Layered Style Guides Back-end Patterns

Ihor Lavrov

January 13, 2020

Agenda

- Why do we need to structure our applications
- What is Big Ball of Mud
- Introduction to NodeJS application layers
- Express JS examples
- Data mapper pattern
- Repository pattern
- Components approach
- Nest JS examples

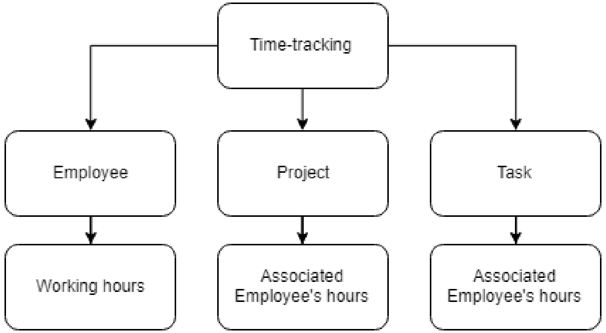
Why do we need to think about application structure

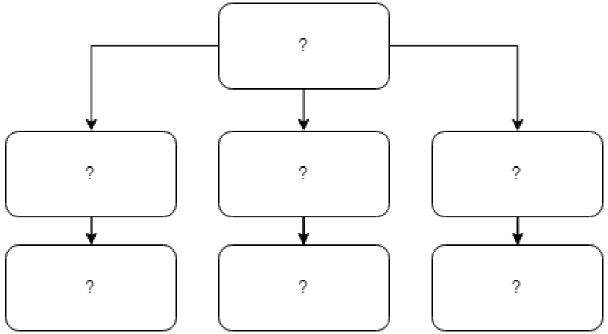
- Easy to add new feature, spending less time to make any change.
- Easy to fix a bug, less time to find a problem and to select solution. Cover correct structured code by unit tests much more easier.
- Keep application maintainable, structured application is much more clear. It requires less time to dive into the code and start to work with it even for newcomers
- Structured application is much more easier to scale. It would be hard to scale application which doesn't have any logical or structural units like classes, modules, layers, etc.
- Following Clean Code principles, everyone would be appreciated.

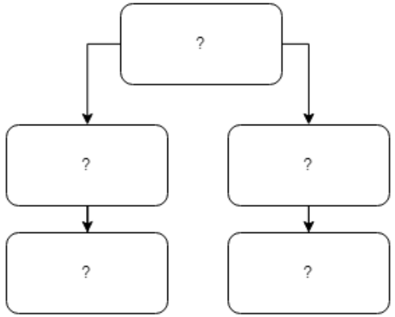
Main issues with unstructured application

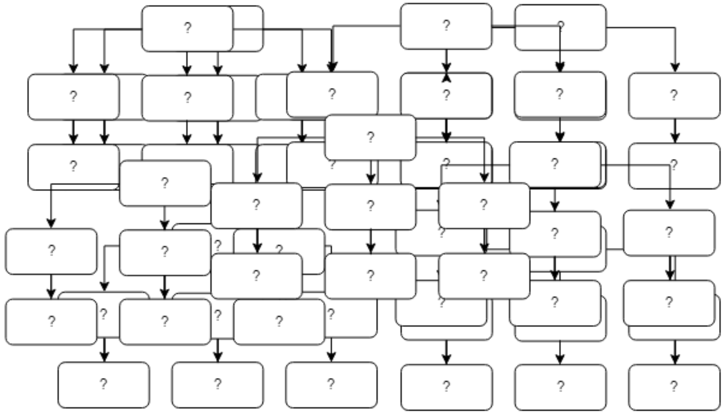
- Unreadable and messy code, making the development process longer and the product itself harder to test
- Useless repetition, making code harder to maintain and manage
- Implementing new features becomes a really challenging task. Since the structure can become a total mess, adding a new feature without messing up existing code can become a real problem

Big ball of mud antipattern









How to manage BBoM

Separation of concerns principle

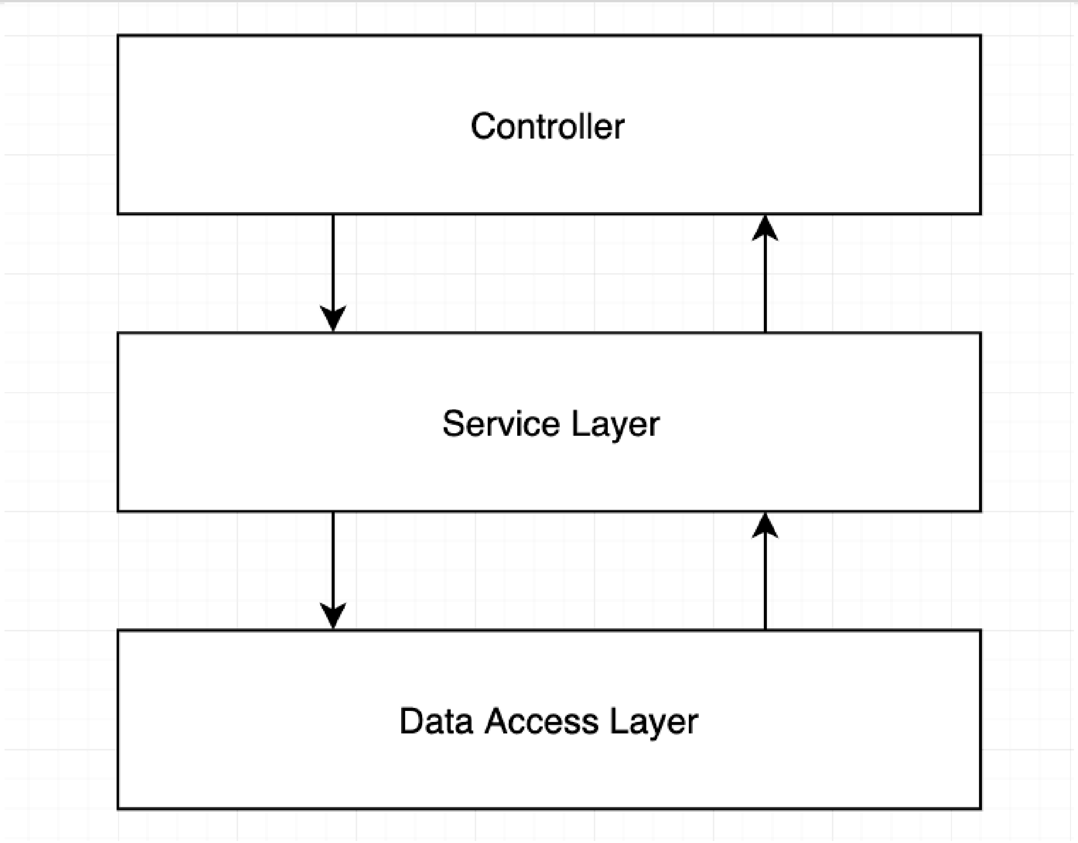
- Comprises the process of separating a system into distinct parts that adhere to a single and unique purpose.
- Aims the managing complexity by establishing a well-organized system.
- Achieved by establishing boundaries.

