

NestJS 의존성 주입 딥다이브

김현우

시작하기 전에

- 이후로 나올 내용은 NestJS를 다뤄보지 않은 사람도 최대한 이해할 수 있도록 풀어썼습니다.
 - 타입스크립트로 백엔드 만드는 사람들이 모두 NestJS를 쓰는 건 아니니깐요!
- 따라서, 먼저 NestJS를 간단히 소개하고 본제로 들어가려 합니다.

NestJS?

- 타입스크립트 기반 백엔드 프레임워크
- 여러 기능 중, 대표적으로 “의존성 주입”
- 의존성 주입이 어떻게 동작하는지 봅시다.
 - v11.1.5 기준으로 설명합니다.
 - 다만 핵심 부분이 그리 빠르게 바뀌는 건 아니라서 다른 버전에서도 크게 다르진 않습니다.



서론

NestJS 구성 요소

- Provider
- Controller
- Module
- ...

NestJS 구성 요소: Provider

- 다른 곳으로 주입될 수 있는 많은 것들
 - 단순 클래스, 레포지토리, config 객체, ...
- 여러 종류가 있습니다.
- 이 중 가장 기본적인 형태인 **단순 클래스**를 어떻게 처리하는지 살펴볼 예정입니다.

```
@Injectable()
export class Calculator {
  add(a: number, b: number): number {
    return a + b;
  }
}
```

서론

사실 이미 예전에 글로 썼는데요...

Nest.js는 실제로 어떻게 의존성을 주입해줄까?

러리 · 2022년 7월 10일

통계 수정 삭제

NestJS

typescript

NestJS 구성 요소: Controller

- 요청을 받는 곳
 - `@Controller``
 - `@Get()`, @Post()`, ...`
- 오른쪽 코드는 `GET /add`` 요청을 받아 `2 + 3`` 을 계산
 - `calculator`` 인스턴스가 **주입**된다!

```
@Controller()
export class AppController {
  constructor(private readonly calculator: Calculator) {}

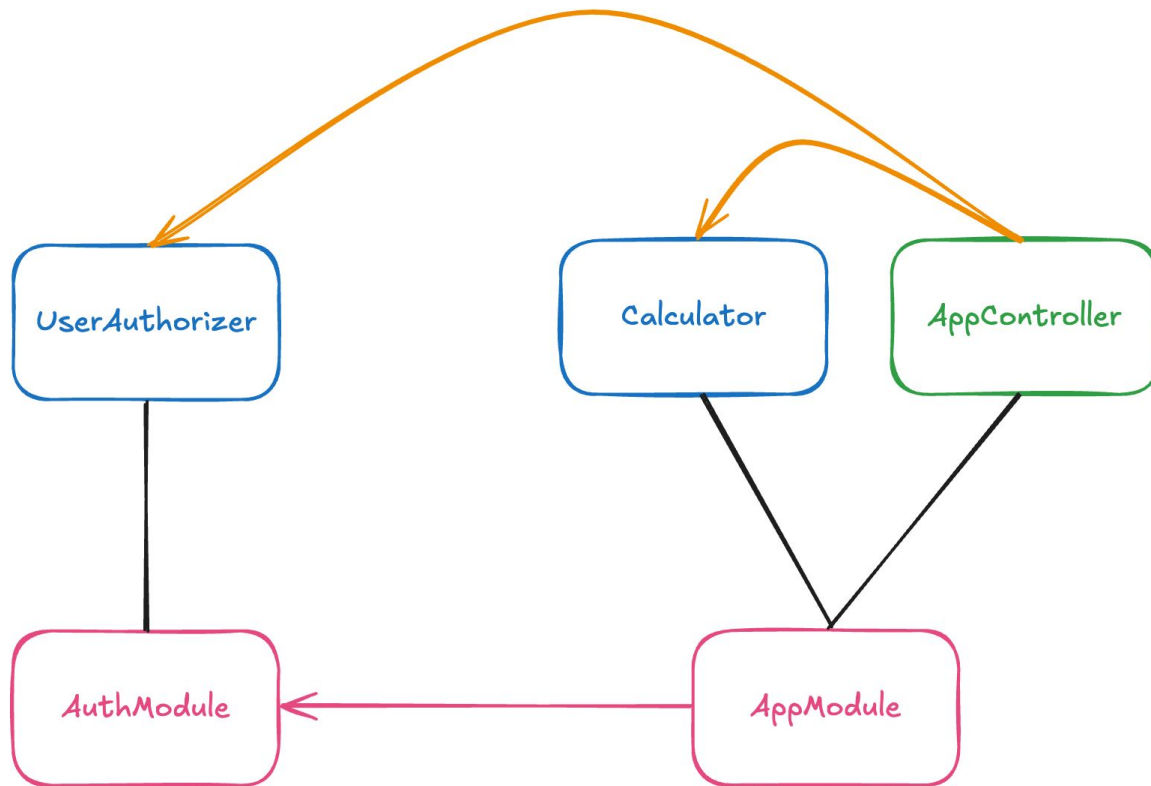
  @Get('/add')
  addNumbers(): number {
    return this.calculator.add(2, 3);
  }
}
```

NestJS 구성 요소: Module

- 동일 모듈에 속한 Provider는 서로를 주입 받을 수 있음
- Controller는 동일 모듈에 속한 Provider를 주입 받을 수 있음
- 다른 모듈을 임포트하여, 다른 모듈의 프로바이더를 주입 받을 수도 있음

```
@Module({  
  imports: [],  
  providers: [Calculator],  
  controllers: [AppController],  
  exports: [],  
})  
export class AppModule {}
```


서론 예시



의존성 주입?

- 단순히 생성자에 파라미터를 넣어두는 것만으로도 NestJS가 알아서 인스턴스를 만들어 주입해줍니다.
- 혹은 `@Inject`를 사용하여 무엇을 주입 받을지 직접 가리킬 수도 있습니다.

```
@Controller()
export class AppController {
  constructor(
    private readonly calculator: Calculator,

    @Inject('CalculatorConfig')
    private readonly calculatorConfig: CalculatorConfig,
  ) {}
}
```

그래서 우리는?

- 오른쪽 코드에서 ``calculator`` 와 ``calculatorConfig`` 가
 1. 어떻게 만들어져서
 2. 어떻게 생성자로 들어왔는지를
- 코드 레벨에서 살펴보겠습니다.

```
@Controller()  
export class AppController {  
  constructor(  
    private readonly calculator: Calculator,  
  
    @Inject('CalculatorConfig')  
    private readonly calculatorConfig: CalculatorConfig,  
  ) {}  
}
```

본론

세 단계로 나뉩니다.

1. 자신이 어떤 의존성을 주입 받아야 하는지 기록
2. 프로바이더 간 의존성 관계 기록
3. 실제 의존성 주입 처리

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

메타데이터?

