معماری نرم افزاری Onion (پیاز)

مقدمه:

معماری Onion توسط Jeffrey Palermo جهت تولید برنامه هایی با قابلیت نگهداری، اطمینان و تست پذیری بالا معرفی گردید. این معماری چالش های رایج را با استفاده از 3 تا N لایه، حل میکند که هر لایه جهت ارتباط با لایه دیگر از اینترفیس ها استفاده می نماید.

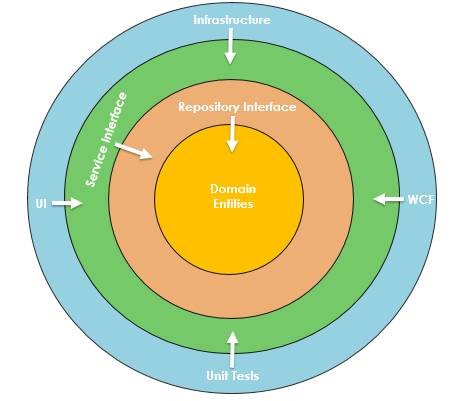
به طور خلاصه معماری onion سبب ایجاد سیستم های جامع و کاملی میشود که قابلیت نگهداری و تست پذیری بالایی دارند و تغییر تکنولوژی در این سیستم ها راحت تر است.

**اصول معماری Onion**

**اتصال لایه ها**

نحوه کارکرد این معماری اتصال لایه های سیستم با یکدیگر به گونه ایست که لایه های مختلف مانند پیاز در کنار همدیگر قرار میگیرند و لایه های بالایی توانایی تغییر و تکامل کمتری نسبت به لایه های پایینی در سیستم را دارا می باشند.

ارتباط لایه ها به یکدیگر به صورت درونی می باشد یعنی لایه بالاتر فقط وابستگی به لایه های درونی خود دارد و هیچ وابستگی به لایه بیرونی خود ندارد. (شکل 1)



شکل 1

**منطق ارتباط لایه ها**

ارتباط لایه ها با یکدیگر با استفاده از Interface ها صورت میگیرد و این امر باعث میشود که وابستگی لایه ها در سطح کد به یکدیگر به حداقل ترین حالت خود برسد.

**لایه های معماری Onion**

**Domain**

این لایه در مرکز سیستم قرار دارد و شامل Entity ها (موجودیت ها) و رفتار های این موجودیت ها می باشد و هیچ نوع وابستگی به سایر بخش های سیستم ندارد. این بخش شامل اجرای رفتارهای تعیین شده برای موجودیت ها مانند اعتبار سنجی و ... نیز می باشد.

**Repository**

این سرویس پلی بین زیر ساخت های خارجی و Domain می باشد. لایه Domain معمولا به اطلاعات و عملکرد های خارجی نیاز دارد ولی نباید وابستگی مستقیمی به آنها داشته باشد و در مقابل این بخش باید به موارد تعریف شده در لایه Domain وابستگی داشته باشد.

**Service**

این بخش وظیفه ارتباط بین دنیای خارج از سیستم مرکزی با آن را دارد و تعامل مستقیم با سرویس دهنده ها و یا UI را انجام میدهد

**مواردی که باید در نظر داشته باشید:**

* این معماری بیشتر در برنامه نویسی شی گرا مورد استفاده قرار میگیرد ولی اصول آن را میتوان در مفاهیم وسیع تری به کار برد.
* یکی از اهداف اصلی این معماری افزایش قابلیت نگهداری سیستم می باشد و برای دستیابی به این قابلیت در ابتدای راه اندازی سیستم کار زیادی نیاز هست و در حین توسعه و نگهداری نیز این اصول باید رعایت شود. پیاده سازی ویژگی ها در این معماری با توجه به عبور از چند لایه ممکن است کند تر انجام شود به همین دلیل Jeffery Palermo توصیه کرده که این معماری برای سیستم های کوچک و ساده به کار برده نشود.
* این معماری به توسعه تست محور سیستم (TDD) کمک شایانی میکند.
* استفاده از Dependency Injection در این معماری یک اصل اجتناب ناپذیر است که NestJS این مورد را به خوبی پوشش میدهد.

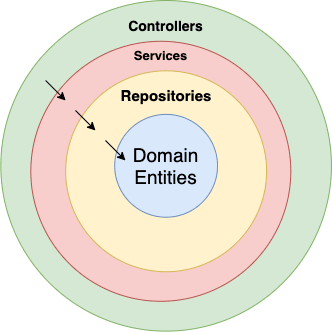
پیاده سازی این معماری در **NestJs**

NestJs یک چهارچوب توسعه نرم افزار در بستر NodeJs میباشد که امکانات مختلفی را در اختیار ما قرار میدهد و مهمترین آن ها Dependency Injection می باشد که یکی از واجبات مورد نیاز در معماری Onion می باشد.

**ایجاد یک پروژه ساده وبلاگ با استفاده از معماری Onion در NestJs**

در ادامه به پیاده سازی سیستم ساده وبلاگ با استفاده از NestJS می پردازیم و پیشفرض در این است، که خواننده اطلاعات کافی در مورد NestJS را داشته باشد.