3 Layers architecture

Rules of 3 Layers approach

- Keep clear separation between the business logic and the API routes
- Use Service layer to handle business logic
- Use config folder to separate configuration files
- Use dependency injection
- Keep correct folder structure

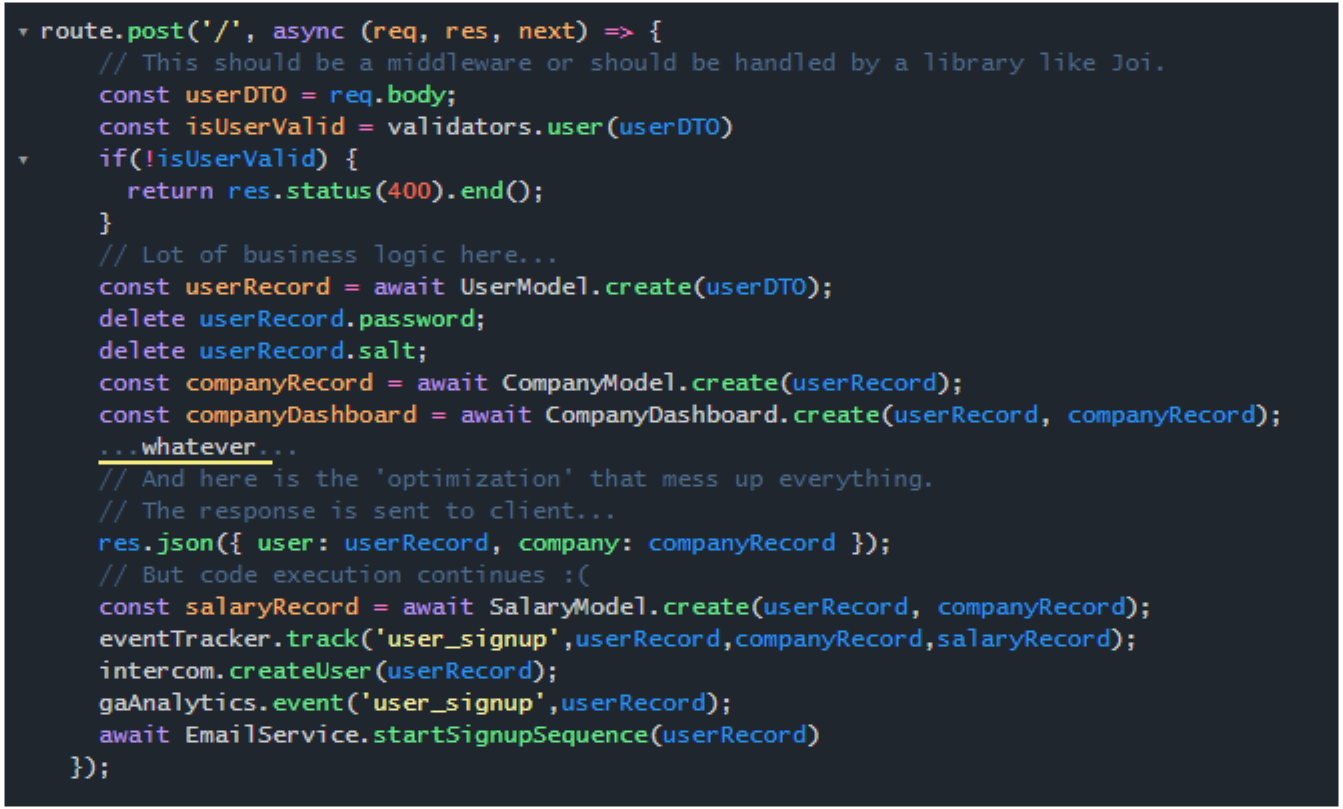
3 Layers structure



Express JS examples

Things you shouldn’t do

Things you shouldn’t do

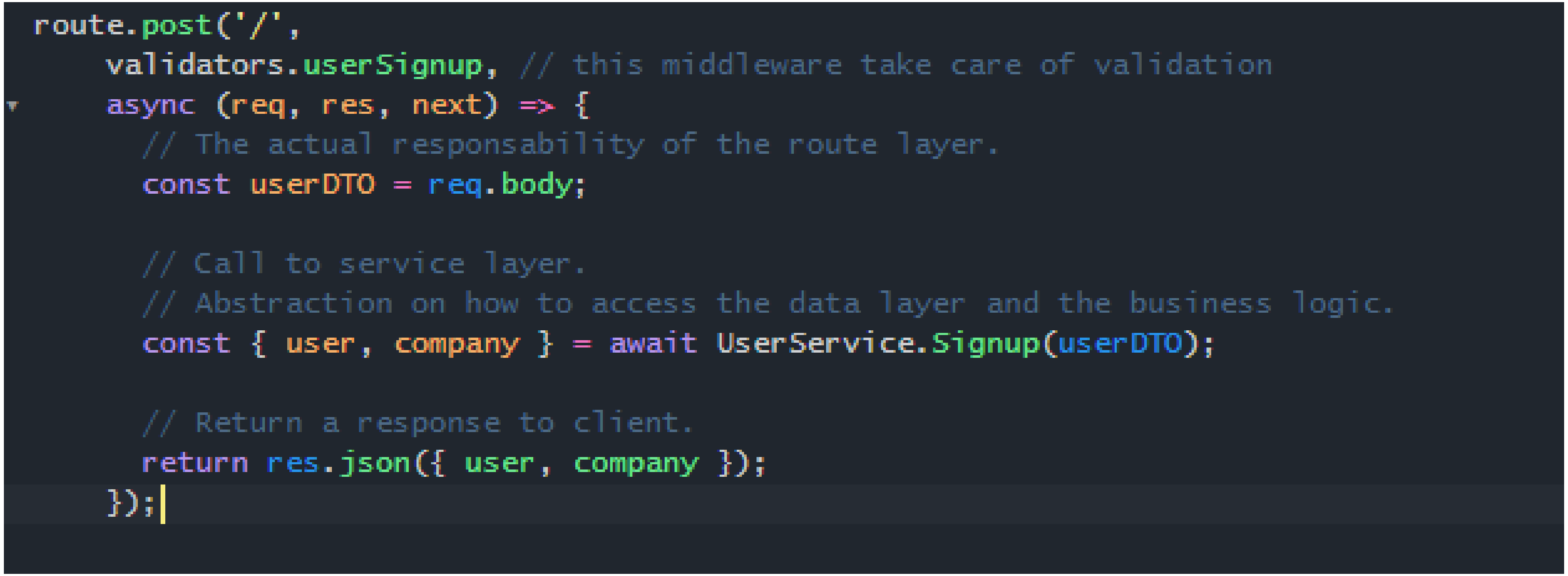


```
route.post('/', async (req, res, next) => {
    // This should be a middleware or should be handled by a library like Joi.
    const userDTO = req.body;
    const isUserValid = validators.user(userDTO)
    if(!isUserValid) {
      return res.status(400).end();
    }
    // Lot of business logic here...
    const userRecord = await UserModel.create(userDTO);
    delete userRecord.password;
    delete userRecord.salt;
    const companyRecord = await CompanyModel.create(userRecord);
    const companyDashboard = await CompanyDashboard.create(userRecord, companyRecord);
    ...whatever...
    // And here is the 'optimization' that mess up everything.
    // The response is sent to client...
    res.json({ user: userRecord, company: companyRecord });
    // But code execution continues :(
    const salaryRecord = await SalaryModel.create(userRecord, companyRecord);
    eventTracker.track('user_signup',userRecord,companyRecord,salaryRecord);
    intercom.createUser(userRecord);
    gaAnalytics.event('user_signup',userRecord);
    await EmailService.startSignupSequence(userRecord)
  });
```

