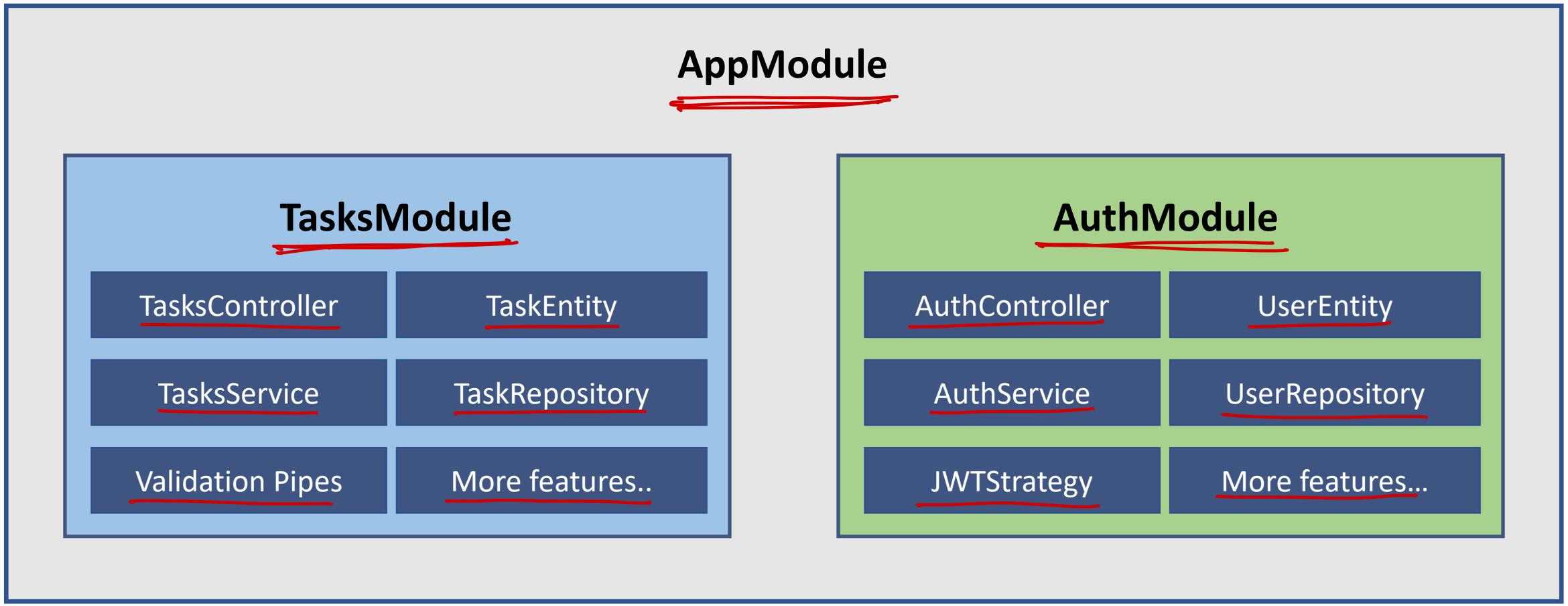
Objectives

API Endpoints

Endpoint	Method	Description
/task	GET	Get all tasks including filters
/task/:id	GET	Get a task details
/task	POST	Create a task
/task/:id	DELETE	Delete a task
/task/:id/status	PATCH	Update task status
/auth/signup	POST	User sign up
/auth/signin	POST	User sign in

What are we going to achieve ?



Objectives

- Develop production-ready REST APIs
- CRUD operations
- Validation
- Error handling
- Data Transfer Objects (DTO)
- System Modularity
- Back end development best practices
- Configuration Management
- Logging
- Security best practices

NestJS Objectives

- NestJS Modules
- NestJS Controllers
- NestJS Services and Providers
- Controller-to-Service communication
- Validation using NestJS Pipes

Persistence Objectives

- Connecting the application to a database (RDBMS)
- Working with relations databases
- Using TypeORM
- Writing simple and complex queries using QueryBuilder
- Performance improvement

Authentication/Authorization objectives

- Signing up and Signing in
- Authentication and authorization
- Protected resources
- Using JWT tokens
- Ownership of tasks by users
- Password hashing