- 데이터의 데이터.
- 타입스크립트에서는 보통 **데코레이터**와 같이 연관지어 많이 사용합니다.
 - 생성자에 붙이거나.. 메서드에 붙이거나.. 파라미터에도 붙고요.

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

타입스크립트도 메타데이터를 줍니다.

- TS → JS 트랜스파일 과정에서 타입스크립트는 메타데이터를 주입해줍니다.
 - ``design:type``: 프로퍼티의 타입
 - ``design:paramtypes``: 파라미터의 타입들
 - ``design:returntype``: 반환 타입
- 예시로 봅시다.

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

타입스크립트도 메타데이터를 줍니다.

```
@EmptyDeco
class ExampleClass {
  @EmptyDeco
  hello: string = "hello";

  constructor(world: number) { /* ... */ }

  @EmptyDeco
  method(): Other { /* ... */ }
}

console.log(Reflect.getMetadata("design:type", ExampleClass.prototype, "hello"));
console.log(Reflect.getMetadata("design:paramtypes", ExampleClass));
console.log(Reflect.getMetadata("design:returntype", ExampleClass.prototype, "method"));
```

```
• λ ~/dev/hello/ npx tsc -p . && node main.js
  [Function: String]
  [ [Function: Number] ]
  [class Other]
```

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

NestJS는 어떻게 활용할까?

- 주로 `design:paramtypes` 과 `design:type` 을 활용합니다.
- 그럼 메타데이터를 받아서 뭘 어떻게 쓸까요?

토큰

- NestJS는 각 Provider를 토큰으로 구분합니다.
 - 문자열, Symbol, 혹은 **생성자 그 자체**까지도 사용할 수 있어요!
 - A 프로바이더를 주입 받고 싶을 때는, A 프로바이더의 토큰만 명시해주면 돼요.
- NestJS는 ``design:paramtypes``를 통해 생성자의 각 파라미터가 **어떤 클래스**의 객체, 즉 **어떤 토큰**을 원하는지 판단합니다.
- 그럼 ``@Inject``를 사용할 때는요?
 - 별도로 기록해둡니다. NestJS는 이것 ``self:paramtypes`` 메타데이터로 저장합니다.

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

코드로 봅시다.

- 프로퍼티에서 사용하는 경우와 생성자에서 사용하는 경우가 혼재되어 있습니다.
- 프로퍼티 내용은 빠면?

```
export function Inject(  
  token?: InjectionToken | ForwardReference,  
): PropertyDecorator & ParameterDecorator {  
  const injectCallHasArguments = arguments.length > 0;  
  
  return (target: object, key: string | symbol | undefined, index?: number) => {  
    let type = token || Reflect.getMetadata('design:type', target, key!);  
    // Try to infer the token in a constructor-based injection  
    if (!type && !injectCallHasArguments) {  
      type = Reflect.getMetadata(PARAMTYPES_METADATA, target, key!)?.[index!];  
    }  
  
    if (!isUndefined(index)) {  
      let dependencies =  
        Reflect.getMetadata(SELF_DECLARED_DEPS_METADATA, target) || [];  
  
      dependencies = [...dependencies, { index, param: type }];  
      Reflect.defineMetadata(SELF_DECLARED_DEPS_METADATA, dependencies, target);  
      return;  
    }  
    let properties =  
      Reflect.getMetadata(PROPERTY_DEPS_METADATA, target.constructor) || [];  
  
    properties = [...properties, { key, type }];  
    Reflect.defineMetadata(  
      PROPERTY_DEPS_METADATA,  
      properties,  
      target.constructor,  
    );  
  }  
};
```

@nestjs/common@11.1.5: inject.decorator.ts#L38-L68 ([link](#))

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

코드로 봅시다.

```
export function Inject(  
  token?: InjectionToken | ForwardReference,  
) : ParameterDecorator {  
  const injectCallHasArguments = arguments.length > 0;  
  
  return (target: object, key: string | symbol | undefined, index: number) => {  
    let type = token || Reflect.getMetadata('design:type', target, key!);  
  
    if (!type && !injectCallHasArguments) {  
      type = Reflect.getMetadata(PARAMTYPES_METADATA, target, key!)?.[index];  
    }  
  
    let dependencies =  
      Reflect.getMetadata(SELF_DECLARED_DEPS_METADATA, target) || [];  
  
    dependencies = [...dependencies, { index, param: type }];  
    Reflect.defineMetadata(SELF_DECLARED_DEPS_METADATA, dependencies, target);  
  };  
}
```

- 자신이 어떤 의존성을 주입 받아야 하는지 ``self:paramtypes`` 메타데이터에 기록합니다.
- 생성자 파라미터는 순서가 중요하므로 **인덱스와 함께** 기록합니다.

본론 #1 : 자신이 어떤 의존성을 주입 받아야 하는지 기록

여기까지 정리하면,

```
@Controller()
export class AppController {
  constructor(
    private readonly calculator: Calculator,

    @Inject('CalculatorConfig')
    private readonly calculatorConfig: CalculatorConfig,
  ) {}
}
```

```
Reflect.getMetadata('design:paramtypes', AppController);
// [ [class Calculator], [Function: Object] ]

Reflect.getMetadata('self:paramtypes', AppController);
// [ { index: 1, param: 'CalculatorConfig' } ]
```

NestJS 의존성 처리 과정

1. 모든 모듈을 순회하며 모듈을 컨테이너에 등록
2. 모든 모듈을 순회하며 모듈의 의존 관계에 대한 트리 생성
3. 모든 모듈을 순회하며 각 프로바이더에 대한 `InstanceWrapper` 생성
4. 모든 모듈의 모든 `InstanceWrapper`를 순회하며 실제로 인스턴스 생성

근데.. 어디서 이런 과정이 시작되는 거죠?

의존성 처리는 어디서 시작될까?

- NestJS를 다뤄보셨다면 우측의 코드가 익숙하실 겁니다.
- 이 중 `NestFactory.create`에 조금 더 집중해봅시다.

```
async function bootstrap() {  
  const app = await NestFactory.create(AppModule);  
  await app.listen(3000);  
}  
bootstrap();
```

의존성 처리는 어디서 시작될까?

- `create`에서 호출되는 코드 중에는
오른쪽과 같은 코드가 있습니다.

```
await ExceptionsZone.asyncRun(  
  async () => {  
    await dependenciesScanner.scan(module);  
    await instanceLoader.createInstancesOfDependencies();  
    dependenciesScanner.applyApplicationProviders();  
  },  
  teardown,  
  this.autoFlushLogs,  
);
```

@nestjs/core@11.1.5: nest-factory.ts#L235-L243 ([link](#))

의존성 처리는 어디서 시작될까?

