Php Laravel

Intro to OOP + Composer

Create a small library package and consume via Composer

1. Create a Project Name “StringUtils” which will be used as a Package for some Other Project.
2. In Terminal Type “composer init”.

```bash
composer init
```

3. Input the Details like Package-name,Description etc:

```json
{
    "name": "aniket/stringutils",
    "description": "its a string utility package",
    "type": "library",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "Utils\\": "Utils/"
        }
    },
    "authors": [
        {
            "name": "Aniket singh"
        }
    ],
    "minimum-stability": "stable",
    "require": {}
}
```

4. Create a Utils folder and add your Class there.

5. Test the class in that folder if wanted using the index.php file
6. Create a New Folder Name “ConsumerApp” in which we will use this StringUtils Package.
7. Use “composer init”
8. Input all the details like done above.
9. In “composer.json” add two lines: (Note: In “require” use the package name to be installed like in our case “aniket/stringutils”)

```json
"repositories": [
    {
        "type": "path",
        "url": "../StringUtils"
    }
]
```

```json
"require": {
    "aniket/stringutils": "dev-main"
},
```

Php Laravel

```
{
    "name": "aniket/consumer-app",
    "description": "Trial App to consume String Utility Package",
    "type": "project",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "": "src/"
        }
    },
    "authors": [
        {
            "name": "Aniket Singh"
        }
    ],
    "minimum-stability": "stable",
    "require": {
        "aniket/stringutils": "dev-main"
    },
    "repositories": [
        {
            "type": "path",
            "url": "../StringUtils"
        }
    ]
}
```

10. Now Run “composer install” it will install the packages in the “require” from the path given in the “repositories”

```
composer install
```

11. Now use the package using the “use” command.

- Package name: `myvendor/consumer-app`
- Description: A test app consuming string-utils
- Author: Your Name your.email@example.com
- Type: `project`
- License: `MIT`

```
{
    "name": "myvendor/my-project",
    "require": {
        "myvendor/string-utils": "dev-main"
    },
    "repositories": [
        {
            "type": "path",
            "url": "../string-utils"
        }
    ]
}
```

Add the library to your project. If it's not on Packagist, you can use a local or VCS repository. For example, add to your project's `composer.json`:

```
"repositories": [
    {
        "type": "vcs",
        "url": "path/to/your/string-utils.g
    }
],
"require": {
    "myvendor/string-utils": "dev-main"
}
```

PSR-12

Key PSR-12 Coding Standards

Php Laravel

1. **General**:
    1. Files must use <?php or <?= tags, with no other variations.
    2. Files must use UTF-8 without BOM.
    3. One file should declare either new symbols (classes, functions, etc.) or perform side effects (e.g., output), but not both.
2. **Indentation and Lines**:

    1. Use 4 spaces for indentation (no tabs).
    2. Lines should not exceed 120 characters; aim for 80 characters where possible.
    3. No trailing whitespace at the end of lines.
    4. Blank lines should be used sparingly to separate logical sections.
    5. Files must end with a single blank line.

3. **Namespaces and Use Statements**:

    1. Namespace declarations must have one blank line after them.
    2. use statements must follow the namespace with one blank line after the block.
    3. Group related use statements together (e.g., all class imports, then functions).

4. **Classes, Properties, and Methods**:

    1. Class names must use StudlyCaps.
    2. Method names must use camelCase.
    3. Opening braces for classes and methods go on the next line.
    4. All properties and methods must have visibility declared (public, protected, or private).
    5. Use one property or method declaration per statement.

5. **Control Structures**:

    1. Use one space after control structure keywords (e.g., if, for, while).
    2. Opening braces for control structures go on the same line.
    3. Parentheses must have no spaces inside them (e.g., if ($condition)).

6. **Operators**:

    1. All binary operators (e.g., +, =, ==) must have a space on either side.
    2. Unary operators (e.g., !, ++) should not have spaces.

7. **Closures and Anonymous Functions**:

    1. Closures must declare function and use with a space after each.
    2. Opening brace goes on the same line as the declaration.

8. **Comments**:

    1. Use // or /* */ for comments; avoid #.
    2. PHPDoc blocks (/** */) are recommended for classes, methods, and properties.

Inheritance & composition • Interfaces & abstract classes • Traits & namespaces • SPL & iterators • Magic methods

Inheritance & composition:

```php
class Aniket
{
    public static int $a = 10;

    public static function add()
    {
        echo self::$a;
    }
}

Aniket::add(); // Outputs: 10
```

```php
<?php
class Person
{
    protected $name;

    public function __construct($name)
    {
        $this->name = $name;
    }

    public function getName()
    {
        return $this->name;
    }
}

class Student extends Person
{
    private $grade;

    public function __construct($name, $grade)
    {
        parent::__construct($name); // Call parent constructor
        $this->grade = $grade;
    }

    public function getInfo()
    {
        return "{$this->name} is in grade {$this->grade}";
    }
}
```

Composition

- **Definition**: A class includes an instance of another class as a property, favoring “has-a” relationships over “is-a” (inheritance).
- **Use**: More flexible than inheritance; promotes loose coupling and reusability.

Inheritance vs. Composition

- **Inheritance**: Use for clear “**is-a**” relationships (e.g., Student is a Person).
- **Composition**: Use for “**has-a**” relationships or when flexibility is needed (e.g., User has an Address).
- **Best Practice**: Prefer composition over inheritance to avoid tight coupling and complex hierarchies.

Php Laravel

```php
<?php
class Address {
    private $city;

    public function __construct($city) {
        $this->city = $city;
    }

    public function getCity() {
        return $this->city;
    }
}

class User {
    private $name;
    private $address;

    public function __construct($name, Address $address) {
        $this->name = $name;
        $this->address = $address; // Composition: User "has" an
        Address
    }

    public function getInfo() {
        return "{$this->name} lives in {$this->address->getCity()}";
    }
}

$address = new Address("New York");
$user = new User("John", $address);
echo $user->getInfo(); // Output: John lives in New York
?>
```

2. Interfaces & Abstract Classes

Key Differences Between Interfaces and Abstract Classes

Feature	Interface	Abstract Class
Declaration	`interface` keyword	`abstract class` keyword
Instantiation	Cannot be instantiated	Cannot be instantiated
Methods	Only abstract methods (implicitly public)	Can have abstract and concrete methods
Properties	Only constants (no instance properties)	Can have properties
Inheritance	Supports multiple interfaces via `implements`	Single inheritance via `extends`
Access Modifiers	Methods are implicitly `public`	Methods can be `public`, `protected`, etc.
Constructors	Cannot have constructors	Can have constructors
Use Case	Define a contract for unrelated classes	Share code among related classes

Key Features of Interfaces

- All methods in an interface are implicitly `public` and abstract (no implementation is provided).
- Interfaces cannot have properties (only constants).
- A class can implement **multiple interfaces**.
- Interfaces are declared using the `interface` keyword.
- Classes that implement an interface must provide implementations for all its methods.
- Interfaces can extend other interfaces.

Key Features of Abstract Classes

- Declared with the `abstract` keyword.
- Cannot be instantiated directly.
- Can have **abstract methods** (no implementation, must be implemented by subclasses) and **concrete methods** (with implementation).
- Can have properties, constructors, and other class features.
- A class can only extend **one abstract class** (no multiple inheritance).
- Abstract methods must be `protected` or `public` (not `private`).

```php
interface Logger {
    public function log(string $message): void;
}

interface Notifier {
    public function notify(string $message): void;
}

class MessageService implements Logger, Notifier {
    public function log(string $message): void {
        echo "Logging: $message\n";
    }

    public function notify(string $message): void {
        echo "Notifying: $message\n";
    }
}

$service = new MessageService();
$service->log("Error occurred");    // Output: Logging:
$service->notify("User notified");  // Output: Notifying
```

```php
abstract class Vehicle {
    protected string $brand;

    public function __construct(string $brand) {
        $this->brand = $brand;
    }

    abstract public function startEngine(): void;

    public function getBrand(): string {
        return $this->brand;
    }
}

class Car extends Vehicle {
    public function startEngine(): void {
        echo "Car engine started for {$this->brand}\n";
    }
}

$car = new Car("Toyota");
$car->startEngine();        // Output: Car engine starte
echo $car->getBrand();      // Output: Toyota
```

3. Traits & Namespaces

Traits in PHP

Traits are a mechanism for code reuse in PHP, allowing you to share methods across multiple classes without relying on inheritance. They are particularly useful for horizontal code reuse, where functionality is shared among unrelated classes.