Deployment Objectives

- Final touches for production use
- Deploying NestJS to AWS
- Deploying front-end application to S3
- Connecting front-end and back-end

Client Server

Client Server Architecture

- The Client-server model is a distributed application structure that partitions task or workload between the providers of a resource or service, called servers, and service requesters called clients
- In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and deliver the data packets requested back to the client
- Clients do not share any of their resources
- Examples of Client-Server Model are Email, World Wide Web, etc

Server

* SERVER
  - program which serves requests
  - types
    - web server :
      - used to serve all web requests
      - used to provide http & https protocols
      - eg.
        - apache
        - IIS
        - nginx
    - file server
      - used to share the files with users
      - eg.
        - NFS - network file system
    - database server
      - used to provide data persistence
      - eg.
        - RDBMS : MySQL, Oracle, SQL Server
        - NO-SQL : MongoDB, Cassandra, HBase
    - application server
      - eg.
        - apache tomcat, JBoss

* URL
  - Uniform Resource Locator
  - used to identify resource uniquely
  - components

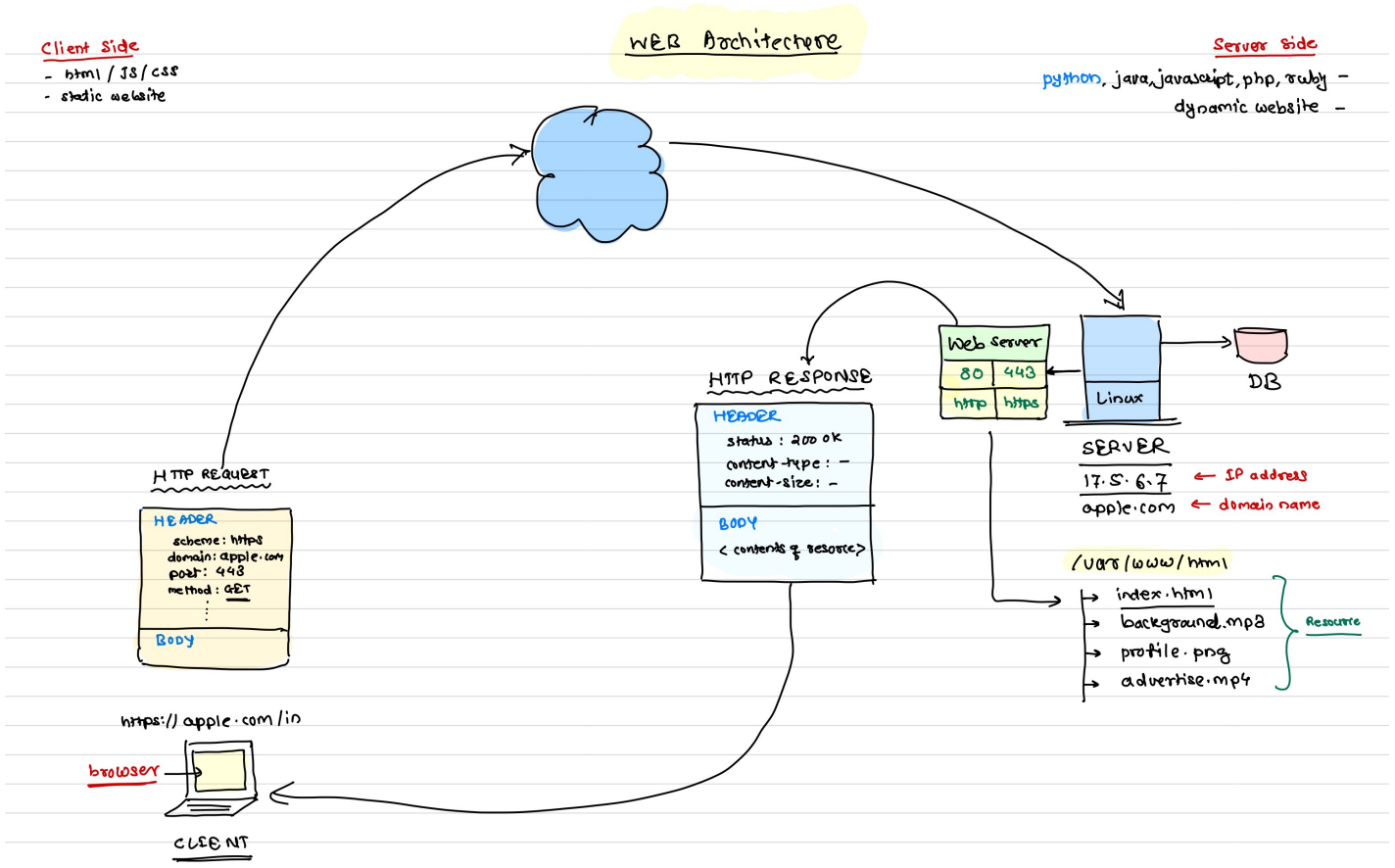
https://google.com/query?q=iphone#top

  - scheme: https
  - IP address or domain name : google.com
  - port no : https ⇒ 443
  - path or file name : query
  - query string : q=iphone
  - hash component : #top

