Design Patterns Day One

Eng Abdel Rahman

Design Vs Architecture

Design refers to the process of creating a detailed plan or blueprint for how a software system will function and look.

Architecture, on the other hand, refers to the high-level structure or framework of the software system. It involves making decisions about the overall organization of the system, such as how the components will be divided into subsystems, how they will communicate with each other, and what technologies will be used.

How a good design looks like

- Understandable and clear
- Reusable
- Extendable
- Easy to maintain
- Loosely coupled

Design Pattern

Design patterns are reusable solutions to commonly occurring software design problems. They provide general solutions to recurring design problems, helping developers to design software that is flexible, extensible, and easy to maintain.

Types of Design Patterns

- Creational patterns, which provide a way to create objects and classes in a flexible and extensible way.
- Structural patterns, which define how objects and classes can be composed to form larger structures and systems.
- Behavioral patterns, which define how objects and classes can interact with each other to accomplish specific tasks.

Architecture Patterns

software architecture patterns are high-level structures that provide a template for designing and implementing software systems.

Common Architecture Types

1. Monolithic architecture: This is a traditional architecture pattern in which all of the components of a software system are combined into a single executable or deployment unit. This pattern is simple to implement and deploy, but can become difficult to maintain and scale as the system grows.
2. Layered architecture: In this pattern, the software system is divided into layers, each of which provides a specific set of services to the layer above it. This allows for a clear separation of concerns and facilitates modularity and reusability.
3. Microservices architecture: This pattern involves breaking down a software system into a set of small, independent services that communicate with each other through APIs. This allows for greater scalability and flexibility, as each service can be developed, deployed, and maintained independently.
4. Service-oriented architecture (SOA): In this pattern, the software system is designed as a set of interconnected services, each of which provides a specific functionality or feature. This allows for greater flexibility and modularity, as services can be developed and deployed independently.

Common Architecture Types (Cont)

1. Event-driven architecture: In this pattern, the software system is designed to respond to events or messages, rather than relying on a central control structure. This allows for greater scalability and flexibility, as different parts of the system can operate independently.
2. Domain-driven design: This pattern involves designing the software system around specific business domains, rather than technical requirements. This allows for a closer alignment between the software system and the business needs it is designed to address.

SOLID Principles

SOLID is a set of five principles that are used to guide the design and development of software systems.

BY following these principles, developers can create software systems that are easier to maintain, extend, and refactor over time. The SOLID principles can also help to reduce code complexity and increase code reusability.

1. Single Responsibility Principle (SRP):

   This principle states that each class or module in a software system should have only one responsibility or reason to change. In other words, a class or module should have only one job to do, and should do it well.

Open/Closed Principle (OCP)

This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In other words, you should be able to add new functionality to a system without changing the existing code.

Liskov Substitution Principle (LSP)

his principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, a subclass should be able to substitute for its superclass without causing any problems.

Interface Segregation Principle (ISP):

This principle states that clients should not be forced to depend on interfaces that they do not use. In other words, you should design interfaces that are specific to the needs of each client, rather than creating a single, general-purpose interface that all clients must use.

Dependency Inversion Principle (DIP)

This principle states that high-level modules should not depend on low-level modules, but both should depend on abstractions. In other words, you should design your software so that the higher-level modules depend on abstract concepts, rather than specific implementations.

IOC

By using IoC and DI, the code becomes more modular, testable, and maintainable, since dependencies are managed by an external container or framework, rather than being tightly coupled within the code. This allows for greater flexibility and easier code reuse.

In summary, IoC stands for "Inversion of Control", which is a software design principle that describes the process of delegating control of the flow of execution to an external container or framework, such as a dependency injection container.

```
pp()->bind(PaymentGateway::class, CustomPaymentGateway::class);
```

DRY Principle

DRY stands for "Don't Repeat Yourself", which is a software development principle that emphasizes the importance of avoiding duplication in code.

The DRY principle states that every piece of knowledge or logic in a system should have a single, unambiguous representation within that system. This means that code should be organized in a way that eliminates redundancy and promotes reuse, making it easier to maintain and modify in the future.

The benefits of following the DRY principle include:

- Reducing the amount of code that needs to be written, which saves time and reduces the risk of errors.
- Improving the maintainability of the code, since changes only need to be made in one place.
- Enhancing the clarity and readability of the code, since there are fewer duplicated sections to confuse the reader.

To apply the DRY principle in practice, developers should strive to:

- Identify and eliminate duplicated code and logic.
- Extract common functionality into reusable modules or functions.
- Avoid copy-and-paste coding.
- Follow established coding standards and conventions.
- Continuously refactor and improve the codebase.

By following the DRY principle, developers can create more efficient, maintainable, and high-quality code.

Find the mistake

Which Solid Principle was violated?

7 Concepts of OOP In Simple Words

Encapsulation

data hiding, access modifiers, information hiding

Encapsulation refers to the practice of bundling data and methods that work on that data within a single unit.

Abstraction

interfaces, abstract classes, inheritance

Abstraction is the process of hiding complex implementation details from the user and exposing only the necessary information.

Inheritance

parent class, child class, subclass, superclass

Inheritance allows classes to inherit properties and behaviors from other classes, which can save time and reduce code duplication.

Polymorphism

method overloading, method overriding, interfaces

Polymorphism means that objects of different types can be treated as if they are of the same type, which allows for more flexible and modular code.

Composition

has-a relationship, object aggregation, object composition

Composition is a design technique in which smaller objects are combined to create larger, more complex objects.

Association

has-a relationship, object dependency

Association describes a relationship between two objects in which one object uses or depends on the other.

Dependency Inversion

dependency injection, inversion of control, interface segregation

DI is a principle that states that high-level modules should not depend on low-level modules, but both should depend on abstractions.

Keivan Damirchi

Composition

objects should be composed of other objects

Schedule -> Appointment

Cart -> Items

How to check a time slot is available in a schedule ?

Design Patterns

1- Singleton

- Create One instance from class
- Creational design pattern

2- Builder

- Create complex objects
- Creational design pattern
- Used in chaining methods like:

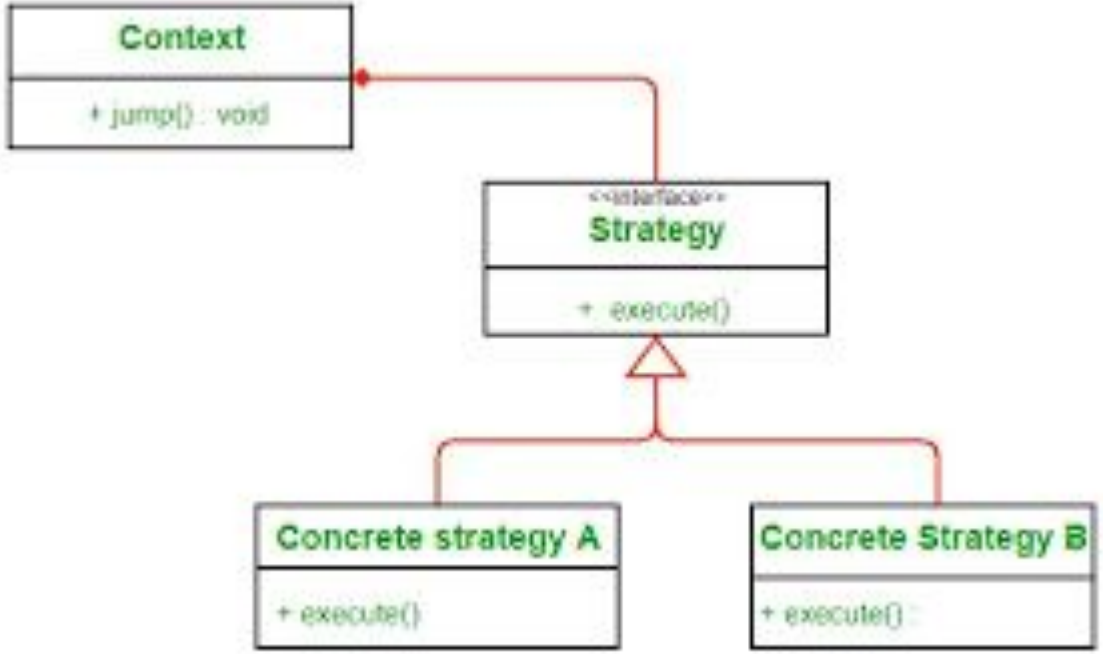
```
$burger = (new BurgerBuilder())
        ->addCheese()
        ->addTomato()
        ->addOnion()
        ->build();
```