Key Features of Traits

- Defined using the `trait` keyword.
- Cannot be instantiated directly.
- Classes can use multiple traits using the `use` keyword.
- Traits can contain concrete methods, abstract methods, and properties.
- Resolve method conflicts using the `insteadof` and `as` operators.
- Traits can be composed (a trait can use another trait).

```php
trait Trait1 {
    public function sayHello(): string {
        return "Hello from Trait1";
    }
}

trait Trait2 {
    public function sayHello(): string {
        return "Hello from Trait2";
    }
}

class MyClass {
    use Trait1, Trait2 {
        Trait1::sayHello insteadof Trait2; // Use Trait1's sayHello
        Trait2::sayHello as sayHelloTrait2; // Rename Trait2's method
    }
}

$instance = new MyClass();
echo $instance->sayHello();        // Output: Hello from Trait1
echo $instance->sayHelloTrait2();  // Output: Hello from Trait2
```

Namespaces in PHP

Namespaces provide a way to organize code by grouping classes, interfaces, traits, and functions under a virtual directory-like structure. They prevent naming conflicts and improve code readability and maintainability.

Key Features of Namespaces

- Declared using the `namespace` keyword at the top of a file.
- Prevent naming collisions by scoping identifiers (e.g., classes, functions).
- Support hierarchical namespaces using backslashes (`\`).
- Allow aliasing with the `use` keyword to simplify long namespace references.
- Can include classes, interfaces, traits, and functions.
- PHP uses the global namespace (`\`) if no namespace is defined.

```php
namespace App\Services;

class UserService {
    public function getUser(): string {
        return "User from App\Services";
    }
}

namespace App\Controllers;

use App\Services\UserService;

class UserController {
    public function index(): string {
        $service = new UserService();
        return $service->getUser();
    }
}
```

Aliasing with `use`

To avoid writing long namespace paths, you can alias classes, interfaces, or traits.

php Collapse

```php
namespace App;

use App\Services\UserService as Service;

class Application {
    public function run(): string {
        $service = new Service();
        return $service->getUser();
    }
}
```

Key Differences Between Traits and Namespaces

Feature	Traits	Namespaces
Purpose	Code reuse across classes	Organize code and avoid naming conflicts
Declaration	`trait` keyword	`namespace` keyword
Scope	Defines reusable methods/properties	Groups classes, traits, etc., logically
Usage	Included in classes with `use`	Declared at file level; imported with `use`
Instantiation	Cannot be instantiated	Not applicable (organizational only)
Multiple Usage	Multiple traits can be used in a class	One namespace per file (typically)

Magic Methods

Magic methods are special methods with names starting and ending with double underscores (e.g., __methodName). They are automatically called by the PHP engine in response to specific events or actions, allowing developers to customize object behavior.

Common Magic Methods

1. **__construct()**
    1. **Purpose**: Called when an object is instantiated with new.
    2. **Use Case**: Initialize object properties or perform setup tasks.

```php
class MyClass {
    public function __construct($name) {
        echo "Object created with name: $name\n";
    }
}
$obj = new MyClass("Test"); // Output: Object created with name: Test
```

2. __destruct()

- **Purpose**: Called when an object is destroyed or the script ends.
- **Use Case**: Clean up resources (e.g., close database connections, release files).

```php
class MyClass {
    public function __destruct() {
        echo "Object destroyed\n";
    }
}
$obj = new MyClass(); // When $obj goes out
```

3. __get($name)

- **Purpose**: Called when accessing an undefined or inaccessible property.
- **Use Case**: Implement dynamic property access or lazy loading.

```php
class MyClass {
    private $data = [];
    public function __get($name) {
        return $this->data[$name] ?? "Property $name not found";
    }
}
$obj = new MyClass();
echo $obj->test; // Output: Property test not found
```

4. `__set($name, $value)`

- **Purpose**: Called when setting a value to an undefined or inaccessible property.
- **Use Case**: Store dynamic properties or enforce validation.

```php
class MyClass {
    private $data = [];
    public function __set($name, $value) {
        $this->data[$name] = $value;
    }
}
$obj = new MyClass();
$obj->test = "Value";
```

5. `__isset($name)`

- **Purpose**: Called when isset() or empty() is used on an undefined or inaccessible property.
- **Use Case**: Check if a dynamic property exists.

```php
class MyClass {
    private $data = ['test' => 'value'];
    public function __isset($name) {
        return isset($this->data[$name]);
    }
}
$obj = new MyClass();
echo isset($obj->test) ? "Exists" : "Not found"; // Output: Exists
```

6. `__unset($name)`

- **Purpose**: Called when unset() is used on an undefined or inaccessible property.
- **Use Case**: Remove dynamic properties.

```php
class MyClass {
    private $data = ['test' => 'value'];
    public function __unset($name) {
        unset($this->data[$name]);
    }
}
$obj = new MyClass();
unset($obj->test);
```

7. `__call($name, $arguments)`

- **Purpose**: Called when an undefined or inaccessible method is invoked.
- **Use Case**: Handle dynamic method calls or create method proxies.

```
class MyClass {
    public function __call($name, $arguments) {
        echo "Called method $name with arguments: " . implode(', ', $arguments)
    }
}
$obj = new MyClass();
$obj->unknownMethod(1, 2); // Output: Called method unknownMethod with argument
```

8. __callStatic($name, $arguments)

- **Purpose**: Called when an undefined or inaccessible static method is invoked.
- **Use Case**: Handle dynamic static method calls.

```
class MyClass {
    public static function __callStatic($name, $arguments) {
        echo "Static method $name called with: " . implode(', ', $arguments)
    }
}
MyClass::unknownStaticMethod(3, 4); // Output: Static method unknownStaticMe
```

9. __toString()

- **Purpose**: Called when an object is used as a string (e.g., in echo or print).
- **Use Case**: Define a string representation of an object.

```
class MyClass {
    public function __toString() {
        return "MyClass object";
    }
}
$obj = new MyClass();
echo $obj; // Output: MyClass object
```

10. __invoke($arguments...)

- **Purpose**: Called when an object is used as a function.
- **Use Case**: Create callable objects.

```
class MyClass {
    public function __invoke($arg) {
        echo "Object called with: $arg\n";
    }
}
$obj = new MyClass();
$obj("test"); // Output: Object called with: test
```

11. __clone()

- **Purpose**: Called when an object is cloned using clone.
- **Use Case**: Customize cloning behavior (e.g., deep copy).

```php
class MyClass {
    public $data = 42;
    public function __clone() {
        $this->data = 0; // Reset on clone
    }
}
$obj1 = new MyClass();
$obj2 = clone $obj1;
echo $obj2->data; // Output: 0
```

12. `__sleep()`

- **Purpose**: Called when an object is serialized with `serialize()`.
- **Use Case**: Specify which properties to serialize.
- **Example**:

```php
class MyClass {
    public $data = "test";
    private $secret = "hidden";
    public function __sleep() {
        return ['data']; // Only serialize $data
    }
}
$obj = new MyClass();
$serialized = serialize($obj);
```

13. `__wakeup()`

- **Purpose**: Called when an object is unserialized with `unserialize()`
- **Use Case**: Reinitialize resources or state after unserialization.
- **Example**:

```php
class MyClass {
    public $data;
    public function __wakeup() {
        $this->data = "restored";
    }
}
$obj = unserialize(serialize(new MyClass()));
echo $obj->data; // Output: restored
```

14. `__debugInfo()`

- **Purpose**: Called when `var_dump()` is used on an object.
- **Use Case**: Control which properties are shown in debugging output.
- **Example**:

```php
class MyClass {
    public $data = "test";
    private $secret = "hidden";
    public function __debugInfo() {
        return ['data' => $this->data]; // Only show $data
    }
}
$obj = new MyClass();
var_dump($obj); // Shows only $data
```

Error types • Exception handling patterns • Custom exceptions • Monolog basics • Config & env mgmt

- **Error types**

1. Error Types

PHP has three main types of errors, each with different severity and handling mechanisms:

Fatal Errors

- **Description**: Severe errors that halt script execution (e.g., calling an undefined function, class not found).
- **Handling**: Cannot be caught with try-catch. Requires checking conditions beforehand or using error_handler.
- **Example**:

```
// Fatal error: undefined function
nonExistentFunction(); // Script stops
```

Warnings

- **Description**: Non-fatal errors that allow the script to continue (e.g., missing file in include, invalid array index).
- **Example**:

```
$array = [];
echo $array[0]; // Warning: Undefined offset: 0
```

Notices

- **Description**: Informational errors for non-critical issues (e.g., accessing an undefined variable).
- **Example**:

```
echo $undefinedVar; // Notice: Undefined variable: undefinedVar
```

Error Reporting

- Control error visibility with error_reporting() or php.ini settings:

```
error_reporting(E_ALL); // Show all errors
ini_set('display_errors', '1'); // Display errors (development only)


ob_clean(); // Clean output buffer to avoid unexpected output
error_reporting(E_ALL);
ini_set('log_errors', 1);
```

```
ini_set('display_errors', 0);
ini_set('error_log', 'error_log.txt');
```

- **Exception handling patterns**

Exceptions are used to handle exceptional conditions (e.g., invalid input, database failures) in a structured way using `try`, `catch`, and `finally` blocks.

