Jul 23, 2023 11 min read 1977 views express.js node.js typescript pnpm

Explore the dynamic integration of TypeScript and Express.js. Uncover TypeScript's benefits in a MEAN stack, harness its power in monorepos, and enhance your Express server with solid type safety.

Build a MEAN web app - Article Series

Introduction

In the previous part, we explored the key components of the **MEAN** stack and how they work together. We also got a good grasp of the monorepo approach and how to structure folders.

In this part, we will take a hands-on approach and focus on coding and experimenting with the monorepo methodology. Following step-by-step instructions, we will learn how to effectively set up and use the monorepo approach, which will allow us to create web applications that are scalable and easy to maintain. We'll gain valuable insights into seamlessly integrating **Express.js** and **TypeScript**, empowering us to build robust and efficient projects with utmost ease. Exciting stuff, right?

What Makes TypeScript a Great Choice for a MEAN Stack?

Certainly! TypeScript needs no introduction, as its powerful features and developer experience are well-known in the web development community. However, let's highlight a few key elements that will significantly benefit our MEAN stack project in a monorepo environment:

- **Full Stack Consistency:** TypeScript brings uniformity to your entire MEAN stack. Since Angular already uses TypeScript, incorporating TypeScript with Express.js on your server side enhances stack consistency and compatibility. You can maintain the same development practices, whether working on the front or back.
- **Shared Type Definitions:** A common challenge in full-stack JavaScript development is keeping data model definitions consistent between the server and client. In a monorepo setup with TypeScript, you can share type definitions (like interfaces or types) across your front and back. This approach ensures that your MongoDB data models, Express.js services, and Angular components agree about the data structures they are dealing with.
- **Scalability and Maintainability:** As monorepo houses multiple projects within a single repository, they allow for easier code sharing and dependency management. Combining this with TypeScript's strong typing helps create scalable and maintainable web applications that can grow with the project's needs.
- **Enhanced Code Collaboration:** In a monorepo project involving multiple developers, TypeScript's type annotations are effective documentation, promoting seamless communication and comprehension among team members. This fosters improved collaboration and minimizes the chances of code conflicts, resulting in smoother development workflows.

Combining TypeScript with Express.js in a monorepo MEAN stack application provides numerous benefits, including improved code quality, productivity, and maintainability. It ensures consistency, enhances collaboration, and enables scalable web development.

> Read more about monorepo in my recently published article. And check out the monorepo tools website for more information.

Now let's shift our focus to another critical aspect of managing projects efficiently: package management.

PNPM: A performant package manager

pnpm is a package manager for JavaScript that is open-source, efficient, and performance-driven. It is often considered an alternative to npm and Yarn. pnpm stands out in monorepo setups by providing the same features as its counterparts but with unique advantages. Its content-addressable filesystem reduces duplication and saves disk space, resulting in faster and more efficient installations than npm or Yarn. pnpm's workspace feature makes managing multiple packages within a repository easy by linking them together for seamless updates across projects, thereby enhancing workflow efficiency.

Here are some key advantages of using pnpm to manage your monorepo:

- **Enhanced Performance:** Due to its efficient storage approach, pnpm installs packages faster than npm or Yarn. When a package is already in the store, and a project requests its installation, pnpm only creates a hard link between the project's **node_modules** directory and the package in the store, resulting in faster installation times.
- **Workspace Management:** pnpm provides first-class support for managing multiple packages in a monorepo through its workspace feature. You can run scripts, install, update, and link packages across the entire workspace, making it easier to manage interdependent packages. Also, it offers out-of-the-box support for monorepos with features like the workspace:* protocol and `--filter` flag.
- **Single instance storage:** pnpm utilizes a distinctive approach called single instance storage, which involves linking packages in a centralized global storage location on your machine. This innovative method ensures that if multiple projects rely on the same package version, it is stored only once, substantially reducing disk space usage.
- **Strong Compatibility:** pnpm integrates seamlessly with npm and Yarn, ensuring compatibility and smooth transition. Migrating existing projects from npm or Yarn to pnpm requires minimal changes to configurations or package structures, making the adoption process straightforward.