Builder (cont)

- Each method return $this except the build method
- Another example of chain

```
$queryBuilder = new QueryBuilder("users");

$query = $queryBuilder->select(["id", "name"])

                ->where("age", ">=", "18")

                ->where("country", "=", "USA")

                ->build();
```

3- Strategy design Pattern



4- Factory design patterns

- The Factory design pattern allows us to create objects without having to specify the exact class name every time.
- This can make our code more flexible and easier to maintain, especially when dealing with complex object creation logic.
- It is a creational design pattern
- Object creational code is hidden in static method (factory method)

```
$object = class::factory_Method(parameter);

$object ->some_method
```

5- Prototype design pattern

- The Prototype design pattern is used when creating objects is either complex or costly. In such cases, instead of creating a new object each time, an existing object is cloned and then modified. This reduces the complexity and cost of creating new objects.

- A practical example of the Prototype design pattern is the creation of a report generator in a web application. Instead of creating a new report each time with all its details, a prototype report is created and stored in the database. When a user wants to generate a report, a clone of the prototype report is retrieved from the database, and its details are modified according to the user's request. This saves time and resources required to generate the report from scratch each time.

Questions

Which design pattern is used to construct complex objects step by step, where different processes can be used to create different representations of the same object?

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

A. Fahmy +

Question

Which design pattern is used to provide a single point of entry for creating objects of different types, depending on the parameters passed to it?

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

Ahmed Zain +

Question

Which design pattern is used to create objects by copying existing ones, thus avoiding expensive creation operations?

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

Nada

Question

Which design pattern allows you to encapsulate interchangeable providers and dynamically select algorithms at runtime

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

Youssef Adel

Question

Which design pattern ensures that only one instance of a class exists and provides a global point of access to it?

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

Esraa ElSayed

Question

Which design pattern can be used to implement a family of algorithms, each represented by a separate class, that can be selected and used at runtime

a. Factory

b. Singleton

c. Builder

d. Prototype

E. Strategy

Maysara safwat

Question

Which creational design pattern is used to encapsulate object creation logic in a separate class, thus making the code more modular and easy to maintain?

a. Factory

b. Singleton

c. Builder

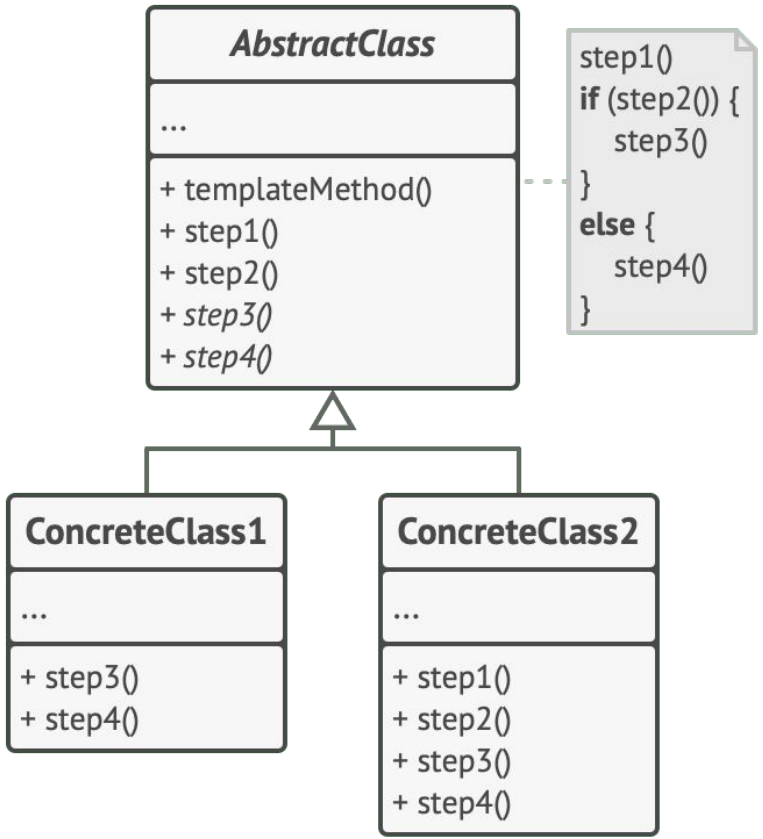
d. Prototype

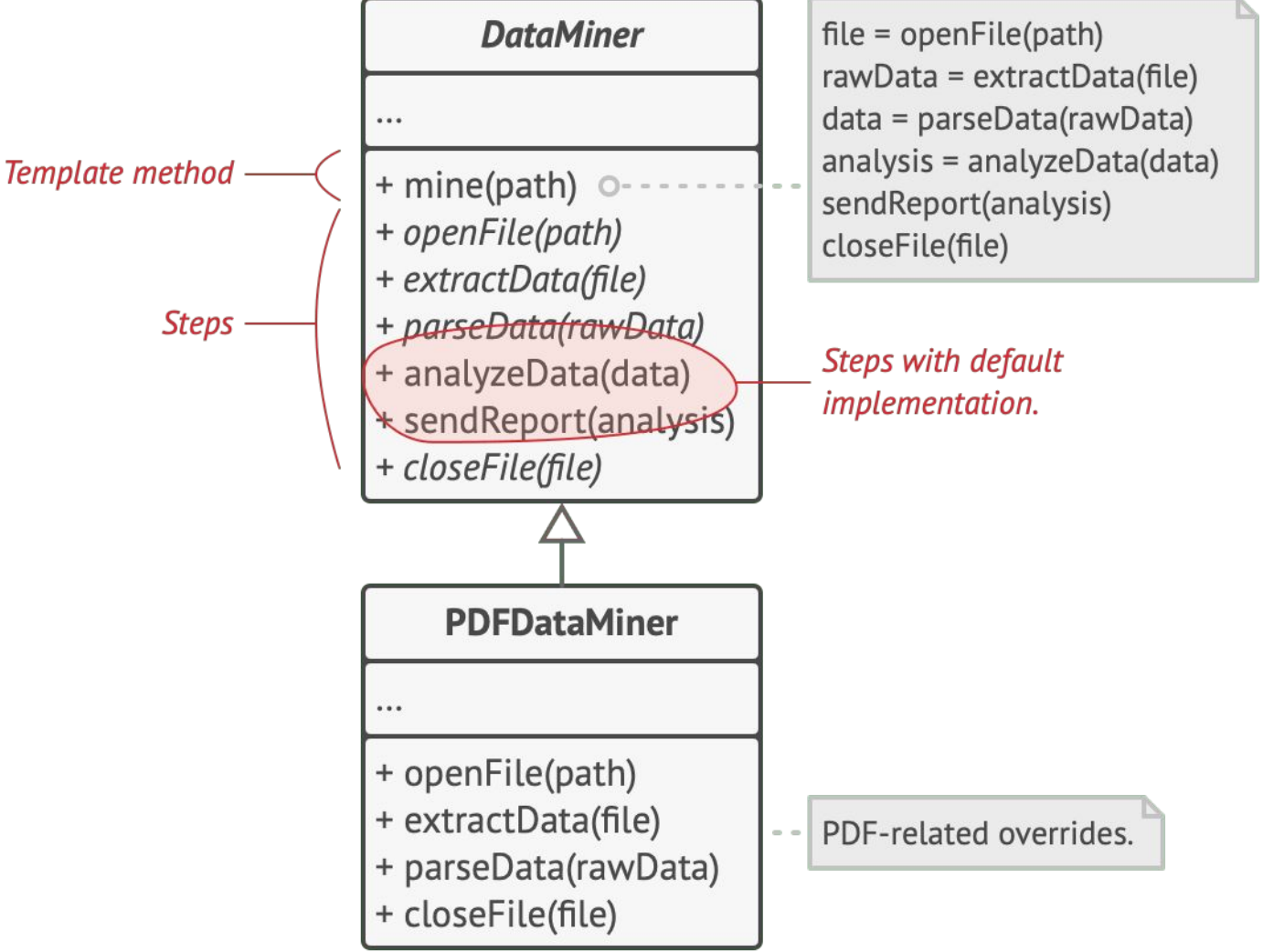
E. Strategy

Mahmoud Elbassyouny

6- Template Method design Pattern

Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes.




