MikroORM

This recipe is here to help users getting started with MikroORM in Nest. MikroORM is the TypeScript ORM for Node.js based on Data Mapper, Unit of Work and Identity Map patterns. It is a great alternative to TypeORM and migration from TypeORM should be fairly easy. The complete documentation on MikroORM can be found here.

> info **info** `@mikro-orm/nestjs` is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the appropriate repository.

Installation

Easiest way to integrate MikroORM to Nest is via `@mikro-orm/nestjs` module. Simply install it next to Nest, MikroORM and underlying driver:

```
$ npm i @mikro-orm/core @mikro-orm/nestjs @mikro-orm/mysql # for
mysql/mariadb
```

MikroORM also supports `postgres`, `sqlite`, and `mongo`. See the official docs for all drivers.

Once the installation process is completed, we can import the `MikroOrmModule` into the root `AppModule`.

```
@Module({
  imports: [
    MikroOrmModule.forRoot({
      entities: ['./dist/entities'],
      entitiesTs: ['./src/entities'],
      dbName: 'my-db-name.sqlite3',
      type: 'sqlite',
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

The `forRoot()` method accepts the same configuration object as `init()` from the MikroORM package. Check this page for the complete configuration documentation.

Alternatively we can configure the CLI by creating a configuration file `mikro-orm.config.ts` and then call the `forRoot()` without any arguments. This won't work when you use a build tools that use tree shaking.

```
@Module({
  imports: [
    MikroOrmModule.forRoot(),
```

```
  ],
  ...
})
export class AppModule {}
```

Afterward, the `EntityManager` will be available to inject across entire project (without importing any module elsewhere).

```
import { MikroORM } from '@mikro-orm/core';
// Import EntityManager from your driver package or `@mikro-orm/knex`
import { EntityManager } from '@mikro-orm/mysql';

@Injectable()
export class MyService {
  constructor(
    private readonly orm: MikroORM,
    private readonly em: EntityManager,
  ) {}
}
```

> info **info** Notice that the `EntityManager` is imported from the `@mikro-orm/driver` package, where driver is `mysql`, `sqlite`, `postgres` or what driver you are using. In case you have `@mikro-orm/knex` installed as a dependency, you can also import the `EntityManager` from there.

Repositories

MikroORM supports the repository design pattern. For every entity we can create a repository. Read the complete documentation on repositories here. To define which repositories should be registered in the current scope you can use the `forFeature()` method. For example, in this way:

> info **info** You should **not** register your base entities via `forFeature()`, as there are no repositories for those. On the other hand, base entities need to be part of the list in `forRoot()` (or in the ORM config in general).

```
// photo.module.ts
@Module({
  imports: [MikroOrmModule.forFeature([Photo])],
  providers: [PhotoService],
  controllers: [PhotoController],
})
export class PhotoModule {}
```

and import it into the root `AppModule`:

```
// app.module.ts
@Module({
```

```
  imports: [MikroOrmModule.forRoot(...), PhotoModule],
})
export class AppModule {}
```

In this way we can inject the `PhotoRepository` to the `PhotoService` using the `@InjectRepository()` decorator:

```
@Injectable()
export class PhotoService {
  constructor(
    @InjectRepository(Photo)
    private readonly photoRepository: EntityRepository<Photo>,
  ) {}
}
```

Using custom repositories

When using custom repositories, we can get around the need for `@InjectRepository()` decorator by naming our repositories the same way as `getRepositoryToken()` method do:

```
export const getRepositoryToken = <T>(entity: EntityName<T>) =>
  `${Utils.className(entity)}Repository`;
```

In other words, as long as we name the repository same was as the entity is called, appending `Repository` suffix, the repository will be registered automatically in the Nest DI container.