Basic Pattern

- **Try**: Code that might throw an exception.
- **Catch**: Handle specific exceptions.
- **Finally**: Executes regardless of whether an exception occurred.
- **Example**:

```php
<?php

declare(strict_types=1);

try {
    $number = 10 / 0; // Throws DivisionByZeroError
} catch (DivisionByZeroError $e) {
    echo "Error: " . $e->getMessage() . "\n";
} finally {
    echo "Cleanup executed.\n";
}
?>
```

```php
declare(strict_types=1);

function processFile(string $file): string
{
    if (!file_exists($file)) {
        throw new RuntimeException("File not found: $file");
    }
    if (!is_readable($file)) {
        throw new UnexpectedValueException("File is not readable:
        $file");
    }
    return file_get_contents($file);
}

try {
    echo processFile("nonexistent.txt");
} catch (RuntimeException $e) {
    echo "Runtime Error: " . $e->getMessage() . "\n";
} catch (UnexpectedValueException $e) {
    echo "Value Error: " . $e->getMessage() . "\n";
} catch (Exception $e) {
    echo "General Error: " . $e->getMessage() . "\n";
}
?>
```

2. **Centralized Exception Handler:**
   - **Purpose**: Define a global exception handler to catch unhandled exceptions.
   - **Use Case**: Log unexpected errors or display user-friendly messages.
   - **Pattern**:

```php
set_exception_handler(function (Throwable $e) {
    error_log("Unhandled exception: " . $e->getMessage());
    echo "An error occurred. Please try again.";
});
```

- **Custom exceptions:**

Custom exceptions allow you to create specific exception types for your application, improving clarity and handling.

Creating a Custom Exception

- Extend the Exception class or a subclass.
- Add custom properties or methods as needed.
- **Example**:

```php
<?php

declare(strict_types=1);

class ValidationException extends Exception
{
    private array $errors;

    public function __construct(string $message, array $errors =
    [], int $code = 0, ?Throwable $previous = null)
    {
        parent::__construct($message, $code, $previous);
        $this->errors = $errors;
    }

    public function getErrors(): array
    {
        return $this->errors;
    }
}
```

```php
function validateUser(array $data): void
{
    $errors = [];
    if (empty($data['name'])) {
        $errors[] = "Name is required.";
    }
    if (!filter_var($data['email'] ?? '', FILTER_VALIDATE_EMAIL)) {
        $errors[] = "Invalid email format.";
    }
    if (!empty($errors)) {
        throw new ValidationException("Validation failed.",
        $errors);
    }
}

try {
    validateUser(['name' => '', 'email' => 'invalid']);
} catch (ValidationException $e) {
    echo $e->getMessage() . "\n";
    foreach ($e->getErrors() as $error) {
        echo "- $error\n";
    }
}
?>
```

• Monolog basics

Monolog Basics

Monolog is a powerful, flexible PHP logging library that adheres to the PSR-3 standard. It's widely used to log messages (e.g., debug, info, errors) to various destinations like files, databases, or third-party services (e.g., Slack, Sentry).

Key Concepts

1. **Logger**: The main class for logging messages, identified by a channel name (e.g., "app").
2. **Handlers**: Define where logs are sent (e.g., files, streams, email, external services).
3. **Processors**: Add extra data to logs (e.g., user ID, request details).
4. **Formatters**: Control the log message format (e.g., JSON, plain text).
5. **Log Levels**: Based on RFC 5424, ranging from DEBUG (least severe) to EMERGENCY (most severe).

Installation

Install Monolog via Composer:

bash

```bash
composer require monolog/monolog
```

Php Laravel

Basic Usage

Here's a simple example of logging to a file:

php Collapse

```php
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// Create a logger instance
$logger = new Logger('app');

// Add a handler to write logs to a file
$logger->pushHandler(new StreamHandler('logs/app.log', Logger::INFO));

// Log messages
$logger->info('User logged in', ['user_id' => 123]);
$logger->error('Database connection failed', ['error' => 'Timeout']);
```

Processors

Add extra data to logs:

php

```php
$logger->pushProcessor(function ($record) {
    $record['extra']['ip'] = $_SERVER['REMOTE_ADDR'];
    return $record;
});
```

Common Handlers

- **StreamHandler**: Logs to a file or stream (e.g., `php://stderr`).
- **RotatingFileHandler**: Creates daily log files (e.g., `app-2025-09-01.log`).
- **SyslogHandler**: Sends logs to the system's syslog.
- **ErrorLogHandler**: Uses PHP's `error_log` function.
- **SlackWebhookHandler**: Sends logs to a Slack channel.

Formatters

Customize log output (e.g., JSON format):

php

```php
use Monolog\Formatter\JsonFormatter;

$handler = new StreamHandler('logs/app.log', Logger::INFO);
$handler->setFormatter(new JsonFormatter());
$logger->pushHandler($handler);
```

Log Levels

- **DEBUG:** Detailed debug info.
- **INFO:** Interesting events (e.g., user login).
- **NOTICE:** Uncommon events.
- **WARNING**: Exceptional occurrences (e.g., deprecated API usage).
- **ERROR:** Runtime errors.
- **CRITICAL:** Critical conditions (e.g., component unavailable).
- **ALERT:** Action must be taken immediately.
- **EMERGENCY:** System is unusable.

Php Laravel

```php
<?php

declare(strict_types=1);

require 'vendor/autoload.php';

use Monolog\Logger;
use Monolog\Handler\StreamHandler;

$logger = new Logger('app');
$logger->pushHandler(new StreamHandler('logs/app.log', Logger::INFO));

$logger->info('Application started.');
$logger->warning('This is a warning.');
$logger->error('An error occurred.');

try {
    throw new Exception('Test exception');
} catch (Exception $e) {
    $logger->error('Caught exception', ['message' => $e->getMessage(),
    'file' => $e->getFile(), 'line' => $e->getLine()]);
}
?>
```

```php
use Monolog\Formatter\JsonFormatter;
use Monolog\Handler\StreamHandler;
use Monolog\Logger;

require_once './vendor/autoload.php';

$formatter = new JsonFormatter();
$formatter->setJsonPrettyPrint( enable: true);
$handler = new StreamHandler( stream: 'php://stdout');
$handler->setFormatter($formatter);

$monolog = new Logger( name: 'channel1');
$monolog->pushHandler($handler);

$monolog->info( message: 'I am info message');
```

```php
<?php

use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Formatter\JsonFormatter;

require "vendor/autoload.php";

$logger = new Logger('App');
$handler = new StreamHandler("Log/App.log", Logger::INFO);    'INFO' is deprecated.

$formatter = new JsonFormatter();
$formatter->setJsonPrettyPrint(true);
$handler->setFormatter($formatter);

$logger->pushHandler($handler);


$logger->pushProcessor(function ($record) {
    $record['extra']['ip'] = 12;
    return $record;
});


$logger->info('Aniket', ['MESSAGE' => "Database Failure", "Status" => 600]);
$logger->warning('AniketW', ['MESSAGE' => "DatabaseW Failure", "Status" => 400]);
$logger->error('AniketE', ['MESSAGE' => "DatabaseE Failure", "Status" => 300]);
```

- **Configuration & Environment Management in PHP**

Configuration & Environment Management in PHP

Managing configuration and environment variables in PHP ensures your application is **secure, portable,** and **adaptable** across environments (e.g., development, staging, production).

