Model-View-Controller

Nest, by default, makes use of the Express library under the hood. Hence, every technique for using the MVC (Model-View-Controller) pattern in Express applies to Nest as well.

First, let's scaffold a simple Nest application using the CLI tool:

```
$ npm i -g @nestjs/cli
$ nest new project
```

In order to create an MVC app, we also need a template engine to render our HTML views:

```
$ npm install --save hbs
```

We've used the `hbs` (Handlebars) engine, though you can use whatever fits your requirements. Once the installation process is complete, we need to configure the express instance using the following code:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { NestExpressApplication } from '@nestjs/platform-express';
import { join } from 'path';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  app.setViewEngine('hbs');

  await app.listen(3000);
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { join } from 'path';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(
    AppModule,
  );

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
```

```
  app.setViewEngine('hbs');

  await app.listen(3000);
}
bootstrap();
```

We told Express that the `public` directory will be used for storing static assets, `views` will contain templates, and the `hbs` template engine should be used to render HTML output.

Template rendering

Now, let's create a `views` directory and `index.hbs` template inside it. In the template, we'll print a `message` passed from the controller:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>App</title>
  </head>
  <body>
    {{ "{{ message }\}" }}
  </body>
</html>
```

Next, open the `app.controller` file and replace the `root()` method with the following code:

```
@@filename(app.controller)
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index')
  root() {
    return { message: 'Hello world!' };
  }
}
```

In this code, we are specifying the template to use in the `@Render()` decorator, and the return value of the route handler method is passed to the template for rendering. Notice that the return value is an object with a property `message`, matching the `message` placeholder we created in the template.

While the application is running, open your browser and navigate to `http://localhost:3000`. You should see the `Hello world!` message.

Dynamic template rendering

If the application logic must dynamically decide which template to render, then we should use the `@Res()` decorator, and supply the view name in our route handler, rather than in the `@Render()` decorator:

> info **Hint** When Nest detects the `@Res()` decorator, it injects the library-specific `response` object. We can use this object to dynamically render the template. Learn more about the `response` object API here.

```
@@filename(app.controller)
import { Get, Controller, Res, Render } from '@nestjs/common';
import { Response } from 'express';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private appService: AppService) {}

  @Get()
  root(@Res() res: Response) {
    return res.render(
      this.appService.getViewName(),
      { message: 'Hello world!' },
    );
  }
}
```

Example

A working example is available here.

Fastify

As mentioned in this chapter, we are able to use any compatible HTTP provider together with Nest. One such library is Fastify. In order to create an MVC application with Fastify, we have to install the following packages:

```
$ npm i --save @fastify/static @fastify/view handlebars
```

The next steps cover almost the same process used with Express, with minor differences specific to the platform. Once the installation process is complete, open the `main.ts` file and update its contents:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { NestFastifyApplication, FastifyAdapter } from '@nestjs/platform-
fastify';
import { AppModule } from './app.module';
import { join } from 'path';
```

```
async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter(),
  );
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    prefix: '/public/',
  });
  app.setViewEngine({
    engine: {
      handlebars: require('handlebars'),
    },
    templates: join(__dirname, '..', 'views'),
  });
  await app.listen(3000);
}
bootstrap();
@@switch
import { NestFactory } from '@nestjs/core';
import { FastifyAdapter } from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import { join } from 'path';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, new FastifyAdapter());
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    prefix: '/public/',
  });
  app.setViewEngine({
    engine: {
      handlebars: require('handlebars'),
    },
    templates: join(__dirname, '..', 'views'),
  });
  await app.listen(3000);
}
bootstrap();
```

The Fastify API is slightly different but the end result of those methods calls remains the same. One difference to notice with Fastify is that the template name passed into the `@Render()` decorator must include a file extension.

```
@@filename(app.controller)
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index.hbs')
```

```
  root() {
    return { message: 'Hello world!' };
  }
}
```

While the application is running, open your browser and navigate to `http://localhost:3000`. You should see the `Hello world!` message.

Example

A working example is available here.