- 여기서,
 1. scan: 1번, 2번, 3번 과정
 2. createInstancesOfDependencies:
4번 과정 (다음 본론)
- 더 깊이 가보면?

```
await ExceptionsZone.asyncRun(  
  async () => {  
    await dependenciesScanner.scan(module);  
    await instanceLoader.createInstancesOfDependencies();  
    dependenciesScanner.applyApplicationProviders();  
  },  
  teardown,  
  this.autoFlushLogs,  
);
```

@nestjs/core@11.1.5: nest-factory.ts#L235-L243 ([link](#))

더 깊이 가보면?

- 여기서,
 1. scanForModules: 1번 과정
 2. scanModulesForDependencies: 2번 과정, 3번 과정
- 1번 과정부터 살펴봅시다.

```
public async scan(  
  module: Type<any>,  
  options?: { overrides?: ModuleOverride[] },  
) {  
  await this.registerCoreModule(options?.overrides);  
  await this.scanForModules({  
    moduleDefinition: module,  
    overrides: options?.overrides,  
  });  
  await this.scanModulesForDependencies();  
  this.addScopedEnhancersMetadata();  
  
  this.calculateModulesDistance();  
  
  this.container.bindGlobalScope();  
}
```

@nestjs/core@11.1.5: scanner.ts#L86-L104 ([link](#))

본론 #2 : 프로바이더 간 의존성 관계 기록 - 1번 과정 (모든 모듈을 순회하며 모듈을 컨테이너에 등록)

scanForModules

1. 모든 모듈을 컨테이너에 등록합니다. (생략)
2. 자신이 참조(imports)하는 모듈 목록을 가져옵니다.
3. 참조하는 모듈 목록을 순회하며 재귀 호출합니다.
4. 이를 통해 모든 모듈을 컨테이너에 등록합니다.

scanForModules

- `MODULE_METADATA.IMPORTS` 메타데이터를 가져와 `modules`에 넣습니다.
- 저 메타데이터는 뭐죠?

```
const modules = !this.isDynamicModule(
  moduleDefinition as Type<any> | DynamicModule,
)
? this.reflectMetadata(
  MODULE_METADATA.IMPORTS,
  moduleDefinition as Type<any>,
)
: [
  ...this.reflectMetadata(
    MODULE_METADATA.IMPORTS,
    (moduleDefinition as DynamicModule).module,
  ),
  ...((moduleDefinition as DynamicModule).imports || []),
];
```

@nestjs/core@11.1.5: scanner.ts#L131-L144 ([link](#))

NestJS의 모듈 시스템

- 모듈에는 여러 정보를 넣을 수 있습니다.
 - 참조할 모듈 (=imports)
 - 프로바이더 등록 (=providers)
 - 컨트롤러 등록 (=controllers)
 - 내보낼 프로바이더 / 모듈 (=exports)
- 근데.. 이 정보는 어디에 저장되지?
 - 메타데이터로!

```
@Module({  
  imports: [],  
  providers: [Calculator],  
  controllers: [AppController],  
  exports: [],  
})  
export class AppModule {}
```

NestJS의 모듈 시스템

```
@Module({
  imports: [],
  providers: [Calculator],
  controllers: [AppController],
  exports: [],
})
export class AppModule {}

console.log(`keys:`, Reflect.getMetadataKeys(AppModule));
console.log(`imports:`, Reflect.getMetadata('imports', AppModule));
console.log(`providers:`, Reflect.getMetadata('providers', AppModule));
console.log(`controllers:`, Reflect.getMetadata('controllers', AppModule));
console.log(`exports:`, Reflect.getMetadata('exports', AppModule));
```

```
keys: [ 'imports', 'providers', 'controllers', 'exports' ]
imports: []
providers: [ [class Calculator] ]
controllers: [ [class AppController] ]
exports: []
```

scanForModules

- `MODULE_METADATA.IMPORTS`가
`imports`입니다.
- 즉, 참조하는 모듈 목록을 가져옵니다.

```
const modules = !this.isDynamicModule(  
  moduleDefinition as Type<any> | DynamicModule,  
)  
? this.reflectMetadata(  
  MODULE_METADATA.IMPORTS,  
  moduleDefinition as Type<any>,  
)  
: [  
  ...this.reflectMetadata(  
    MODULE_METADATA.IMPORTS,  
    (moduleDefinition as DynamicModule).module,  
  ),  
  ...((moduleDefinition as DynamicModule).imports || []),  
];
```

@nestjs/core@11.1.5: scanner.ts#L131-L144 ([link](#))

scanForModules

- 참조하는 모듈의 목록을 순회하며 다시 `scanForModules` 를 호출합니다.
 - 이를 통해 모든 모듈을 등록합니다.
- 즉, 모듈 등록은 **DFS**로 이뤄집니다.

```
let registeredModuleRefs: Module[] = [];  
for (const [index, innerModule] of modules.entries()) {  
  // ...  
  const moduleRefs = await this.scanForModules({  
    moduleDefinition: innerModule,  
    scope: ([] as Array<Type>).concat(scope, moduleDefinition as Type),  
    ctxRegistry,  
    overrides,  
    lazy,  
  });  
  registeredModuleRefs = registeredModuleRefs.concat(moduleRefs);  
}
```

@nestjs/core@11.1.5: scanner.ts#L146-L166 ([link](#))

다음은?

- 컨테이너에 모든 모듈이 등록됐습니다.
- 이제 모든 모듈을 순회하며 모듈 간 의존 관계를 만들 차례입니다.
 - `scanModulesForDependencies``

```
public async scan(  
  module: Type<any>,  
  options?: { overrides?: ModuleOverride[] },  
) {  
  await this.registerCoreModule(options?.overrides);  
  await this.scanForModules({  
    moduleDefinition: module,  
    overrides: options?.overrides,  
  });  
  await this.scanModulesForDependencies();  
  this.addScopedEnhancersMetadata();  
  
  this.calculateModulesDistance();  
  
  this.container.bindGlobalScope();  
}
```

@nestjs/core@11.1.5: scanner.ts#L86-L104 ([link](#))

scanModulesForDependencies

- 모든 모듈을 순회하며 `reflect*`를 호출합니다.
- 여기서,
 1. `reflectImports`: 2번 과정
 2. `reflectProviders`: 3번 과정
- `reflectImports`를 살펴봅시다.

```
public async scanModulesForDependencies(  
  modules: Map<string, Module> = this.container.getModules(),  
) {  
  for (const [token, { metatype }] of modules) {  
    await this.reflectImports(metatype, token, metatype.name);  
    this.reflectProviders(metatype, token);  
    this.reflectControllers(metatype, token);  
    this.reflectExports(metatype, token);  
  }  
}
```