Key Concepts

1. **Environment Variables**: Store sensitive or environment-specific settings (e.g., database credentials, API keys) outside the codebase.
2. **Configuration Files**: Store non-sensitive, structured settings (e.g., app name, log paths) in files like config.php or config.yaml.
3. **Dotenv**: A popular library to load environment variables from a .env file.

A. Environment Variable (.env):

Using Dotenv

Install `vlucas/phpdotenv`:

```bash
composer require vlucas/phpdotenv
```

Create a `.env` file in your project root:

```text
DB_HOST=localhost
DB_NAME=mydb
DB_USER=root
DB_PASS=secret
APP_ENV=development
```

Load `.env` in your PHP application:

```php
use Dotenv\Dotenv;

$dotenv = Dotenv::createImmutable(__DIR__);
$dotenv->load();

$dbHost = $_ENV['DB_HOST']; // Access env variable
echo $dbHost; // Outputs: localhost
```

B. Configuration file (config.php/config.ini):

Store settings in a PHP file (e.g., config.php) or JSON/YAML.

Php Laravel

```php
<?php

declare(strict_types=1);

return [
    'app_name' => 'MyApp',
    'debug' => true,
    'database' => [
        'host' => 'localhost',
        'name' => 'myapp_db',
        'user' => 'root',
        'password' => ''
    ]
];
?>
```

- Usage:

```php
$config = require 'config.php';
echo $config['app_name']; // Output: MyApp
```

```php
<?php

declare(strict_types=1);

require 'vendor/autoload.php';

use Dotenv\Dotenv;

$dotenv = Dotenv::createImmutable(__DIR__);
$dotenv->load();

$config = [
    'app_name' => $_ENV['APP_NAME'] ?? 'DefaultApp',
    'debug' => filter_var($_ENV['DEBUG'] ?? false,
    FILTER_VALIDATE_BOOLEAN),
    'database' => [
        'host' => $_ENV['DB_HOST'] ?? 'localhost',
        'name' => $_ENV['DB_NAME'] ?? 'default_db',
        'user' => $_ENV['DB_USER'] ?? 'root',
        'password' => $_ENV['DB_PASSWORD'] ?? ''
    ]
];

echo $config['app_name'] . "\n"; // Output: MyApp
var_dump($config['debug']); // Output: bool(true)
?>
```

2. **Validation**: Use `Dotenv::createImmutable()->safeLoad()` to ensure required env variables are set:

php Collapse Wrap Copy

```php
$dotenv->required(['DB_HOST', 'DB_NAME', 'DB_USER', 'DB_PASS']);
```

“Config.ini” **“index.php”**

```ini
[database]
APP_NAME=Aniket_App
DB_HOST = localhost
DB_USERNAME = root
DB_DATABASE = User
DB_PASSWORD =
```

```php
<?php
$config = require "config.php";
$config2 = parse_ini_file("config.ini", true);
echo "<pre>";
print_r($config);
print_r($config2['database']);
```

PDO connection • Prepared statements • Transactions • DAO pattern • Pagination

● PDO connection

1. PDO Connection

PDO (PHP Data Objects) is a database abstraction layer that provides a consistent interface for interacting with various databases (e.g., MySQL, PostgreSQL, SQLite).

Setting Up a PDO Connection

- **Requirements**: Database server (e.g., MySQL), credentials, and PDO enabled (default in PHP).
- **Steps**:
  1. Create a PDO instance with DSN (Data Source Name), username, and password.
  2. Set error mode to exceptions for better error handling.
  3. Optionally configure attributes (e.g., fetch mode).

```php
<?php

declare(strict_types=1);

namespace App\Database;

use PDO;
use PDOException;

class DatabaseConnection
{
    private PDO $pdo;

    public function __construct(string $dsn, string $username, string $password)
    {
        try {
            $this->pdo = new PDO($dsn, $username, $password, [
                PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
                PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
                PDO::ATTR_EMULATE_PREPARES => false,
            ]);
        } catch (PDOException $e) {
            throw new PDOException('Connection failed: ' . $e->getMessage(), (int) $e->getCode());
        }
    }

    public function getPdo(): PDO
    {
        return $this->pdo;
    }
}
```

Php Laravel

```php
// Usage
try {
    $dsn = 'mysql:host=localhost;dbname=myapp_db;charset=utf8mb4';
    $db = new DatabaseConnection($dsn, 'root', '');
    echo 'Connected successfully!';
} catch (PDOException $e) {
    echo 'Error: ' . $e->getMessage();
}
?>
```

OR

1. PDO Connection

PDO = a way to connect PHP to your database.

It's like plugging in a cable so PHP can talk to MySQL.

```php
<?php
$dsn = "mysql:host=localhost;dbname=testdb;charset=utf8mb4";
$user = "root";
$pass = "";

try {
    $pdo = new PDO($dsn, $user, $pass);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    echo "Connected!";
} catch (PDOException $e) {
    echo "Connection failed: " . $e->getMessage();
}
```

Key Notes

- **DSN Format**: Varies by database (e.g., ***mysql:host=localhost;dbname=myapp_db*** for MySQL).
- **Attributes**:
  - **PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION**: Throws exceptions on errors.
  - **PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC**: Returns associative arrays.
  - **PDO::ATTR_EMULATE_PREPARES** => false: Uses native prepared statements.

2. Prepared Statements

Prepared statements prevent SQL injection by separating SQL logic from user input, using placeholders (? or named :param).

Using Prepared Statements

- **Types**:
  - Positional (?): Bind values by position.
  - Named (:name): Bind values by parameter name.
- **Methods**:

- prepare($sql): Prepares the SQL statement.
- bindValue($param, $value, $type): Binds a value to a placeholder.
- execute(): Executes the prepared statement.
- fetch(), fetchAll(): Retrieve results.

```php
try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4',
    'root', '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);

    // Named parameters
    $stmt = $pdo->prepare('SELECT * FROM users WHERE email = :email AND status
    = :status');
    $stmt->bindValue(':email', 'john@example.com', PDO::PARAM_STR);
    $stmt->bindValue(':status', 'active', PDO::PARAM_STR);
    $stmt->execute();
    $user = $stmt->fetch();

    if ($user) {
        echo "Found user: {$user['name']}\n";
    } else {
        echo "No user found.\n";
    }

    // Positional parameters
    $stmt = $pdo->prepare('INSERT INTO users (name, email) VALUES (?, ?)');
    $stmt->execute(['Jane Doe', 'jane@example.com']);
    echo "User inserted.\n";
} catch (PDOException $e) {
    echo 'Error: ' . $e->getMessage();
}
?>
```

Best Practices

- Always use prepared statements for user input.
- Explicitly bind parameter types (e.g., `PDO::PARAM_INT`, `PDO::PARAM_STR`) for clarity.
- Avoid concatenating user input directly into SQL queries.
- Reuse prepared statements for performance in loops.

3. Transactions

Transactions ensure multiple database operations are executed atomically (all succeed or all fail), maintaining data integrity.

Transaction Methods

- beginTransaction(): Starts a transaction.
- commit(): Saves changes if all operations succeed.
- rollBack(): Reverts changes if an error occurs.

Php Laravel

```php
namespace App\Database;

use PDO;
use PDOException;

try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4',
    'root', '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    ]);

    $pdo->beginTransaction();

    $stmt = $pdo->prepare('UPDATE accounts SET balance = balance - ? WHERE
    user_id = ?');
    $stmt->execute([100, 1]); // Withdraw $100 from user 1

    $stmt = $pdo->prepare('UPDATE accounts SET balance = balance + ? WHERE
    user_id = ?');
    $stmt->execute([100, 2]); // Deposit $100 to user 2

    $pdo->commit();
    echo "Transaction completed.\n";
} catch (PDOException $e) {
    $pdo->rollBack();
    echo "Transaction failed: " . $e->getMessage() . "\n";
}
```

```php
try {
    $pdo->beginTransaction();

    // Take 100 from Alice
    $pdo->prepare("UPDATE accounts SET balance = balance - 100 WHERE name = ?")
        ->execute(['Alice']);

    // Add 100 to Bob
    $pdo->prepare("UPDATE accounts SET balance = balance + 100 WHERE name = ?")
        ->execute(['Bob']);

    $pdo->commit(); // ✅ everything worked
} catch (Exception $e) {
    $pdo->rollBack(); // ❌ undo changes
    echo "Error: " . $e->getMessage();
}
```

```php
catch (Throwable $e) {
    if ($pdo->inTransaction()) {
        $pdo->rollBack();
    }
    // log and rethrow or handle
    throw $e;
```

4. DAO Pattern

The **Data Access Object (DAO)** pattern separates database logic from business logic, providing a clean interface for data operations.