Overall, pnpm provides a reliable and efficient solution for effectively managing large-scale projects, making it a compelling option for monorepo-based development.

> You can use npm or Yarn workspaces instead of pnpm if you prefer.

Let's explore how to implement a monorepo structure using pnpm.

Create a monorepo structure using PNPM

Before installing pnpm, make sure that Node.js is already installed on your system. You can check if Node.js is installed by executing the following command:

```
node -v
# v20.4.0
```

If you don't have Node.js installed, you can download it from the official website. npm is bundled with Node.js, so you'll get both when you install Node.js.

As of the day this article is published, I am utilizing the latest version available, `20.4.0`.

> I highly recommend using **nvm (Node Version Manager)** if you simultaneously work on multiple JavaScript projects. This tool allows you to install, manage, and switch between multiple versions of Node.js on your machine. It's a fantastic tool that can make your work much more efficient.

After setting up Node.js and npm, the next step is to install pnpm. You can use **Brew** (if it's already installed) or npm as an alternative.

```
# using brew
brew install pnpm
# using npm
npm install -g pnpm
```

For more information on installing pnpm, please refere to the **installation guide**, which provides various options.

To ensure that pnpm has been installed correctly, you can verify its version:

```
pnpm -v
# 8.6.7
```

To set up the root of your monorepo, create a new directory. As an example, I will name mine **bugsight**.

Create an **apps** directory within the root directory. Make separate directories for your **client** and **server** applications inside the **apps** directory.

Next, create a **packages** directory at the root level. This will hold shared code that can be used by both the **client** and **server** applications. For instance, you can create a **tsconfig** package that houses your shared TypeScript configuration.

Here is the expected directory structure for your monorepo:

```
bugsight
├── apps
├   ├── server
├   └── client
└── packages
    ├── tsconfig
    └── ...
```

> A monolithic application is a singular entity requiring all its components to be deployed collectively. A monorepo, on the other hand, is different. It can host several independent applications within the same repository, sharing local packages while enabling independent deployment. Therefore, monorepos offer greater flexibility in deployment compared to monoliths.

To generate a top-level **package.json** file, initialize pnpm in the root directory.

```
pnpm init
```

Once you have initiated the process, you will notice that the **package.json** file has been generated.

package.json

```
1    {
2        "name": "bugsight",
3        "version": "1.0.0",
```

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

> At this point, the outcome seems comparable to using **npm**. Nonetheless, it's important to mention that the `pnpm init` command operates comparably to `npm init --yes`. This implies that it automatically assigns default values when initializing, but if you want to personalize these values, you can do so after the initial configuration.

To set up pnpm workspaces within our monorepo, we need to make a **pnpm-workspace.yaml** file at the top level. This file outlines the layout of the monorepo and identifies the paths for the workspaces or packages contained within it. pnpm will then recognize and handle these sub-packages accordingly.

pnpm-workspace.yaml

```
packages:
    - 'apps/*'
    - 'packages/*'
```

It's worth mentioning that you can use the **.npmrc** file to control package management in a pnpm monorepo. This file significantly influences pnpm's operations and the monorepo structure. However, these settings require careful consideration to align with your project's requirements. Read more about it in the official docs.

> Managing a monorepo workspace can be easier with various tools such as Nx, Turborepo, and Bazel.For a small monorepo like ours, these tools might seem overkill initially. The built-in workspace features of package managers like pnpm, Yarn, or npm might be sufficient for managing a small monorepo.

> However, as your project grows, you might start feeling the need for more advanced features, such as improved handling of inter-package dependencies, sophisticated code-sharing, and optimized build processes. This is where these tools can be helpful.

After setting up our monorepo structure using pnpm, it's time to move on to the following section. In this step, we will work on creating a basic Express server.

Create a minimal Express server

To start, go to the server folder and initialize a package.json file.

```
pnpm init
```

After using `pnpm init`, the **package.json** file is generated automatically with default values without any prompts. You must access the **package.json** file and modify the main field value. Specifically, please change it to `src/index.js` instead of `index.js`.

apps/server/package.json

```
{
    "name": "server",
    "version": "1.0.0",
    "description": "",
    "main": "src/index.js",
    ...
}
```

To set up your server code, create a **src** directory. Inside this directory, create an **index.js** file.