@nestjs/core@11.1.5: scanner.ts#L202-L211 ([link](#))

reflectImports

1. `imports` 메타데이터를 불러와서
2. 모듈 인스턴스에 등록합니다.

```
public async reflectImports(  
  module: Type<unknown>,  
  token: string,  
  context: string,  
) {  
  const modules = [  
    ...this.reflectMetadata(MODULE_METADATA.IMPORTS, module),  
    ...this.container.getDynamicMetadataByToken(  
      token,  
      MODULE_METADATA.IMPORTS as 'imports',  
    )!,  
  ];  
  for (const related of modules) {  
    await this.insertImport(related, token, context);  
  }  
}
```

@nestjs/core@11.1.5: scanner.ts#L213-L228 ([link](#))

reflectImports

1. `imports` 메타데이터를 불러와서
2. 모듈 인스턴스에 등록합니다.

```
public async addImport(  
  relatedModule: Type<any> | DynamicModule,  
  token: string,  
) {  
  if (!this.modules.has(token)) {  
    return;  
  }  
  const moduleRef = this.modules.get(token)!;  
  const { token: relatedModuleToken } =  
    await this.moduleCompiler.compile(relatedModule);  
  const related = this.modules.get(relatedModuleToken)!;  
  moduleRef.addImport(related);  
}
```

@nestjs/core@11.1.5: container.ts#L238-L250 ([link](#))

reflectImports

- 결과적으로 모든 모듈이 자신이 참조하는 모듈을 imports로 갖게 됩니다.
- 즉, **모듈 의존성 트리**가 만들어집니다.

```
public async addImport(  
  relatedModule: Type<any> | DynamicModule,  
  token: string,  
) {  
  if (!this.modules.has(token)) {  
    return;  
  }  
  const moduleRef = this.modules.get(token)!;  
  const { token: relatedModuleToken } =  
    await this.moduleCompiler.compile(relatedModule);  
  const related = this.modules.get(relatedModuleToken)!;  
  moduleRef.addImport(related);  
}
```

@nestjs/core@11.1.5: container.ts#L238-L250 ([link](#))

scanModulesForDependencies

- `reflectImports`를 모두 봤으니, 이제
`reflectProviders`를 볼 차례입니다.
 - `InstanceWrapper`를 생성한다고
하는데, 애가 누구죠?

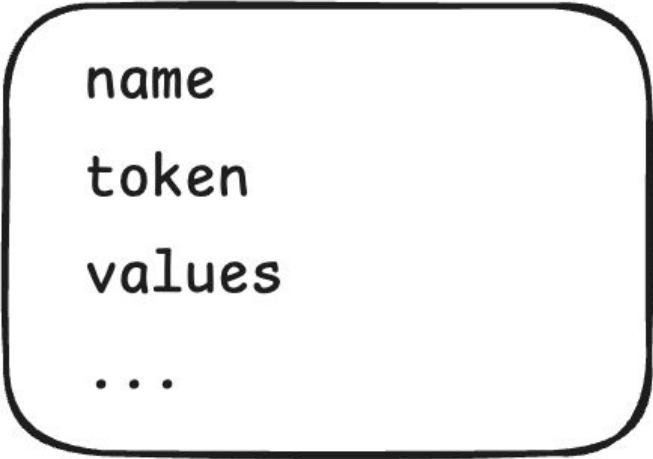
```
public async scanModulesForDependencies(  
  modules: Map<string, Module> = this.container.getModules(),  
) {  
  for (const [token, { metatype }] of modules) {  
    await this.reflectImports(metatype, token, metatype.name);  
    this.reflectProviders(metatype, token);  
    this.reflectControllers(metatype, token);  
    this.reflectExports(metatype, token);  
  }  
}
```

@nestjs/core@11.1.5: scanner.ts#L202-L211 ([link](#))

InstanceWrapper?

- 이름 그대로 **인스턴스의 래퍼**
- 처음에는 비어있다 의존성 처리 과정에서 값 (=인스턴스)이 채워집니다.
- 실제 인스턴스 외에도 **메타데이터**를 저장하기 위해 사용합니다.
 - 이름, 토큰, ...
- `reflectProviders`는 모든 모듈의 `InstanceWrapper`를 만드는 역할인데요,

InstanceWrapper



```
name
token
values
...
```

본론 #2 : 프로바이더 간 의존성 관계 기록 - 3번 과정 (모든 모듈을 순회하며 각 프로바이더에 대한 `InstanceWrapper` 생성)

reflectProviders

1. `providers` 메타데이터를 불러와서
2. 모듈 인스턴스에 등록합니다.

```
public reflectProviders(module: Type<any>, token: string) {  
  const providers = [  
    ...this.reflectMetadata(MODULE_METADATA.PROVIDERS, module),  
    ...this.container.getDynamicMetadataByToken(  
      token,  
      MODULE_METADATA.PROVIDERS as 'providers',  
    )!,  
  ];  
  providers.forEach(provider => {  
    this.insertProvider(provider, token);  
    this.reflectDynamicMetadata(provider, token);  
  });  
}
```

@nestjs/core@11.1.5: scanner.ts#L230-L242 ([link](#))

본론 #2 : 프로바이더 간 의존성 관계 기록 - 3번 과정 (모든 모듈을 순회하며 각 프로바이더에 대한 `InstanceWrapper` 생성)

reflectProviders

1. `providers` 메타데이터를 불러와서
2. 모듈 인스턴스에 등록합니다.

```
public addProvider(  
  provider: Provider,  
  token: string,  
  enhancerSubtype?: EnhancerSubtype,  
): string | symbol | Function {  
  const moduleRef = this.modules.get(token);  
  // ...  
  const providerKey = moduleRef.addProvider(provider, enhancerSubtype!);  
  const providerRef = moduleRef.getProviderByKey(providerKey);  
  
  DiscoverableMetaHostCollection.inspectProvider(this.modules, providerRef);  
  
  return providerKey as Function;  
}
```

@nestjs/core@11.1.5: container.ts#L252-L270 ([link](#))

addProvider

- 모듈 인스턴스에 등록할 때
`InstanceWrapper`가 생성됩니다.
 - Provider 유형에 따라 다르게 생성됩니다.
 - ex) 단순 클래스 프로바이더

```
this._providers.set(  
  provider,  
  new InstanceWrapper({  
    token: provider,  
    name: (provider as Type<Injectable>).name,  
    metatype: provider as Type<Injectable>,  
    instance: null,  
    isResolved: false,  
    scope: getClassScope(provider),  
    durable: isDurable(provider),  
    host: this,  
  }),  
);
```

@nestjs/core@11.1.5: module.ts#L266-L278 ([link](#))

여기까지 정리하면,

- `NestFactory.create`를 호출하면 의존성 처리가 시작됩니다.
- 모든 모듈을 **DFS**로 순회하며 컨테이너에 등록합니다.
- 등록한 모듈을 모두 순회하며,
 - a. `imports` 메타데이터를 읽어 **모듈 의존성 트리**를 만듭니다.
 - b. `providers` 메타데이터를 읽어 각 프로바이더에 대한 **`InstanceWrapper`**를 생성합니다.
- 의존성 방향도 알았고.. 어떤 프로바이더가 어떤 모듈에 있는지도 알았으니..
- 이제 진짜 인스턴스만 만들면 되겠네요?