Key Concepts

- **DAO Class**: Encapsulates database operations for a specific entity (e.g., User).
- **Dependency**: Typically depends on a PDO connection.
- **Methods**: CRUD operations (Create, Read, Update, Delete).
- **Example** (Assumes a users table with id, name, email columns):

```php
class UserDao
{
    private PDO $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function findById(int $id): ?array
    {
        try {
            $stmt = $this->pdo->prepare('SELECT * FROM users WHERE id = :id');
            $stmt->bindValue(':id', $id, PDO::PARAM_INT);
            $stmt->execute();
            $user = $stmt->fetch();
            return $user ?: null;
        } catch (PDOException $e) {
            throw new PDOException('Failed to find user: ' . $e->getMessage());
        }
    }

    public function create(string $name, string $email): int
    {
        try {
            $stmt = $this->pdo->prepare('INSERT INTO users (name, email) VALUES (:name, :email)');
            $stmt->bindValue(':name', $name, PDO::PARAM_STR);
            $stmt->bindValue(':email', $email, PDO::PARAM_STR);
            $stmt->execute();
            return (int) $this->pdo->lastInsertId();
        } catch (PDOException $e) {
            throw new PDOException('Failed to create user: ' . $e->getMessage());
        }
    }
```

```php
}

// Usage
try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4', 'root', '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);

    $userDao = new UserDao($pdo);
    $id = $userDao->create('John Doe', 'john@example.com');
    echo "Created user with ID: $id\n";

    $user = $userDao->findById($id);
    if ($user) {
        echo "Found user: {$user['name']}\n";
    }
} catch (PDOException $e) {
    echo 'Error: ' . $e->getMessage();
}
?>
```

5. Pagination

Pagination splits large datasets into smaller pages, improving performance and user experience.

Approach

- Use SQL **LIMIT** and **OFFSET** to fetch a subset of records.
- Calculate total pages based on total records and page size.
- Pass the current page via query parameters (e.g., ?page=2).
- **Example** (Web-based pagination with a users table):

Pagination = showing results in pages (like Google search results). Instead of showing **all users at once**, you show 5 per page.

```php
try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4', 'root', '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);

    $page = isset($_GET['page']) ? max(1, (int) $_GET['page']) : 1;
    $perPage = 5;
    $offset = ($page - 1) * $perPage;

    // Get total records
    $stmt = $pdo->query('SELECT COUNT(*) FROM users');
    $totalRecords = (int) $stmt->fetchColumn();
    $totalPages = (int) ceil($totalRecords / $perPage);

    // Fetch paginated data
    $stmt = $pdo->prepare('SELECT * FROM users LIMIT :limit OFFSET :offset');
    $stmt->bindValue(':limit', $perPage, PDO::PARAM_INT);
    $stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
    $stmt->execute();
    $users = $stmt->fetchAll();

    // Output
    header('Content-Type: text/html; charset=UTF-8');
?>
```

Php Laravel

```php
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>User Pagination</title>
</head>
<body>
    <h1>Users (Page <?php echo $page; ?> of <?php echo $totalPages; ?>)</h1>
    <ul>
        <?php foreach ($users as $user): ?>
            <li><?php echo htmlspecialchars($user['name']) . ' (' . htmlspecialchars($user['email']) .
            ')'; ?></li>
        <?php endforeach; ?>
    </ul>
    <nav>
        <?php if ($page > 1): ?>
            <a href="?page=<?php echo $page - 1; ?>">Previous</a>
        <?php endif; ?>
        <?php if ($page < $totalPages): ?>
            <a href="?page=<?php echo $page + 1; ?>">Next</a>
        <?php endif; ?>
    </nav>
</body>
</html>
<?php
} catch (PDOException $e) {
    echo 'Error: ' . htmlspecialchars($e->getMessage());
}
?>
```

```php
$limit = 5;
$page = isset($_GET['page']) ? (int)$_GET['page'] : 1;
$offset = ($page - 1) * $limit;

$stmt = $pdo->prepare("SELECT * FROM users LIMIT :limit OFFSET :offset");
$stmt->bindValue(':limit', $limit, PDO::PARAM_INT);
$stmt->bindValue(':offset', $offset, PDO::PARAM_INT);
$stmt->execute();
$users = $stmt->fetchAll();

foreach ($users as $user) {
    echo $user['name'] . " - " . $user['email'] . "<br>";
}

// Count total users
$total = $pdo->query("SELECT COUNT(*) FROM users")->fetchColumn();
$totalPages = ceil($total / $limit);

// Page links
for ($i = 1; $i <= $totalPages; $i++) {
    echo "<a href='?page=$i'>$i</a> ";
}
```

XSS/CSRF/SQLi prevention • Password hashing (bcrypt/argon2) • Rate limiting (concept) • Input validation libraries • Secure file handling

- ## XSS/CSRF/SQLi prevention

1. Cross-Site Scripting (XSS) Prevention

- **Description**: XSS occurs when malicious scripts are injected into web pages viewed by users, often via user input displayed without sanitization.
- **Prevention**:
    - Escape output with htmlspecialchars() or equivalent to convert special characters (e.g., <, >) to HTML entities.
    - Use Content Security Policy (CSP) headers to restrict script sources.

- Validate and sanitize input to reject malicious content.

```php
$userInput = "<script>alert('Hacked!');</script>";
echo htmlspecialchars($userInput, ENT_QUOTES, 'UTF-8');
// Outputs: &lt;script&gt;alert('Hacked!');&lt;/script&gt;
```

Content Security Policy (CSP):

- Add a CSP header to restrict script sources:

header("Content-Security-Policy: default-src 'self'; script-src 'self' trusted.cdn.com");

```
:("Content-Security-Policy: default-src 'self'; script-src 'self' trusted.cdn.com");
```

Avoid `eval()` **or** `innerHTML`:

- Never evaluate user input as code.
- Use safe DOM manipulation (e.g., `textContent` instead of `innerHTML` in JavaScript).

Framework Helpers:

- In Laravel: Use Blade's `{{ $variable }}` (auto-escapes) instead of `{!! $variable !!}`.
- In Symfony: Use Twig's `{{ variable|e }}` for escaping.

2. CSRF (Cross-Site Request Forgery) Prevention

CSRF tricks a user's browser into making unintended requests (e.g., changing a password) using their active session.

Prevention Techniques

1. **CSRF Tokens**:

- Generate a unique, unpredictable token for each form and validate it on submission.
- Example in PHP:

```php
session_start();
// Generate token
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
?>
<form method="POST" action="/update">
    <input type="hidden" name="csrf_token" value="<?php echo htmlspecialchars($_SES
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
```

Php Laravel

Validate on server:

```php
session_start();
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
        die('CSRF validation failed');
    }
    // Process form
}
```

2. **SameSite Cookies:**

- Set the `SameSite` attribute on session cookies to prevent cross-origin requests:

```php
session_set_cookie_params([
    'lifetime' => 0,
    'path' => '/',
    'domain' => '',
    'secure' => true, // HTTPS only
    'httponly' => true, // Prevent JavaScript access
    'samesite' => 'Strict' // or 'Lax'
]);
session_start();
```

Verify HTTP Methods:

- Use POST or PUT for state-changing actions (e.g., updating data) instead of GET.
- Check the request method:

```php
if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
    http_response_code(405);
    die('Method Not Allowed');
}
```

3. SQL Injection (SQLi) Prevention

SQLi occurs when an attacker injects malicious SQL code into a query, potentially exposing or manipulating database data.

