Mongo

Nest supports two methods for integrating with the MongoDB database. You can either use the built-in TypeORM module described here, which has a connector for MongoDB, or use Mongoose, the most popular MongoDB object modeling tool. In this chapter we'll describe the latter, using the dedicated `@nestjs/mongoose` package.

Start by installing the required dependencies:

```
$ npm i @nestjs/mongoose mongoose
```

Once the installation process is complete, we can import the `MongooseModule` into the root `AppModule`.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [MongooseModule.forRoot('mongodb://localhost/nest')],
})
export class AppModule {}
```

The `forRoot()` method accepts the same configuration object as `mongoose.connect()` from the Mongoose package, as described here.

Model injection

With Mongoose, everything is derived from a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection. Schemas are used to define Models. Models are responsible for creating and reading documents from the underlying MongoDB database.

Schemas can be created with NestJS decorators, or with Mongoose itself manually. Using decorators to create schemas greatly reduces boilerplate and improves overall code readability.

Let's define the `CatSchema`:

```
@@filename(schemas/cat.schema)
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument } from 'mongoose';

export type CatDocument = HydratedDocument<Cat>;

@Schema()
export class Cat {
  @Prop()
  name: string;
```

```
  @Prop()
  age: number;

  @Prop()
  breed: string;
}

export const CatSchema = SchemaFactory.createForClass(Cat);
```

> info **Hint** Note you can also generate a raw schema definition using the `DefinitionsFactory` class (from the `nestjs/mongoose`). This allows you to manually modify the schema definition generated based on the metadata you provided. This is useful for certain edge-cases where it may be hard to represent everything with decorators.

The `@Schema()` decorator marks a class as a schema definition. It maps our `Cat` class to a MongoDB collection of the same name, but with an additional “s” at the end - so the final mongo collection name will be `cats`. This decorator accepts a single optional argument which is a schema options object. Think of it as the object you would normally pass as a second argument of the `mongoose.Schema` class' constructor (e.g., `new mongoose.Schema(_, options)`). To learn more about available schema options, see this chapter.

The `@Prop()` decorator defines a property in the document. For example, in the schema definition above, we defined three properties: `name`, `age`, and `breed`. The schema types for these properties are automatically inferred thanks to TypeScript metadata (and reflection) capabilities. However, in more complex scenarios in which types cannot be implicitly reflected (for example, arrays or nested object structures), types must be indicated explicitly, as follows:

```
@Prop([String])
tags: string[];
```

Alternatively, the `@Prop()` decorator accepts an options object argument (read more about the available options). With this, you can indicate whether a property is required or not, specify a default value, or mark it as immutable. For example:

```
@Prop({ required: true })
name: string;
```

In case you want to specify relation to another model, later for populating, you can use `@Prop()` decorator as well. For example, if `Cat` has `Owner` which is stored in a different collection called `owners`, the property should have type and ref. For example:

```
import * as mongoose from 'mongoose';
import { Owner } from '../owners/schemas/owner.schema';
```

```
// inside the class definition
@Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' })
owner: Owner;
```

In case there are multiple owners, your property configuration should look as follows:

```
@Prop({ type: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' }] })
owner: Owner[];
```

Finally, the **raw** schema definition can also be passed to the decorator. This is useful when, for example, a property represents a nested object which is not defined as a class. For this, use the `raw()` function from the `@nestjs/mongoose` package, as follows:

```
@Prop(raw({
  firstName: { type: String },
  lastName: { type: String }
}))
details: Record<string, any>;
```

Alternatively, if you prefer **not using decorators**, you can define a schema manually. For example:

```
export const CatSchema = new mongoose.Schema({
  name: String,
  age: Number,
  breed: String,
});
```

The `cat.schema` file resides in a folder in the `cats` directory, where we also define the `CatsModule`. While you can store schema files wherever you prefer, we recommend storing them near their related **domain** objects, in the appropriate module directory.

Let's look at the `CatsModule`:

```
@@filename(cats.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { Cat, CatSchema } from './schemas/cat.schema';

@Module({
  imports: [MongooseModule.forFeature([{ name: Cat.name, schema: CatSchema
}])],
  controllers: [CatsController],
  providers: [CatsService],
```

```
})
export class CatsModule {}
```

The `MongooseModule` provides the `forFeature()` method to configure the module, including defining which models should be registered in the current scope. If you also want to use the models in another module, add MongooseModule to the `exports` section of `CatsModule` and import `CatsModule` in the other module.

Once you've registered the schema, you can inject a `Cat` model into the `CatsService` using the `@InjectModel()` decorator:

```
@@filename(cats.service)
import { Model } from 'mongoose';
import { Injectable } from '@nestjs/common';
import { InjectModel } from '@nestjs/mongoose';
import { Cat } from './schemas/cat.schema';
import { CreateCatDto } from './dto/create-cat.dto';

@Injectable()
export class CatsService {
  constructor(@InjectModel(Cat.name) private catModel: Model<Cat>) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
import { Model } from 'mongoose';
import { Injectable, Dependencies } from '@nestjs/common';
import { getModelToken } from '@nestjs/mongoose';
import { Cat } from './schemas/cat.schema';

@Injectable()
@Dependencies(getModelToken(Cat.name))
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
  }

  async create(createCatDto) {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll() {
    return this.catModel.find().exec();
```

```
  }
}
```

Connection

At times you may need to access the native Mongoose Connection object. For example, you may want to make native API calls on the connection object. You can inject the Mongoose Connection by using the `@InjectConnection()` decorator as follows:

```
import { Injectable } from '@nestjs/common';
import { InjectConnection } from '@nestjs/mongoose';
import { Connection } from 'mongoose';

@Injectable()
export class CatsService {
  constructor(@InjectConnection() private connection: Connection) {}
}
```

Multiple databases

Some projects require multiple database connections. This can also be achieved with this module. To work with multiple connections, first create the connections. In this case, connection naming becomes **mandatory**.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/test', {
      connectionName: 'cats',
    }),
    MongooseModule.forRoot('mongodb://localhost/users', {
      connectionName: 'users',
    }),
  ],
})
export class AppModule {}
```

> warning **Notice** Please note that you shouldn't have multiple connections without a name, or with the same name, otherwise they will get overridden.

With this setup, you have to tell the `MongooseModule.forFeature()` function which connection should be used.

```
@Module({
  imports: [
    MongooseModule.forFeature([{ name: Cat.name, schema: CatSchema }],
'cats'),
  ],
})
export class CatsModule {}
```

You can also inject the `Connection` for a given connection:

```
import { Injectable } from '@nestjs/common';
import { InjectConnection } from '@nestjs/mongoose';
import { Connection } from 'mongoose';

@Injectable()
export class CatsService {
  constructor(@InjectConnection('cats') private connection: Connection) {}
}
```

To inject a given `Connection` to a custom provider (for example, factory provider), use the `getConnectionToken()` function passing the name of the connection as an argument.

```
{
  provide: CatsService,
  useFactory: (catsConnection: Connection) => {
    return new CatsService(catsConnection);
  },
  inject: [getConnectionToken('cats')],
}
```

If you are just looking to inject the model from a named database, you can use the connection name as a second parameter to the `@InjectModel()` decorator.

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(@InjectModel(Cat.name, 'cats') private catModel: Model<Cat>)
{}
}
@@switch
@Injectable()
@Dependencies(getModelToken(Cat.name, 'cats'))
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
```

```
    }
  }
```

Hooks (middleware)

Middleware (also called pre and post hooks) are functions which are passed control during execution of asynchronous functions. Middleware is specified on the schema level and is useful for writing plugins (source). Calling `pre()` or `post()` after compiling a model does not work in Mongoose. To register a hook **before** model registration, use the `forFeatureAsync()` method of the `MongooseModule` along with a factory provider (i.e., `useFactory`). With this technique, you can access a schema object, then use the `pre()` or `post()` method to register a hook on that schema. See example below:

```
@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.pre('save', function () {
            console.log('Hello from pre save');
          });
          return schema;
        },
      },
    ]),
  ],
})
export class AppModule {}
```

Like other factory providers, our factory function can be `async` and can inject dependencies through `inject`.

```
@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        imports: [ConfigModule],
        useFactory: (configService: ConfigService) => {
          const schema = CatsSchema;
          schema.pre('save', function() {
            console.log(
              `${configService.get('APP_NAME')}: Hello from pre save`,
            ),
          });
          return schema;
        },
```

```
        inject: [ConfigService],
      },
    ]),
  ],
})
export class AppModule {}
```

Plugins

To register a plugin for a given schema, use the `forFeatureAsync()` method.