인스턴스를 생성합시다.

- 아까 앞에서 본 코드에서...
- `createInstanceOfDependencies``
를 살펴볼 차례입니다.

```
await ExceptionsZone.asyncRun(  
  async () => {  
    await dependenciesScanner.scan(module);  
    await instanceLoader.createInstancesOfDependencies();  
    dependenciesScanner.applyApplicationProviders();  
  },  
  teardown,  
  this.autoFlushLogs,  
);
```

@nestjs/core@11.1.5: nest-factory.ts#L235-L243 ([link](#))

인스턴스 생성 과정

1. 모든 모듈의 모든 프로바이더를 동시에 순회하며 프로바이더 로딩 시작
2. 프로바이더의 의존성 목록을 가져와 먼저 자신이 속한 모듈에서 탐색
 - a. 존재한다면 해당 `InstanceWrapper`의 의존성 처리 먼저 완료시킴
 - b. 존재하지 않는다면 참조하고 있는 모듈에서 프로바이더를 탐색
3. 의존성이 **모두 인스턴스화 완료되면** 이들을 활용하여 대상 프로바이더 인스턴스화 진행
4. 이걸 **모든 프로바이더**에 대해 진행하면 의존성 처리가 완료됩니다.
 - 복잡하고 사용되는 메서드도 많습니다. (주요 메서드만 14개..)
 - 다 살펴보면 당연히 좋겠지만 시간 상 이유로 주요 로직 몇 개만 살펴보겠습니다.

의존하는 프로바이더 먼저 로딩

- `loadInstance` 메서드의 한 부분입니다.
- 인스턴스를 생성하려면 생성자를 호출해야 합니다. 이를 위해 생성자 내 의존성을 먼저 처리한 후, 콜백을 호출합니다.
 - `resolveConstructorParams` 역할

```
const t0 = this.getNowTimestamp();
const callback = async (instances: unknown[]) => {
  const properties = await this.resolveProperties(
    wrapper,
    moduleRef,
    inject as InjectionToken[],
    contextId,
    wrapper,
    inquirer,
  );
  const instance = await this.instantiateClass(
    instances,
    wrapper,
    targetWrapper,
    contextId,
    inquirer,
  );
  this.applyProperties(instance, properties);
  wrapper.initTime = this.getNowTimestamp() - t0;
  settlementSignal.complete();
};
await this.resolveConstructorParams<T>(
  wrapper,
  moduleRef,
  inject as InjectionToken[],
callback,
  contextId,
  wrapper,
  inquirer,
);
```

@nestjs/core@11.1.5: injector.ts#L166-L195 ([link](#))

의존하는 프로바이더 먼저 로딩

- 생성자 파라미터에 대한 인스턴스를 만들면 콜백을 호출합니다.
- 콜백에서는 이를 활용해 대상 프로바이더의 인스턴스를 만듭니다.
 - `instantiateClass` 역할
- 그런 다음 만들어진 인스턴스에 프로퍼티를 주입하면서 **프로바이더 하나의 의존성 처리**가 완료됩니다.

```
const t0 = this.getNowTimestamp();
const callback = async (instances: unknown[]) => {
  const properties = await this.resolveProperties(
    wrapper,
    moduleRef,
    inject as InjectionToken[],
    contextId,
    wrapper,
    inquirer,
  );
  const instance = await this.instantiateClass(
    instances,
    wrapper,
    targetWrapper,
    contextId,
    inquirer,
  );
  this.applyProperties(instance, properties);
  wrapper.initTime = this.getNowTimestamp() - t0;
  settlementSignal.complete();
};
await this.resolveConstructorParams<T>({
  wrapper,
  moduleRef,
  inject as InjectionToken[],
  callback,
  contextId,
  wrapper,
  inquirer,
});
```

@nestjs/core@11.1.5: injector.ts#L166-L195 ([link](#))

생성자 내 의존성을 인식하는 방법

- 생성자의 의존성을 처리하는 과정에서 사용하는 메서드입니다.
- 여기서 앞서 본론 #1에서 살펴봤던 의존성 목록을 합쳐 사용하는 모습을 볼 수 있습니다.
- 이 과정을 통해 NestJS에게 생성자에 어떤 **프로바이더를 주입해야 하는지** 알려줄 수 있습니다.

```
public reflectConstructorParams<T>(type: Type<T>): any[] {  
  const paramtypes = [  
    ... (Reflect.getMetadata(PARAMTYPES_METADATA, type) || []),  
  ];  
  const selfParams = this.reflectSelfParams<T>(type);  
  
  selfParams.forEach(({ index, param }) => (paramtypes[index] = param));  
  return paramtypes;  
}  
  
public reflectSelfParams<T>(type: Type<T>): any[] {  
  return Reflect.getMetadata(SELF_DECLARED_DEPS_METADATA, type) || [];  
}
```

@nestjs/core@11.1.5: injector.ts#L433-L441 ([link](#)), #L447-L449 ([link](#))

인스턴스화!

- `instantiateClass` 메서드의 한 부분입니다.
- `new`를 사용해 새로운 인스턴스를 만드는 모습을 볼 수 있습니다.
 - `instances`는 앞서 생성자 내 의존성 처리 과정에서 생성된 인스턴스들입니다.
 - 즉, 생성자의 파라미터입니다.
- 이걸 모든 프로바이더에 반복하면 끝납니다.

```
public async instantiateClass<T = any>(  
  instances: any[],  
  /* ... */  
) : Promise<T> {  
  // ...  
  
  if (isNil(inject) && isInContext) {  
    instanceHost.instance = wrapper.forwardRef  
      ? Object.assign(  
        instanceHost.instance,  
        new (metatype as Type<any>)(...instances),  
      )  
      : new (metatype as Type<any>)(...instances);  
  
    instanceHost.instance = this.instanceDecorator(instanceHost.instance);  
  } else if (isInContext) {  
    // ...  
  }  
  instanceHost.isResolved = true;  
  return instanceHost.instance;  
}
```