Service layer

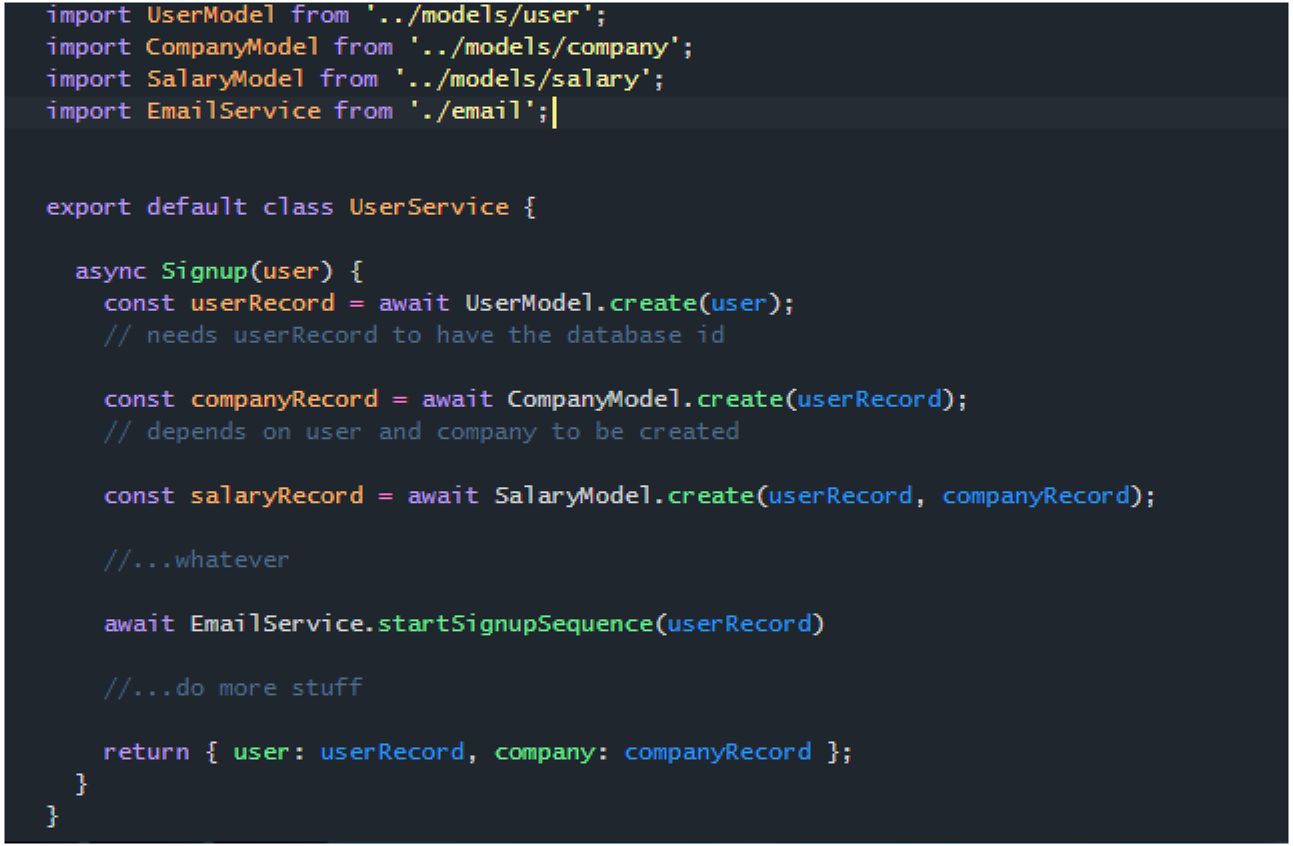
- Move your code away from the express.js router
- Don't pass the **req** or **res** object to the service layer
- Don't return anything related to the HTTP transport layer like a status code or headers from the service layer.