> The **src** folder is a conventional location for your project's source code. It provides a dedicated space for your application's code, separate from other files such as **configurations**, **documentation**, or **dependencies**. This separation helps maintain an organized codebase, making it easier to build tools to identify source files, manage tests, and control what's included in version control.

Add a straightforward JavaScript code to validate the setup that will print a value.

apps/server/index.js

```
const print = message => {
    console.log(message);
};

print('JavaScript is AWESOME');
```

To execute this JavaScript file, use the following command from the server folder:

```
node src/index.js
```

The "**JavaScript is AWESOME**" message should be printed in your terminal.

Let's install **Express**, which will allow us to create a server.

```
pnpm add express --filter server
```

> The `--filter` flag is a powerful feature pnpm provides to manage packages in a monorepo setup. This flag lets you specify which packages you want your commands to apply. Consequently, it allows you to execute pnpm commands targeted at specific folders from any location within your project.
>
> Read more about the filter flag.

> To ensure proper execution of commands, using the same workspace `name` in your **package.json **file as you specify in the commands is essential. For example, if the property `name` in **package.json** is `server` within the **server** folder, use `server` as the value for the `--filter` flag to aim the corresponding workspace.

In a monorepo setup, running commands from the root folder is often convenient to avoid frequent folder switching.

```
pnpm --filter <package-name> <command>
# pnpm --filter server dev
```

For a well-organized and structured codebase, creating a **server.js** file that exports a `start` method is recommended. This method can be utilized in our entry point **index.js**, which may include additional functionalities besides starting an express server.

apps/server/server.js

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

const start = () => {
    const app = express();

    app.use('/', (req, res, next) => {
        return res.send('JavaScript is AWESOME');
    });

    app.listen(4000, () => {
        console.info(`Server running on port 4000...`);
    });
};

module.exports = { start };
```

For the sake of simplicity, we have hardcoding the port number. However, later, we will explore how to handle **environment variables**.

To start the server, invoke the start method in the main **index.js** file.

apps/server/index.js

```
const server = require('./server');

server.start();
```

We can start the server using the same method we used for running the JavaScript file previously.

```
node src/index.js
# Server running on port 4000...
```

To access your server, open your favorite browser and navigate to http://localhost:4000, which invokes the endpoint `/`. Once there, you should observe the message "**JavaScript is AWESOME**" displayed on the page.

Hot reloading with Nodemon

> We'll use **Nodemon** instead of the `--watch` flag introduced in Node.js 19 for our project. Nodemon provides advanced features that better suit our needs, particularly TypeScript integration. Currently, the node `--watch` option lacks sufficient customization, so we have chosen Nodemon as our preferred solution.

Nodemon is a utility that provides hot reloading for Node.js applications, allowing real-time updates without manually stopping and restarting the server. This feature significantly speeds up the development process, as developers can instantly see the effects of their changes.

Nodemon is typically installed as a development dependency. You can install it via:

```
pnpm add -D nodemon --filter server
```

After installing Nodemon, you can use it to launch your application by replacing `node` with `nodemon` in your command. This feature enables hot reloading, which automatically restarts the server when code changes are detected.

```
nodemon src/index.js
# Server running on port 4000...
```

It's a good practice to include a script to launch the server. While the current command may be short and straightforward, as time progresses, it may require additional arguments depending on the instance (e.g., **prod**, **test**, **dev**).

apps/server/package.json

```
{
    "name": "server",
    "version": "1.0.0",
    "description": "",
    "main": "src/index.ts",
    "scripts": {
        "dev": "nodemon src/index.ts",
    }
    ...
}
```

You can include a script in your root **package.json** file to run the server from any folder. This script will allow you to start the server quickly regardless of your current working directory.

package.json

```
{
    "name": "bugsight",
    "version": "1.0.0",
    "description": "",
    "scripts": {
        "dev:server": "pnpm --filter server dev"
    },
    ...
}
```

Similar to adding dependencies, we use the `--filter` flag to specify the package and an additional argument to indicate the script that should be executed (`dev`).

To run the server from the root folder, use this command:

```
pnpm run dev:server
```

Alternatively, you can include the `-w` flag in your command if you prefer to run it from any other folder.