DataMiner
...
+ mine(path)
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ analyzeData(data)
+ sendReport(analysis)
+ closeFile(file)
Template method
Steps
file = openFile(path)
rawData = extractData(file)
data = parseData(rawData)
analysis = analyzeData(data)
sendReport(analysis)
closeFile(file)
Steps with default implementation.
PDFDataMiner
...
+ openFile(path)
+ extractData(file)
+ parseData(rawData)
+ closeFile(file)
PDF-related overrides.

Components of template method design pattern

1. Abstract class: The abstract class defines the basic steps of the algorithm in a template method. This template method typically calls other abstract methods that represent specific steps of the algorithm.

2. Concrete classes: The concrete classes are subclasses of the abstract class that provide concrete implementations for the abstract methods defined in the abstract class.

Hook methods

A hook is an optional step with an empty body. A template method would work even if a hook isn't overridden. Usually, hooks are placed before and after crucial steps of algorithms, providing subclasses with additional extension points for an algorithm.

In other words, a Hook Method is a placeholder in the template method that allows subclasses to inject custom behavior at specific points in the algorithm's execution. This allows the template method to be more flexible and adaptable to different situations, without requiring modifications to the overall structure of the algorithm.

Use of Template Method design pattern

The Template Method pattern is often used in frameworks and libraries to provide a standard way of performing a task that can be customized by users of the framework.

Example

We need to make a small package to call APIs a certain sequence to help developers follow good practise it in calling any API. The package will also support ready made classes to call many famous APIs such as GitHub, OpenWeather, ..etc

The sequence is :

- Preparing the request
- Logging the request
- Parsing the response
- Log the response
- Pre_response_hook
- If success send success response
- If failed send error response

Check the code of the example

Template method VS strategy

The Template pattern is similar to the Strategy pattern. These two patterns differ in scope and in methodology. Strategy is used to allow callers to vary an entire algorithm, like how to calculate different types of tax, while Template Method is used to vary steps in an algorithm.

7- Observer design pattern

In event-driven programming, the Observer pattern is commonly used to handle events that occur in a program

The Observer design pattern is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

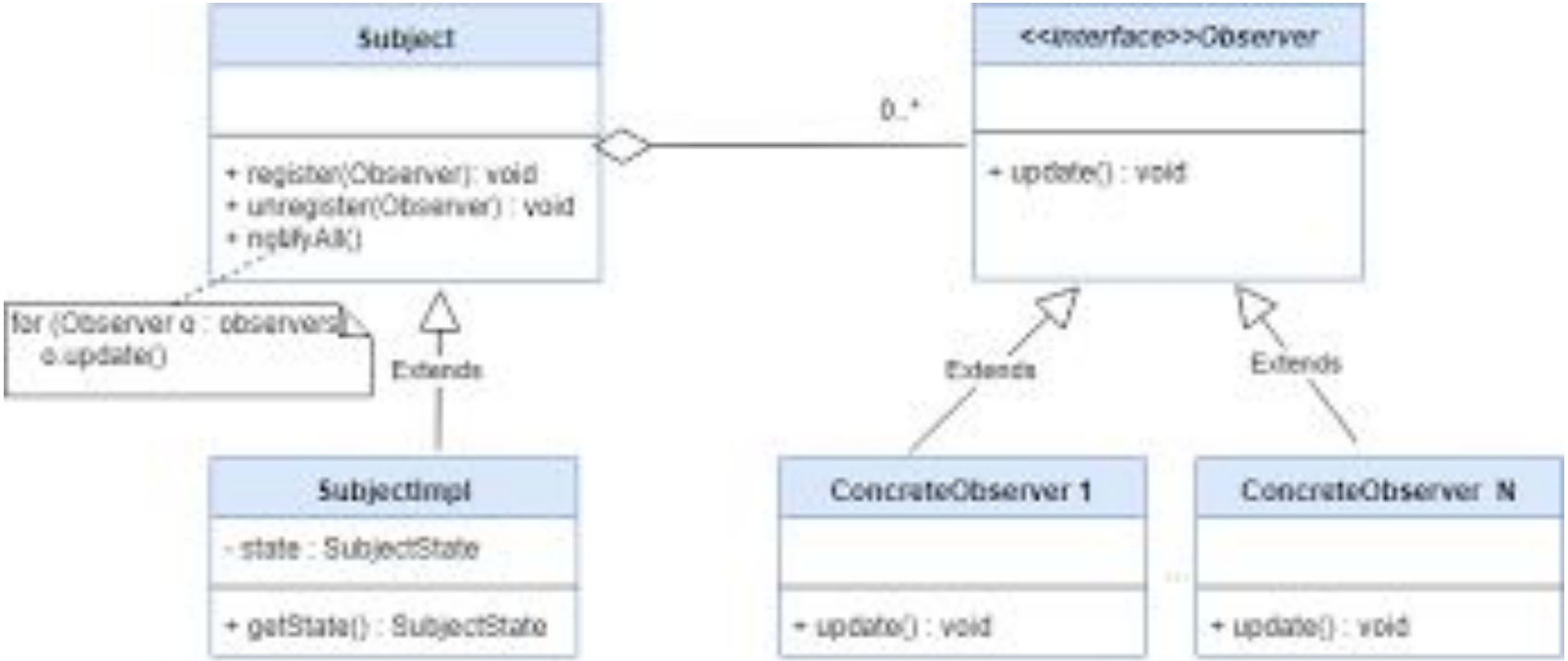
In this pattern, there are two main entities: the Subject and the Observer. The Subject is the object that maintains a list of its dependents, or Observers, and notifies them automatically of any changes to its state. The Observer is the object that is interested in the state of the Subject and registers itself with the Subject to receive updates.

The subject Class

- Has an array of Observers
- Has an Attach Method to add_Observer
- Has a Detach to remove Observer
- Has a Notify or trigger Method to be called when the event is fired to call the Update Methods of all observers.

The Observer Class

All Observer should just have an Update Method to be called when subject is changed.


Subject
+ register(Observer): void
+ unregister(Observer) : void
+ notifyAll()
for (Observer o : observers)
o.update()
0..*
<<interface>>Observer
+ update() : void
Extends
Extends
Extends
SubjectImpl
- state : SubjectState
+ getState() : SubjectState
ConcreteObserver 1
+ update() : void
ConcreteObserver N
+ update() : void

Example

When a booking is made in a hotel we desire to send the booking to the Accounting API used. Send Notifications to user with the Voucher via Email and SMS, Make the room reservation via the hotel API.

- Booking becomes a Subject to be Observed
- Accountant class, Notification class, Reservation class All are Observers that should be notified with New Booking to act accordingly

Check the code of the example

Model Events

Eloquent provides a set of events that you can use to hook into the model's lifecycle and execute code when certain actions occur.

You can listen to specific Model like Booking in previous example

Let say you want to change a behaviour that happened when you add a new record in All Models mot just one or two you can do that

Model Events in Eloquent

1. Retrieving Models: These events are triggered when querying for a model or when retrieving a model instance.
   - `retrieved`: This event is called after the model has been retrieved from the database.
   - `retrieved:{$class}`: This event is called after a model of a specific class has been retrieved from the database.