Controller layer



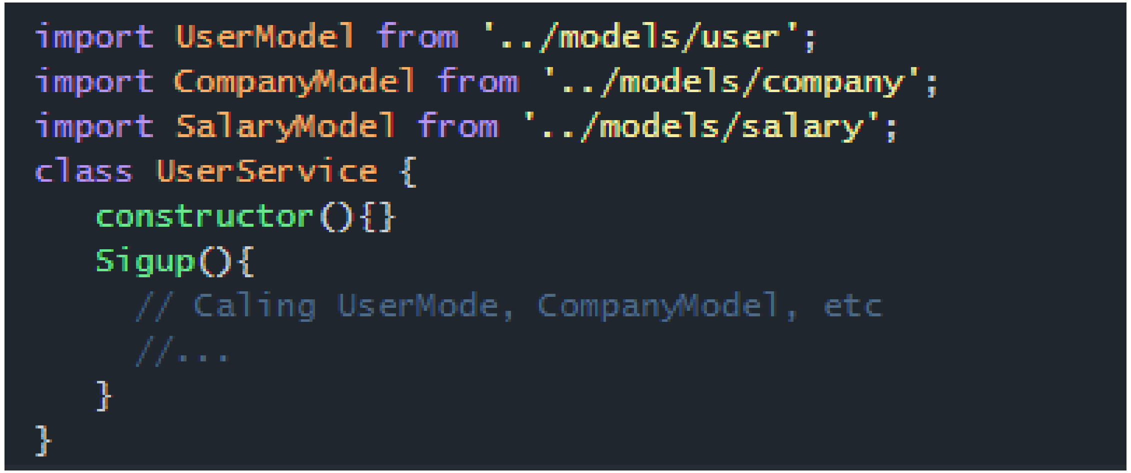
```
route.post('/',
    validators.userSignup, // this middleware take care of validation
    async (req, res, next) => {
      // The actual responsability of the route layer.
      const userDTO = req.body;

      // Call to service layer.
      // Abstraction on how to access the data layer and the business logic.
      const { user, company } = await UserService.Signup(userDTO);

      // Return a response to client.
      return res.json({ user, company });
    });
```

Service layer



```
import UserModel from '../models/user';
import CompanyModel from '../models/company';
import SalaryModel from '../models/salary';
import EmailService from './email';


export default class UserService {

  async Signup(user) {
    const userRecord = await UserModel.create(user);
    // needs userRecord to have the database id

    const companyRecord = await CompanyModel.create(userRecord);
    // depends on user and company to be created

    const salaryRecord = await SalaryModel.create(userRecord, companyRecord);

    //...whatever

    await EmailService.startSignupSequence(userRecord)

    //...do more stuff

    return { user: userRecord, company: companyRecord };
  }
}
```

Direct dependencies (Antipattern)



```
import UserModel from '../models/user';
import CompanyModel from '../models/company';
import SalaryModel from '../models/salary';
class UserService {
   constructor(){}
   Sigup(){
     // Caling UserMode, CompanyModel, etc
     //...
   }
}
```

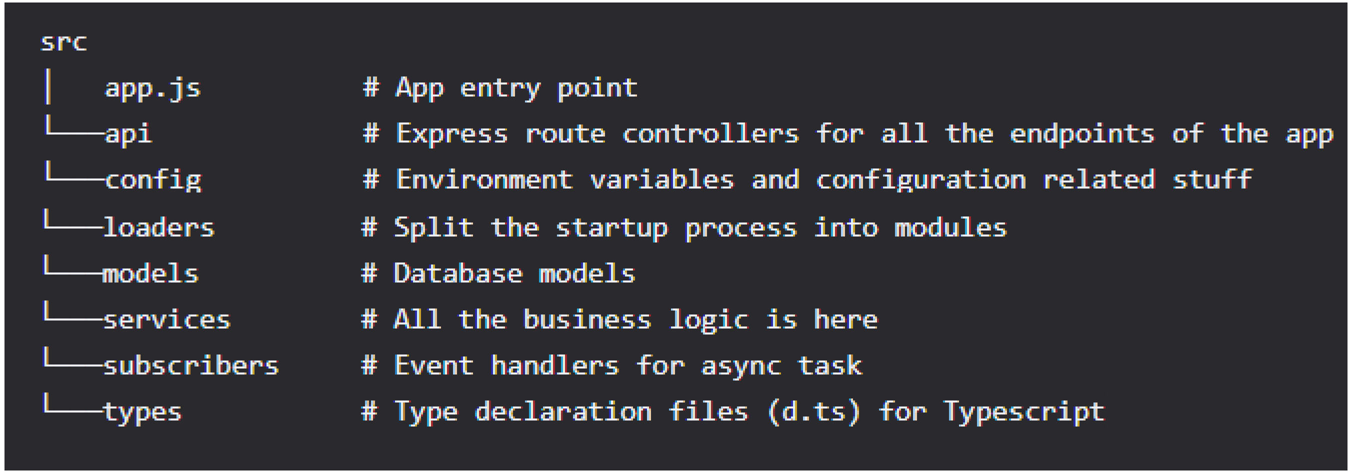
Dependency injection

```
export default class UserService {
  constructor(userModel, companyModel, salaryModel) {
    this.userModel = userModel;
    this.companyModel = companyModel;
    this.salaryModel = salaryModel;
  }
  getMyUser(userId) {
    // models available throug 'this'
    const user = this.userModel.findById(userId);
    return user;
  }
}
```

```
const userServiceInstance = new UserService(userModel, companyModel, salaryModelMock);
```