```
pnpm -w run dev:server
```

Now that we have explored Nodemon and its benefits for server development let's move on to the next section, where we will discuss integrating TypeScript with Express.

Integrate TypeScript with Express.js

First, we must create a shared package for our TypeScript configuration. Inside the **packages** directory, create a new folder named **tsconfig**.

Using your terminal, navigate to the **tsconfig** folder and initialize a **package.json** file.

```
pnpm init
```

Changing the package name to `@bugsight/tsconfig` is recommended to prevent naming conflicts with other npm packages. Using a scoped name, you can avoid issues with existing packages on the npm registry if you decide to publish your packages.

packages/tsconfig/package.json

```
{
    "name": "@bugsight/tsconfig",
    "version": "1.0.0",
    "description": "",
    ...
}
```

Install **TypeScript** as a development dependency. In this workspace, TypeScript is the sole dependency required.

```
pnpm add -D typescript
```

To create a shared tsconfig file for our server, client, or other packages, we need to generate the **tsconfig.base.json** file, which serves as the configuration file for TypeScript. This file guides the TypeScript compiler during the compilation process to transform our project into JavaScript. You can manually create the **tsconfig.base.json** file or utilize the `npx tsc --init` command, which generates a default file with preset compiler options.

packages/tsconfig/tsconfig.base.json