```
@Module({
  imports: [
    MongooseModule.forFeatureAsync([
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.plugin(require('mongoose-autopopulate'));
          return schema;
        },
      },
    ]),
  ],
})
export class AppModule {}
```

To register a plugin for all schemas at once, call the `.plugin()` method of the `Connection` object. You should access the connection before models are created; to do this, use the `connectionFactory`:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/test', {
      connectionFactory: (connection) => {
        connection.plugin(require('mongoose-autopopulate'));
        return connection;
      }
    }),
  ],
})
export class AppModule {}
```

Discriminators

Discriminators are a schema inheritance mechanism. They enable you to have multiple models with overlapping schemas on top of the same underlying MongoDB collection.

Suppose you wanted to track different types of events in a single collection. Every event will have a timestamp.

```
@@filename(event.schema)
@Schema({ discriminatorKey: 'kind' })
export class Event {
  @Prop({
    type: String,
    required: true,
    enum: [ClickedLinkEvent.name, SignUpEvent.name],
  })
  kind: string;

  @Prop({ type: Date, required: true })
  time: Date;
}

export const EventSchema = SchemaFactory.createForClass(Event);
```

> info **Hint** The way mongoose tells the difference between the different discriminator models is by the "discriminator key", which is `__t` by default. Mongoose adds a String path called `__t` to your schemas that it uses to track which discriminator this document is an instance of. You may also use the `discriminatorKey` option to define the path for discrimination.

`SignedUpEvent` and `ClickedLinkEvent` instances will be stored in the same collection as generic events.

Now, let's define the `ClickedLinkEvent` class, as follows:

```
@@filename(click-link-event.schema)
@Schema()
export class ClickedLinkEvent {
  kind: string;
  time: Date;

  @Prop({ type: String, required: true })
  url: string;
}

export const ClickedLinkEventSchema =
SchemaFactory.createForClass(ClickedLinkEvent);
```

And `SignUpEvent` class:

```
@@filename(sign-up-event.schema)
@Schema()
export class SignUpEvent {
  kind: string;
  time: Date;

  @Prop({ type: String, required: true })
  user: string;
}

export const SignUpEventSchema =
SchemaFactory.createForClass(SignUpEvent);
```

With this in place, use the `discriminators` option to register a discriminator for a given schema. It works on both `MongooseModule.forFeature` and `MongooseModule.forFeatureAsync`:

```
@@filename(event.module)
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [
    MongooseModule.forFeature([
      {
        name: Event.name,
        schema: EventSchema,
        discriminators: [
          { name: ClickedLinkEvent.name, schema: ClickedLinkEventSchema },
          { name: SignUpEvent.name, schema: SignUpEventSchema },
        ],
      },
    ]),
  ]
})
export class EventsModule {}
```

Testing

When unit testing an application, we usually want to avoid any database connection, making our test suites simpler to set up and faster to execute. But our classes might depend on models that are pulled from the connection instance. How do we resolve these classes? The solution is to create mock models.

To make this easier, the `@nestjs/mongoose` package exposes a `getModelToken()` function that returns a prepared injection token based on a token name. Using this token, you can easily provide a mock implementation using any of the standard custom provider techniques, including `useClass`, `useValue`, and `useFactory`. For example:

```
@Module({
  providers: [
    CatsService,
    {
      provide: getModelToken(Cat.name),
      useValue: catModel,
    },
  ],
})
export class CatsModule {}
```

In this example, a hardcoded `catModel` (object instance) will be provided whenever any consumer injects a `Model<Cat>` using an `@InjectModel()` decorator.

Async configuration

When you need to pass module options asynchronously instead of statically, use the `forRootAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
MongooseModule.forRootAsync({
  useFactory: () => ({
    uri: 'mongodb://localhost/nest',
  }),
});
```

Like other factory providers, our factory function can be `async` and can inject dependencies through `inject`.

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    uri: configService.get<string>('MONGODB_URI'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you can configure the `MongooseModule` using a class instead of a factory, as shown below:

```
MongooseModule.forRootAsync({
  useClass: MongooseConfigService,
});
```

The construction above instantiates `MongooseConfigService` inside `MongooseModule`, using it to create the required options object. Note that in this example, the `MongooseConfigService` has to implement the `MongooseOptionsFactory` interface, as shown below. The `MongooseModule` will call the `createMongooseOptions()` method on the instantiated object of the supplied class.

```
@Injectable()
export class MongooseConfigService implements MongooseOptionsFactory {
  createMongooseOptions(): MongooseModuleOptions {
    return {
      uri: 'mongodb://localhost/nest',
    };
  }
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `MongooseModule`, use the `useExisting` syntax.

```
MongooseModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

Example

A working example is available here.