Laravel life cycle

1. Web Server:

الـ **Web Server** هو اللي بيستقبل أول حاجة الطلب بتاع المستخدم. يعني لما تفتح صفحة في المتصفح، المتصفح بيبعت طلب بيشوف إيه اللي جاي من المتصفح وبيبعت الطلب ده لبرنامج web server الـ (زي Apache أو Nginx) للـ عشان يعالجه PHP.

2. Index.php:

في لارافيل. الملف ده هو البداية لأي `public` اللي موجود في مجلد `index.php` بيشوفها هي الملف PHP أول حاجة وبعد كده، Composer بتاع autoloader طلب جاي. هو زي نقطة البداية اللي بتجهز شوية حاجات أولية زي تحميل الـ بيشغل تطبيق لارافيل.

3. Kernel:

هو اللي Kernel في لارافيل بيشيل مسئولية إدارة الطلبات والردود. يعني الـ **Kernel** بعد ما يحصل الإعدادات الأولية، الـ بيتعامل مع الطلبات وبيمررها للـ middleware (الحاجات اللي بتتعامل مع الطلب قبل أو بعد ما يوصل للتطبيق).

4. Service Providers:

هما اللي بيقوموا بإعداد وربط الخدمات في لارافيل. يعني لو عندك حاجة زي قاعدة بيانات، أو **Service Providers** الـ بتاع لارافيل عشان تقدر تستخدمها في أي container بيحطوها في الـ Service Providers إيميل، أو أي خدمة تانية، الـ مكان في التطبيق.

5. Routing:

الرابط اللي المستخدم بيطلبه)) URI اللي هيتنفذ. يعني الـ **Route** في المرحلة دي، لارافيل بيبدأ يفحص الطلب ويربطه بالـ ولما يلاقي الـ `routes/api.php`. أو `routes/web.php` اللي بتكون موجودة في الملفات Routes بيتطابق مع الـ Route بينفذ الـ المتطابق، controller أو الـ closure المرتبط به.

6. Controller:

الـ **Controller** Controller الصح، هو بيروح لـ Route هو المسئول عن التعامل مع الطلبات. يعني لما لارافيل يوصل للـ اللي بتتعامل مع قاعدة البيانات)، ويجهز الرد اللي هيرجع للمستخدم) Models عشان يعالج البيانات، يتفاعل مع الـ.

7. Model:

الـ **Models** محتاج يجيب بيانات أو يغير بيانات في قاعدة Controller في لارافيل بتتعامل مع قاعدة البيانات. يعني لو الـ Model البيانات، بيستخدم الـ عشان يسهل التعامل مع البيانات دي.

8. Response:

الرد اللي هيرجع للمستخدم). الرد ده ممكن يكون **Response** يعالج الطلب، بيجهز الـ Controller بعد ما الـ HTML أو أي نوع تاني من الردود (API response لو هو JSON عشان يظهر في المتصفح) أو.

9. Middleware Again:

أو يتأكد من الأمان)، Headers تاني ممكن يعدل في الرد ده (مثلاً يضيف له **Middleware** قبل ما الرد يروح للمستخدم، الـ زي ما حصل في البداية قبل ما يوصل للتطبيق.

10. Sending the Response:

بيبعت الرد للمستخدم في المتصفح Web Server والـ ،Web Server وأخيراً، بعد ما يتجهز الرد، لارافيل بيرجع الرد ده للـ عشان يظهر له الصفحة أو البيانات.

وبكده تخلص دورة حياة الطلب في لارافيل.

solid principles

1- المبدا الاول Single Responsibility

له مسئولية واحدة فقط يقوم بيها وله سبب واحد فقط فى التغيير Model او Class او Function معناها ان كل
مثلا المحاسب هو ال بيقوم بعملية الحسابات في الشركة (مسئولية واحدة)غير مسئول مثلا عن التسويق
Cohesive كمبدأ المسئولية الواحدة بيجعل الكلاس اكثر تماس
متي نحتاج الي هذا المبدا (مبدا المسئولية الواحدة)؟
لما الاقي كلاس معين يقوم باكثر من مسئولية في نفس الوقت.
ذي مثلا كلاس واحد يقوم بقراءة البيانات وحفظ البيانات في قاعدة البيانات والتحقق من سلامة البيانات
ده عبارة عن انذار لاستخدام هذا المبدا واقسم الكلاس ده الي كلاسات صغيرة كل كلاس مسئول عن
مسئولية واحدة فقط
الحل
هعمل كلاس يكون مسئول عن قراءة البيانات ... وكلاس مسئول عن حفظ البيانات في قاعده البيانات
وكلاس مسئول عن التحقق من سلامة البيانات وكدا انا طبقت مبدا Single Responsibility