```
{
    "compilerOptions": {
        "noImplicitAny": true,
        "allowSyntheticDefaultImports": true,
        "forceConsistentCasingInFileNames": true,
        "strict": true,
        "alwaysStrict": true,
        "useUnknownInCatchVariables": true,
        "allowUnreachableCode": false,
        "noImplicitReturns": true,
        "noUncheckedIndexedAccess": true,
        "noFallthroughCasesInSwitch": true,
        "exactOptionalPropertyTypes": true
    }
}
```

> These rules are just a few examples that can be beneficial whether you're working with **Angular** or **Express**. That's why I included them in the shared package. However, you can adjust your TypeScript settings according to your preferences. Additionally, in the future, I may introduce additional rules to further enhance the configuration.

Please refer to the official docs for the complete list of the available rules.

Now that we've finished configuring our package, we can add it as a dependency to the **package.json** file for the server.

`apps/server/package.json`

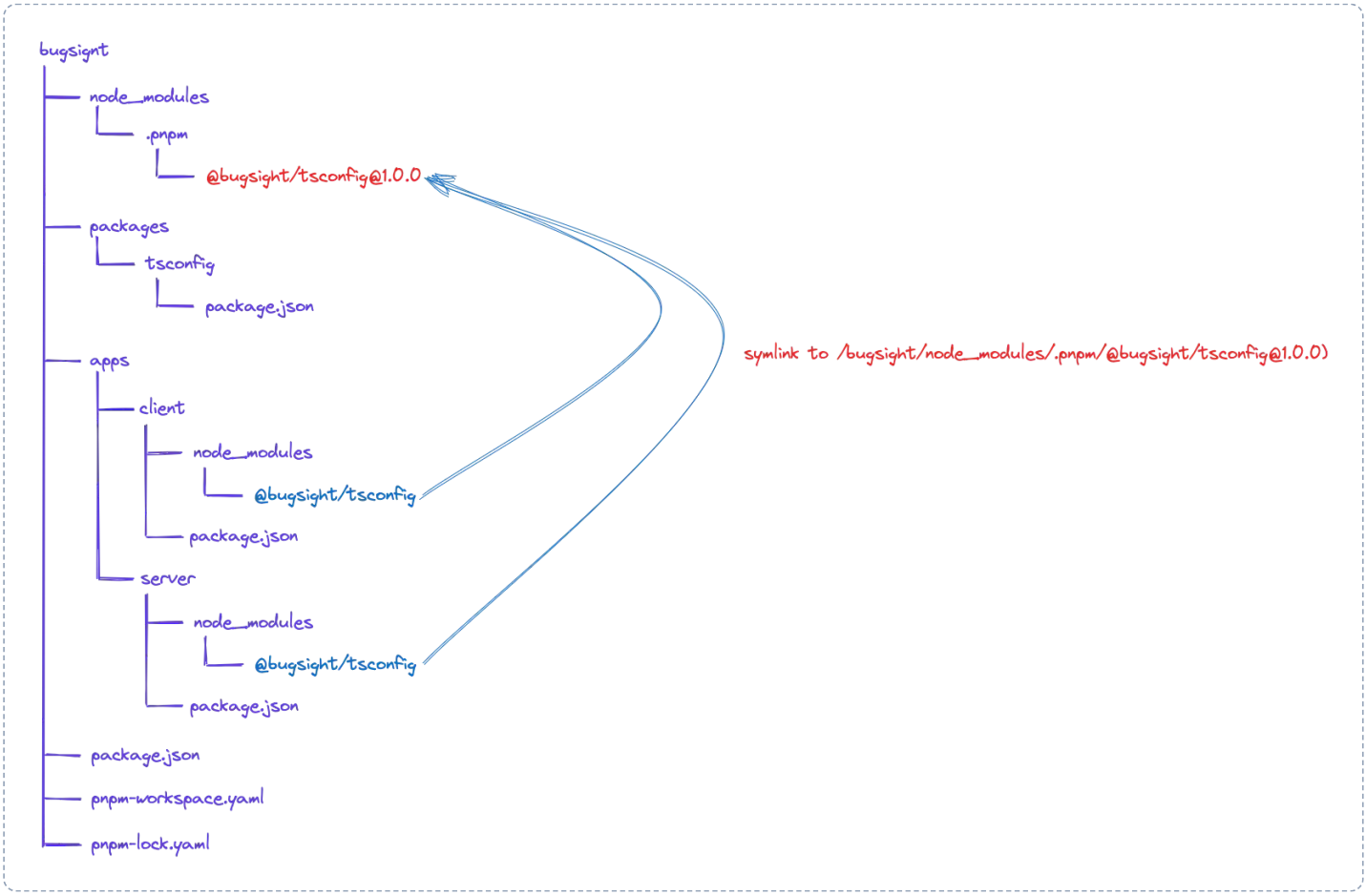
```
{
    "name": "server",
    "main": "src/index.js",
    "dependencies": {
        "@bugsight/tsconfig": "workspace:*",
        "express": "^4.18.2"
    }
    ...
}
```

pnpm has introduced the `workspace:*` protocol. When you add a package to a project using this protocol, you can be sure you're using the most up-to-date code and maintaining consistency across projects.

To proceed, please execute the following command to install the required package or any missing package:

```
pnpm install --filter server
```

Let's take a close look at how pnpm treats the installed packages.



bugsight is the root of your monorepo, and **node_modules** at the root is the shared node_modules. **client** and **server** are your applications, each with a **node_modules** directory containing symlinks to dependencies in the shared **node_modules** folder.

The shared `tsconfig` package is installed in the root **node_modules** directory and symlinked into each application's **node_modules**. The client and server can use the shared `tsconfig` package while maintaining disk efficiency and avoiding duplicate installations.

Monorepos often lead to many **node_modules** directories, which can clutter your workspace using an editor like **VSCode**. However, **VSCode** offers a way to minimize this visual clutter. Modifying your settings allows you to hide all **node_modules** folders from the explorer panel.

To do this, open your settings file and add the following:

```
"files.exclude": {
    "**/node_modules": true
}```
```

Also, you can avoid polluting search results by excluding node_modules and pnpm-lock file:

```
"search.exclude": {
    "**/node_modules": true,
    "pnpm-lock.yaml": true
}
```

> I recommend using a dedicated profile in your Visual Studio Code for this project. Creating a separate profile allows you to configure specific settings, extensions, and preferences tailored to the requirements of this project without interfering with other projects and causing conflicts.
>
> Read more about profiles and how to use them effectively.

Next, let's create a **tsconfig.json** file within the server directory. This will help to extend the base configuration and include extra settings specifically for the server.

apps/server/tsconfig.json