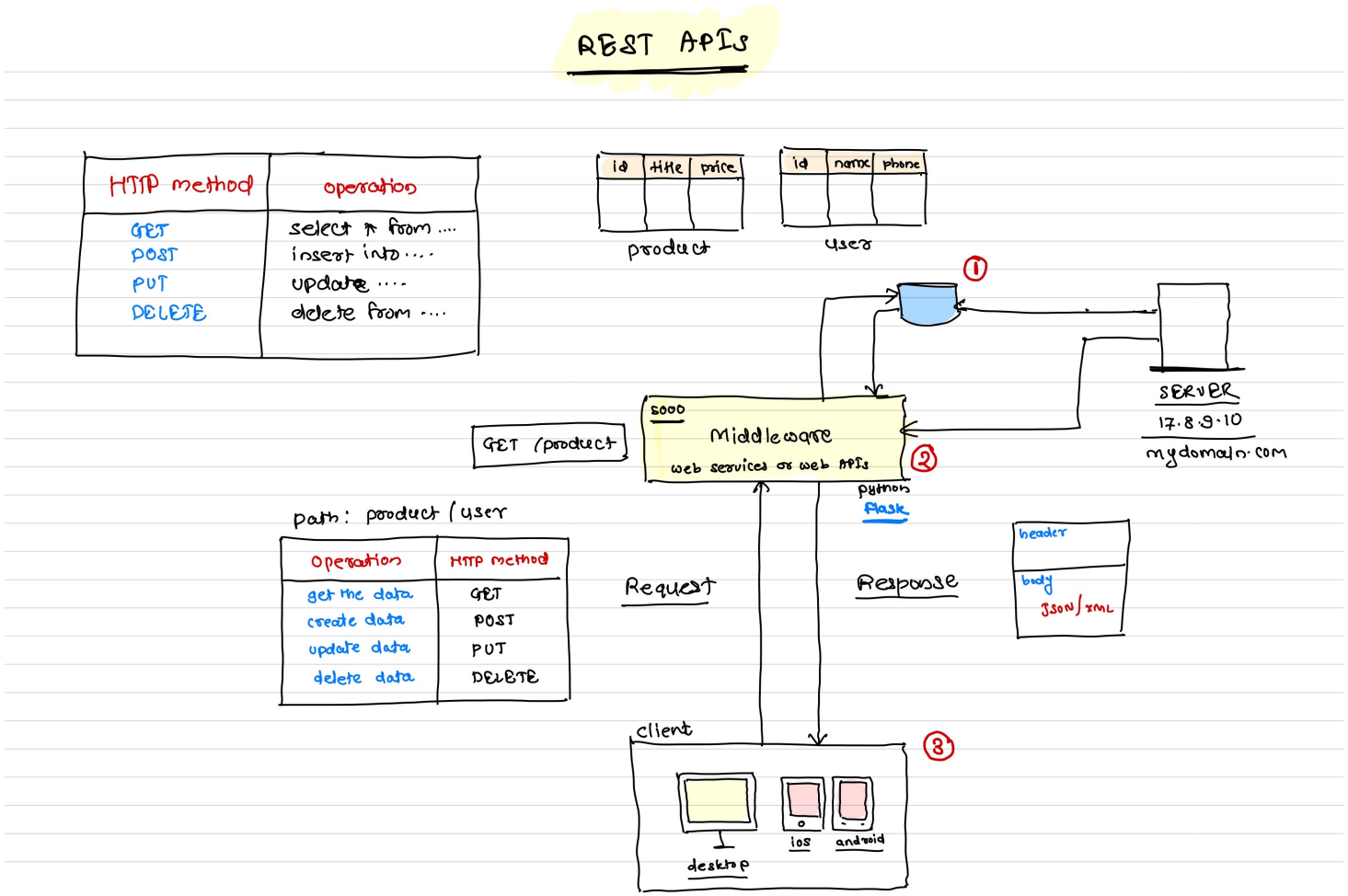
---

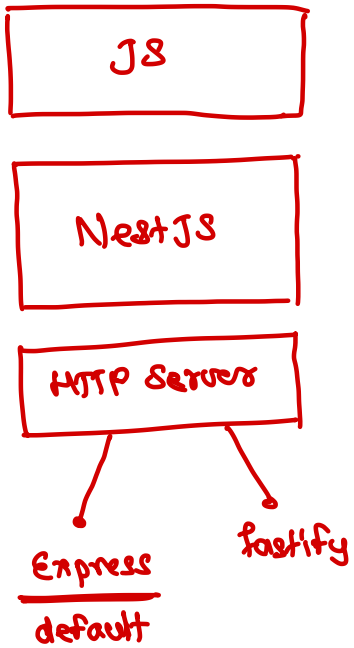
* Status code
  * 1XX : informational / debugging messages (101 / 102)
  * 2XX : success (200)
  * 3XX : migration / redirection (301 / 302)
  * 4XX : client error (404)
  * 5XX : server error (500)

Request-Response Pattern



REST Design Pattern





NestJS

Introduction

- Nest (NestJS) is a framework for building efficient, scalable NodeJs server-side applications
- It uses progressive JavaScript, is built with and fully supports TypScript (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming).
- Under the hood, Nest makes use of robust HTTP Server frameworks like Express (the default) and optionally can be configured to use Fastify as well!
- Nest provides a level of abstraction above these common Node.js frameworks (Express/Fastify), but also exposes their APIs directly to the developer
- This gives developers the freedom to use the myriad of third-party modules which are available for the underlying platform

Philosophy

- The rise of popular web technologies such as Angular, React and Vue has massively improved the development experience
- However, while nodejs for server side has plenty of libraries, none of them efficiently solves the main problem of architecture
- NestJS provides an out-of-the-box application architecture which allows developers and teams to create highly testable, scalable, loosely coupled and easy-to-maintain applications

Features

- Typescript support
- Nest CLI to initialize and develop your applications
- Excellent Documentation
- Easy unit testing just like Angular
- Highly scalable
- Built for large scale enterprise applications
- Open-source (MIT license)

Platform

- Nest aims to be a platform-agnostic framework
- Platform independence makes it possible to create reusable logical parts that developers can take advantage of across several different types of applications
- Technically, Nest is able to work with any Node HTTP framework once an adapter is created
- There are two HTTP platforms supported out-of-the-box
  - platform-express
    - Express is a well-known minimalist web framework for node
    - It's a battle tested, production-ready library with lots of resources implemented by the community
    - The @nestjs/platform-express package is used by default
  - platform-fastify
    - Fastify is a high performance and low overhead framework highly focused on providing maximum efficiency and speed

Nest CLI

- The NestJS CLI is a command-line interface tool that helps you to initialize and develop your applications
- It has many benefits – from scaffolding a project to building a well structured application
- It is possible to generate a project and generate schematics in a project using CLI
- The leaves almost no room for mistakes in terms of project structure
- Using CLI the generate projects and schematics can save you a lot of time writing boilerplate code

Installation

- To get started, you can either scaffold the project with the **Nest CLI**

```
$ npm install -g @nestjs/cli
$ nest new project-name
```

