SWC

SWC (Speedy Web Compiler) is an extensible Rust-based platform that can be used for both compilation and bundling. Using SWC with Nest CLI is a great and simple way to significantly speed up your development process.

> info **Hint** SWC is approximately **x20 times faster** than the default TypeScript compiler.

Installation

To get started, first install a few packages:

```
$ npm i --save-dev @swc/cli @swc/core
```

Getting started

Once the installation process is complete, you can use the `swc` builder with Nest CLI, as follows:

```
$ nest start -b swc
# OR nest start --builder swc
```

> info **Hint** If your repository is a monorepo, check out this section.

Instead of passing the `-b` flag you can also just set the `compilerOptions.builder` property to `"swc"` in your `nest-cli.json` file, like so:

```
{
  "compilerOptions": {
    "builder": "swc"
  }
}
```

To customize builder's behavior, you can pass an object containing two attributes, `type` (`"swc"`) and `options`, as follows:

```
"compilerOptions": {
  "builder": {
    "type": "swc",
    "options": {
      "swcrcPath": "infrastructure/.swcrc",
    }
  }
}
```

To run the application in watch mode, use the following command:

```
$ nest start -b swc -w
# OR nest start --builder swc --watch
```

Type checking

SWC does not perform any type checking itself (as opposed to the default TypeScript compiler), so to turn it on, you need to use the `--type-check` flag:

```
$ nest start -b swc --type-check
```

This command will instruct the Nest CLI to run `tsc` in `noEmit` mode alongside SWC, which will asynchronously perform type checking. Again, instead of passing the `--type-check` flag you can also just set the `compilerOptions.typeCheck` property to `true` in your `nest-cli.json` file, like so:

```
{
  "compilerOptions": {
    "builder": "swc",
    "typeCheck": true
  }
}
```

CLI Plugins (SWC)

The `--type-check` flag will automatically execute **NestJS CLI plugins** and produce a serialized metadata file which then can be loaded by the application at runtime.

SWC configuration

SWC builder is pre-configured to match the requirements of NestJS applications. However, you can customize the configuration by creating a `.swcrc` file in the root directory and tweaking the options as you wish.

```
{
  "$schema": "https://json.schemastore.org/swcrc",
  "sourceMaps": true,
  "jsc": {
    "parser": {
      "syntax": "typescript",
      "decorators": true,
      "dynamicImport": true
    },
    "baseUrl": "./"
```

```
  },
  "minify": false
}
```

Monorepo

If your repository is a monorepo, then instead of using `swc` builder you have to configure `webpack` to use `swc-loader`.

First, let's install the required package:

```
$ npm i --save-dev swc-loader
```

Once the installation is complete, create a `webpack.config.js` file in the root directory of your application with the following content:

```
const swcDefaultConfig = require('@nestjs/cli/lib/compiler/defaults/swc-defaults').swcDefaultsFactory().swcOptions;

module.exports = {
  module: {
    rules: [
      {
        test: /\.ts$/,
        exclude: /node_modules/,
        use: {
          loader: 'swc-loader',
          options: swcDefaultConfig,
        },
      },
    ],
  },
};
```

Monorepo and CLI plugins

Now if you use CLI plugins, `swc-loader` will not load them automatically. Instead, you have to create a separate file that will load them manually. To do so, declare a `generate-metadata.ts` file near the `main.ts` file with the following content:

```
import { PluginMetadataGenerator } from '@nestjs/cli/lib/compiler/plugins';
import { ReadonlyVisitor } from '@nestjs/swagger/dist/plugin';

const generator = new PluginMetadataGenerator();
generator.generate({
```

```
  visitors: [new ReadonlyVisitor({ introspectComments: true, pathToSource:
__dirname })],
  outputDir: __dirname,
  watch: true,
  tsconfigPath: 'apps/<name>/tsconfig.app.json',
});
```

> info **Hint** In this example we used `@nestjs/swagger` plugin, but you can use any plugin of your choice.

The `generate()` method accepts the following options:

`watch`	Whether to watch the project for changes.
`tsconfigPath`	Path to the `tsconfig.json` file. Relative to the current working directory (`process.cwd()`).
`outputDir`	Path to the directory where the metadata file will be saved.
`visitors`	An array of visitors that will be used to generate metadata.
`filename`	The name of the metadata file. Defaults to `metadata.ts`.
`printDiagnostics`	Whether to print diagnostics to the console. Defaults to `true`.