```
{
    "extends": "@bugsight/tsconfig/tsconfig.base.json",
    "include": ["src/**/*"],
    "compilerOptions": {
        "module": "commonjs",
        "target": "es6",
        "esModuleInterop": true,
        "moduleResolution": "node",
        "outDir": "dist",
        "baseUrl": "./src"
    }
}
```

It's important to mention that we refer to the **tsconfig.base.json** file without using a relative path. Instead, we refer to it by name because it's installed as a dependency in the **node_modules** directory.

> When using TypeScript, it does not run directly. It goes through a process called transpilation, which converts it into JavaScript. The transpiler uses the settings defined in the **tsconfig.json** file to decide how to transform the

TypeScript code into JavaScript.

Let's change the file extensions of **index.js** and **server.js** to **.ts**. Moreover, update the files' content to use ES6 modules, which require the use of `import` and `export` syntax, as demonstrated below:

apps/server/server.ts

```ts
import express from 'express';

const start = () => {
    const app = express();

    app.use('/', (req, res, next) => {
        return res.send('JavaScript is AWESOME');
    });

    app.listen(4000, () => {
        console.info(`Server running on port 4000...`);
    });
};

export { start };
```

apps/server/index.ts

```ts
import * as server from './server';

server.start();
```

You will encounter errors that appear in your ts files. This is a good sign that your TypeScript configuration is correctly set up.

```ts
import express from 'express';    Could not find a declaration file for module 'express'.

const start = () => {
    const app = express();

    app.use('/', (req, res, next) => {    Parameter 'req' implicitly has an 'any' type.
        return res.send('JavaScript is AWESOME.');
    });

    app.listen(4000, () => {
        console.info(`Server running on port 4000...`);
    });
};

export { start };
```

The first error with the express import indicates that the `express` package is written in JavaScript and has no exported modules.

The second error occurs because the three parameters have an implicit `any` type, which goes against the `noImplicitAny` rule in our base tsconfig. This error is actually connected to the first error. If the `express` package had exported modules, the parameters would have received proper types automatically.

Although Node.js and Express are coded in JavaScript, TypeScript's type-checking can still be utilized by incorporating type definitions from the DefinitelyTyped repository on GitHub. This repository provides top-notch TypeScript definitions for various JavaScript libraries, including Node.js and Express. To avoid creating type declarations from the beginning, you can efficiently locate these type-declarations utilizing the `@types/{packageName}` pattern.

Use this command to install type definitions as development dependencies (`-D`). This is recommended as these definitions are only required during development and are not necessary for the production build.

```
pnpm add -D @types/node @types/express --filter server
```

The package associated with `@types/node` contains definitions for APIs such as `file` , `process` , and `path` . Installing these types should resolve any errors in your TypeScript file.

We need to update the **package.json** file to use the TypeScript version.

apps/server/package.json

```
{
    "name": "server",
    "version": "1.0.0",
    "description": "",
    "main": "src/index.ts",
    "scripts": {
        "dev": "nodemon src/index.ts"
    }
}
```

Let's attempt to start up the server.

```
pnpm run dev:server
```

An error is expected to occur.

```
~/Desktop/bugsight/packages/tsconfig (1.138s)
pnpm -w run dev:server

> bugsight@1.0.0 dev:server /Users/ayoubkhial/Desktop/bugsight
> pnpm --filter server dev


> server@1.0.0 dev /Users/ayoubkhial/Desktop/bugsight/apps/server
> nodemon src/index.ts

