tsconfig.json

* define things like which files should be “compiled, which directory to compile them to, and which version of JavaScript to emit.

Table

Description automatically generated

## SOLID Principle

**Single Responsibility**

* a class or method should be focused on doing one thing and one thing only.

Common violation

* presentation logic
* file/database read/write
  + when your class or method becomes responsible for handling the persistence layers specifics.

Example

* before

Text

Description automatically generated

* after

Text

Description automatically generated

**Open/Close Principle**

* open to extension
* close to modification
* don’t add stuff to completed class
  + Solution (new functionality): add functionality by extending

**Liskov Substitution Principle**

* A subclass should be used whenever its base class can be used, without altering the functionality in a blocking way.
* Think twice when using inheritance

**The Interface Segregation Principle**

* A class should not depend on methods that it does not need to implement
* Splitting interface into smaller ones

How to recognize the issue

* Don’t know how to implement an interface method
* The interface method does not belong in the class

How to avoid

* Segregate interfaces (split up)
* Only implement methods it needs

**The dependency Inversion Principle**

* Classes and modules should depend on abstractions instead of concrete implementations
* In typescript, it boils down to one concept 🡪 interface
  + Abstractions == interfaces

Loose coupling

**Design Patterns**

* Don’t reinvent the wheel
* Every time you are struggling to come up with an architectural breakthrough, there is probably a design pattern

**Creational**

* All about instantiating objects
* How objects are instantiated
* How many of them we are allowed to create

Singleton pattern

* Have only a single instance of a specific class throughout the entire application
* Benefits
  + Shared state
    - More common in mobile apps or console apps; not common in web apps and APIs
  + Avoid long initializations
    - Ex. A class that has very complex logic in its constructor
  + Cross-class communication
  + Perfectly represents unique items
* Key
  + Use class.\_instance = this

Factory Pattern

* A combination of the single responsibility and open/close principles combined

Loose Coupling

* All about interfaces
* Software parts that communicate with each other have little to no knowledge of each other’s actual implementation
* What to use
  + Single responsibility principle
  + Separation of concerns
  + Factory pattern/object pool
  + Dependency Injection
* Benefits
  + Easy to work with large projects
  + Swap implementation
  + Testability
  + Components grow independently

The Object Pool

* A pool of pre-initialized objects whose initialization is heavyweight
* Every time we need such an object we take one from the pool
* Once we are done with it, we return it back to the pool
* Is it depreciated

**Structural**

* About the apps’ structure
* Situation for structural patterns
  + Have complex classes or objects that are composed from smaller objects
  + Structural patterns offer most efficient solutions to this type of situation

**Behavioural**

* Ways applications flow and how objects communicate with each other

**Dependency Injection**

* a technique in which an object receives other objects that it depends on, called dependencies.
* Instead of the client specifying which service it will use, the injector tells the client what service to use. The 'injection' refers to the passing of a dependency (a service) into the client that uses it.

Ts Specific

* New is not allowed
  + Be careful to not abuse New

Traditional vs dependency injection flow of control

* Traditional

Text

Description automatically generated

* Inverted flow of control

Text

Description automatically generated

Dependency injection container

* an object that knows how to instantiate and configure objects.
* Every time you create a new component in your app, you register it with the container

**Decorator**

* An experimental TypeScript feature
* Widely used Angular

How to use

* Have to enable 2 fields in tsconfig.json file

Types

* Class, Method, Property, Accessors

Method Decorator

* Arguments:
  + The class constructor
  + The method’s name
  + The property descriptor
* Hook
  + A decorator that will allow us to call certain methods before the method that the decorator decorates
  + Allows use not to extend the class to include methods that are called before the actual method
  + The decorator factory that binds hook to the bar method only runs once

Constructor decorator

* Only accepts only one argument
  + Constructor function
* Best practice
  + Always use decorator factories

Dependency **Injection**

* all about removing dependencies from your code.