Prevention Techniques

1. **Use Prepared Statements and Parameterized Queries**:

- Avoid concatenating user input into SQL queries.
- Use PDO or MySQLi with prepared statements:

Php Laravel

```php
// PDO Example
$pdo = new PDO('mysql:host=localhost;dbname=mydb', 'user', 'pass');
$stmt = $pdo->prepare('SELECT * FROM users WHERE email = :email');
$stmt->execute(['email' => $userInput]);
$user = $stmt->fetch();
```

```php
// MySQLi Example
$mysqli = new mysqli('localhost', 'user', 'pass', 'mydb');
$stmt = $mysqli->prepare('SELECT * FROM users WHERE email = ?');
$stmt->bind_param('s', $userInput);
$stmt->execute();
$result = $stmt->get_result();
```

2. **Escape User Input (Fallback):**
   - If prepared statements aren't feasible, escape input using database-specific functions:

```php
$escaped = $mysqli->real_escape_string($userInput);
$query = "SELECT * FROM users WHERE email = '$escaped'"; // Less secure, use
```

5. **Validate and Sanitize Input:**
   - Validate input types (e.g., email format, numeric IDs):

```php
if (!filter_var($userInput, FILTER_VALIDATE_EMAIL)) {
    die('Invalid email');
}
```

- Password Hashing (bcrypt/Argon2)

Secure password hashing protects user passwords by storing them in a way that's computationally expensive to reverse, even if the database is compromised.

- **bcrypt**: A widely used, battle-tested algorithm based on Blowfish. It's the **default** in PHP's password hashing functions and is known for simplicity and security.
- **Argon2**: A newer algorithm, winner of the 2015 Password Hashing Competition. It's highly configurable, resistant to GPU-based attacks, and suitable for modern applications.

Php Laravel

1. Using bcrypt in PHP

PHP's built-in `password_hash()` and `password_verify()` functions use bcrypt by default, making it simple to implement.

Installation

No external libraries are needed—bcrypt is built into PHP (since 5.5).

Hashing a Password

php Collapse Wrap

```php
$password = "user_password123";
$hash = password_hash($password, PASSWORD_BCRYPT, ['cost' => 12]);
echo $hash;
// Example output: $2y$12$8Kj... (60-character hash with salt and cost)
```

Options:

- **cost:** Controls computational effort (default is 10). Higher values (e.g., 12) are more secure but slower. Adjust based on server performance (aim for ~100-300ms per hash).

Verifying a Password

php

```php
$userInput = "user_password123";
$storedHash = "$2y$12$8Kj..."; // From database

if (password_verify($userInput, $storedHash)) {
    echo "Password is valid!";
} else {
    echo "Invalid password.";
}
```

2. Using Argon2 in PHP

Argon2 is available in PHP 7.2+ and is the recommended choice for modern applications due to its resistance to GPU-based attacks and configurability. It has three variants:

- **Argon2i**: Optimized for password hashing (resists side-channel attacks).
- **Argon2d**: Optimized for cryptocurrency mining (faster, less common for passwords).
- **Argon2id**: Hybrid of both, recommended for most use cases.

Installation

Argon2 requires the `libsodium` extension or PHP 7.2+ with Argon2 support compiled in. Check with:

php Collapse Wrap Co

```php
if (defined('PASSWORD_ARGON2ID')) {
    echo "Argon2 is available!";
}
```

Hashing a Password

php Collapse

```php
$password = "user_password123";
$hash = password_hash($password, PASSWORD_ARGON2ID, [
    'memory_cost' => 65536, // 64MB (in kibibytes)
    'time_cost' => 4,       // Number of iterations
    'threads' => 2          // Parallelism factor
]);
echo $hash;
// Example output: $argon2id$v=19$m=65536,t=4,p=2$... (variable length)
```

- **Options:**
  - `memory_cost`: Memory usage (default: 64MB). Increase for stronger security.
  - `time_cost`: Number of iterations (default: 2). Higher is slower and more secure.
  - `threads`: Parallelism (default: 2). Adjust based on server CPU cores.

Verifying a Password

php

```php
$userInput = "user_password123";
$storedHash = "$argon2id$v=19$m=65536,t=4,p=2$...";

if (password_verify($userInput, $storedHash)) {
    echo "Password is valid!";
} else {
    echo "Invalid password.";
}
```

php

```php
$start = microtime(true);
password_hash("test", PASSWORD_BCRYPT, ['cost' => 12]);
$time = microtime(true) - $start;
echo "Hashing took $time seconds";
```

- **Rate Limiting (Concept)**

Concept of Rate Limiting

Rate limiting restricts the number of requests (e.g., API calls, login attempts, form submissions) a client can make to a server in a specific period (e.g., per second, minute, or hour). It mitigates threats like brute-force attacks, denial-of-service (DoS) attacks, and API abuse while ensuring system stability.

Key Components

1. **Identifier**: Determines who is making the request (e.g., user ID, IP address, API key).
2. **Limit**: The maximum number of allowed requests (e.g., 100 requests/hour).
3. **Time Window**: The duration for tracking requests (e.g., 1 hour, 1 minute).
4. **Action**: What happens when the limit is exceeded (e.g., block request, return HTTP **429** Too Many Requests).

How It Works

- Track requests per identifier in a storage system (e.g., in-memory like Redis, database, or session).
- For each request, check if the limit is exceeded within the time window.
- If exceeded, reject the request; otherwise, allow it and update the counter.

Implementation Options

- **Database**: Store request counts with timestamps.
- **Redis**: Use expiring keys for high-performance tracking (e.g., INCR with EXPIRE).
- **Session**: Track requests per session (less reliable for anonymous users).
- **Third-Party**: Use libraries like bandwidth-throttle/token-bucket.

Identify the client (e.g., `$_SERVER['REMOTE_ADDR']` for IP or user ID after authentication).

```php
<?php

declare(strict_types=1);

namespace App\Security;

class RateLimiter
{
    private PDO $pdo;
    private int $limit = 5;
    private int $window = 60; // Seconds

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function check(string $identifier): bool
    {
        $stmt = $this->pdo->prepare('SELECT COUNT(*) FROM requests WHERE identifier
        = :id AND timestamp > :time');
        $stmt->bindValue(':id', $identifier, PDO::PARAM_STR);
        $stmt->bindValue(':time', time() - $this->window, PDO::PARAM_INT);
        $stmt->execute();
        $count = (int) $stmt->fetchColumn();

        if ($count >= $this->limit) {
            return false;
        }

        $stmt = $this->pdo->prepare('INSERT INTO requests (identifier, timestamp)
        VALUES (:id, :time)');
        $stmt->bindValue(':id', $identifier, PDO::PARAM_STR);
        $stmt->bindValue(':time', time(), PDO::PARAM_INT);
        $stmt->execute();

        return true;
    }
}

// Usage (assumes `requests` table: identifier, timestamp)
try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4', 'root',
    '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    ]);
    $limiter = new RateLimiter($pdo);
    $ip = $_SERVER['REMOTE_ADDR'];

    if (!$limiter->check($ip)) {
        http_response_code(429);
        echo 'Too many requests.';
        exit;
    }
    echo 'Request allowed.';
} catch (Exception $e) {
    echo 'Error: ' . htmlspecialchars($e->getMessage(), ENT_QUOTES, 'UTF-8');
}
?>
```

- **Input Validation Libraries**

Input validation libraries simplify and standardize input validation, reducing errors and improving security.

