ConfigurableModuleBuilder

August 19, 2026

ConfigurableModuleBuilder

Kind: Class

Source: `packages/common/module-utils/configurable-module.builder.ts`

Part of: Common

Factory that lets you create configurable modules and
provides a way to reduce the majority of dynamic module boilerplate.

`ConfigurableModuleBuilder` generates the dynamic-module infrastructure needed to configure a NestJS module without manually writing repetitive `forRoot`, `forRootAsync`, options-token, and provider boilerplate. It lets module authors customize static method names, factory method names, and extra module-definition options before producing a `ConfigurableModuleHost`.

Methods

Method	Signature	Returns
`setExtras`	`setExtras(extras: ExtraModuleDefinitionOptions, transformDefinition: ( definition: DynamicModule, extras: ExtraModuleDefinitionOptions, ) => DynamicModule)`	`void`
`setClassMethodName`	`setClassMethodName(key: StaticMethodKey)`	`void`
`setFactoryMethodName`	`setFactoryMethodName(key: FactoryClassMethodKey)`	`void`
`build`	`build()`	`ConfigurableModuleHost< ModuleOptions, StaticMethodKey,`

Method	Signature	Returns
		`FactoryClassMethodKey, ExtraModuleDefinitionOptions >`

Properties

Property	Type
`staticMethodKey`	`StaticMethodKey`
`factoryClassMethodKey`	`FactoryClassMethodKey`
`extras`	`ExtraModuleDefinitionOptions`
`transformModuleDefinition`	`( definition: DynamicModule, extraOptions: ExtraModuleDefinitionOptions, ) => DynamicModule`
`logger`	`any`

Where it refuses work

- `ConfigurableModuleBuilder` stops the work with an early return when `!extras`, in 2 places.
- `ConfigurableModuleBuilder` stops the work with an early return when `options.inject && options.provideInjectionTokensFrom`.
- `ConfigurableModuleBuilder` stops the work with an early return when `options.useFactory`.

Diagram

```
graph LR
  A[ConfigurableModuleBuilder] --> B[Configure method names]
  A --> C[Configure extras]
  B --> D[build()]
  C --> D
  D --> E[ConfigurableModuleHost]
  E --> F[ConfigurableModuleClass]
  E --> G[MODULE_OPTIONS_TOKEN]
  E --> H[OPTIONS_TYPE / ASYNC_OPTIONS_TYPE]
  F --> I[Application Module Imports]
```

Usage

```ts
import { Module } from '@nestjs/common';
import { ConfigurableModuleBuilder } from '@nestjs/common';

export interface EmailModuleOptions {
  apiKey: string;
  sender: string;
}

const {
  ConfigurableModuleClass,
  MODULE_OPTIONS_TOKEN,
} = new ConfigurableModuleBuilder<EmailModuleOptions>()
  .setClassMethodName('forRoot')
  .setFactoryMethodName('createEmailOptions')
  .setExtras(
    { isGlobal: false },
    (definition, extras) => ({
      ...definition,
      global: extras.isGlobal,
    }),
  )
  .build();

@Module({
  providers: [
    {
      provide: 'EMAIL_OPTIONS',
      useExisting: MODULE_OPTIONS_TOKEN,
    },
  ],
  exports: ['EMAIL_OPTIONS'],
})
export class EmailModule extends ConfigurableModuleClass {}

// In another module:
// EmailModule.forRoot({
//   apiKey: process.env.EMAIL_API_KEY!,
//   sender: 'noreply@example.com',
// })
```

AI Coding Instructions

- Create one `ConfigurableModuleBuilder<ModuleOptions>` per configurable module and export the generated options token when other providers need access to configuration.
- Call `setClassMethodName()` only when the default `register` method does not match the module's public API convention, such as `forRoot`.

- Use `setExtras()` for module-definition concerns such as `global`, not for runtime options that application providers consume.
- Extend the generated `ConfigurableModuleClass` in the final `@Module()` class so generated synchronous and asynchronous registration methods remain available.
- Keep custom factory method names aligned with async configuration factories when using `registerAsync` / `forRootAsync`-style integration.