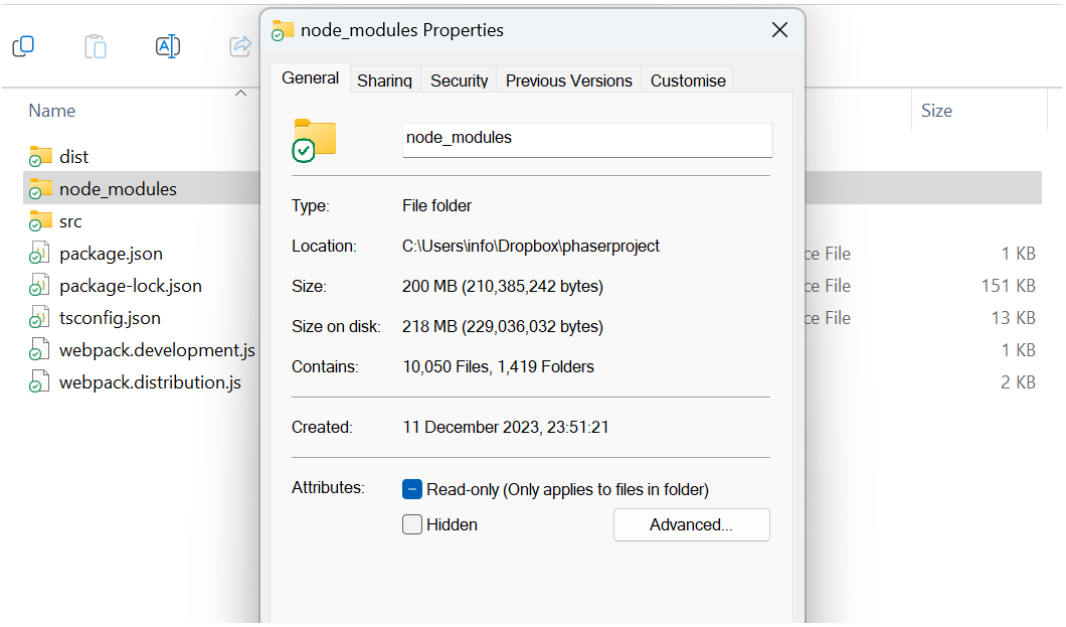
Remember: splitting your programming scripts into smaller, modular pieces offers several benefits that contribute to better code organization, maintainability, testing, readability and collaboration.

## Moving and distributing your whole project

Sometimes you may need to move the project from one disk to another, or to distribute somewhere.

Have you wondered how much hard disk space such a project takes up?

While the space taken up by the project itself depends on the number of scripts and assets you are using, it is interesting to note how much space the **node_modules** folder takes up:




That's more than 200MB!

And where did this folder come from?

The **node_modules** folder is a directory commonly created in projects that use Node.js.

When you install dependencies for a Node.js project using npm, such as Phaser, TypeScript and all npm plugins, these dependencies are downloaded and stored in the **node_modules** folder.

Imagine hosting a dozen or so projects on your hard drive, or having to

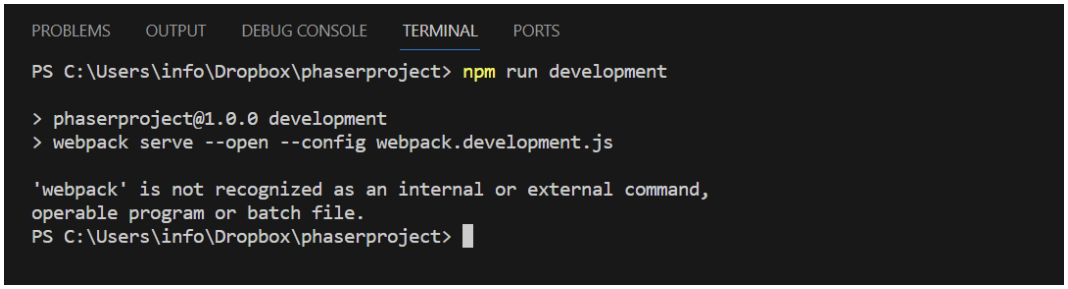
share some of them with collaborators, or make them downloadable from a website as I usually do with the tutorials I post on my blog.

Fortunately, when we are not working on a project or need to deploy it, we can get rid of **node_modules** folder.