Popular Libraries

- **Respect/Validation**: Flexible, chainable validation rules.
  - Install: *composer require respect/validation*
- **Symfony Validator**: Robust, reusable validator (part of Symfony).
  - Install: *composer require symfony/validator*
- **Rakkit/Validator**: Lightweight, inspired by Laravel's validator.
  - Install: *composer require rakit/validation*

- **Installation:**

bash

```bash
composer require respect/validation
```

```php
namespace App\Security;

require 'vendor/autoload.php';

use Respect\Validation\Validator as v;
use Respect\Validation\Exceptions\ValidationException;

try {
    $data = [
        'name' => $_POST['name'] ?? '',
        'email' => $_POST['email'] ?? '',
        'age' => $_POST['age'] ?? '',
    ];

    v::key('name', v::stringType()->notEmpty()->length(2, 50))
     ->key('email', v::email())
     ->key('age', v::intVal()->min(18))
     ->assert($data);

    echo 'Validation passed!';
} catch (ValidationException $e) {
    echo 'Validation errors:<br>';
    foreach ($e->getMessages() as $message) {
        echo "- {$message}<br>";
    }
}
?>
```

Php Laravel

```php
$email = $_POST['email'] ?? '';
$validator = v::email()->length(1, 255);

try {
    $validator->assert($email);
    echo "Valid email: " . htmlspecialchars($email, ENT_QUOTES, 'UTF-8');
} catch (Exception $e) {
    $logger->warning('Invalid email input', ['email' => $email, 'ip' => $_
    die('Invalid email: ' . $e->getMessage());
}
```

PHP's Built-in `filter_var()`

- **Description:** Not a library but a built-in PHP function for basic validation/sanitization.
- **Usage:**

php Collapse Wra

```php
$email = $_POST['email'] ?? '';
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
    $logger->warning('Invalid email', ['email' => $email, 'ip' => $_SERVE
    die('Invalid email');
}
echo "Valid email: " . htmlspecialchars($email, ENT_QUOTES, 'UTF-8');
```

1. Install Valitron with Composer

Run this in your project folder:

bash

```bash
composer require vlucas/valitron
```

```php
<?php
require 'vendor/autoload.php';

use Valitron\Validator;

// Example data
$data = [
    'email' => 'test@example.com',
    'age'   => 17,
    'name'  => ''
];

// Pass your input data (could be $_POST, $_GET, JSON, etc.)
$v = new Validator($data);

// Add validation rules
$v->rule('required', ['email', 'age', 'name']);
$v->rule('email', 'email');
$v->rule('integer', 'age');
$v->rule('min', 'age', 18);
$v->rule('lengthMin', 'name', 3);
```

```php
// Run validation
if ($v->validate()) {
    echo "✅ Data is valid!";
} else {
    // Get validation errors
    print_r($v->errors());
}
```

- ## Secure File Handling

Secure file handling prevents vulnerabilities like unauthorized access, malicious uploads, or directory traversal.

Key Principles

- Validate file types and sizes.
- Use secure storage locations (outside web root).
- Generate unique filenames to prevent overwrites.
- Check MIME types with finfo for reliability.

```php
use PDO;
use PDOException;

try {
    $pdo = new PDO('mysql:host=localhost;dbname=myapp_db;charset=utf8mb4', 'root',
    '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    ]);

    if ($_SERVER['REQUEST_METHOD'] === 'POST' && isset($_FILES['file'])) {
        $file = $_FILES['file'];
        $uploadDir = __DIR__ . '/uploads/';
        $allowedTypes = ['image/jpeg', 'image/png'];
        $maxSize = 2 * 1024 * 1024; // 2MB

        // Validate file error
        if ($file['error'] !== UPLOAD_ERR_OK) {
            throw new RuntimeException('Upload error: ' . $file['error']);
        }

        // Validate size
        if ($file['size'] > $maxSize) {
            throw new RuntimeException('File too large. Max 2MB.');
        }

        // Validate MIME type
        $finfo = finfo_open(FILEINFO_MIME_TYPE);
        $mime = finfo_file($finfo, $file['tmp_name']);
        finfo_close($finfo);
        if (!in_array($mime, $allowedTypes)) {
            throw new RuntimeException('Invalid file type. Only JPEG/PNG allowed.');
        }

        // Generate unique filename
        $filename = uniqid('img_', true) . '.' . pathinfo($file['name'],
        PATHINFO_EXTENSION);
        $destination = $uploadDir . $filename;

        // Move file
        if (!move_uploaded_file($file['tmp_name'], $destination)) {
            throw new RuntimeException('Failed to move file.');
        }

        // Log to database
        $stmt = $pdo->prepare('INSERT INTO uploads (filename, user_id) VALUES
        (:filename, :user_id)');
        $stmt->execute(['filename' => $filename, 'user_id' => 1]);
        echo "File uploaded: {$filename}";
    }
} catch (Exception $e) {
    http_response_code(400);
    echo 'Error: ' . htmlspecialchars($e->getMessage(), ENT_QUOTES, 'UTF-8');
}
?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>File Upload</title>
</head>
<body>
    <form method="post" enctype="multipart/form-data">
        <label for="file">Upload Image:</label>
        <input type="file" name="file" id="file">
        <input type="submit" value="Upload">
    </form>
</body>
</html>
```

- ## SPL & iterators

What is SPL?

The **Standard PHP Library (SPL)** is a collection of built-in classes, interfaces, and functions in PHP designed to solve common problems, such as data structure manipulation, iteration, and file handling. It's available by default in PHP 5.0+ and includes:

- **Data Structures**: Classes like ***SplFixedArray, SplStack, SplQueue, SplHeap,*** etc.
- **Iterators**: Interfaces and classes for iterating over data (e.g., ***arrays, files, directories***).
- **File Handling**: Classes like **SplFileObject** for simplified file operations.
- **Interfaces**: **Iterator, IteratorAggregate, Countable**, etc., for custom iteration.
- **Autoloading:** Support for custom autoloaders.(**spl_autoload_register()**)

SPL Iterators

Iterators allow you to traverse data structures (arrays, objects, files, etc.) in a standardized way, implementing the **Iterator** or **IteratorAggregate** interfaces. SPL

provides several iterator classes to handle specific use cases, making it easier to process collections or files securely and efficiently.

Core Iterator Interfaces

- **Iterator**: Defines methods for iteration (current(), key(), next(), rewind(), valid()).
- **IteratorAggregate**: Allows objects to define a custom iterator via getIterator().

Instead of using traditional foreach over arrays, iterators let you iterate over complex objects in a consistent way.

📌 Basic Iterator Interface

The core is the `Iterator` **interface**, which defines 5 methods you must implement:

```php
interface Iterator extends Traversable {
    public function current();   // Return current element
    public function key();       // Return current key
    public function next();      // Move forward to next element
    public function rewind();    // Rewind back to first element
    public function valid();     // Check if current position is valid
}
```

```php
<?php
class MyCollection implements Iterator {
    private $items = [];
    private $position = 0;

    public function __construct(array $items) {
        $this->items = $items;
    }

    public function current() {
        return $this->items[$this->position];
    }

    public function key() {
        return $this->position;
    }

    public function next() {
        ++$this->position;
    }

    public function rewind() {
        $this->position = 0;
    }

    public function valid() {
        return isset($this->items[$this->position]);
    }
}

$collection = new MyCollection(["Apple", "Banana", "Orange"]);
foreach ($collection as $item) {
    echo $item . "\n"; // Output: Apple Banana Orange
}
?>
```

Php Laravel

```php
<?php
class MyCollection implements IteratorAggregate {
    private $items;

    public function __construct(array $items) {
        $this->items = $items;
    }

    public function getIterator() {
        return new ArrayIterator($this->items);
    }
}

$collection = new MyCollection(["Apple", "Banana", "Orange"]);
foreach ($collection as $item) {
    echo $item . "\n"; // Output: Apple Banana Orange
}
?>
```

Key SPL Iterator Classes

- `ArrayIterator`: Iterates over arrays or objects.
- `DirectoryIterator`: Iterates over files and directories in a filesystem.
- `SplFileObject`: Iterates over lines in a file.
- `RecursiveArrayIterator`: Iterates over nested arrays.
- `LimitIterator`: Limits the number of items in iteration.
- `FilterIterator`: Filters items during iteration based on a condition.