**لایه های پروژه در NestJs**



**Domain**

هسته برنامه ما می باشد و جهت توسعه نرم افزار در مرحله اول تمام توجه ما به این لایه می باشد و بقیه لایه ها با توجه به این لایه پیاده سای می شوند.

در پروژه تعریف شده ما این لایه شامل مقالات وبلاگ می باشد بنابر این در ابتده Interface آن را تعریف میکنیم:

export interface IArticle {

  id: number;

  body: string;

  createdAt: Date;

  updatedAt: Date;

  title: string;

}

**Repository**

این لایه در تعامل با لایه Domain می باشد و کار دریافت یا حذف Entity ها را به عده دارد و با پایگاه داده در تعامل هست و میتوان از آن برای استفاده از هر پایگاه داده ای استفاده کرد.

در پروژه تعریف شده یک Repository برای SQlite تعریف شده و به سادگی میتوان آن را باتوجه به معماری پیاده سازی شده جایگزین سایر پایگاه های داده و یا حتی سرویس های ذخیره سازی ابری کرد.

در ادامه کد مربوط به Interface این لایه آمده است:

export interface class ArticleRepository {

  abstract get(id: number): Promise<IArticle>;

  abstract delete(id: number): Promise<boolean>;

  abstract save(input: Partial<IArticle>): Promise<IArticle>;

  abstract update(input: Partial<IArticle>): Promise<boolean>;

  abstract getList(

    skip: number,

    limit: number,

  ): Promise<{ result: IArticle[]; total: number }>;

}

در لایه Reposytory معمولا توابع یکسانی برای تمامی Entity ها پیاده سازی میشود و پیشنهاد میشود از قابلیت Generics در TypeScript برای کد نویسی کمتر استفاده شود. پس کد بالا را به صورت بهینه و یک کلاس Abstract به حالت زیر تغییر میدهیم:

export abstract class BaseRepository<T> {

  abstract get(id: number): Promise<T>;

  abstract delete(id: number): Promise<boolean>;

  abstract save(input: Partial<T>): Promise<T>;

  abstract update(input: Partial<T>): Promise<boolean>;

  abstract getList(

    skip: number,

    limit: number,

  ): Promise<{ result: T[]; total: number }>;

}

**Service**

این لایه وظیفه دارد تا اطلاعات مورد نیاز را از لایه Repository تهیه کرده و به لایه بالاتر (Controller) ارجاع دهد.

export interface IArticleService {

  create(title: string, body: string): Promise<IArticle>;

  update(id: number, title?: string, body?: string): Promise<boolean>;

  get(id: number): Promise<IArticle>;

  getList(

    skip: number,

    limit: number,

  ): Promise<{ result: IArticle[]; total: number }>;

  delete(id: number): Promise<boolean>;

}

**Controller**

نهایتا لایه های درونی در تعامل با یکدیگر سرویسی را ارایه میکنند که ما در Controller که بیرونی ترین لایه میباشد از آنها استفاده میکنیم. این لایه وظیفه ارایه سرویس به لایه دیگری را ندارد بنابر این نیازی به ایجاد Interface برای آن نیست و میتوانیم آن را مستقیما کد نویسی کنیم.

@Controller('article')

export class ArticleController {

  constructor(

    @Inject('ARTICLE\_SERVICE\_TOKEN')

    protected service: IArticleService,

  ) {}

  @UseInterceptors(ClassSerializerInterceptor)

  @Get(':id')

  getArticle(@Param() id: number): Promise<Article> {

    return this.service.get(id);

  }

  @Put()

  createArticle(@Body() article: CreateArticleDto) {

    return this.service.create(article.title, article.body);

  }

  @Post()

  updateArticle(@Body() article: UpdateArticleDto) {

    return this.service.update(article.id, article.title, article.body);

  }

  @Delete(':id')

  deleteArticle(@Param() id: number) {

    return this.deleteArticle(id);

  }

  @UseInterceptors(ClassSerializerInterceptor)

  @Get('list/:skip/:limit')

  getListOfArticles(

    @Param('skip') skip: number,

    @Param('limit') limit: number,

  ) {

    return this.service.getList(skip ?? 0, limit ?? 1);

  }

}

**میتوان در لایه بیرونی رفتار های مختلفی را انتظار داشت مثلا به موازات کنترلری که برای سرویس دهی با پروتکل Rest نوشته شده میتوان کنترلر دیگری نوشت که با GRPC یا حتی تحت کنسول نیز اراییه خدمات کند.**

پس از تعریف Interface های لایه های مختلف حالا باید کد آن ها را پیاده سازی نماییم.

نکته: در پروژه تکمیل شده با توجه به این که از TypeOrm استفاده شده است و این پکیج خود توانمایی ایجاد Repository را دارد نیازی به کدنویسی آن نبوده ولی برای درک بهتر این لایه نیز کد نویسی شده است.

لینک پروژه: <https://github.com/sajjad-ser/nestjs-onion>