# Laravel

1. **Docker:** Docker is a tool for running applications and services in small, light-weight "containers" which do not interfere with your local machine's installed software or configuration. This means you don't have to worry about configuring or setting up complicated development tools such as web servers and databases on your local machine.
2. **Sail:** Laravel Sail is a light-weight command-line interface for interacting with Laravel's default Docker development environment.
3. **Laravel Vapor:** Laravel Vapor is a serverless deployment platform for Laravel, powered by AWS.
4. **PHP & Blade:**
5. **Livewire:** alternative of JavaScript
6. **Inertia:** single page application process alternative of Vue , react
7. **Laravel Breeze:** simple authentication package with tailwind and bootstrap.
8. **Jet stream :** full authentication package. All API, auth, token , CSS bootstrap.
9. **Nginx:** server for load management.
10. **Laravel Forge:**

# **Request Life cycle:**

1. user send a request

2. request go to public/index.php:

a. check server is maintaining mood

* then load maintenance.php file

b. otherwise load vendor/autoload.php

c. load bootstrap/app.php

# **Design pattern**

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

there are 23 design patterns which can be classified in three categories:

1. Creational,
2. Structural
3. Behavioral patterns

**Creational patterns:**

1. Abstract factory
2. Builder
3. ~~Dependency Injection~~
4. Factory method
5. ~~Lazy initialization~~
6. Multiton
7. Object pool
8. Prototype
9. Resource acquisition is initialization
10. Singleton

**Structural :**

1. adapter,
2. Bridge
3. Composite
4. Decorator
5. ~~Delegation~~
6. ~~Extension object~~
7. Facade
8. Flyweight
9. ~~Front controller~~
10. ~~Marker~~
11. ~~Module~~
12. Proxy
13. ~~Twin~~

**Behavioral patterns**

1. ~~Blackboard~~
2. Chain of responsibility
3. Command
4. Interpreter
5. Iterator
6. Mediator
7. Memento
8. Null object
9. Observer or Publish/subscribe
10. ~~Servant~~
11. ~~Specification~~
12. State
13. Strategy
14. Template method
15. Visitor
16. ~~Fluent Interface~~

# **Dependency injection**

**Dependency:**

Before know dependency injection we need to know what is dependency. Dependency or dependent means relying on something for support. As example, if we want to go somewhere we depend on car.

In programming we say When class A uses some functionality of class B, then it’s said that class A has a dependency of class B. if we want to use other class method, we need to create the object of that class. Without create object we cannot use method of other class. Here class A create the object of class B and then use class B method.

Code:

Class engine{

Function show(){

Count<<”this is goo”;

}

}

Class Car {

Car yahamaEngine = new Engine();

YahamaEngine.show();

}

In above, Car use show method by create object of Engine class. So, we called Car is dependence of engine class.

Here is some problem of class dependency:

1. Class is not testable
2. Code is not extensible
3. Single responsibility
4. Lifetime of Object:

To solve this type of problem we use dependency injection.

**Dependency injection:**

In dependency injection dependence class object are create by someone else and dependable class use this object. because dependencies can be injected at runtime rather than at compile time

The 3 Types of Dependency Injection---

1. constructor injection,
2. method injection,
3. property injection.

**Constructor Injection:**

Constructor injection is the process of using the constructor to pass in the dependencies of a class. You should use constructor injection when your class has a dependency that the class requires in order to work properly.

If your class cannot work without a dependency, then inject it via the constructor.

you should use constructor injection when the dependency in question has a lifetime longer than a single method. Dependencies passed into the constructor should be useful to the class in a general way, with its use spanning multiple methods in the class. If a dependency is used in only one spot, method injection

Checking for null is necessary and is boilerplate code. Protecting against null being passed as a parameter is called the guard pattern

public class CustomerBusinessLogic{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess){

\_dataAccess = custDataAccess;

}

public CustomerBusinessLogic(){

\_dataAccess = new CustomerDataAccess();

}

public string ProcessCustomerData(int id){

return \_dataAccess.GetCustomerName(id);

}

}

public interface ICustomerDataAccess{

string GetCustomerName(int id);

}

public class CustomerDataAccess: ICustomerDataAccess{

public CustomerDataAccess(){

}

public string GetCustomerName(int id){

return "Dummy Customer Name";

}

}

**Property Injection (setter injection):**

You should use property injection in case the dependency is truly optional

Property Injection however causes Temporal Coupling and when writing Line of Business applications, your dependencies should never be optional: you should instead apply the Null Object pattern.

property injection is considered bad in 98% of all scenarios because it hides dependencies and there is no guarantee that the object will be injected when the class is created.

The built-in IoC container does not support property injection. You will have to use a third-party IoC container.

Code:

public class CustomerBusinessLogic{

public CustomerBusinessLogic(){

}

public string GetCustomerName(int id){

return DataAccess.GetCustomerName(id);

}

public ICustomerDataAccess DataAccess { get; set; }

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

\_customerBL.DataAccess = new CustomerDataAccess();

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

**Method Injection:**

Thus, method injection is useful in two scenarios: when the implementation of dependency will vary, and when the dependency needs to be renewed after each use. In both cases, it’s up to the caller to decide what implementation to pass to the method.

interface IDataAccessDependency{

void SetDependency(ICustomerDataAccess customerDataAccess);

}

public class CustomerBusinessLogic : IDataAccessDependency{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic() { }

public string GetCustomerName(int id){

return \_dataAccess.GetCustomerName(id);

}

public void SetDependency(ICustomerDataAccess customerDataAccess){

\_dataAccess = customerDataAccess;

}

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

((IDataAccessDependency)\_customerBL).SetDependency(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

**Inversion of Control (IoC)**