2. Saving Models: These events are triggered when creating or updating a model.
   - `creating`: This event is called before the model is saved for the first time.
   - `created`: This event is called after the model has been saved for the first time.
   - `updating`: This event is called before the model is updated.
   - `updated`: This event is called after the model has been updated.
   - `saving`: This event is called before the model is saved, either for the first time or as an update.
   - `saved`: This event is called after the model has been saved, either for the first time or as an update.
   - `restoring`: This event is called before a soft-deleted model is restored from the database.
   - `restored`: This event is called after a soft-deleted model has been restored from the database.

Events in Eloquent

3. Deleting Models: These events are triggered when deleting a model.
   - `deleting`: This event is called before the model is deleted.
   - `deleted`: This event is called after the model has been deleted.
   - `forceDeleting`: This event is called before a model is force deleted.
   - `restoring`: This event is called before a soft-deleted model is restored from the database.
   - `restored`: This event is called after a soft-deleted model has been restored from the database.

Eloquent and Observer design pattern

In Laravel's Eloquent ORM, the Observer pattern is used to enable listening to specific database events, such as when a model is created, updated, or deleted. This allows you to perform additional actions or modify the data before or after the database operation occurs.

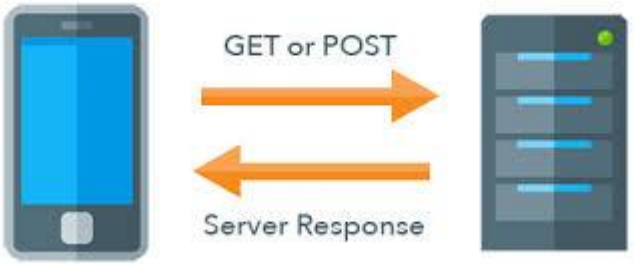
Sequence of Actions:

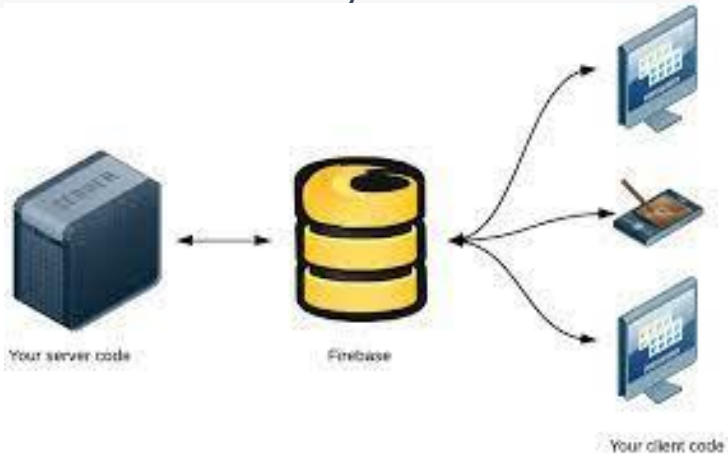
1. First, create a Booking model using `php artisan make:model Booking`.
2. Create an Observer class for the Booking model, using `php artisan make:observer BookingObserver --model=Booking`.
3. define the Observer class in the `boot` method of a Service Provider class. For example, create a `App\Providers\EventServiceProvider`class, and add the Observer to the `$listen` array:
4. Register the `EventServiceProvider` in the `config/app.php` configuration file

Check Code of Laravel

Notifying the Mobile that a change happened in any Model

A real-time database is a database system that allows for real-time data synchronization and instant updates.





Check the code of firebase Observer

8- Repository Design pattern

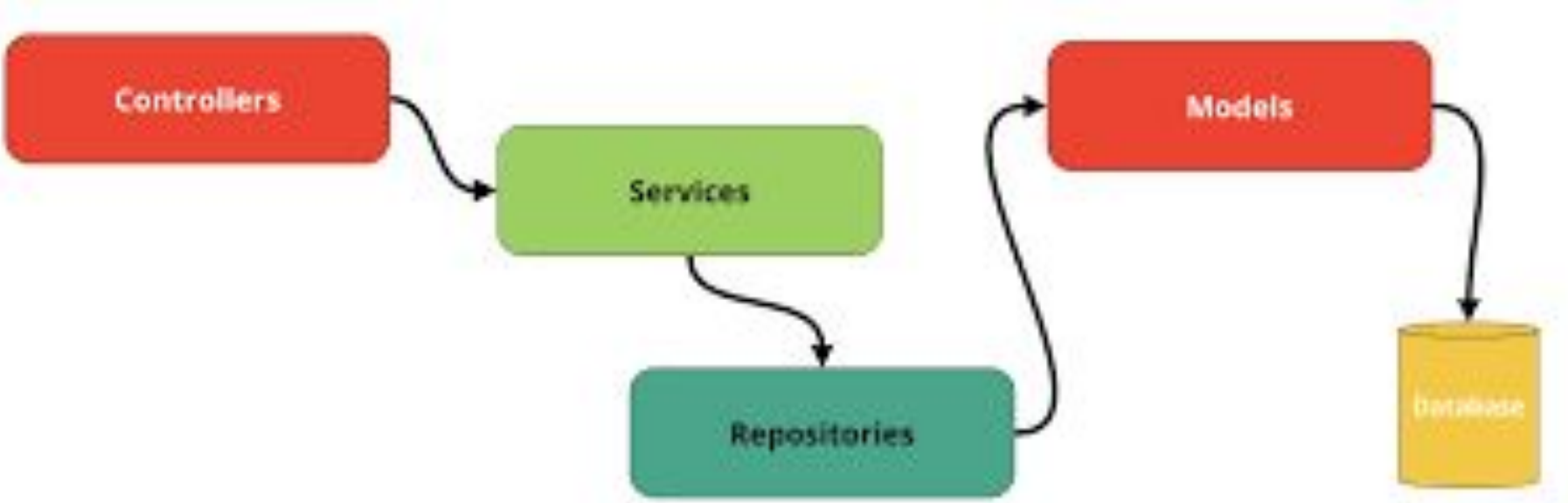
The repository design pattern is a software design pattern that separates the data access code and data manipulation code from the rest of the application. It provides a layer of abstraction between the application logic and the persistence layer, making it easier to switch to different data sources, databases or ORMs.

Repository Cont

The repository pattern involves creating a class that handles all the database interactions for a particular model or entity. This class is called a repository and it provides a set of methods for retrieving and manipulating data, such as `findAll()`, `findById()`, `create()`, `update()`, and `delete()`. The repository also provides a level of abstraction over the database query language, making it easier to change the database technology in the future.

Check the code

Controller, service, Repository and Model



Make a Laravel app that uses Repository DP to

- Has a User Model
- Has a User Repository
- Has an Authentication Service
- Has a Login Controller

Check The code

9- Facade design pattern

- Wrapper to cover complexity of calling multiple classes
- Wrapper to hide complexity of legacy system
- Wrapper to hide complexity of calling a 3rd party system and decrease the cost of change.
- Used extensively in laravel.
- It’s a structural design pattern.