2- المبدا الثاني Open-Close Principle

معناه ان بيسمح بالاضافة وغير مسموح بالتعديل
Open > Extended مسموح بالتوسع والاضافه
Close > Modified غير مسموح بالتعديل علي الكود الحالي
لانه بيسمحوا اننا نضيف علي الكود اللى موجود بداخلة Abstract & Interface ممكن استخدامه هذا المبدا باستخدام ال

3- المبدا الثالث Liskov Substitution Principle

لو عند كلاس للاب وكلاس للابن اقدر اتبادل الادوار بنهم من غير ما ابوظ البرنامج
T يساوي الاوبجكت بتاع ال S و كلاس لإ ممكن اخلى الاوبجكت بتاع ال S لو عندي اتنين كلاس
Ex:

```
T t = new T( ); S s = new S( );
```

ممكن اقول

```
T t = new T( ); Or T t = new S( );
```

معناها لو الاب مش موجود ممكن الابن يقوم بنفس الدور

هو مش المفروض يستخدمها Non Implemented مخليش مثلا كلاس اضع فيه دوال

4- المبدا الرابع Interface Segregation Principle

هو مش المفروض يستخدمها Non Implemented مخليش مثلا كلاس اضع فيه دوال

طبيعي لما اي كلاس مثلا يورث منه لازم يطبق كل الدوال ال داخله Interface مثال: انا لما يكون عندي
ISP طيب ممكن الكلاس مش عاوز يطبق كل الدوال ال فيه في الحالة دي بنستخدم مبدا
لوحده Interface وهي بنقوم بوضع الدوال ال محتاجها الكلاس في

5- المبدا الخامس Dependency Inversion Principle

Abstract لازم يعتمدوا الاتنين على ال Low Level Module مينفعش يعتمد علي ال High Level Module معناها ان ال
و العكس صحيح Concrete يعتمد على تفاصيل ال Abstraction مينفعش ال
High Level Module و ال Low Level Module ما الفرق بين ال
اخر Module ال بيعتمد على Module هو ال << High Level Module
اخر او بيتعمد عليه لكن فى اضيق الحدود Module ال مش بيعتمد على Module هو ال << Low Level Module

شرح SOLID بالبلدي:
S - مبدأ المسؤولية الواحدة (SRP):
يعني إن كل كلاس (فئة) لازم يكون عنده مهمة واحدة بس.
زي ما تقول: كلاس واحد يختص بكل حاجة متعلقة بالمستخدم (زي التسجيل)، وكلاس تاني يكون مسؤول عن قاعدة البيانات. متخلطش بين المهام في نفس الكلاس.
O - مبدأ الفتح والإغلاق (OCP):
الكلاس لازم يكون مفتوح للتوسيع (يعني ممكن تضيف له مميزات جديدة)، ولكن مغلق للتعديل (ما تعدلش على الكود القديم).
زي ما تقول: لو عايز تضيف خصائص جديدة، تقدر تعمل كلاس جديد يضيف الوظائف دي من غير ما تعدل في الكود القديم.
L - مبدأ الاستبدال (LSP):
لو عندك كلاس كبير، لازم تقدر تستبدل أي كلاس مشتق منه (أي كلاس مخصص) وتستمر الشغل زي ما هو من غير ما يخرب الكود.
يعني لو عندك دالة بتتعامل مع الكلاس الأب، المفروض تشتغل بنفس الطريقة مع أي كلاس فرعي (مشتق).
I - مبدأ تقسيم الواجهات (ISP):
العميل (أي كلاس تاني بيستخدم الكلاس ده) مش لازم يعتمد على الوظائف اللي مش محتاجها.
يعني بدل ما تعمل واجهة فيها شوية دوال مالهاش علاقة ببعضها، قسمها لواجهات أصغر وأكثر تحديدًا.
D - مبدأ عكس الاعتماديات (DIP):
الكلاسات الكبيرة (اللي فيها منطق معقد) مش لازم تعتمد على الكلاسات الصغيرة (اللي فيها تفاصيل تنفيذية)، لكن الكل يعتمد على واجهات.
يعني خلي الكود بتاعك أكثر مرونة، لو استخدمت واجهات أو كلاس تجريدي، الكلاس الكبير مش هيتأثر لو الكلاس الصغير اتغير

ملخص ل Git

Git is a distributed version control system that enables developers to collaborate efficiently on projects. It allows multiple people to work on the same codebase simultaneously without conflicts. Here are the key concepts in Git:

Repository: A storage space where your project resides. It can be local on your machine or hosted on a remote server.