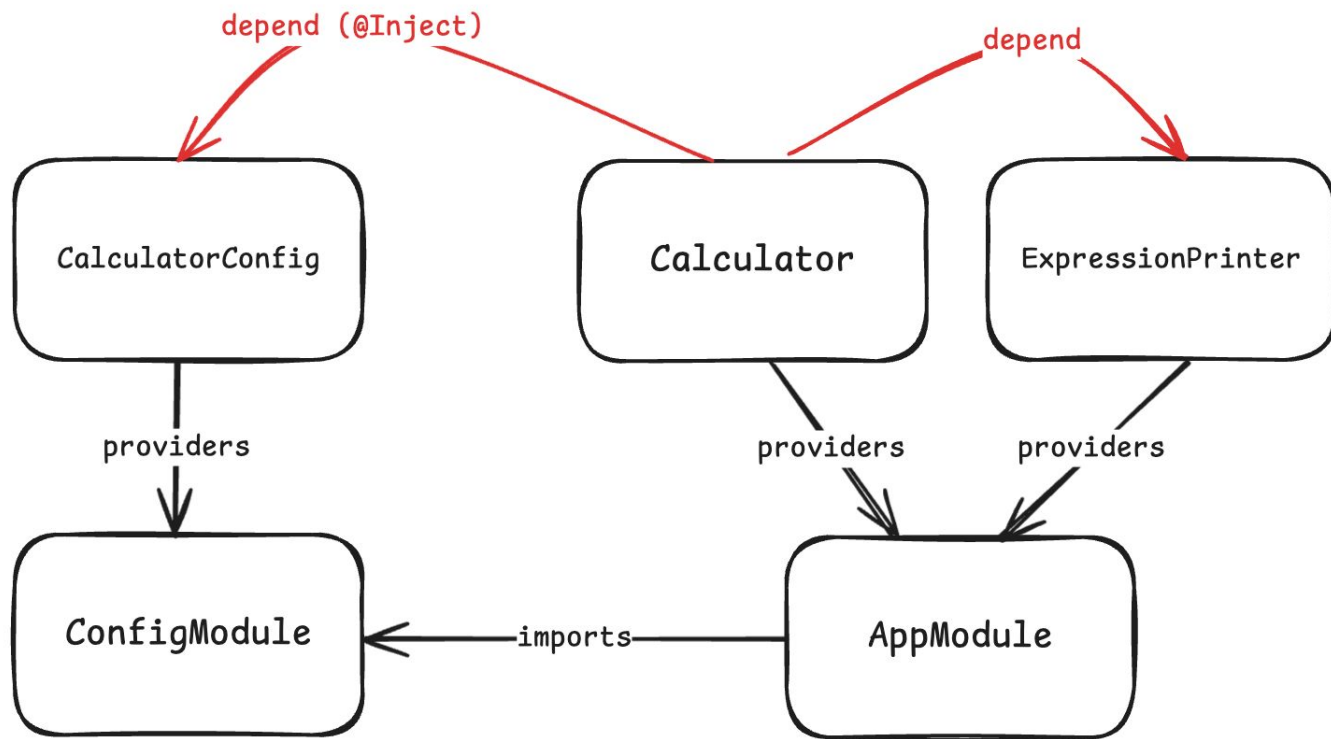
@nestjs/core@11.1.5: injector.ts#L806-L848 ([link](#))

여기까지 정리하면,

- 모든 프로바이더에 대해,
 - a. 생성자 내 의존성 처리
 - b. 프로퍼티 내 의존성 처리
 - c. 완료되면 처리했던 의존성 적용하여 새로운 인스턴스 생성!
- 위 작업이 끝나면 의존성 주입 처리가 완료되고, 서버가 켜지기 시작합니다.

예시

의존성 처리 예시 상황



예시

초기 상황

CalculatorConfig

Calculator

ExpressionPrinter

ConfigModule

AppModule

예시

메타데이터 설정

CalculatorConfig

Calculator

ExpressionPrinter

```
design:paramtypes: [Object, ExpressionPrinter]  
self:paramtypes: [{index: 0, param: CalculatorConfig}]
```

ConfigModule

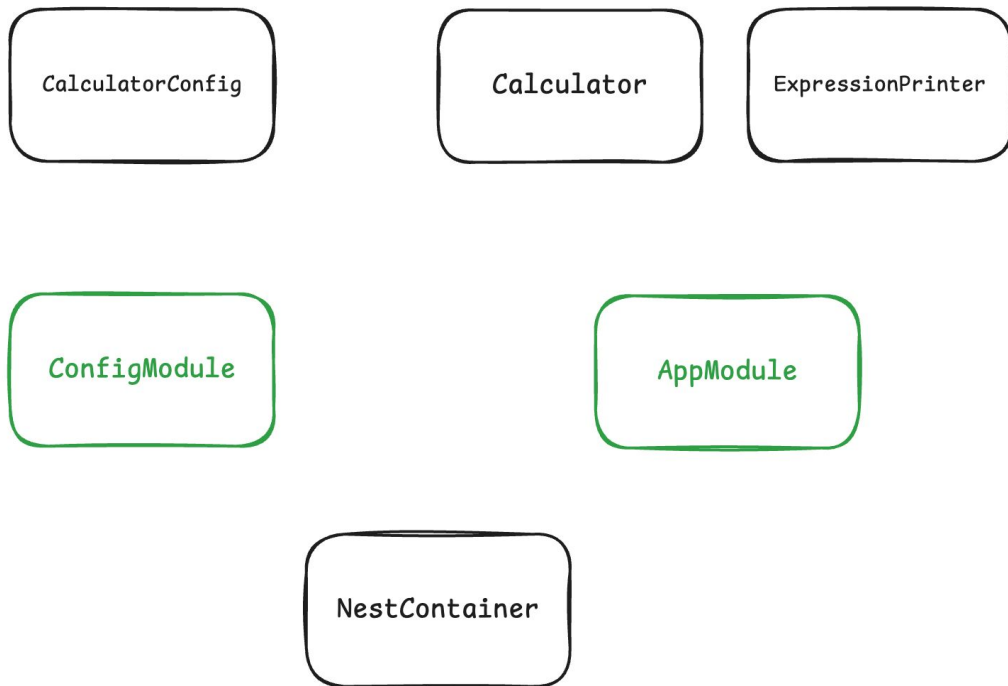
AppModule

```
providers: [CalculatorConfig]  
exports: [CalculatorConfig]
```

```
imports: [ConfigModule]  
providers: [Calculator, ExpressionPrinter]
```