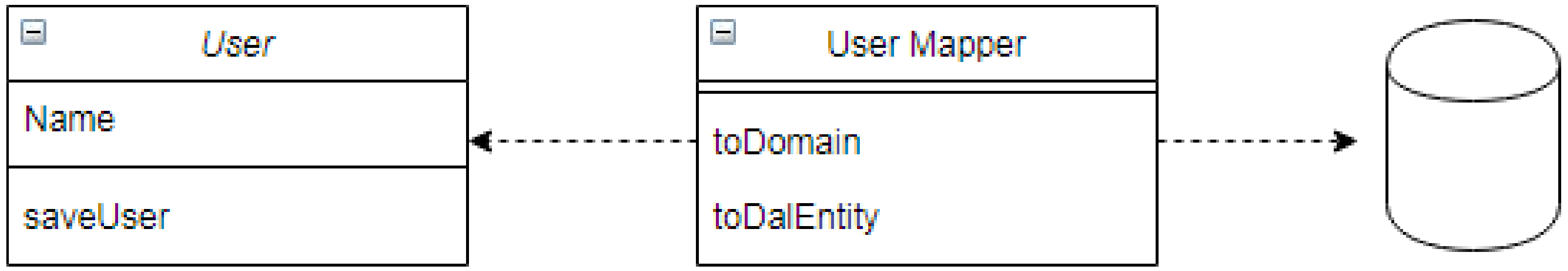
Folder structure

Keep correct folder structure

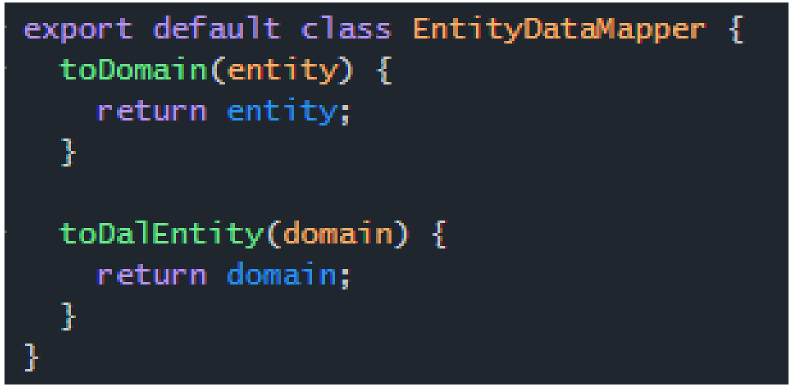


```
src
│   app.js          # App entry point
└───api             # Express route controllers for all the endpoints of the app
└───config          # Environment variables and configuration related stuff
└───loaders         # Split the startup process into modules
└───models          # Database models
└───services        # All the business logic is here
└───subscribers     # Event handlers for async task
└───types           # Type declaration files (d.ts) for Typescript
```

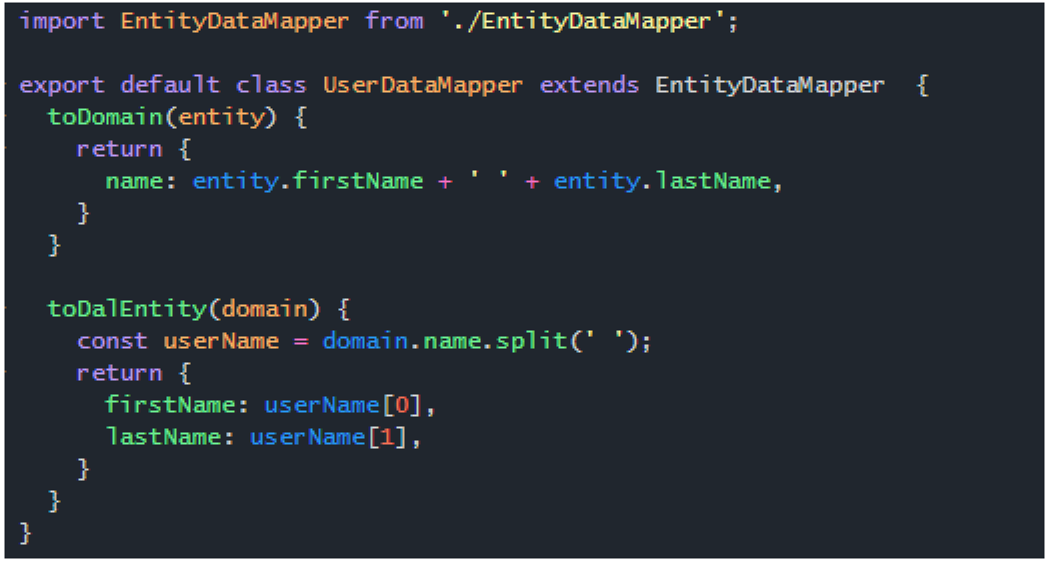
Data Mapper Pattern



Data Mapper Pattern

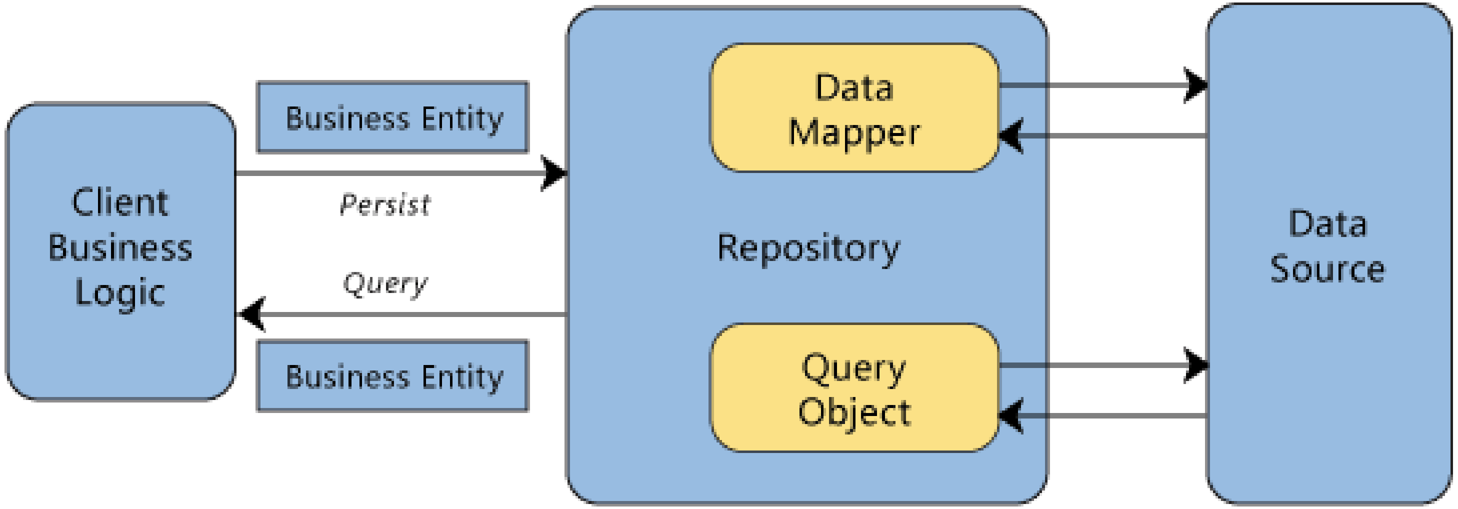


```
export default class EntityDataMapper {
  toDomain(entity) {
    return entity;
  }

  toDalEntity(domain) {
    return domain;
  }
}
```