9- Facade Design Pattern

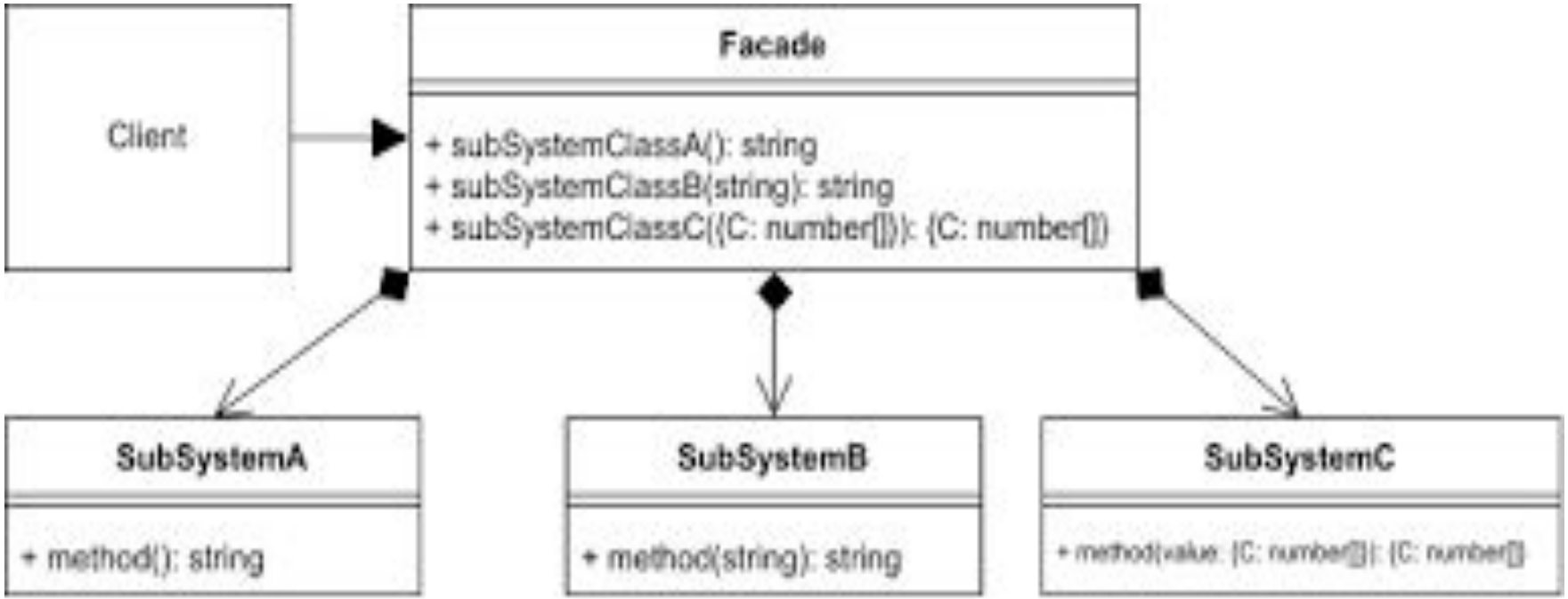
When we use a third-party plugin, we usually have to learn the API and adapt our code to its interface. If we directly use the API of the plugin throughout our codebase, changing the plugin or replacing it with another one can become very costly because we have to modify all the places in our codebase where we used the API of the plugin.

However, if we use a Facade to access the functionality of the plugin, we can isolate the plugin behind the Facade and make it easier to change or replace. The Facade provides a simple and consistent interface to the plugin that can be used throughout our codebase. If we need to change the plugin or replace it with another one, we only need to update the implementation of the Facade, and the rest of the codebase remains unchanged.

9- Facade Interface

The Facade design pattern provides a simplified interface to a complex system. It allows you to hide the complexities of the system behind a simple and easy-to-use interface, which makes it easier for developers to use the system without having to worry about its underlying details.

Using a Facade can make it easier to refactor complex legacy systems by isolating dependencies and providing a simpler and more consistent interface to components. This can help reduce the risk of introducing bugs or breaking the system when making changes or updates.


Client
Facade
+ subSystemClassA(): string
+ subSystemClassB(string): string
+ subSystemClassC({C: number[]}): {C: number[]}
SubSystemA
+ method(): string
SubSystemB
+ method(string): string
SubSystemC
+ method(value: {C: number[]}): {C: number[]}

Make two facades

1- A Geo Facade to wrape the code of calling a thired party GEO service instead of exposing the code in the entire project.

2- A payment Facade that provide a simple interface to Payment service , Order Service and Accountant Service so it can provide interface to make an order , send notification about the order and send the order to the accountant system.

Check the code

Collection Types

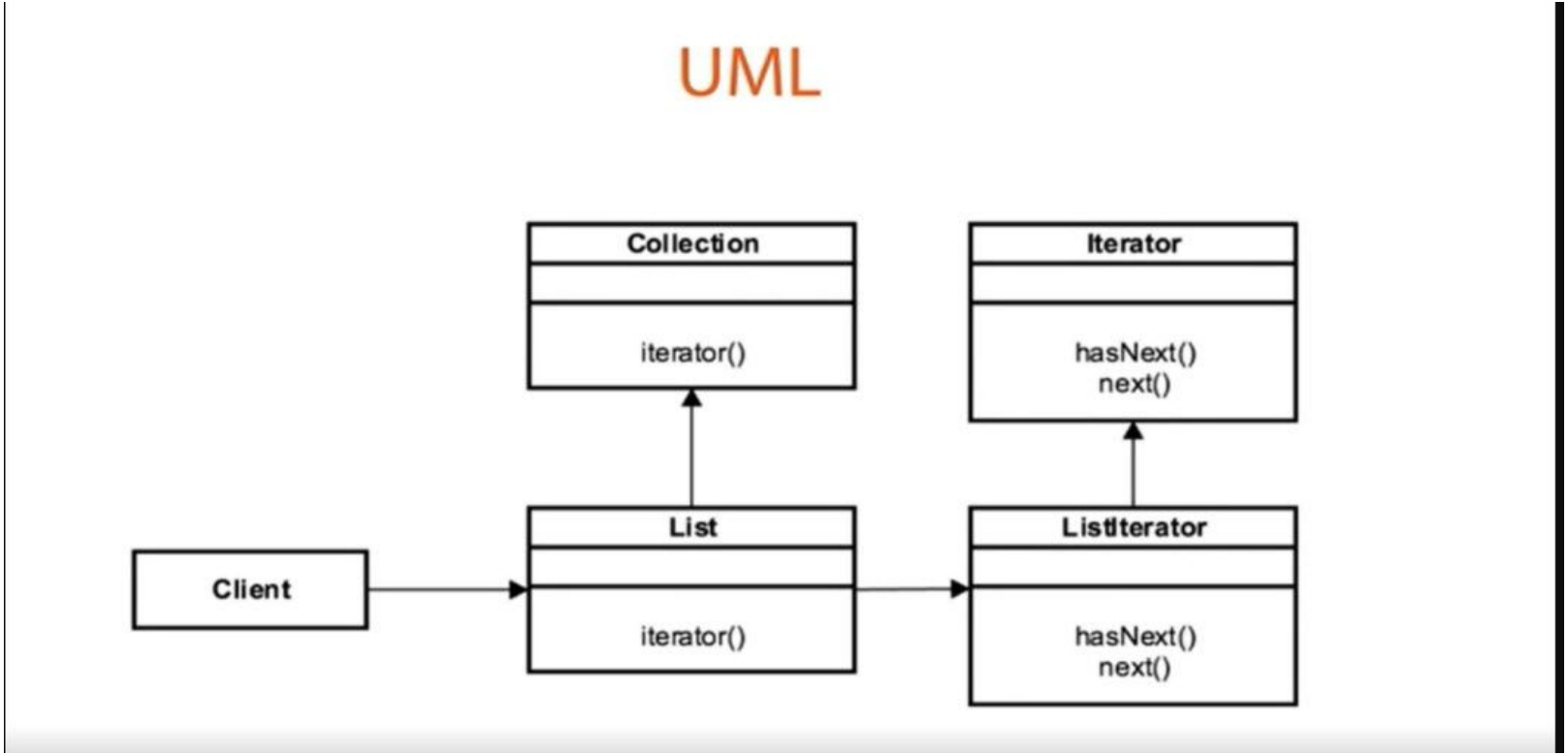
1. Array: A collection of elements of the same data type, stored in contiguous memory locations, and accessed by their index.
2. Linked List: A collection of elements that are linked together by pointers, with each element containing a value and a reference to the next element.
3. Map/Dictionary: A collection of key-value pairs, where each key maps to a unique value. Maps can be implemented using hash tables, binary trees, or other data structures.
4. Stack: A collection of elements that supports last-in-first-out (LIFO) access. Stacks can be implemented using arrays or linked lists.
5. Queue: A collection of elements that supports first-in-first-out (FIFO) access. Queues can be implemented using arrays or linked lists.

10- Iterator design pattern

- One of the main goals of the Iterator design pattern is to provide a common interface for iterating over different types of collections, so that you can use the same logic to traverse them regardless of their underlying implementation.