Logically once deleted, you will no longer be able to test or create final builds of your project as the system will return an error, as in this case where we try to run

```
npm run development
```

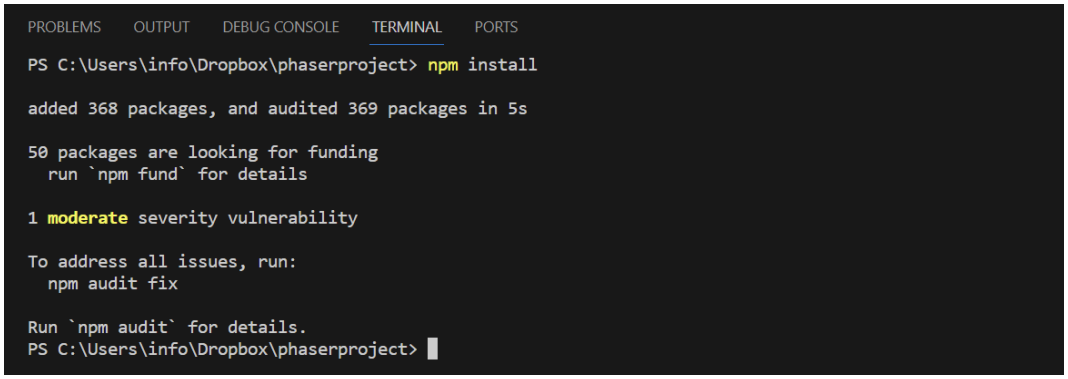
this way:




But never mind: just execute

```
npm install
```

and all the dependencies will be installed again, creating a new **node_modules** folder.




At this time you can keep working on the project.

## Some more boring words to introduce Vite

In the realm of programming, change is a constant and inevitable reality.

Languages, frameworks, libraries, and development paradigms are continually being updated or replaced, often in an effort to address new challenges that emerge as technology and user needs evolve.

Tools that seem imperative today may become obsolete in a few years, overtaken by solutions that are more efficient, scalable, or easy to use.

This pace of transformation is not limited to technical tools, but also extends to development practices and methodologies, such as adopting new approaches to project management or code optimization.

Programmers, therefore, cannot afford to stand still: they must cultivate a mindset of continuous learning, remaining curious and open to new things.

Adapting to this rapidly changing landscape means not only acquiring new technical skills, but also developing the ability to recognize changes that bring value and those that risk being merely passing fads.

It is a challenge that, while requiring commitment, also offers tremendous opportunities for personal and professional growth, enabling the creation of innovative, efficient and sustainable solutions in a constantly changing technological world.

This is why I am also introducing Vite, another build tool which may overtake webpack in the near future.

## What is Vite?

Vite is a modern build tool and development server designed to improve the development experience by providing fast and efficient workflows for frontend projects.

It was created by Evan You, the creator of Vue.js, and is particularly popular in the Vue.js ecosystem but can also be used with other frameworks like React, Svelte, and vanilla JavaScript. And Phaser, of course.

There are some key fatures which are really useful in Phaser HTML5 game development:

**Support for Hot Module Replacement (HMR)**: Vite has a highly efficient HMR system, which only updates the changed parts of the code without needing a full page reload. This results in a smooth and quick development experience.

**Instant Start**: Unlike traditional bundlers that require an initial bundle to be created before starting the dev server, Vite serves files directly from the source, meaning there's no need for a slow startup time.

**Optimized Build with Rollup**: For production builds, Vite uses Rollup, a highly efficient bundler that optimizes code for smaller bundle sizes and better performance.

**TypeScript and JSX Support**: Vite includes built-in support for TypeScript, JSX, and other modern JavaScript features without the need for complex configuration.

Is Vite better than webpack? It's a complex question, because every developer has his own habits and way of working.

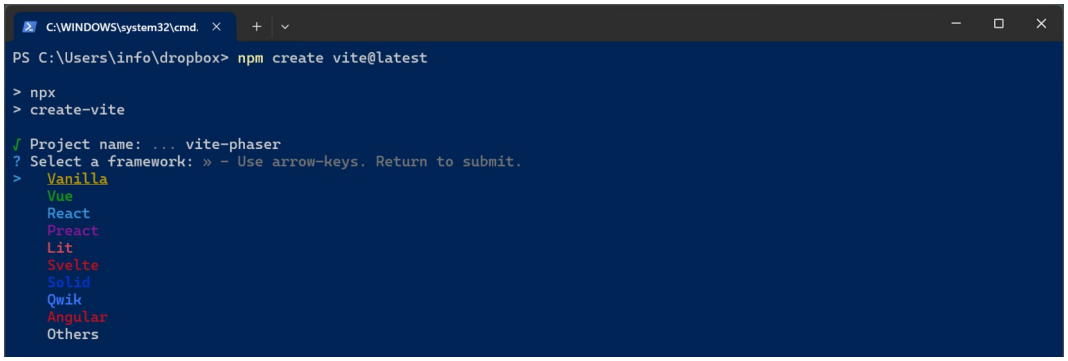
My very personal preference, regarding HTML5 game development with Phaser and TypeScript, leans toward Vite, but I recommend that you try both Vite and webpack, then draw your own conclusions.