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node src/index.ts`
sh: ts-node: command not found
[nodemon] failed to start process, "ts-node" exec not found
[nodemon] Error
    at Bus.<anonymous> (/Users/ayoubkhial/Desktop/bugsight/node_modules/.pnpm/nodemon@3.0.1/node_modules/nodemon/lib/nodemon.js:158:25)
    at Bus.emit (node:events:524:35)
    at ChildProcess.<anonymous> (/Users/ayoubkhial/Desktop/bugsight/node_modules/.pnpm/nodemon@3.0.1/node_modules/nodemon/lib/monitor/run.js:199:11)
    at ChildProcess.emit (node:events:512:28)
    at ChildProcess._handle.onexit (node:internal/child_process:293:12)
```

We need to use a **ts-node** package to execute the TypeScript file in a Node.js environment. This package compiles the TypeScript file on the fly, converting it to a JavaScript file that can be executed with `nodemon`.

> With the release of version 1.19.0, Nodemon now includes built-in support for TypeScript files, thanks to **ts-node** integration. This means that manual configuration is no longer necessary. While the node CLI is still used to execute JavaScript files by default, Nodemon will automatically switch to using ts-node for TypeScript files. This makes the development process more straightforward when working with TypeScript and Nodemon.

Once you rerun the development script, the TypeScript-based Express server should run without issues. Visit **https://localhost:4000** to confirm that the message is displayed on the screen.

Next, we will include two additional scripts: `build` and `prod`. These scripts play vital roles in the development, testing, and deployment phases of your project.

apps/server/package.json

```
{
    "name": "server",
    "version": "1.0.0",
    "description": "",
    "main": "src/index.ts",
    "scripts": {
        "build": "tsc --build",
        "dev": "nodemon src/index.ts",
        "prod": "npm run build && node ./dist/index.js"
    }
}
```

The purpose of the `build` script is to compile our TypeScript code into JavaScript. It's specified with the `tsc` command. This command translates our TypeScript files into JavaScript counterparts, outputting them in the **./dist** directory as defined by the `outDir` parameter in** tsconfig.json**.

The `prod` script serves our application in production mode. It starts the production server by running the compiled `index.js` from the `./dist` directory. This script is typically executed after the application has been built using the `build` script.

Just like with the `dev` script, you can include both scripts in the route **package.json** file so they can be accessed from anywhere.

apps/server/package.json

```
{
    "name": "bugsight",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "build:server": "pnpm --filter server build",
        "dev:server": "pnpm --filter server dev",
        "prod:server": "pnpm --filter server prod"
    }
}
```

Pushing to GitHub

Head to GitHub and create a new repository. You don't need to initialize it with a **README** or **.gitignore**.

In your local project directory, use the terminal to initialize **Git** (if not already done) by using the command:

```
pnpm run dev:server
```

Then, add the URL of your new GitHub repository with the command:

```
git remote add origin <your-github-repo-url>
# git remote add origin https://github.com/ayoubkhial/bugsight-app.git
```

This connects your local repository to the remote one on GitHub.

Before you push your code to a GitHub repository, setting up a **.gitignore** file to specify which files or directories should not be included in the repository is essential. This is especially crucial for a monorepo setup to avoid cluttering your repo with unnecessary files.

In your project's root directory, create a new file named **.gitignore**.

```
.gitignore

1    node_modules
2    dist
3    .vscode
4    .DS_Store
```

Using this configuration, you can prevent specific directories and files, such as **node_modules**, **dist** (usually containing compiled code), **.DS_Store** (created by macOS), and the **.vscode** folder containing your IDE workspace settings, from being included when pushing to GitHub.

Now you can stage, commit and push your local commits to the remote repository

```
git add .
git commit -m "Initial commit"
git push -u origin master
```

conclusion

In conclusion, integrating TypeScript in a MEAN stack can significantly enhance your development experience by providing robust typing and intuitive error-checking.

You can find the complete code source in this repository; feel free to give it a star ⭐.

If you want to keep up with this series, consider subscribing to my newsletter to receive updates as soon as I publish an article.

Read more

- Everything you need to know about monorepos, and the tools to build them
- Learn more about server-client modal (3-tier architecture)
- Check out the 2023 Stackoverflow survey to see the popularity of the MEAN stack technologies

0 reactions

0 comments

Write Preview Aa

Sign in to comment

Sign in with GitHub

Subscribe to my newsletter

Get notified whenever I share new articles by subscribing to my newsletter. Also, I share exciting tools and resources I found while surfing the web.

What should I call you? What's your email? Subscribe

Designed by Ayoub. The code source is available on Github.