- the Iterator pattern allows you to iterate over a collection of objects without having to know the internal structure of that collection. It's a useful pattern for decoupling the traversal logic from the collection itself, making your code more modular and flexible.

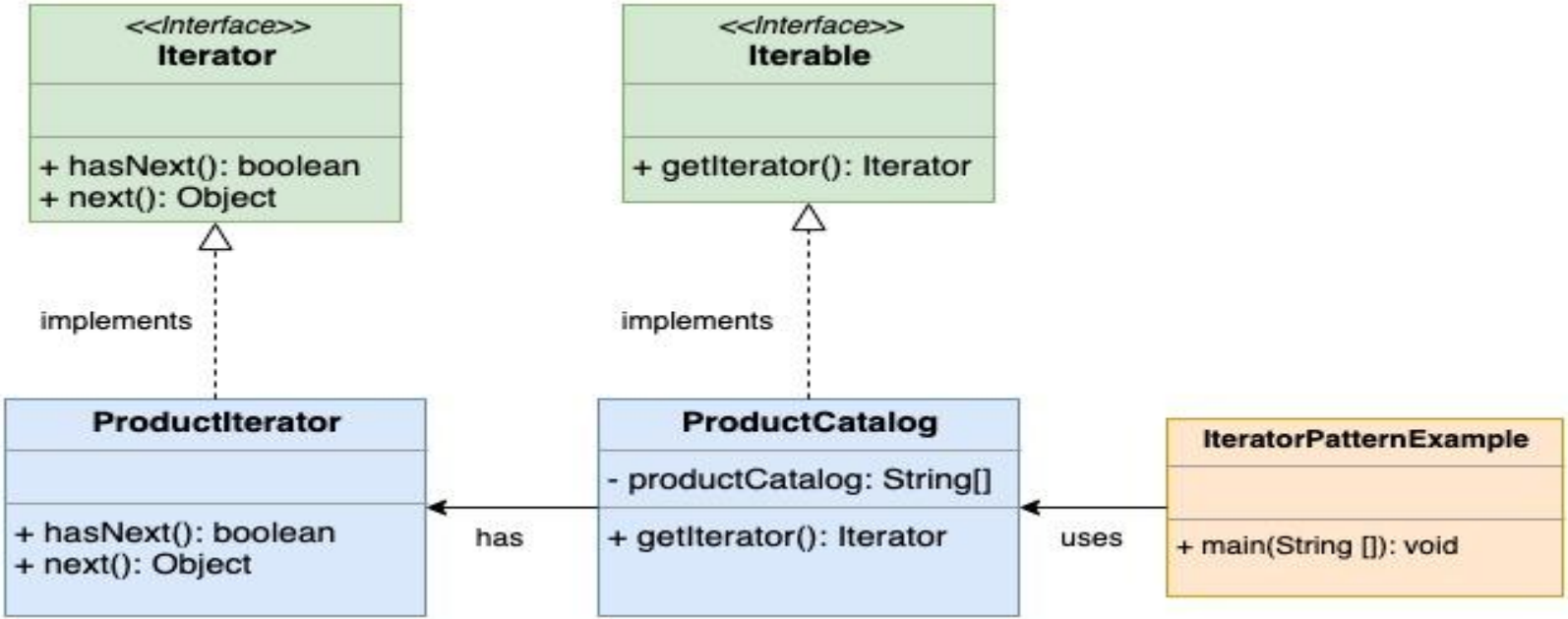
Iterator Components



iTERATOR design pattern In Java

The `Iterable` interface has only one method, `iterator()`, that returns an instance of the `Iterator` interface. The `Iterator` interface then has three methods: `hasNext()`, `next()`, and `remove()`.

Iterator Components



Iterator Components

1. Iterator interface: Defines a common interface for all iterators, specifying methods for traversing the collection and retrieving elements.
2. Concrete Iterator: Implements the Iterator interface and provides the actual implementation for iterating over a specific collection.
3. Aggregate interface: Defines a common interface for all collections, specifying a method for creating an iterator object.
4. Concrete Aggregate: Implements the Aggregate interface and provides the actual collection of elements to be iterated over.

Check the Code

1- Q and A

Which design pattern is used to provide a simple interface for a complex subsystem?

A. Façade

B. Observer

C. Singleton

D. Builder

2- Q and A

Which design pattern is used to ensure a class has only one instance and provide a global point of access to it?

A. Singleton

B. Factory

C. Prototype

D. Iterator

3- Q and A

Which design pattern is used to allow one object to notify others of changes to its state?

A. Façade

B. Observer

C. Builder

D. Repository

4- Q and A

You are developing a web application that needs to persist data to a database. Which design pattern can be used to provide a layer of abstraction between the application and the database, and simplify data access and manipulation?

a) Singleton

b) Repository

c) Builder

d) Prototype

5- Q and A

Which design pattern is used to provide a way to access elements of an aggregate object without exposing its underlying representation?

A. Iterator

B. Observer

C. Template Method

D. Façade

6- Q and A

Which design pattern is used to provide a way to copy an existing object without the need to create a new one from scratch?

A. Prototype

B. Observer

C. Strategy

D. Repository

7- Q and A

You want to create an object that requires a lot of configuration, with different optional and mandatory parameters. Which design pattern can be used to simplify the object creation process?

a) Singleton

b) Observer

c) Builder

d) Iterator

8- Q and A

Which design pattern is used to provide a way to create objects based on a set of input parameters?

A. Factory

B. Observer

C. Template Method

D. Prototype

9- Q and A

You have an Order class that needs to notify the Billing class when a new order is placed. Which design pattern can be used to implement this behavior?

a) Builder

b) Observer

c) Façade

d) Singleton

10- Q and A

You are building a web application that has two separate login systems, one for the main application and another for a third-party service that is integrated with the application. When a user logs in to the main application, you want to automatically log them in to the third-party service as well, so they don't need to enter their credentials twice. Which design pattern can be used to make one login system notifies the other one.

a) Factory

b) Builder

c) Observer

d) Strategy

11- Q and A

You want to define an algorithm in a base class, but allow subclasses to implement specific steps of the algorithm. Which design pattern can be used to achieve this?

a) Builder

b) Façade

c) Template Method

d) Iterator

12- Q and A

You want to create multiple objects that share the same properties and behavior, without having to duplicate the code for each object. Which design pattern can be used to achieve this?

a) Observer

b) Factory

c) Prototype

d) Builder

13- Q and A

You are developing a Laravel application, and you need to implement a feature that sends a notification to a user whenever their account is updated or when a new product is added or when a new promotion is introduced and other casses might be added in the future . Which design pattern can be used to observe changes to all needed models and trigger notifications to the user?

a) Factory

b) Builder

c) Model Observer

d) Singleton

14- Q and A

You are creating a package for developers and you want to support hock functions which design pattern you will use

a) Observer

b) Factory

c) Prototype

d) Template Method

15- Q and A

You have a collection of objects that you want to traverse in a specific order, but without exposing the collection's internal structure. Which design pattern can be used to achieve this?

a) Observer

b) Template Method

c) Façade

d) Iterator

16- Q and A

You are working on a legacy codebase that has a complex and tightly coupled system of modules and classes. You need to refactor the codebase to simplify the system and make it more modular. Which design pattern can be used to create a simple interface that hides the complexity of the underlying system and provides a unified entry point for the client code?

a) Prototype

b) Observer

c) Strategy

d) Facade

17- Q and A

You are working on a Laravel application that allows users to create and save models, but you need to change the behavior so that all models are saved with a different time zone. Which design pattern can be used to observe the creation of new models and modify their properties before they are saved to the database?

a) Facade

b) Strategy

c) Model Observer

d) Builder

18- Q and A

You are developing a web application that needs to integrate with the Zoho customer relationship management (CRM) system. The application should store new customer data in the Zoho system whenever a new user is registered on the site. Which design pattern can be used to create a simple interface that wraps the complexity of the Zoho API and provides a unified entry point for the client code?