# Creating a TypeScript Vite project

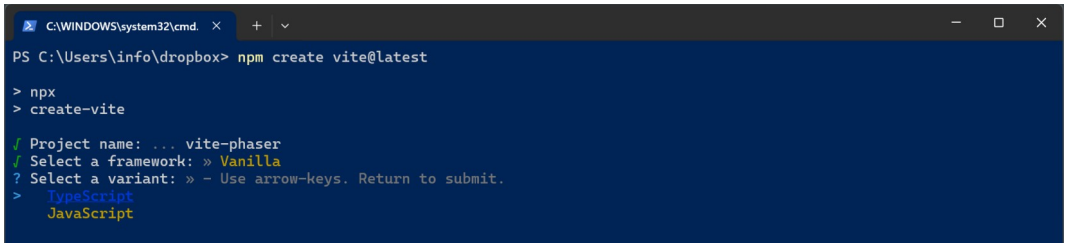
With PowerShell go into the directory in which you want to create your TypeScript Vite project, then write:

```
npm create vite@latest
```

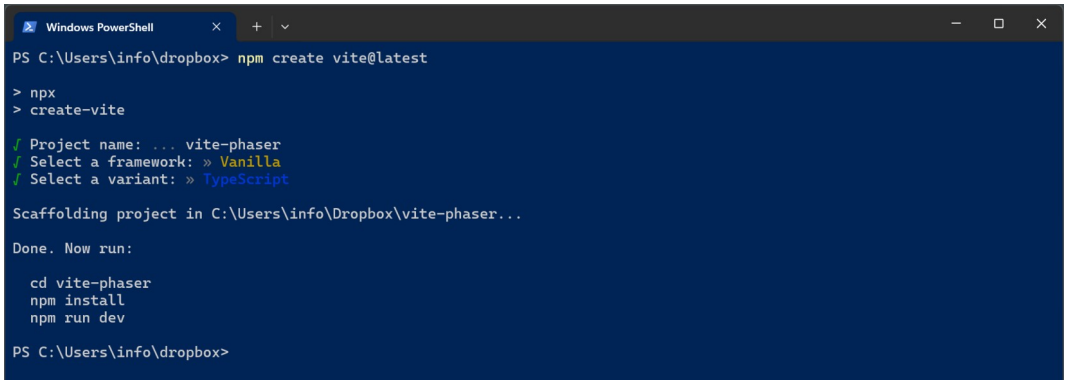
Enter project name then choose **Vanilla**, this way:




Then choose **TypeScript**, this way:




Now you will be prompted to enter some more commands from the shell.



Stop using PowerShell and let's make things quicker.

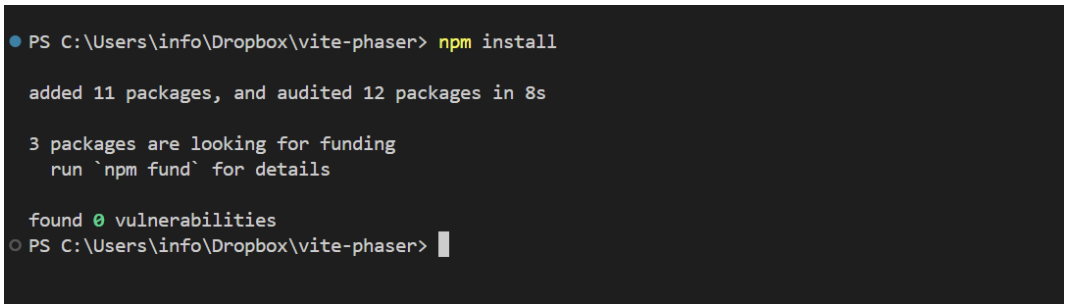
Launch Visual Studio Code, opening the folder Vite just created.

In my case the folder is **vite-phaser** because this is how I called the project.

Now open a terminal with **Terminal > New Terminal**, then enter

```
npm install
```

This way:




Now your project is ready to be run, with

```
npm run dev
```

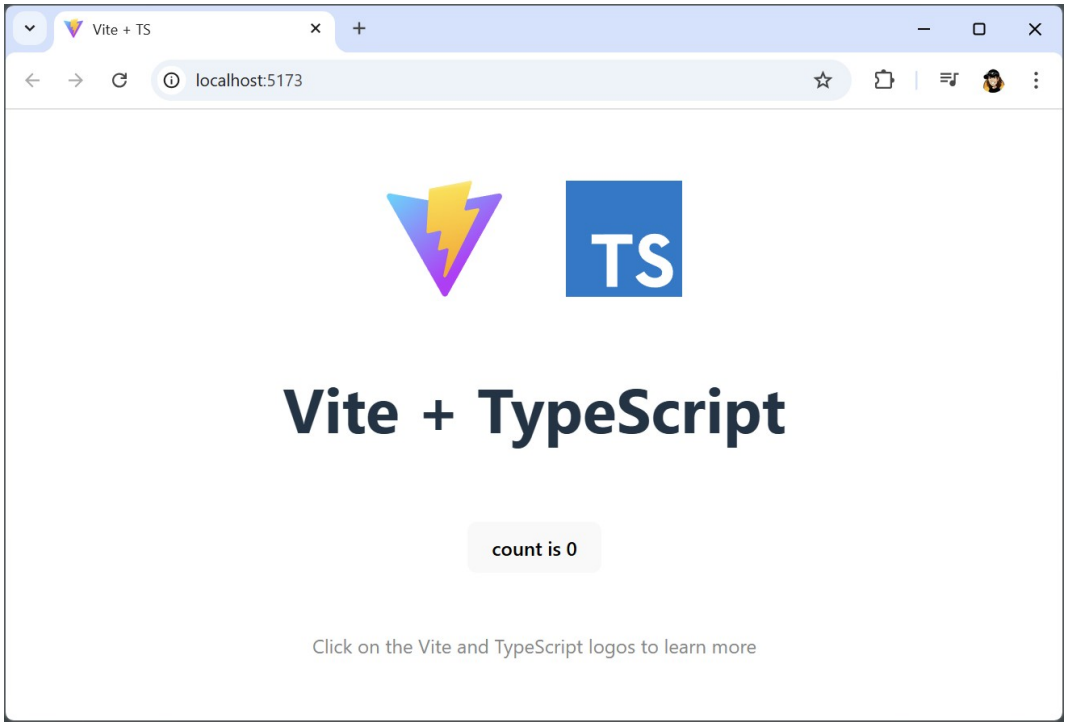
This way:


If you are using Dropbox you may get an error:


It says the resource is busy and locked. There are various workaround to

solve this issue, but the simplest thing you can do is to temporarily puase Dropbox syincing.

Now, if you point your browser to **http://localhost:5173/**, or any address shown in the terminal, you should see the default Vite project, this one:




At this time, we are ready to make some changes, but first let's see how Hot Module Replacement works.

## What is Hot Module Replacement?

Both Live Reloading, explained before, and Hot Module Replacement are techniques used in development tools to improve the developer experience by automatically updating the browser to reflect changes in the code without requiring manual refreshes.

However, they differ in how they work and what they update.

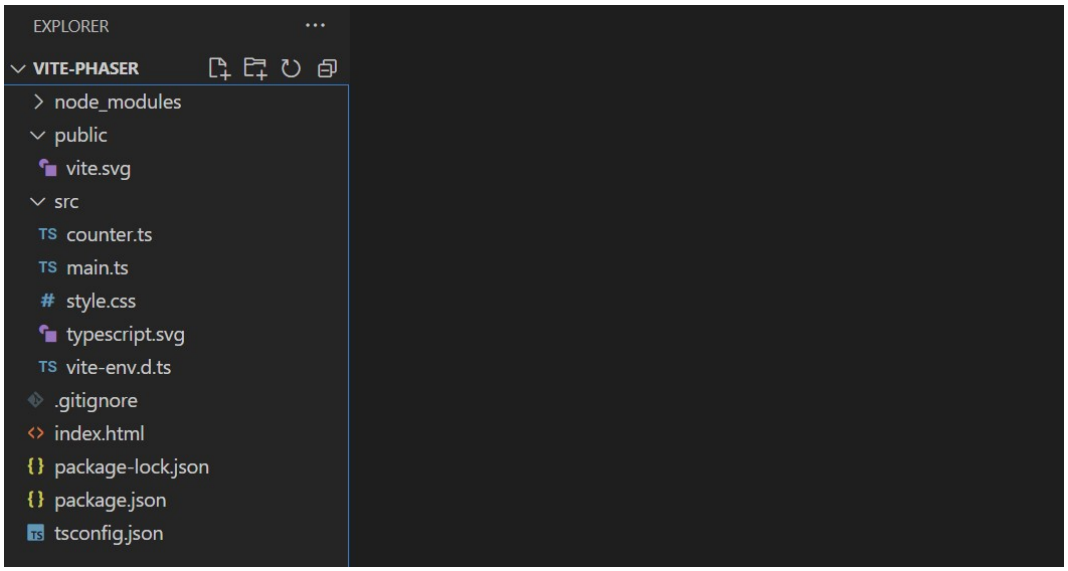
**Live Reloading** automatically refreshes the entire browser page when a file change is detected in the project: the development server detects a change in the source code, then the browser reloads the entire page to reflect the updated content.

**Hot Module Replacement** is a more advanced technique that updates only the changed modules in a running application without reloading the entire page: the development server detects a change in the source code, then only the specific modules that have changed are replaced in the browser, so the application continues to run without a full reload.

This is very important because will allow us to speed up the development and debug process.

# The structure of a Vite project

Let's have a look at file tree in **Explorer** panel:




From top to bottom we can find:

**node_modules** folder: contains all the dependencies that are required for the application to run, and must not be touched.

**public** folder: special directory used to store static assets that should be served as they are, without any processing or transformation during the build process. All files placed in this folder are copied directly to the root of the final build output without modifications, making it an ideal place for assets like images, fonts, or static configuration files.

**src** folder: special directory where the core source code of our application resides. It contains all the TypeScript scripts and other files which will be processed, transformed, and bundled by Vite when creating a production build.