Finally, you can run the `generate-metadata` script in a separate terminal window with the following command:

```
$ npx ts-node src/generate-metadata.ts
# OR npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

Common pitfalls

If you use TypeORM/MikroORM or any other ORM in your application, you may stumble upon circular import issues. SWC doesn't handle **circular imports** well, so you should use the following workaround:

```
@Entity()
export class User {
  @OneToOne(() => Profile, (profile) => profile.user)
  profile: Relation<Profile>; // <--- see "Relation<>" type here instead
of just "Profile"
}
```

> info **Hint** `Relation` type is exported from the `typeorm` package.

Doing this prevents the type of the property from being saved in the transpiled code in the property metadata, preventing circular dependency issues.

If your ORM does not provide a similar workaround, you can define the wrapper type yourself:

```
/**
 * Wrapper type used to circumvent ESM modules circular dependency issue
 * caused by reflection metadata saving the type of the property.
 */
export type WrapperType<T> = T; // WrapperType === Relation
```

For all circular dependency injections in your project, you will also need to use the custom wrapper type described above:

```
@Injectable()
export class UserService {
  constructor(
    @Inject(forwardRef(() => ProfileService))
    private readonly profileService: WrapperType<ProfileService>,
  ) {};
}
```

Jest + SWC

To use SWC with Jest, you need to install the following packages:

```
$ npm i --save-dev jest @swc/core @swc/jest
```

Once the installation is complete, update the `package.json`/`jest.config.js` file (depending on your configuration) with the following content:

```
{
  "jest": {
    "transform": {
      "^.+\\.(t|j)s?$": ["@swc/jest"]
    }
  }
}
```

Additionally you would need to add the following `transform` properties to your `.swcrc` file: `legacyDecorator`, `decoratorMetadata`:

```
{
  "$schema": "https://json.schemastore.org/swcrc",
  "sourceMaps": true,
  "jsc": {
```

```
    "parser": {
      "syntax": "typescript",
      "decorators": true,
      "dynamicImport": true
    },
    "transform": {
      "legacyDecorator": true,
      "decoratorMetadata": true
    },
    "baseUrl": "./"
  },
  "minify": false
}
```

If you use NestJS CLI Plugins in your project, you'll have to run `PluginMetadataGenerator` manually. Navigate to this section to learn more.

Vitest

Vitest is a fast and lightweight test runner designed to work with Vite. It provides a modern, fast, and easy-to-use testing solution that can be integrated with NestJS projects.

Installation

To get started, first install the required packages:

```
$ npm i --save-dev vitest unplugin-swc @swc/core @vitest/coverage-c8
```

Configuration

Create a `vitest.config.ts` file in the root directory of your application with the following content:

```
import swc from 'unplugin-swc';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    globals: true,
    root: './',
  },
  plugins: [
    // This is required to build the test files with SWC
    swc.vite({
      // Explicitly set the module type to avoid inheriting this value
from a `.swcrc` config file
      module: { type: 'es6' },
    }),
```

```
  ],
});
```

This configuration file sets up the Vitest environment, root directory, and SWC plugin. You should also create a separate configuration file for e2e tests, with an additional `include` field that specifies the test path regex:

```
import swc from 'unplugin-swc';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    include: ['**/*.e2e-spec.ts'],
    globals: true,
    root: './',
  },
  plugins: [swc.vite()],
});
```

Additionally, you can set the `alias` options to support TypeScript paths in your tests:

```
import swc from 'unplugin-swc';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  test: {
    include: ['**/*.e2e-spec.ts'],
    globals: true,
    alias: {
      '@src': './src',
      '@test': './test',
    },
    root: './',
  },
  resolve: {
    alias: {
      '@src': './src',
      '@test': './test',
    },
  },
  plugins: [swc.vite()],
});
```

Update imports in E2E tests

Change any E2E test imports using `import * as request from 'supertest'` to `import request from 'supertest'`. This is necessary because Vitest, when bundled with Vite, expects a default import

for supertest. Using a namespace import may cause issues in this specific setup.

Lastly, update the test scripts in your package.json file to the following:

```
{
  "scripts": {
    "test": "vitest run",
    "test:watch": "vitest",
    "test:cov": "vitest run --coverage",
    "test:debug": "vitest --inspect-brk --inspect --logHeapUsage --
threads=false",
    "test:e2e": "vitest run --config ./vitest.config.e2e.ts"
  }
}
```

These scripts configure Vitest for running tests, watching for changes, generating code coverage reports, and debugging. The test:e2e script is specifically for running E2E tests with a custom configuration file.

With this setup, you can now enjoy the benefits of using Vitest in your NestJS project, including faster test execution and a more modern testing experience.

> info **Hint** You can check out a working example in this repository