```
// `**./author.entity.ts**`
@Entity()
export class Author {
  // to allow inference in `em.getRepository()`
  [EntityRepositoryType]?: AuthorRepository;
}

// `**./author.repository.ts**`
@Repository(Author)
export class AuthorRepository extends EntityRepository<Author> {
  // your custom methods...
}
```

As the custom repository name is the same as what `getRepositoryToken()` would return, we do not need the `@InjectRepository()` decorator anymore:

```
@Injectable()
export class MyService {
```

```
  constructor(private readonly repo: AuthorRepository) {}
}
```

Load entities automatically

> info **info** `autoLoadEntities` option was added in v4.1.0

Manually adding entities to the entities array of the connection options can be tedious. In addition, referencing entities from the root module breaks application domain boundaries and causes leaking implementation details to other parts of the application. To solve this issue, static glob paths can be used.

Note, however, that glob paths are not supported by webpack, so if you are building your application within a monorepo, you won't be able to use them. To address this issue, an alternative solution is provided. To automatically load entities, set the `autoLoadEntities` property of the configuration object (passed into the `forRoot()` method) to `true`, as shown below:

```
@Module({
  imports: [
    MikroOrmModule.forRoot({
      ...
      autoLoadEntities: true,
    }),
  ],
})
export class AppModule {}
```

With that option specified, every entity registered through the `forFeature()` method will be automatically added to the entities array of the configuration object.

> info **info** Note that entities that aren't registered through the `forFeature()` method, but are only referenced from the entity (via a relationship), won't be included by way of the `autoLoadEntities` setting.

> info **info** Using `autoLoadEntities` also has no effect on the MikroORM CLI - for that we still need CLI config with the full list of entities. On the other hand, we can use globs there, as the CLI won't go thru webpack.

Serialization

> warning **Note** MikroORM wraps every single entity relation in a `Reference<T>` or a `Collection<T>` object, in order to provide better type-safety. This will make Nest's built-in serializer blind to any wrapped relations. In other words, if you return MikroORM entities from your HTTP or WebSocket handlers, all of their relations will NOT be serialized.

Luckily, MikroORM provides a serialization API which can be used in lieu of `ClassSerializerInterceptor`.

```

```

```
@Entity()
export class Book {
  @Property({ hidden: true }) // Equivalent of class-transformer's
`@Exclude`
  hiddenField = Date.now();

  @Property({ persist: false }) // Similar to class-transformer's
`@Expose()`. Will only exist in memory, and will be serialized.
  count?: number;

  @ManyToOne({
    serializer: (value) => value.name,
    serializedName: 'authorName',
  }) // Equivalent of class-transformer's `@Transform()`
  author: Author;
}
```

Request scoped handlers in queues

> info **info** `@UseRequestContext()` decorator was added in v4.1.0

As mentioned in the docs, we need a clean state for each request. That is handled automatically thanks to the `RequestContext` helper registered via middleware.

But middlewares are executed only for regular HTTP request handles, what if we need a request scoped method outside of that? One example of that is queue handlers or scheduled tasks.

We can use the `@UseRequestContext()` decorator. It requires you to first inject the `MikroORM` instance to current context, it will be then used to create the context for you. Under the hood, the decorator will register new request context for your method and execute it inside the context.

```
@Injectable()
export class MyService {
  constructor(private readonly orm: MikroORM) {}

  @UseRequestContext()
  async doSomething() {
    // this will be executed in a separate context
  }
}
```

Using `AsyncLocalStorage` for request context

By default, the `domain` api is used in the `RequestContext` helper. Since `@mikro-orm/core@4.0.3`, you can use the new `AsyncLocalStorage` too, if you are on up to date node version:

```
// create new (global) storage instance
const storage = new AsyncLocalStorage<EntityManager>();

@Module({
  imports: [
    MikroOrmModule.forRoot({
      // ...
      registerRequestContext: false, // disable automatic middleware
      context: () => storage.getStore(), // use our AsyncLocalStorage
instance
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}

// register the request context middleware
const app = await NestFactory.create(AppModule, { ... });
const orm = app.get(MikroORM);

app.use((req, res, next) => {
  storage.run(orm.em.fork(true, true), next);
});
```

Testing

The `@mikro-orm/nestjs` package exposes `getRepositoryToken()` function that returns prepared token based on a given entity to allow mocking the repository.

```
@Module({
  providers: [
    PhotoService,
    {
      provide: getRepositoryToken(Photo),
      useValue: mockedRepository,
    },
  ],
})
export class PhotoModule {}
```

Example

A real world example of NestJS with MikroORM can be found here