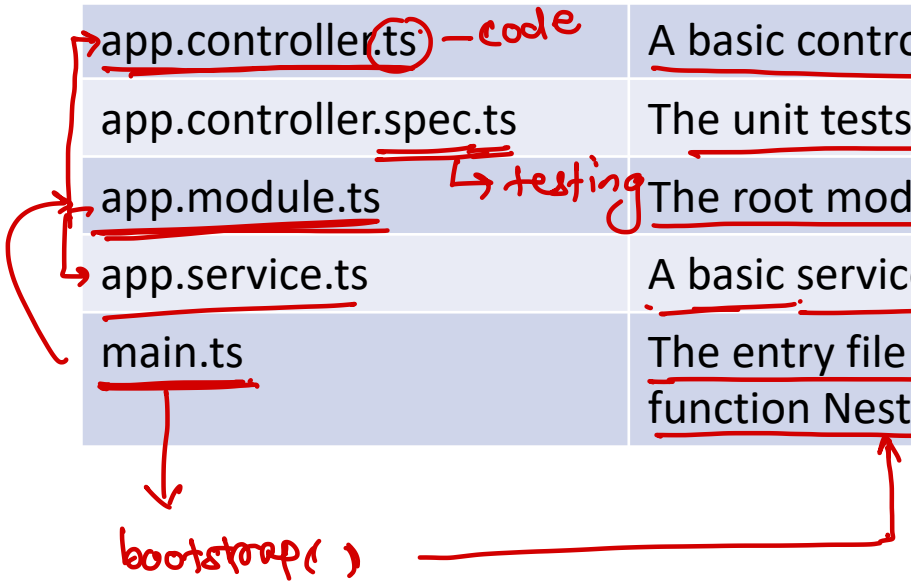
- Or clone a starter project

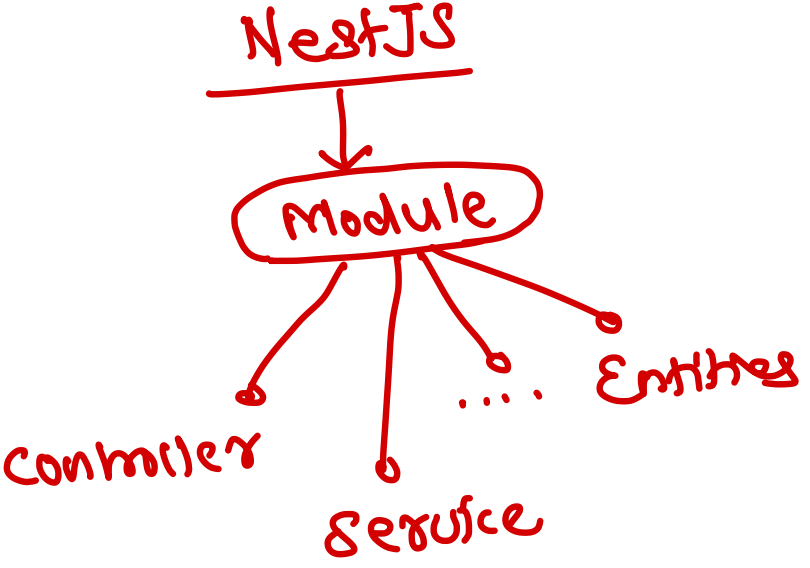
```
$ git clone https://github.com/nestjs/typescript-starter.git project
$ cd project
$ npm install
$ npm run start
```

Project Contents

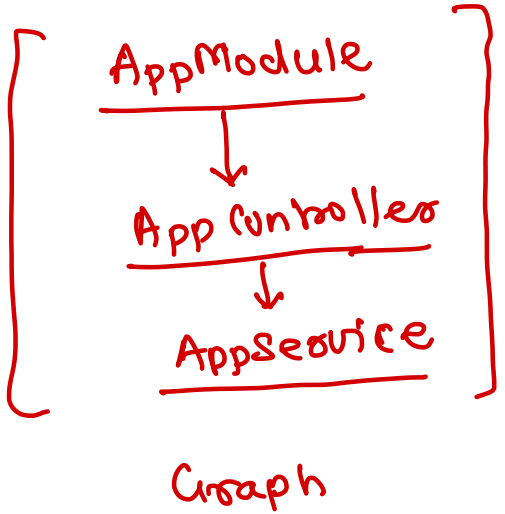
- When you create Nest project you will get a project will following structure

File	Usage
app.controller.ts	A basic controller with a single route
app.controller.spec.ts	The unit tests for the controller.
app.module.ts	The root module of the application.
app.service.ts	A basic service with a single method.
main.ts	The entry file of the application which uses the core function NestFactory to create a Nest application instance.