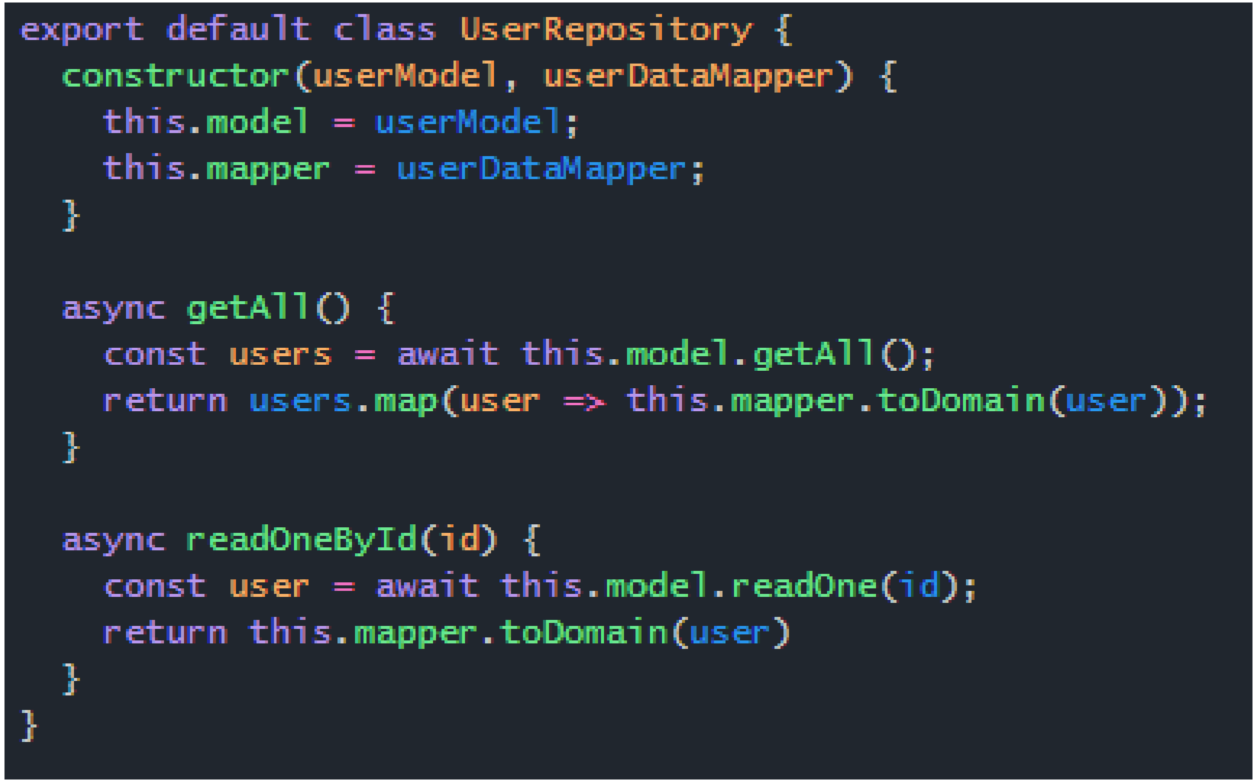


```
import EntityDataMapper from './EntityDataMapper';

export default class UserDataMapper extends EntityDataMapper  {
  toDomain(entity) {
    return {
      name: entity.firstName + ' ' + entity.lastName,
    }
  }

  toDalEntity(domain) {
    const userName = domain.name.split(' ');
    return {
      firstName: userName[0],
      lastName: userName[1],
    }
  }
}
```

Repository pattern



Repository pattern



```
export default class UserRepository {
  constructor(userModel, userDataMapper) {
    this.model = userModel;
    this.mapper = userDataMapper;
  }

  async getAll() {
    const users = await this.model.getAll();
    return users.map(user => this.mapper.toDomain(user));
  }

  async readOneById(id) {
    const user = await this.model.readOne(id);
    return this.mapper.toDomain(user)
  }
}
```

Repository pattern

Pros of usage repository

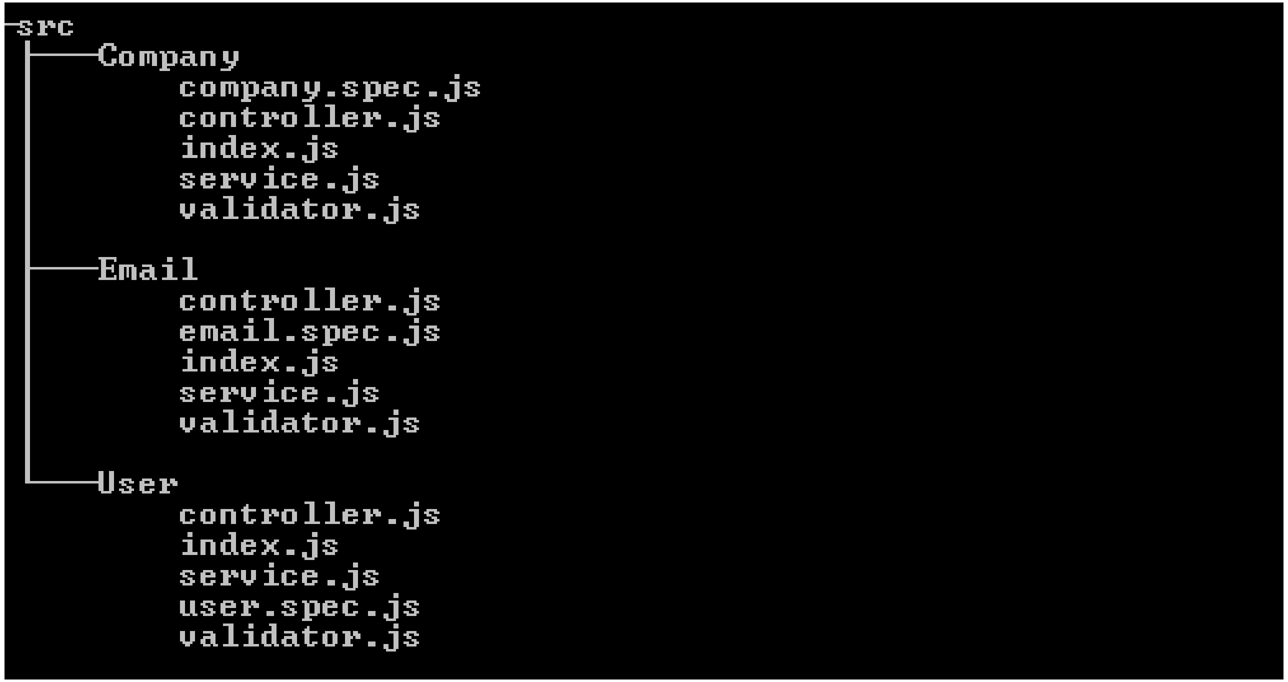
- It centralizes the data logic or Web service access logic.
- It provides a substitution point for the unit tests.
- It provides a flexible architecture that can be adapted as the overall design of - the application evolves.