**src\vite-env.d.ts**: TypeScript declaration file that provides type definitions for Vite-specific features and configuration. Don't touch it as we will remove it.

**.gitignore**: configuration file used in Git repositories to specify intentionally untracked files that Git should ignore. It helps prevent certain files, directories, or patterns from being included in version control. If you don't use Git, you can delete it.

**index.html**: the entry point of the application. Think about it as the home page.

**package.json**: core component of the project, used to manage project metadata, dependencies, scripts, and configurations.

**package-lock.json**: file automatically generated that ensures consistency in dependency management. Unlike package.json, this shouldn't be manually edited.

**tsconfig.json**: configuration file used in TypeScript projects to specify compiler options, defining how TypeScript should transpile code and what features to enable.

Time to add Phaser to the project.

## Adjusting Vite project to work with Phaser

In order to work with Phaser, we must first install (guess what?) Phaser, but at the moment let's have a look at **package.json** file:

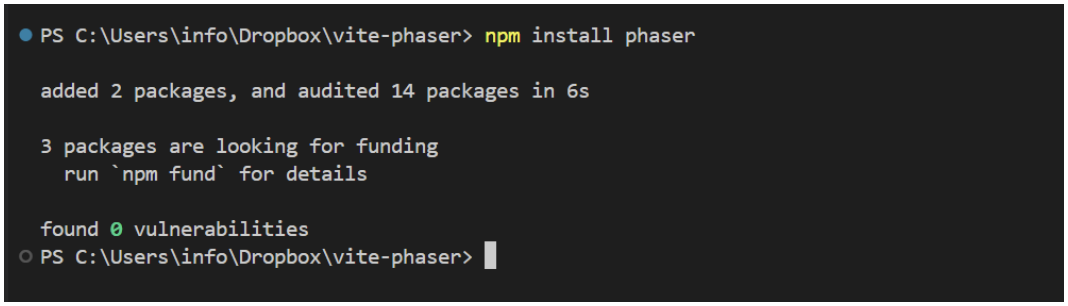
```
{
  "name": "vite-phaser",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "typescript": "~5.6.2",
    "vite": "^6.0.5"
  }
}
```

This is the default Vite package.json file, with some scripts and the latest TypeScript version.

To install Phaser, and then see what changes, write in Terminal:

`npm install phaser`

This way:




The latest Phaser version will be instlled.

Now if you look at **package.json**, you should see these new lines:

```
{
  "name": "vite-phaser",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "typescript": "~5.6.2",
    "vite": "^6.0.5"
  },
  "dependencies": {
    "phaser": "^3.87.0"
  }
}
```

Now the project knows we are using Phaser.

In the same file, change these two lines this way:

```
{
  "name": "vite-phaser",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite --port 8080 --open",
    "build": "tsc && vite build",
    "preview": "vite preview --port 8080 --open"
  },
  "devDependencies": {
    "typescript": "~5.6.2",
    "vite": "^6.0.5"
  },
  "dependencies": {
```

```
    "phaser": "^3.87.0"
  }
}
```

What happened? When we launched

`npm run dev`

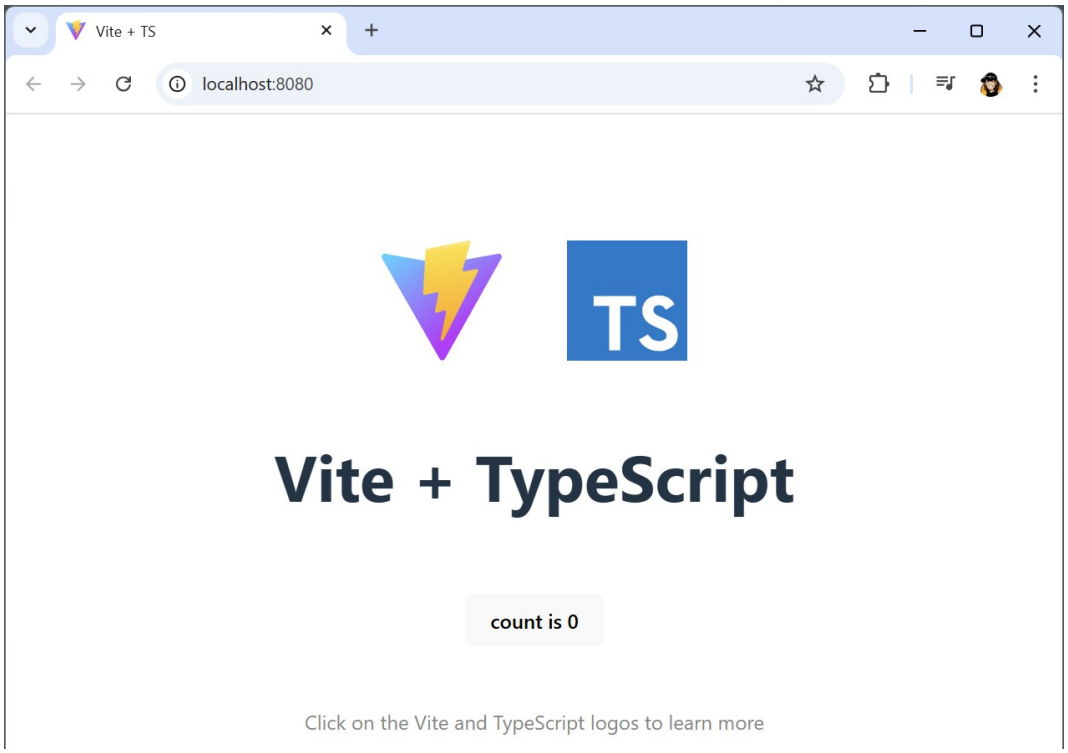
from Terminal, we just executed the script at line 7.