Commit: A snapshot of the repository at a specific point in time, containing all the changes made.

Branch: A separate line of development in your repository. The default branch is typically called "main" or "master."

Merge: The process of integrating changes from one branch into another.

Clone: Creating a local copy of an existing repository.

Pull: Fetching changes from a remote repository and merging them into your local repository.

Push: Sending your local commits to a remote repository.

Common Git Commands:

git init: Initialize a new Git repository.

git clone [url]: Clone an existing repository from a remote server.

git status: Check the status of your working directory.

git add [file]: Stage changes for the next commit.

git commit -m "message": Commit staged changes with a descriptive message.

git branch: List all branches in the repository.

git checkout [branch]: Switch to a different branch.

git merge [branch]: Merge changes from one branch into the current branch.

git pull: Fetch and merge changes from a remote repository.
git push: Push local commits to a remote repository.

Classes and Objects

- **Class**: A **template** or **blueprint** used to create objects. The class contains **properties** (data) and **methods** (functions) that define the behavior of an object.

Practical Example: Consider a `Car` class. This class contains properties like color and speed, and methods like `accelerate()` (to speed up the car) and `brake()` (to slow it down).

- **Object**: An **instance** of a class. When an object is created from a class, we can access the properties and methods defined in that class.

Practical Example: When you create a new car using the `Car` class, the car becomes an object. This car could be red (color) and capable of speeding up or slowing down using the class methods.

2. Inheritance

- **Inheritance** allows a new class to **inherit** properties and methods from an existing class. This enables code reuse without rewriting it.

Practical Example: Imagine a `SportsCar` class inheriting from the `Car` class. The `SportsCar` class will inherit all the properties and methods from `Car`, like speed and color, but it might also add a new property like `turboBoost` to further increase speed.

3. Encapsulation

- **Encapsulation** means hiding the internal details of data so that the data is protected and cannot be accessed or modified directly. Data is handled through **methods** provided by the class.

Practical Example: In a `BankAccount` class, we might have a `balance` property, but it cannot be accessed directly. Instead, methods like `deposit()` (for deposits) and `withdraw()` (for withdrawals) are used to safely modify the balance.

4. Polymorphism

- **Polymorphism** means that the same method can operate on different types of objects, and in each case, it can have a **different implementation** based on the type of object.

Practical Example: If you have an `Animal` class with a `makeSound()` method, the `Dog` class might implement it to print "Bark!", while the `Cat` class might implement it to print "Meow!". The same method, but the behavior differs depending on the object.

5. Abstraction

- **Abstraction** means hiding the complex details and focusing on the essential features. In other words, the **implementation** is hidden and only the **interface** is provided for the developer to use.

Practical Example: Consider a `RemoteControl` class with methods like `powerOn()` and `powerOff()`. However, the inner workings (like switching between channels) are hidden from the user, and the user doesn't need to understand the complex details to operate it.

6. Interfaces

- An **interface** is a **contract** that defines a set of methods that any class must implement. The interface does not provide any **implementation** but only the method definitions.

Practical Example: Imagine an interface called `Flyable` containing a method `fly()`. Classes like `Bird` and `Airplane` can implement this interface and define how each flies, but each will implement `fly()` differently.

Core Objectives of OOP:

- **Code Reusability**: Through **inheritance** and **interfaces**, you can create new classes based on existing ones.
- **Reducing Redundancy**: With **encapsulation** and **polymorphism**, you can use the same code in multiple places without having to rewrite it.
- **Improving Maintenance and Scalability**: Using **abstraction** and **interfaces** makes it easier to modify the code or add new features without affecting the existing code.

Key Concepts:

- **Inheritance**, **Encapsulation**, **Polymorphism**, **Abstraction**, and **Interfaces** are the foundational principles of OOP that help organize code, making it easier to maintain and scale.