Components approach

Components approach

- Too many folders should be opened to understand how application work
- Big applications structure could be too complicated
- Path is too long when you are going to include some module inside

Components approach

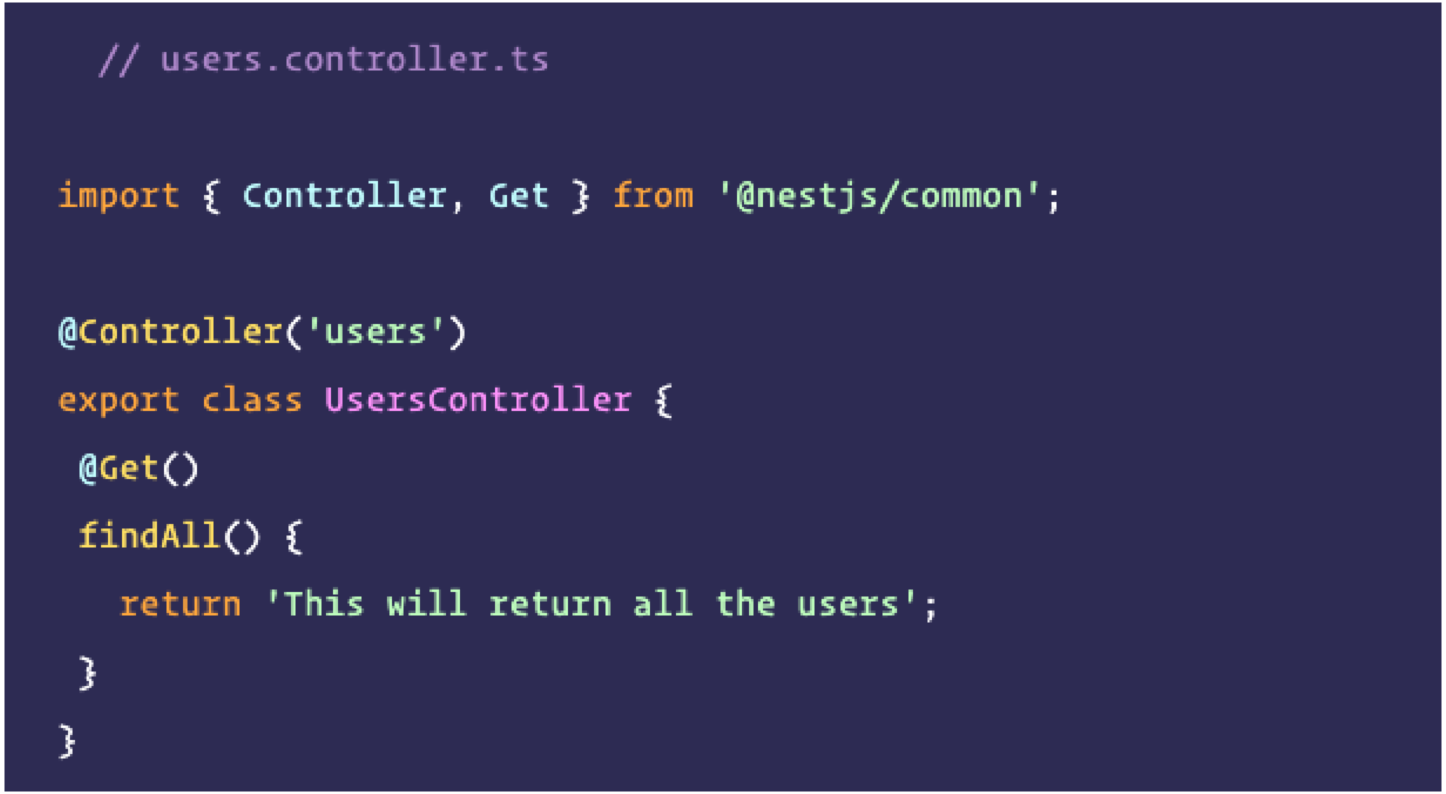


```
─src
 ├───Company
 │       company.spec.js
 │       controller.js
 │       index.js
 │       service.js
 │       validator.js
 │
 ├───Email
 │       controller.js
 │       email.spec.js
 │       index.js
 │       service.js
 │       validator.js
 │
 └───User
         controller.js
         index.js
         service.js
         user.spec.js
         validator.js
```

Components approach

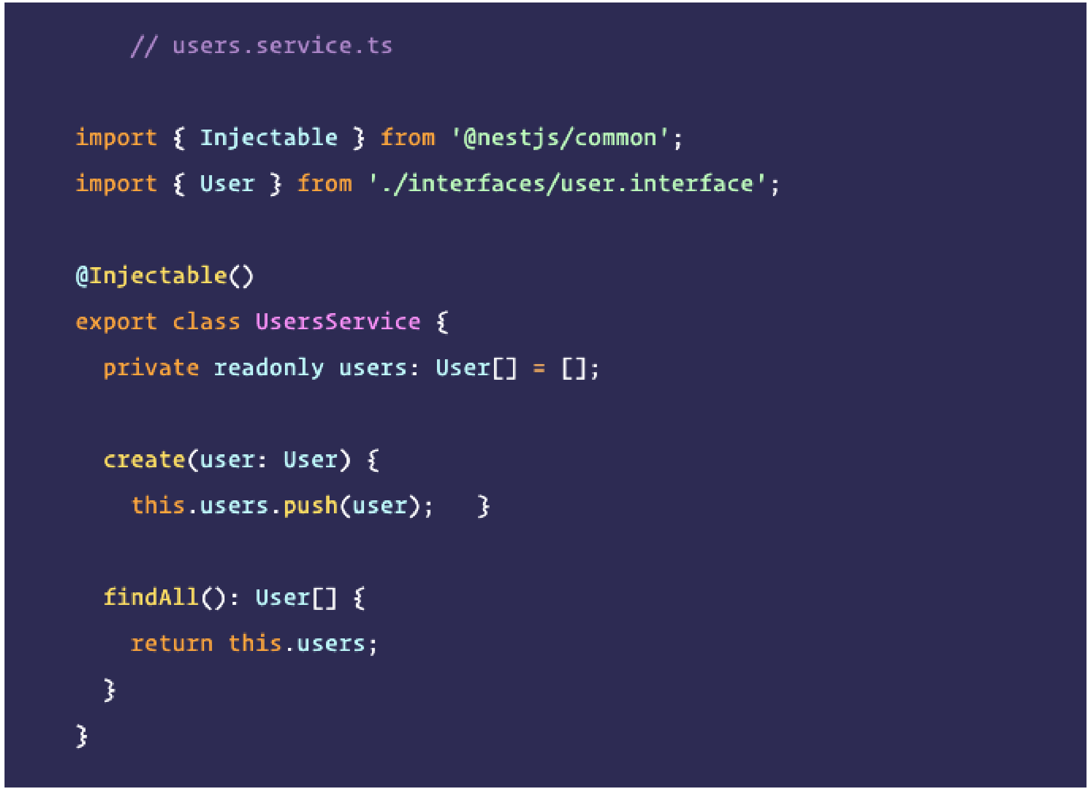
- Encapsulated logic inside of components allows to hide implementation from component’s client
- All details grouped in the same place inside the component, includes accessing to the database
- Simple and intuitive structure allows to reduce time of maintenance
- It allows to keep layered architecture inside the component
- It’s a step behind microservice architecture