Now with **--port 8080** we set the port to 8080 and with **--open** we automatically open the browser window at the end of the process.

Same concept is applied to line 9. Try by yourself, write in Terminal

`npm run dev`

And you will see the page automatically open at **http://localhost:8080/**.




Do you see it? Now it's port 8080.

## Remove unnecessary files

Since this course is focused on HTML5 game development with Phaser and TypeScript, we can safely remove some files to make file tree simpler.

I removed **.gitignore**, everything inside **public** folder and everything inside **src** folder except **main.ts** and **style.css**. Since I also removed **vite-env.d.ts** which contains some TypeScript information, I am going to add a couple of lines to **tsconfig.json**:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "module": "ESNext",
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "skipLibCheck": true,

    /* added by me */
    "types": ["vite/client"],
    "strictPropertyInitialization": false,

    /* Bundler mode */
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "isolatedModules": true,
    "moduleDetection": "force",
    "noEmit": true,

    /* Linting */
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true,
    "noUncheckedSideEffectImports": true
  },
  "include": ["src"]
}
```

The first line added acts in the place of **vite-env.d.ts**, while next line sets **strictPropertyInitialization** to **false** to prevent TypeScript to raise an error when a class property was declared but not set in the constructor.

It's a matter of convenience.

Now it's time to write some code to turn this project into the official Phaser example you can find at https://labs.phaser.io/view.html?src=src\game%20objects\images\image%20rotation.js.

First, let's rewrite **index.html**:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Phaser + TypeScript + Vite</title>
    <script type="module" src="/src/main.ts"></script>
  </head>
  <body>
    <div id="thegame"></div>
  </body>
</html>
```

We can see three important things:

1 – There isn't any stylesheet. This does not mean that we will not include styles in our web page, but we will do it somewhere else, in a more efficient way that also allows for their minification.

2 – We are including a TypeScript file. I know web browser can't run TypeScript files, but don't worry: Vite will take care of everything.

Also look at the path: **/src/main.ts**. This is the file we left in **src** folder.

3 – In the body there is a div whose id is **thegame**. Phaser project will be rendered inside that element.

Talking about css, let's edit **style.css** in **src** folder:

```
/* remove margin and padding from all elements */
* {
  padding : 0;
  margin : 0;
}

/* set body background color */
body {
  background-color : #000000;
}

/* Disable browser handling of all panning and zooming gestures. */
canvas {
  touch-action : none;
}
```

And now the most important file, **main.ts** in **src** folder:

```
import 'phaser';
import './style.css';

class PlayGame extends Phaser.Scene {
    image : Phaser.GameObjects.Image;
    constructor() {
        super('PlayGame');
    }
    preload() : void {
        this.load.image('logo', 'assets/phaser3-logo.png');
    }
    create() : void {
        this.image = this.add.image(400, 300, 'logo');
    }
    update() : void {
        this.image.rotation += 0.01;
    }
}

let configObject : Phaser.Types.Core.GameConfig = {
    scale : {
        mode        : Phaser.Scale.FIT,
```

```
        autoCenter  : Phaser.Scale.CENTER_BOTH,
        parent      : 'thegame',
        width       : 800,
        height      : 600
    },
    scene: PlayGame
};

new Phaser.Game(configObject);
```

Explaining the source code is beyond the scope of this guide, but it's just the TypeScript version of the official Phaser example.

