Module reference

Nest provides the `ModuleRef` class to navigate the internal list of providers and obtain a reference to any provider using its injection token as a lookup key. The `ModuleRef` class also provides a way to dynamically instantiate both static and scoped providers. `ModuleRef` can be injected into a class in the normal way:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(private moduleRef: ModuleRef) {}
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }
}
```

> info **Hint** The `ModuleRef` class is imported from the `@nestjs/core` package.

Retrieving instances

The `ModuleRef` instance (hereafter we'll refer to it as the **module reference**) has a `get()` method. This method retrieves a provider, controller, or injectable (e.g., guard, interceptor, etc.) that exists (has been instantiated) in the **current** module using its injection token/class name.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private service: Service;
  constructor(private moduleRef: ModuleRef) {}

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  onModuleInit() {
    this.service = this.moduleRef.get(Service);
```

```
  }
}
```

> warning **Warning** You can't retrieve scoped providers (transient or request-scoped) with the `get()` method. Instead, use the technique described below. Learn how to control scopes here.

To retrieve a provider from the global context (for example, if the provider has been injected in a different module), pass the `{{ '{' }} strict: false {{ '}' }}` option as a second argument to `get()`.

```
this.moduleRef.get(Service, { strict: false });
```

Resolving scoped providers

To dynamically resolve a scoped provider (transient or request-scoped), use the `resolve()` method, passing the provider's injection token as an argument.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private transientService: TransientService;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.transientService = await
this.moduleRef.resolve(TransientService);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.transientService = await
this.moduleRef.resolve(TransientService);
  }
}
```

The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique **context identifier**. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

```

```

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}
```

To generate a single instance across multiple `resolve()` calls, and ensure they share the same generated DI container sub-tree, you can pass a context identifier to the `resolve()` method. Use the `ContextIdFactory` class to generate a context identifier. This class provides a `create()` method that returns an appropriate unique identifier.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService, contextId),
      this.moduleRef.resolve(TransientService, contextId),
    ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}
@@switch
```

```
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService, contextId),
      this.moduleRef.resolve(TransientService, contextId),
    ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}
```

info **Hint** The `ContextIdFactory` class is imported from the `@nestjs/core` package.

Registering `REQUEST` provider

Manually generated context identifiers (with `ContextIdFactory.create()`) represent DI sub-trees in which `REQUEST` provider is `undefined` as they are not instantiated and managed by the Nest dependency injection system.

To register a custom `REQUEST` object for a manually created DI sub-tree, use the `ModuleRef#registerRequestByContextId()` method, as follows:

```
const contextId = ContextIdFactory.create();
this.moduleRef.registerRequestByContextId(/* YOUR_REQUEST_OBJECT */,
contextId);
```

Getting current sub-tree

Occasionally, you may want to resolve an instance of a request-scoped provider within a **request context**. Let's say that `CatsService` is request-scoped and you want to resolve the `CatsRepository` instance which is also marked as a request-scoped provider. In order to share the same DI container sub-tree, you must obtain the current context identifier instead of generating a new one (e.g., with the `ContextIdFactory.create()` function, as shown above). To obtain the current context identifier, start by injecting the request object using `@Inject()` decorator.

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(REQUEST) private request: Record<string, unknown>,
  ) {}
}
```

```
@@switch
@Injectable()
@Dependencies(REQUEST)
export class CatsService {
  constructor(request) {
    this.request = request;
  }
}
```

> info **Hint** Learn more about the request provider here.

Now, use the `getByRequest()` method of the `ContextIdFactory` class to create a context id based on the request object, and pass this to the `resolve()` call:

```
const contextId = ContextIdFactory.getByRequest(this.request);
const catsRepository = await this.moduleRef.resolve(CatsRepository,
contextId);
```

Instantiating custom classes dynamically

To dynamically instantiate a class that **wasn't previously registered** as a **provider**, use the module reference's `create()` method.

```
@@filename(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private catsFactory: CatsFactory;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
@@switch
@Injectable()
@Dependencies(ModuleRef)
export class CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}
```

This technique enables you to conditionally instantiate different classes outside of the framework container.