NestJS
Module
Controller
Service
...
Entities

Module



NestJS Module

- A module is a class annotated with a @Module() decorator
- The @Module() decorator provides metadata that Nest makes use of to organize the application structure
- Each application has at least one module – the root module which is the starting point for the application
- The root module is the starting point Nest uses to build the application graph - the internal data structure Nest uses to resolve module and provider relationships and dependencies
- Modules are an effective way to organize components by a closely related set of capabilities
- It is a good practice to have a folder per module, containing the modules' components
- Modules are singletons, therefore a module can be imported by multiple other modules

Creating a module

- A module is defined by annotating a class with the @Module decorator
- The decorator provides metadata to organize the application structure
- The @Module() decorator takes a single object whose properties describe the module

Parameter	Usage
providers	the providers that will be instantiated by the Nest injector and that may be shared at least across this module
controllers	the set of controllers defined in this module which have to be instantiated
imports	the list of imported modules that export the providers which are required in this module
exports	the subset of providers that are provided by this module and should be available in other modules which import this module

Module

```
import { Module } from '@nestjs/common'
import { TasksModule } from './tasks/tasks.module'

@Module({
  imports: [TasksModule],
  controllers: [],
  providers: [],
  exports: [],
})
export class AppModule {}
```

decorator

metadata

Controllers

NestJS Controllers

- Controllers are responsible for handling incoming **requests** and returning **responses** to the client
- The **routing** mechanism controls which controller receives which requests
- Controllers are bound to a specific path (for example, /task for task resource)
- Contain handlers, which handle endpoints and request methods (GET, POST, DELETE etc)
- Can take advantage of dependency injection to consume provides within the same module

Handlers

- Handlers are simple methods withing the controller class, decorated with decorators such as
  - @Get
  - @Post
  - @Delete
  - @Put
  - @Patch
- Handler decorators may accept parameters