a) Prototype

b) Facade

c) Singleton

d) Builder

19- Q and A

You have integrated your web application with the Zoho CRM system using the Facade design pattern to simplify the API calls. You now need to ensure that the Zoho system is notified whenever a new user is registered on the site. Which design pattern can be used to observe the creation of new users and trigger the Zoho API call to store the customer data?

a) Observer

b) Strategy

c) Factory

d) Prototype

20- Q and A

You are building a payment processing system that needs to support multiple providers of payment gateways, such as PayPal, Stripe, and Authorize.net. Each provider has a different API and set of features, but the system should be able to switch between providers dynamically without changing the code. Which design pattern can be used to encapsulate the algorithmic behavior of each provider in separate classes and enable the client code to choose the appropriate strategy at runtime?

a) Singleton

b) Prototype

c) Facade

d) Strategy

Famous software Architecture Patterns

- SaaS
- Service - Repository - Model
- Micro Services

SaaS

SaaS (Software as a Service) is a software model where the software application is provided to the users over the internet as a service. In this model:

1. The users typically do not own the software, but instead pay for its usage on a subscription basis.
2. Its cheaper for the users and more profitable for the publisher
3. it allows users to access software applications from anywhere with an internet connection
4. It removes the need for users to manage the software infrastructure themselves.

Isolation Types

One of the key considerations in designing a SaaS application is the data isolation between different tenants (i.e., users of the application).

There are two main approaches to data isolation in SaaS applications: record-based isolation and database-based isolation.

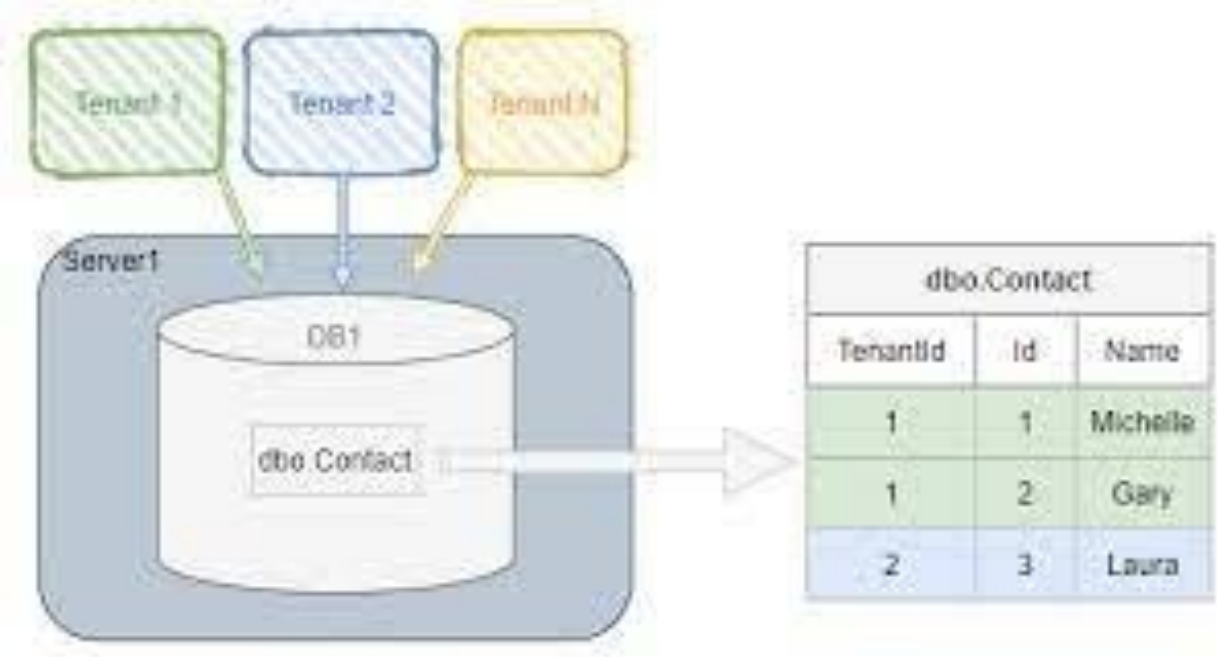
- Record Isolation
- Database isolation

Record Isolation

Record-based isolation involves storing data from different tenants in the same database tables, but with a tenant identifier (e.g., a user ID) associated with each record.

This approach is simpler to implement than database-based isolation, but it requires careful attention to the security and privacy of the tenant data. For example, it is important to ensure that queries always include the appropriate tenant identifier to prevent data leakage between tenants.

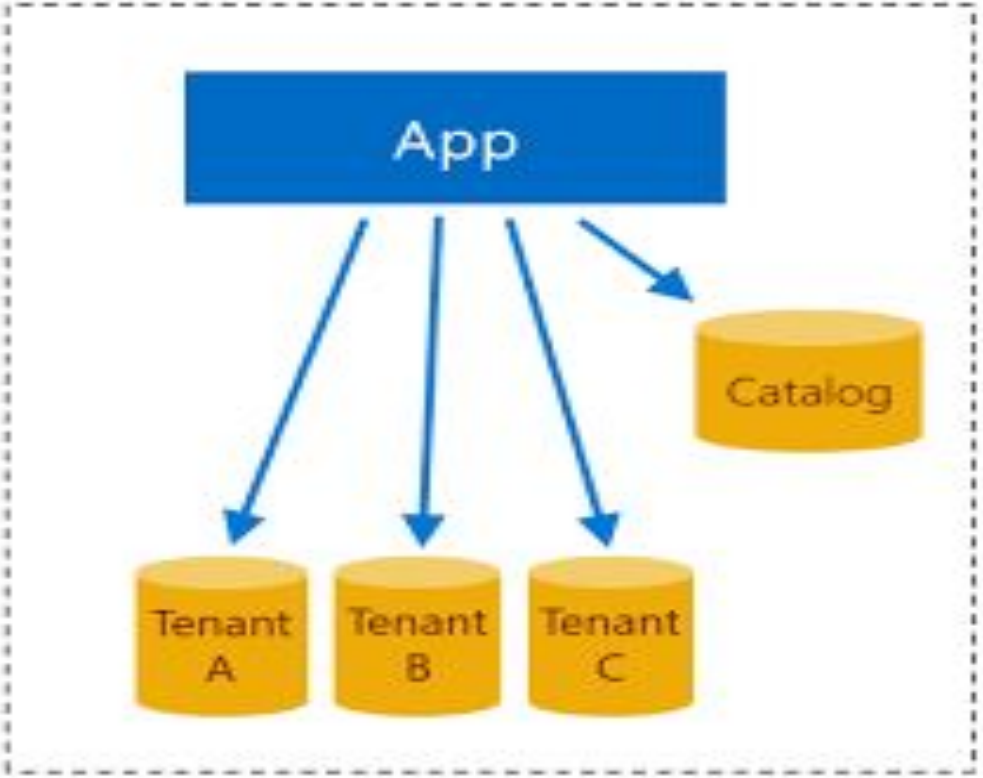
Record Isolation



DB isolation

Database-based isolation involves storing each tenant's data in a separate database, either on the same server or on different servers. This approach provides stronger data isolation between tenants, but it also requires more infrastructure and management overhead

DB isolation



Building Multi Tenant app with Laravel

Use Tenancy for laravel package here: https://tenancyforlaravel.com/?utm_campaign=ln2020t&utm_source=LN&utm_medium=article