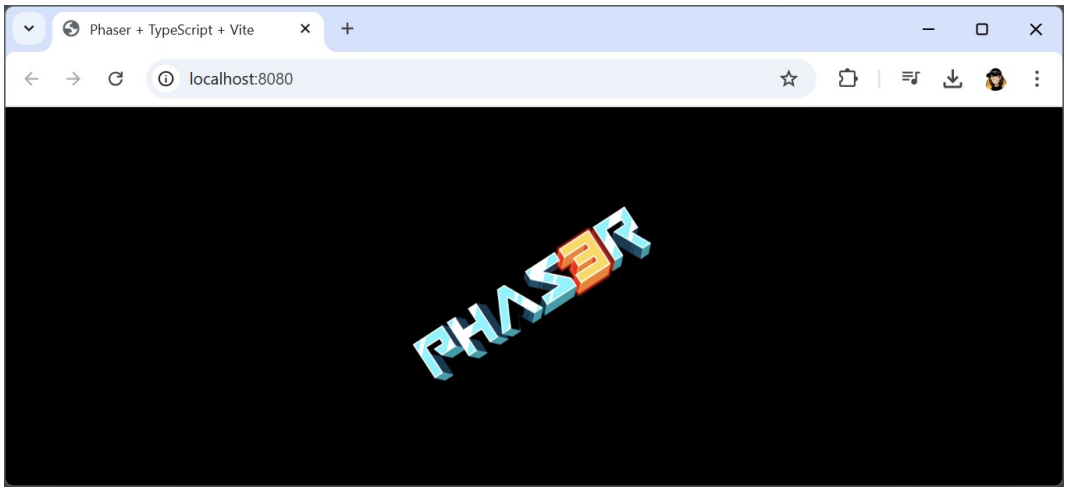
However, look at the second line: here is where we include **style.css**.

You will also need the Phaser logo to be copied in **public/assets** folder:

Now everything is ready to launch the Phaser project from Terminal with

`npm run dev`

and you should see your Phaser page opening with something like this:




This is running in our local server. How to export it?

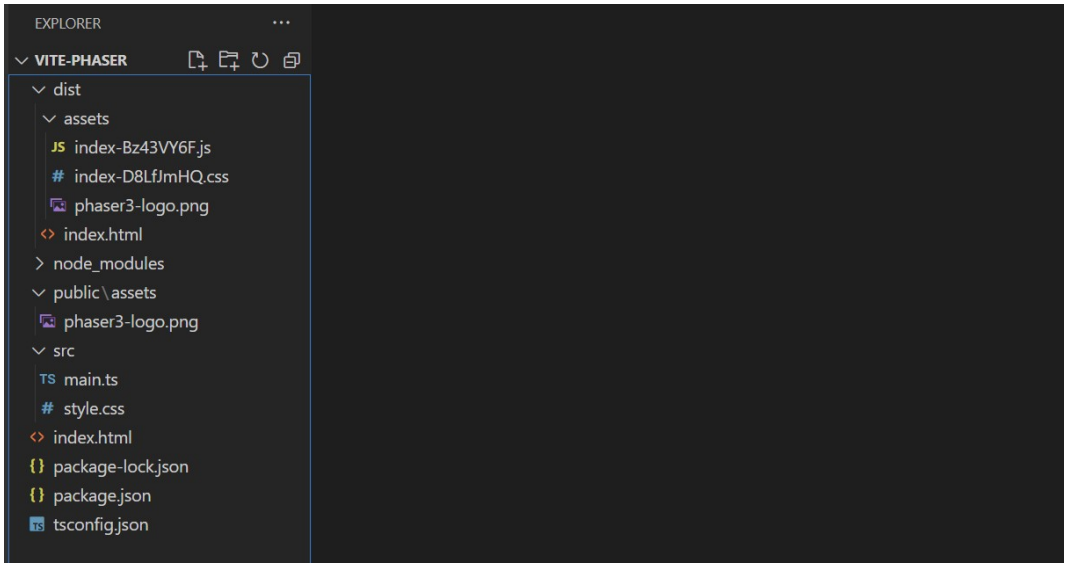
# Exporting Phaser Vite project for distribution

Vite has a built in script to export project, so from Terminal just enter

**`npm run build`**

This way:


Now you should find your project ready for distribution in **dist** folder; let's have a look at it:




What those strange names in assets folder are?

It's the way Vite handles files built for distribution. Never mind file names, there's everything we need.

Let's have a look at **index.html**:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Phaser + TypeScript + Vite</title>
    <script type="module" crossorigin src="/assets/index-
Bz43VY6F.js"></script>
    <link rel="stylesheet" crossorigin href="/assets/index-
D8LfJmHQ.css">
  </head>
  <body>
    <div id="thegame"></div>
  </body>
</html>
```

As you can see, it changed a bit according to new file names, and the style sheet has been included.

You probably noticed we received a warning: **Some chunks are larger than 500 kB after minification**. It's ok, but we don't want to see this message anymore.

Vite reads, if available, a configuration file called **vite.config.js** in root folder.

Let's create it with this content:

```
export default {
    build: {
        chunkSizeWarningLimit: 1500
    }
}
```

Now the limit is 1500KB, but you can set ever higher, and the warning

won’t pop up anymore.

Let’s have a look at the css file:

```
*{padding:0;margin:0}body{background-color:#000}canvas{touch-
action:none}
```

It has been minified. Good. But if you look at the js file you will find a lot of comments like this one:

```
CSS
/* @author       samme
 * @copyright    2013-2024 Phaser Studio Inc.
 * @license      {@link https://opensource.org/licenses/MIT|MIT
License}
*/
```

This is absolutely no good because we want to reduce its size as much as we can.

This is when Terser comes into play.

## What is Terser?

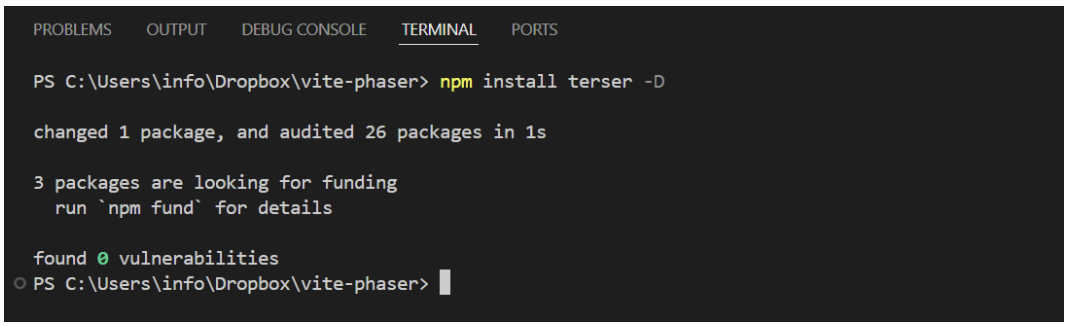
Terser is a modern JavaScript minification tool designed to compress and optimize JavaScript code.

It reduces the size of JavaScript files by removing unnecessary characters like whitespace and comments, and minimizing JavaScript files for faster loading times and better performance.

Let's install Terser by writing in Terminal

```
npm install terser -D
```

This way:




Your **package.json** file should change this way:

```
{
  "name": "vite-phaser",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite --port 8080 --open",
    "build": "tsc && vite build",
    "preview": "vite preview --port 8080 --open"
  },
  "devDependencies": {
    "terser": "^5.37.0",
    "typescript": "~5.6.2",
    "vite": "^6.0.5"
  },
```

```
  "dependencies": {
    "phaser": "^3.87.0"
  }
}
```

What does that **-D** at the end of **npm install terser** mean?

The **-D** flag at the end of an **npm install** command stands for **--save-dev**.

It tells npm to install the specified package and add it as a development dependency in your project's **package.json** file.

Now let's configure it by adding some lines to **vite.config.js** file:

```
export default {
    build: {
        chunkSizeWarningLimit: 1500,
        minify: 'terser',
        terserOptions: {
            format: {
                comments: false
            }
        }
    }
}
```

We are telling Terser to remove the comments. There could be many more options to set, but this goes beyond the scope of this guide.

Now write in Terminal once more

```
npm run build
```

and look at the content of the JavaScript file in **dist** folder; there shouldn't be any comment.

In my case the total size dropped from 1,473.41kB to 1,190.65kB.

**dist** folder now contains the game ready to be published, but how to test it? With a local web server?

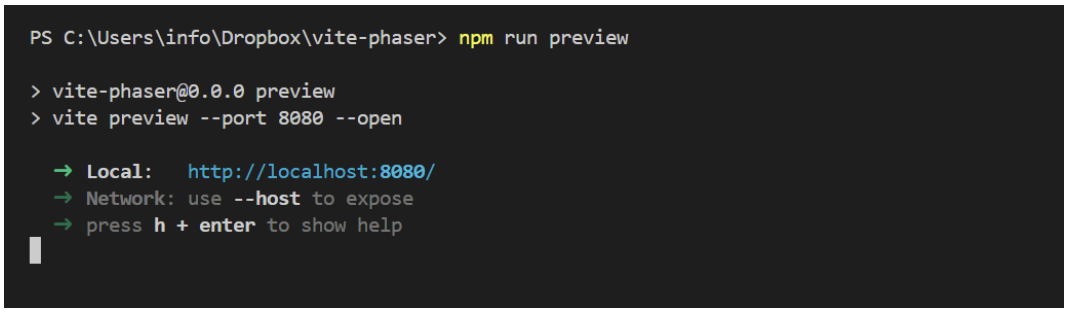
## Testing your distributable project

Vite has everything you need to test the content in **dist** folder.

From Terminal, just enter

npm run preview

this way:




and a new browser window will open with the content of **dist** folder running in it.

When you are done with development, if you want to save space on your hard disk, you can delete **dist** and **node_modules** folders, as well as **package-lock.json**.

To start working again on that project, you just need to make npm create them again with

```
npm install
```

And now you can use Vite for your Phaser projects.

# Thank you for reading me!

I really hope this guide has been useful to you and you enjoyed reading it as much as I enjoyed writing it.

Your feedback is very important to me, so in case you find any inaccuracies, any mistakes, or any concepts that you feel could have been better explained, please do not hesitate to let me know.

You can send your feedback by dropping a mail to info@emanueleferonato.com.

Where to go now? You can check my blog at https://www.emanueleferonato.com/ where you will find a lot of examples and prototypes using Phaser and TypeScript.