Controller

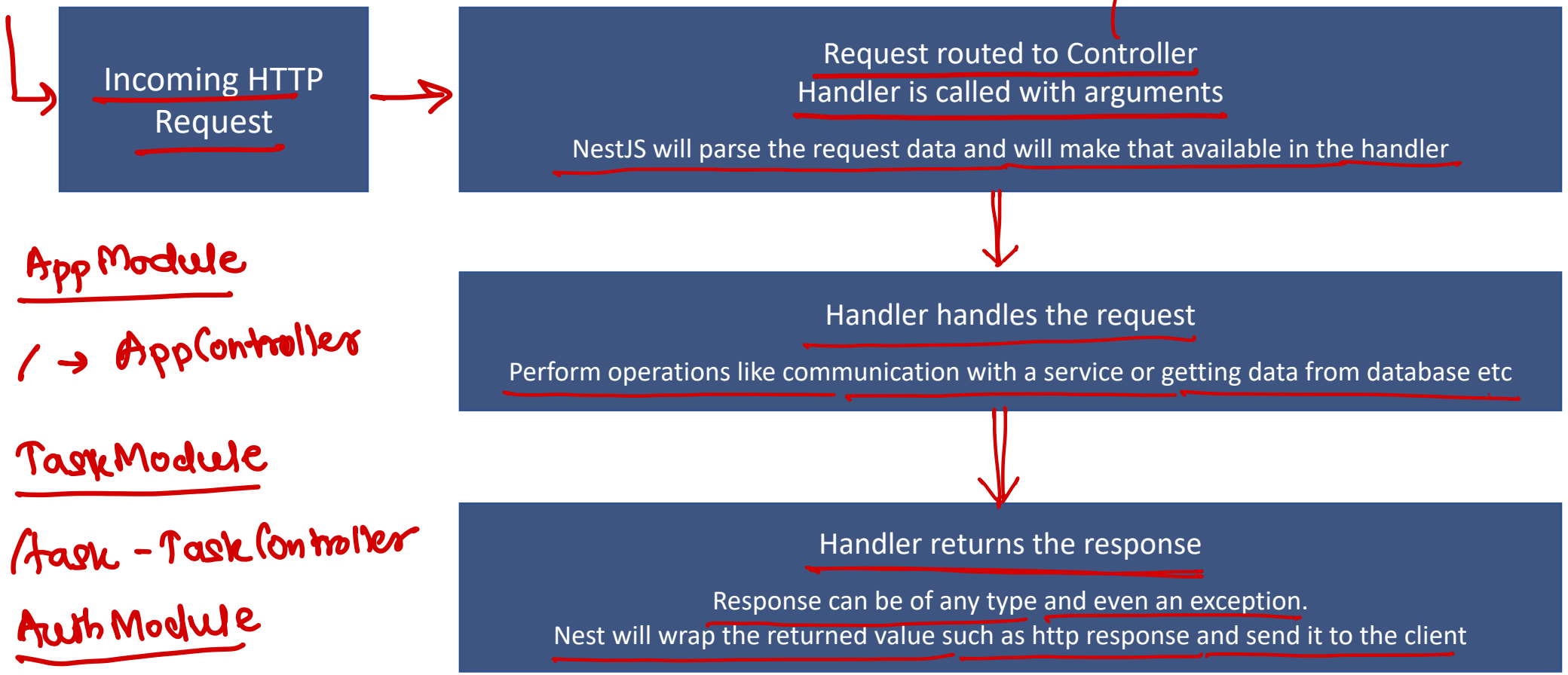
```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) { }

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

What happens behind the scene ?

/task



AppModule

/ → AppController

TaskModule

/task - TaskController

AuthModule

/auth → AuthModule

Providers

NestJS Providers

(Dependency Injection)

- Providers are a fundamental concept in Nest
- Many of the basic Nest classes may be treated as a provider services, repositories, factories, helpers
- The main idea of a provider is that it can **inject** dependencies; this means objects can create various relationships with each other, and the function of "wiring up" instances of objects can largely be delegated to the Nest runtime system
- A provider is simply a class annotated with an @Injectable() decorator.
- Providers must be provided to the module for them to be usable
- Can be exported from a module

NestJS Service

- Service is defined as provider
- Not all providers are services