예시

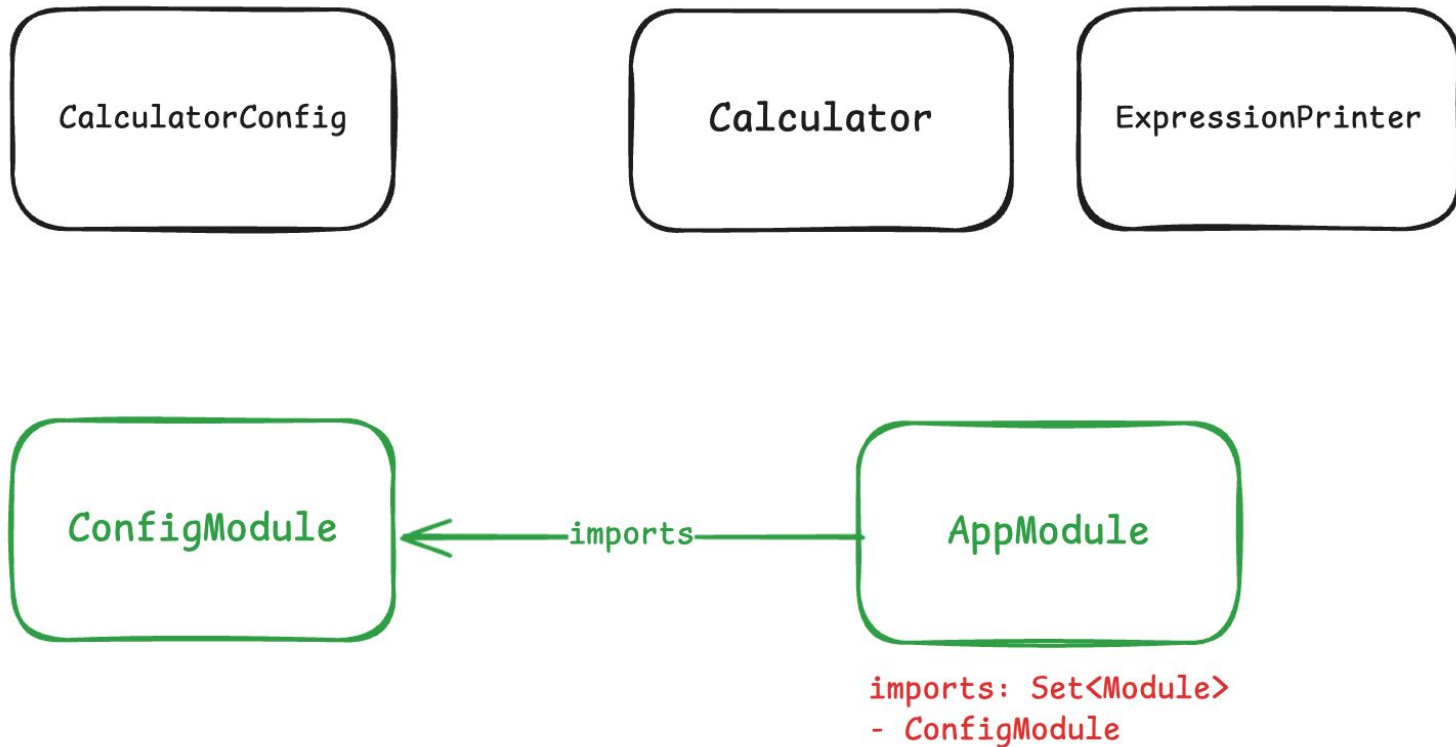
모듈 인스턴스 생성 및 컨테이너에 모듈 등록



```
modules: Map<string, Module>  
- AppModule  
- ConfigModule
```