InversifyJS

* Goal: write code that adheres to the dependency inversion principle
* Dependency inversion principle
  + High-level modules should not depend on low-level modules. Both should depend on abstractions
  + Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.
* String indexes issue solution
  + Export a global object a matching between the interface type and a symbol

DependencyScopes

* Singleton
* Transient: create a new instance
* Request

**Structural design pattern**

Decorator pattern

* Respects the open/closed principle
* Dynamically adding responsibilities to an object at runtime
* Decorator models the features that can be added to the based class
* Goal
  + Combine a single concrete class with one or more decorators
  + Ex

Diagram

Description automatically generated

* Using an experimental decorator is not necessary

**Adapter Pattern**

* Solve the struggles of extended use of third-party libraries
* Converts an interface of a class to one expected by the consumer

Repository

* Something like an interface standing between the persistence layer and the rest of your application

Persistence layer

* The filesystem of the server that serving the application

**Façade Pattern**

* The goal: simplify an interface
* Is the application of the principle of least interest
  + AKA: Law of Demeter: every component should have little knowledge of how other components work and only communicate with a few specific close friends
* Ex. Use external APIs
  + Social media and authentication services
  + Software provided as a service
  + Enterprise APIs
* Flow: application – service RESTful API

Diagram

Description automatically generated

* + Ex: MailChimp

before

Diagram

Description automatically generated

* + After

Diagram

Description automatically generated

* + - The façade can extract the logic even further, acting as a bridge between our app and the service.
  + Façade layer

Diagram

Description automatically generated

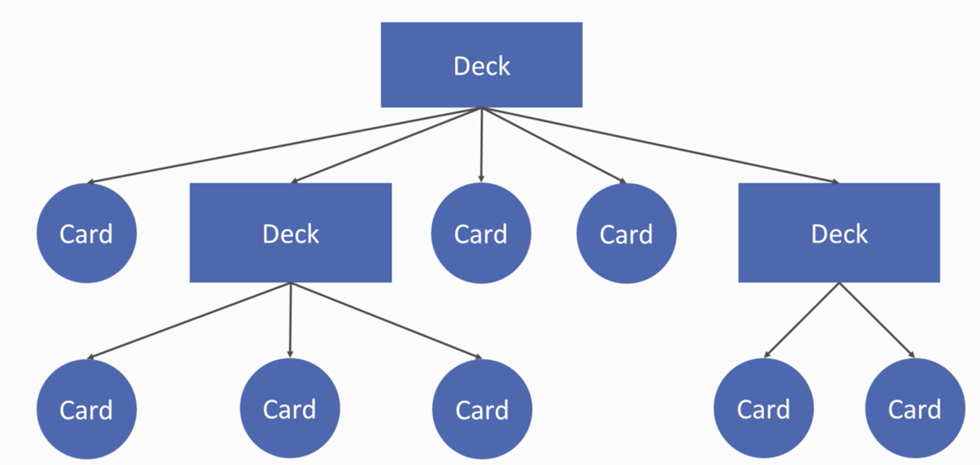
Composite Pattern

* describes a group of objects that are treated the same way as a single instance of the same type of object.
* Implementing the composite pattern lets clients treat individual objects and compositions uniformly.
* Ex

Diagram

Description automatically generated

* Used to create part-whole collections in the form of tree-like structures that can contain both individual items and collections as well
  + Part-whole collection
* Use case
  + you want to represent part-whole hierarchies of objects.
  + you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.



**Behavioral patterns**

Strategy pattern

* Choose algorithm at runtime
* Code receives run-time instructions as to which in a family of algorithms to use.
* Ex. Error handler
  + Display & store

Diagram

Description automatically generated

Observer pattern

* an object, named the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
* Use case
  + mainly used for implementing distributed event handling systems,

A picture containing text, electronics

Description automatically generated

Command pattern

* encapsulate all information needed to perform an action or trigger an event later.
* Don’t want to know when it finishes and only want to know when it starts
* Allow to store all the information needed to perform certain task into a single object

**Messaging Patterns**

The Publisher Subscriber pattern

* senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be.