- It is common concept within development and not exclusive to the NestJS
- Services are singleton objects when wrapped with @Injectable() decorator, which means the same instance will be shared across the application
- Service is a main source for business logic, e.g. service will be called from controller to validate the data, create an item in the database and return a response

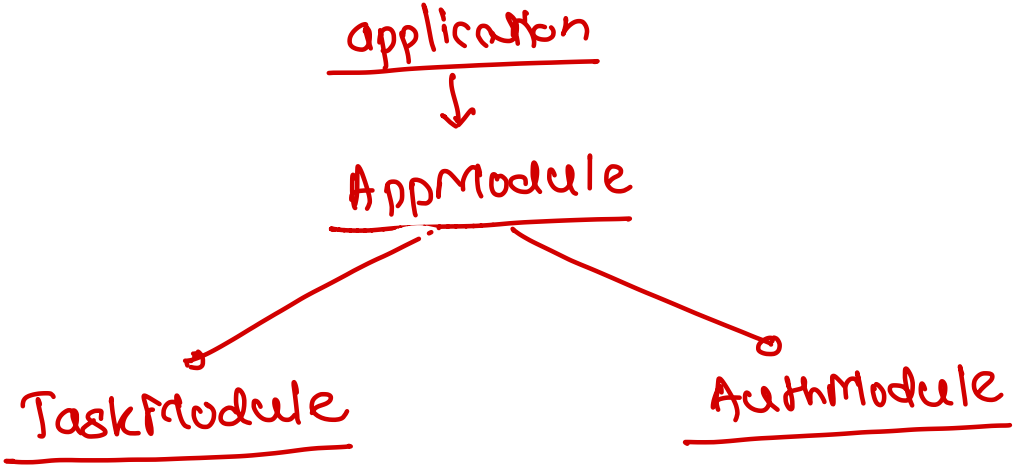
Dependency Injection in NestJS

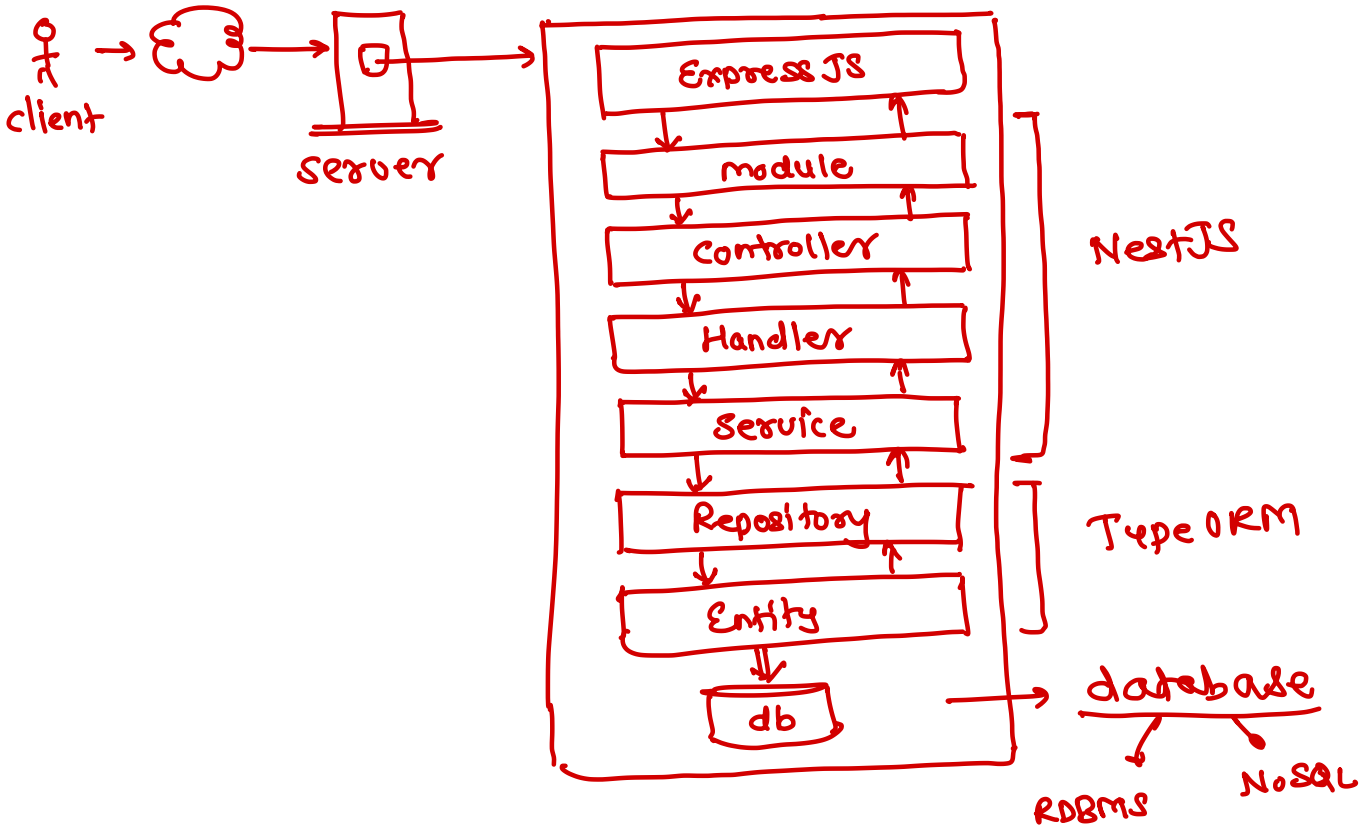
- Any component within NestJS ecosystem can inject a provider that is decorated with @Injectable()
- We define the dependencies in the constructor of the class
- NestJS takes care of injecting required object for us and it will be then available as a class property

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) { }

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```


application
AppModule
TaskModule
AuthModule


client
server
ExpressJS
module
controller
Handler
Service
Repository
Entity
db
NestJS
TypeORM
database
RDBMS
NoSQL