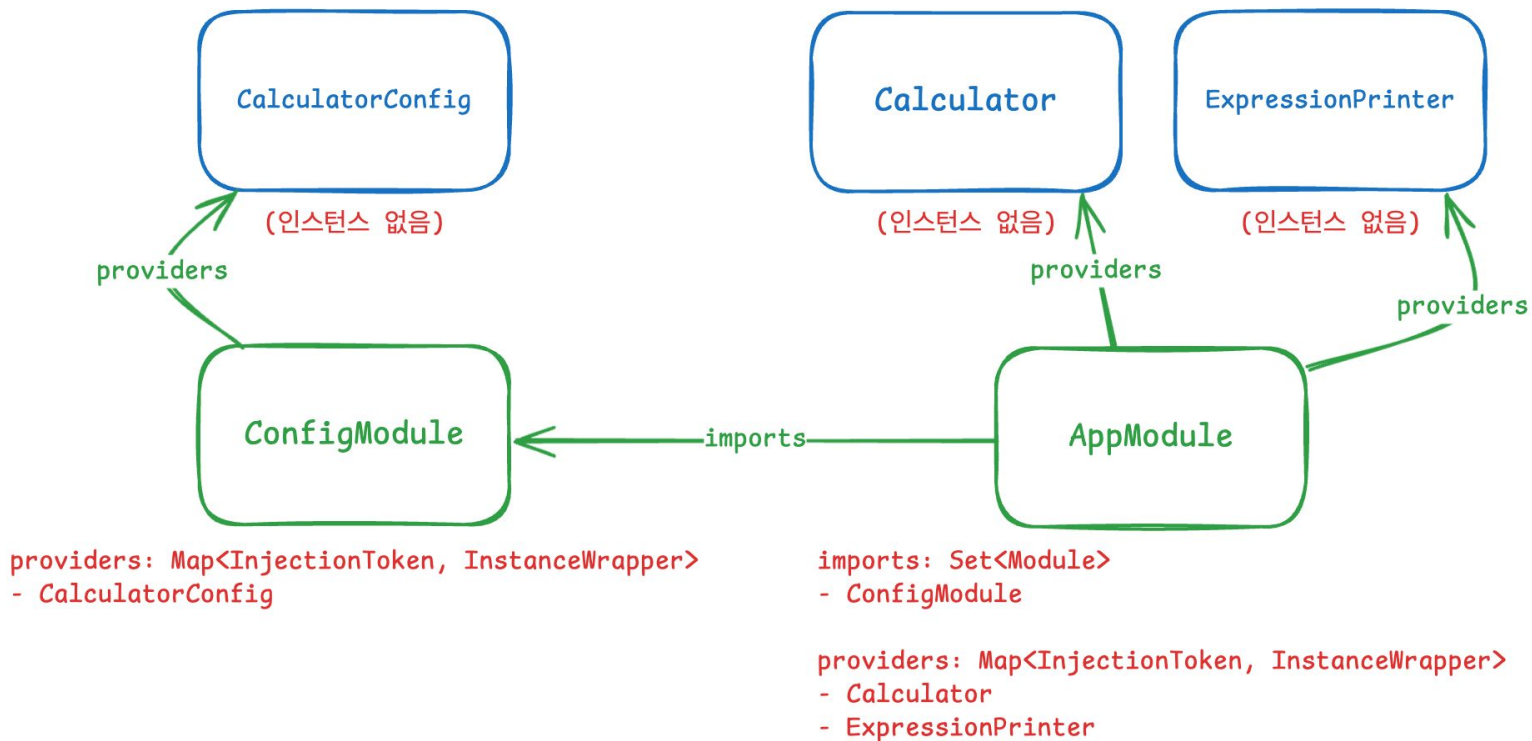
예시

모듈 간 의존 트리 생성



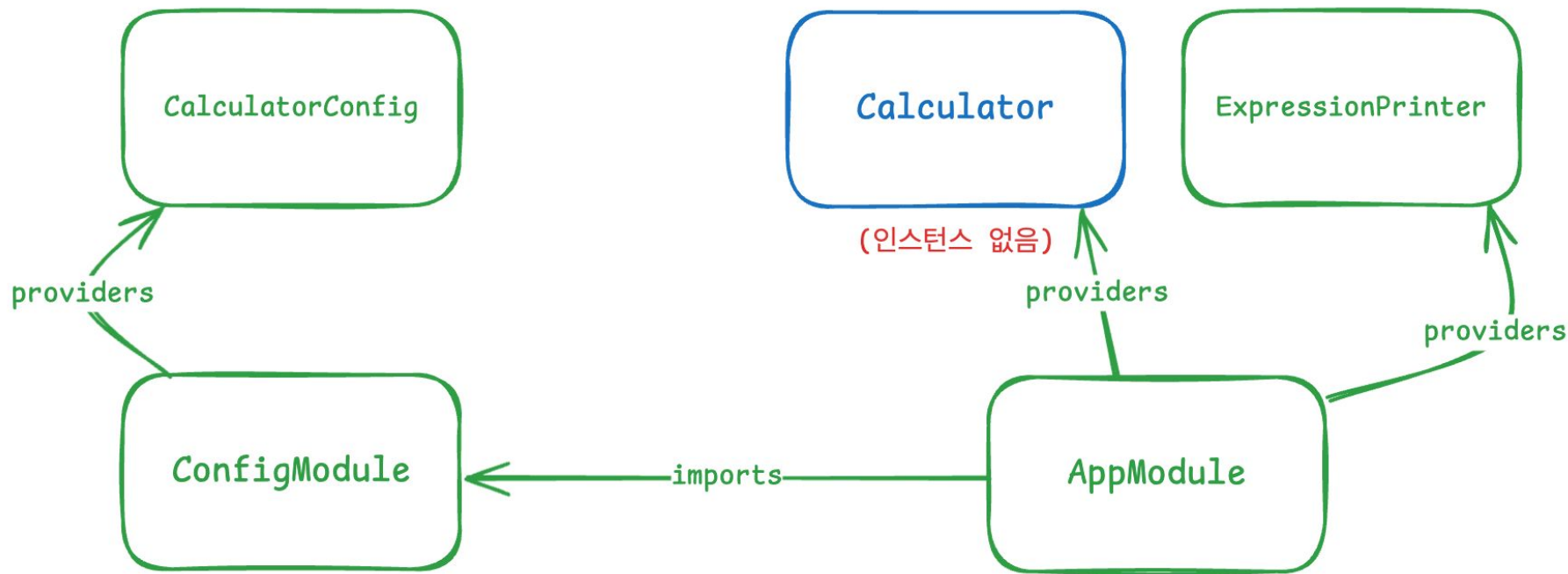
예시

`InstanceWrapper` 생성 및 모듈 등록

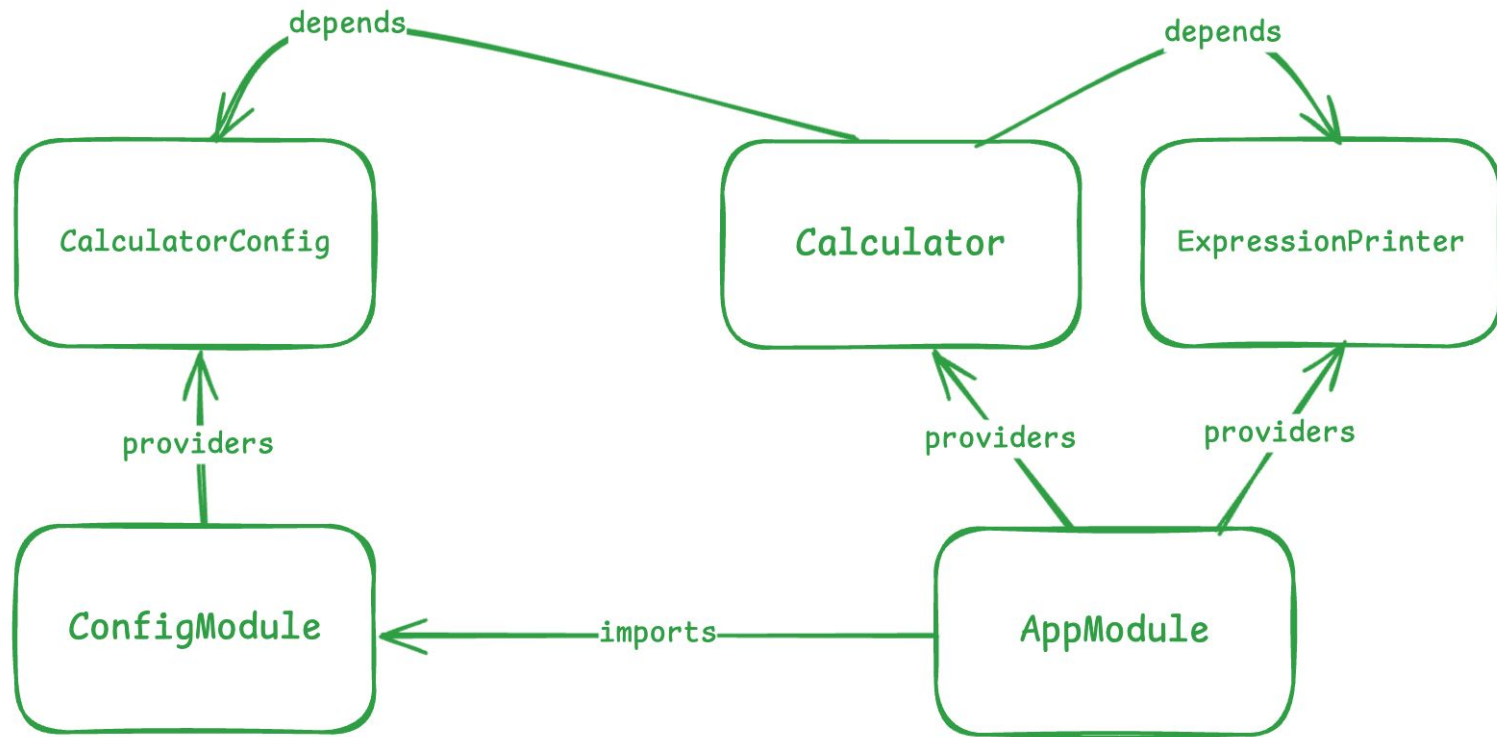


예시

의존하는 프로바이더의 인스턴스 생성



예시
끝!



마무리

- NestJS 의존성 주입 과정을 깊게 알아봤습니다.
- 최대한 NestJS 관련 지식 없이도 이해하실 수 있도록 준비했는데, 괜찮았으면 좋겠습니다.
- “사사뚝 코마없”
 - 사람 사는 거 똑같고 코딩에 마법은 없다.
 - 결국 안에서 우리를 위해 열심히 의존성을 처리해주고 있었습니다.
- 감사합니다.