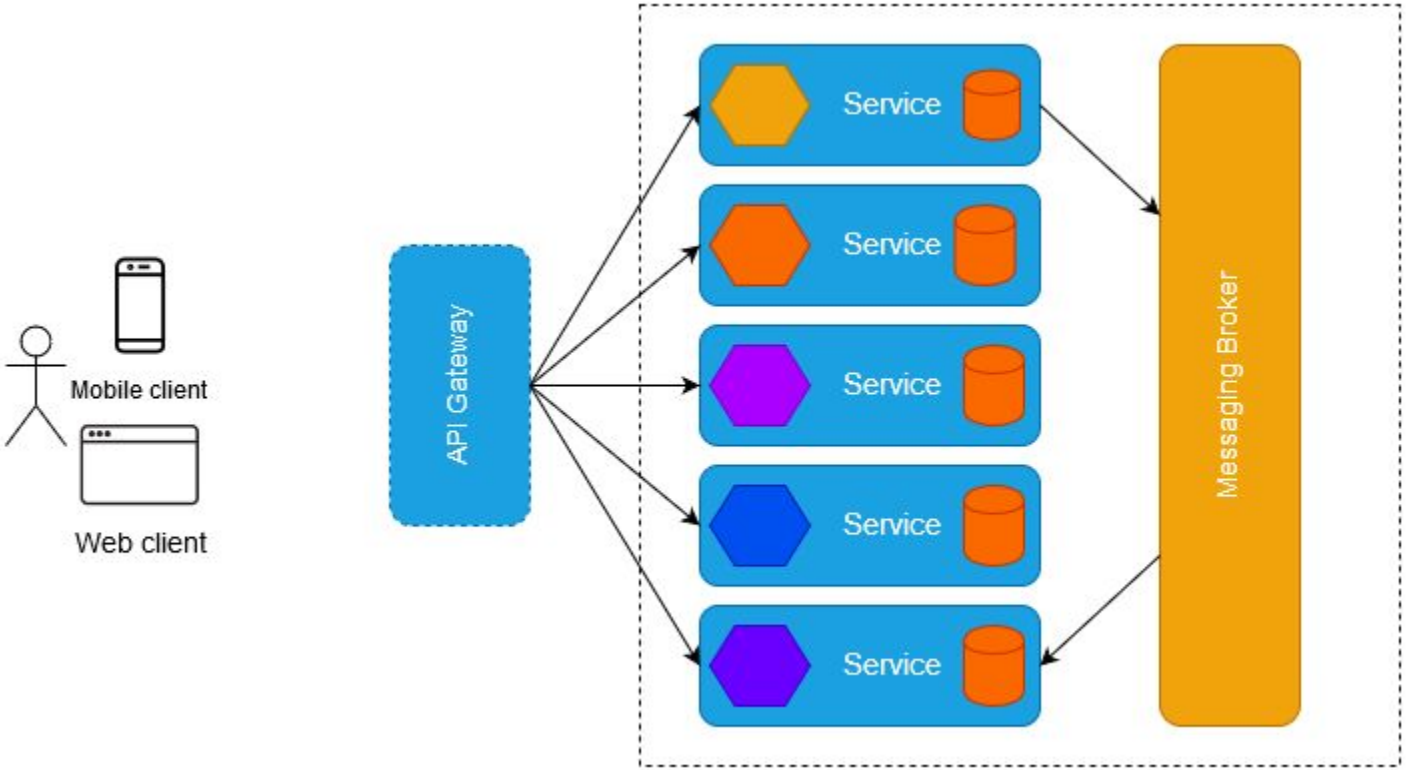
Microservices when to use them

- Complex applications with a lot of functionality
- Large-scale applications with high traffic and scalability requirements
- Applications that require high availability and fault tolerance
- Applications that require frequent updates and releases
- Applications that need to be developed and deployed independently by multiple teams
- Applications that require different technologies and programming languages for different components
- Applications that require flexible deployment and scaling options
- Applications that require easy integration with other systems and services
- Good DevOps team
- Skillable Developers
- Not recommended for startups in first stages.

Microservices when not To use them

- Simple applications with minimal functionality
- Small-scale applications with low traffic and scalability requirements
- Applications that do not require high availability and fault tolerance
- Applications that do not require frequent updates and releases
- Applications that can be developed and deployed by a single team
- Applications that use a single technology and programming language for all components
- Applications that do not require flexible deployment and scaling options
- Juniors Teams
- Not recommended for startups in their first stages

MicroServices Components



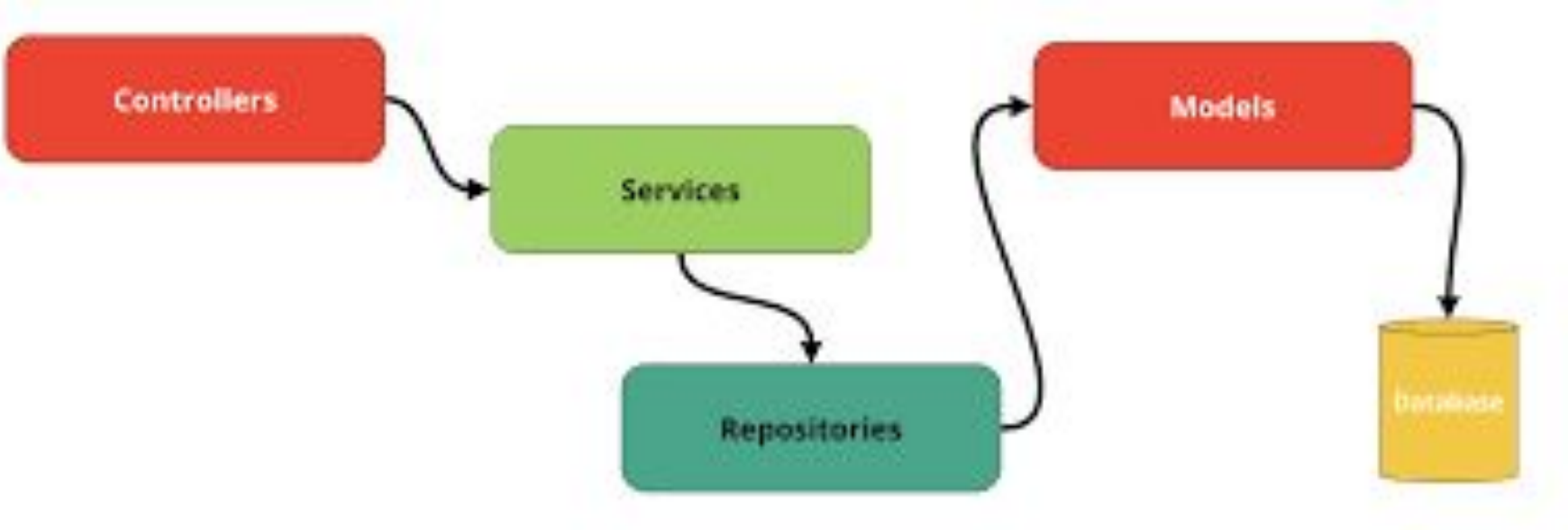
MicroServices Components

1. Services: Microservices are built around the concept of services, which are self-contained units of functionality that can be independently deployed and scaled. Each service is responsible for a specific business capability and communicates with other services using APIs.
2. APIs: APIs (Application Programming Interfaces) are the means by which microservices communicate with each other. APIs define the methods and protocols used for communication and data exchange between different services.
3. Data storage: Each microservice may have its own database or storage solution. This helps to ensure that each service is loosely coupled and can be independently scaled and maintained.

MicroServices Cont

1. Containers: Containers are lightweight, portable units that allow microservices to be easily deployed and run in any environment. Containerization helps to reduce dependencies and simplifies deployment and scaling.
2. Service registry: A service registry is a database that keeps track of all the available microservices in the system, their location, and their current status. This helps services to discover and communicate with each other.
3. Broker: A broker is a messaging system that enables asynchronous communication between microservices. Brokers help to decouple services and improve scalability and reliability.
4. Gateways: Gateways are components that provide a single entry point for external requests to the microservices system. They help to simplify and secure the communication between external clients and the microservices.

Controller , Service, Repository Pattern



Controller , Service, Repository Pattern

The Controller, Service, and Repository patterns are commonly used in software development, particularly in web applications.

Components

Controller:

Separates the concerns of handling HTTP requests from other application logic

Services

Encapsulates complex business logic

Controller , Service, Repository Pattern

Repository

- Abstracts away the details of data storage and retrieval, allowing for easy switching between different storage solutions.

Evaluate PHP Projects