Nest JS examples. Controller



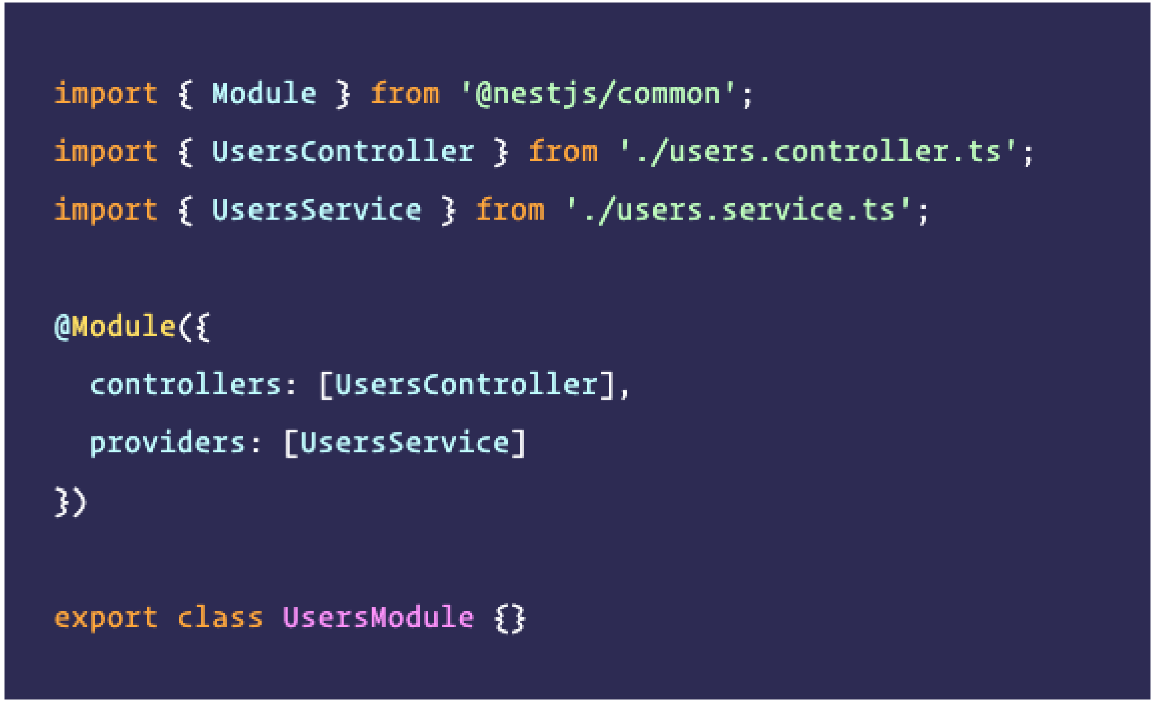
```
  // users.controller.ts

import { Controller, Get } from '@nestjs/common';

@Controller('users')
export class UsersController {
 @Get()
 findAll() {
   return 'This will return all the users';
 }
}
```

Nest JS examples. Service



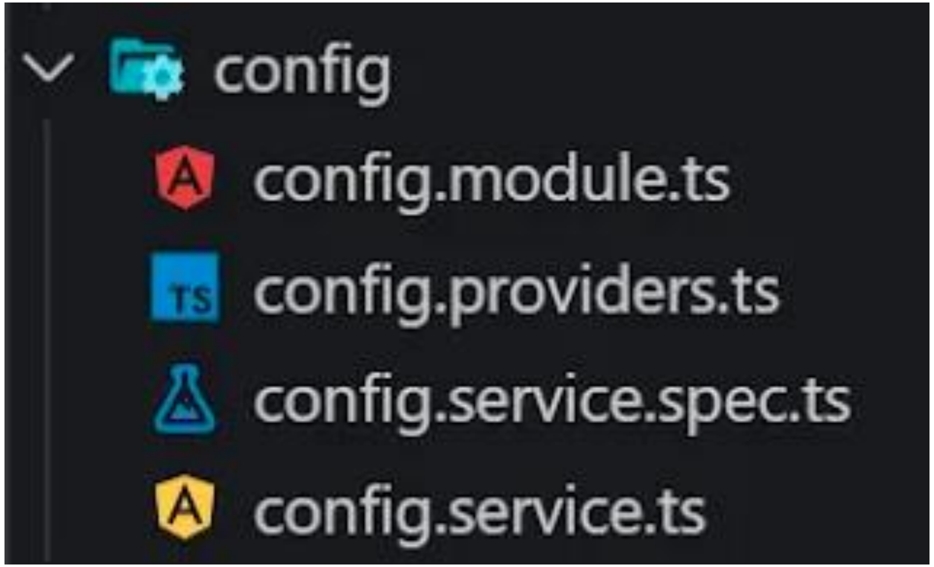
```
    // users.service.ts

import { Injectable } from '@nestjs/common';
import { User } from './interfaces/user.interface';

@Injectable()
export class UsersService {
  private readonly users: User[] = [];

  create(user: User) {
    this.users.push(user);   }

  findAll(): User[] {
    return this.users;
  }
}
```

Nest JS examples. Module



```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller.ts';
import { UsersService } from './users.service.ts';

@Module({
  controllers: [UsersController],
  providers: [UsersService]
})

export class UsersModule {}
```

Nest JS examples. Folder structure



```
config
    config.module.ts
    config.providers.ts
    config.service.spec.ts
    config.service.ts
```

Questions!

Thanks for listening!

Links in notes