➢ Common SPL Iterator Classes

1. **ArrayIterator**
   - Iterates over arrays/objects like a wrapper.

```php
$array = ["a" => 1, "b" => 2, "c" => 3];
$iterator = new ArrayIterator($array);

foreach ($iterator as $key => $value) {
    echo "$key => $value\n";
}
```

2. **FilterIterator:**
   - Filters items during iteration based on a custom condition.
   - **Usage:**

```php
class EmailFilterIterator extends FilterIterator {
    public function accept() {
        return filter_var($this->current(), FILTER_VALIDATE_EMAIL);
    }
}

$emails = ['alice@example.com', 'invalid', 'bob@example.com'];
$iterator = new EmailFilterIterator(new ArrayIterator($emails));

foreach ($iterator as $email) {
    $logger->info('Valid email', ['email' => $email]);
    echo htmlspecialchars($email, ENT_QUOTES, 'UTF-8') . "\n";
}
```

3. **RecursiveArrayIterator:**
   - Iterates over nested arrays.
   - **Usage:**

```php
$data = ['users' => ['Alice' => ['email' => 'alice@example.com'], 'Bob' => ['email'
$iterator = new RecursiveIteratorIterator(new RecursiveArrayIterator($data));

foreach ($iterator as $key => $value) {
    $logger->info('Nested data', ['key' => $key, 'value' => $value]);
    echo htmlspecialchars("$key: $value\n", ENT_QUOTES, 'UTF-8');
}
```

4. **DirectoryIterator:**
   - Iterates over files and directories in a filesystem.
   - **Usage** (with secure file handling):

php Collapse Wrap Copy

```php
$dir = new DirectoryIterator(__DIR__ . '/uploads');
foreach ($dir as $fileInfo) {
    if ($fileInfo->isFile() && in_array($fileInfo->getExtension(), ['jpg', 'png']))
        $logger->info('Processing file', ['file' => $fileInfo->getFilename()]);
        echo htmlspecialchars($fileInfo->getFilename(), ENT_QUOTES, 'UTF-8') . "\n";
    }
}
```

5. **LimitIterator:**
   - Limits the number of items iterated (useful for pagination or rate limiting).
   - **Usage:**

php Collapse Wrap

```php
$data = range(1, 100);
$iterator = new LimitIterator(new ArrayIterator($data), 0, 5); // First 5 items

foreach ($iterator as $number) {
    $logger->info('Limited item', ['number' => $number]);
    echo $number . "\n";
}
```

6. **RegexIterator:**
   - Filters items based on a regular expression.
   - **Usage:**

php Collapse Wrap

```php
$files = ['image.jpg', 'doc.pdf', 'photo.png'];
$iterator = new RegexIterator(new ArrayIterator($files), '/\.(jpg|png)$/i');

foreach ($iterator as $file) {
    $logger->info('Valid image file', ['file' => $file]);
    echo htmlspecialchars($file, ENT_QUOTES, 'UTF-8') . "\n";
}
```

RecursiveDirectoryIterator

- Iterates directories **recursively**.

php

```php
$dir = new RecursiveDirectoryIterator(".");
$iterator = new RecursiveIteratorIterator($dir);

foreach ($iterator as $file) {
    echo $file . "\n";
}
```

SPL Data Structures

SPL provides data structures that work well with iterators, enhancing collection handling:

1. **SplFixedArray**:

- Fixed-size array, faster than native arrays for large datasets.
- Usage:

```php
$array = new SplFixedArray(3);
$array[0] = 'Alice';
$array[1] = 'Bob';
$array[2] = 'Charlie';

foreach ($array as $item) {
    echo htmlspecialchars($item, ENT_QUOTES, 'UTF-8') . "\n";
}
```

2. **SplDoublyLinkedList**:

- A linked list supporting FIFO/LIFO iteration.

```php
$list = new SplDoublyLinkedList();
$list->push('Alice');
$list->push('Bob');

foreach ($list as $item) {
    echo htmlspecialchars($item, ENT_QUOTES, 'UTF-8') . "\n";
}
```

3. **SplStack:**

```php
<?php
$stack = new SplStack();
$stack->push("Apple");
$stack->push("Banana");
$stack->push("Orange");

foreach ($stack as $item) {
    echo $item . "\n"; // Output: Orange Banana Apple
}
?>
```

Php Laravel

WEEK-3

Relational modeling • Keys & constraints • 1NF/2NF/3NF • ER diagrams • Data types & collation

- ## Relational modeling

Relational Modeling is the process of designing a database using tables (relations) to store data, where each table represents an entity, and relationships between entities are defined using keys.

Key Concepts

- **Entities**: Real-world objects (e.g., Users, Orders) represented as tables.
- **Attributes**: Properties of entities (e.g., name, email) represented as columns.
- **Relationships**: Connections between entities (e.g., a User places many Orders).
  - **One-to-One**: Each record in Table A relates to one record in Table B (e.g., User to Profile).
  - **One-to-Many**: One record in Table A relates to multiple records in Table B (e.g., User to Orders).
  - **Many-to-Many**: Multiple records in Table A relate to multiple records in Table B, requiring a junction table (e.g., Students to Courses via Enrollments).

Example

Consider an e-commerce database:

- **Entities:** `Users`, `Orders`, `Products`, `Order_Items` (junction table for many-to-many between `Orders` and `Products`).
- **Relationships:**
  - A `User` has many `Orders` (one-to-many).
  - An `Order` contains many `Products` via `Order_Items` (many-to-many).

```sql
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    order_date DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE products (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);

CREATE TABLE order_items (
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders(id),
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Keys & Constraints

Keys and constraints enforce data integrity and define relationships in a relational database.

Keys

- **Primary Key**: A unique identifier for each row in a table (e.g., StudentID in a Students table). It must be unique and non-null.
- **Foreign Key**: A column (or set of columns) in one table that references the primary key of another table, establishing a relationship (e.g., DepartmentID in Students referencing DepartmentID in Departments).
- **Candidate Key**: A set of attributes that could uniquely identify a row (e.g., Email or StudentID could both be candidate keys).
- **Composite Key**: A primary key made of multiple columns (e.g., OrderID + ProductID in an OrderDetails table).

1. Candidate Key

- **Definition:** Any column (or set of columns) that can uniquely identify a row.
- A table can have **multiple candidate keys**.
- From these, **one is chosen as the Primary Key**.

Example:

```sql
Employees(emp_id, email, phone)
```

- Candidate Keys = `{emp_id}`, `{email}`, `{phone}` (all three uniquely identify employees).

2. Primary Key

- **Definition:** The candidate key that is **chosen** as the main unique identifier.
- Each table can have **only one primary key**.
- Cannot be `NULL`.

Example:

```sql
PRIMARY KEY (emp_id)
```

Here, `emp_id` is the primary key (chosen from the candidate keys).

3. Composite Key

- **Definition:** A key made up of **two or more columns** that together uniquely identify a row.
- Used when **no single column** can uniquely identify records.

Example (student-course enrollments):

```sql
Enrollments(student_id, course_id, grade)
```

- Neither `student_id` alone nor `course_id` alone is unique.
- But `(student_id, course_id)` together is unique → **Composite Key**.

```sql
PRIMARY KEY (student_id, course_id)
```

Constraints

- **NOT NULL**: Ensures a column cannot be null.
- **UNIQUE**: Ensures column values are unique.
- **FOREIGN KEY**: Ensures valid references to another table.
- **CHECK**: Enforces a condition (e.g., price > 0) (supported in MySQL 8.0+).
- **DEFAULT**: Sets a default value for a column.
- **PRIMARY KEY**: Combines NOT NULL and UNIQUE to uniquely identify rows.

```sql
CREATE TABLE products (
    id INT NOT NULL,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL DEFAULT 0 CHECK(price > 100),
    PRIMARY KEY (id, name)
);
```

- ## Normalization (1NF, 2NF, 3NF)

Normalization organizes data to eliminate redundancy and anomalies (insertion, deletion, update) by decomposing tables into smaller, related tables. It uses normal forms (NF).

First Normal Form (1NF)

- **Rule:** All attributes must be atomic (indivisible), and each table must have a primary key with no repeating groups.
- **Example:**
  - **Non-1NF:** A `Students` table with a column `Courses` containing multiple values like "Math, Physics".
  - **1NF Fix:** Split `Courses` into a separate table with one course per row, linked by `StudentID`.
  - **Result:**

text Collapse Wrap

```text
Students: [StudentID, Name]
StudentCourses: [StudentID, Course]
```

Second Normal Form (2NF)

- **Rule:** Must be in 1NF, and all non-key attributes must depend on the entire primary key (no partial dependency). Applies to tables with composite keys.
- **Example:**
  - **Non-2NF:** `OrderDetails [OrderID, ProductID, OrderDate, ProductName]` where `OrderDate` depends only on `OrderID` and `ProductName` depends only on `ProductID`.
  - **2NF Fix:** Split into:

text Collapse Wrap Co

```text
Orders: [OrderID, OrderDate]
Products: [ProductID, ProductName]
OrderDetails: [OrderID, ProductID]
```

Third Normal Form (3NF)

- **Rule:** Must be in 2NF, and no non-key attribute depends on another non-key attribute (no transitive dependency).
- **Example:**
  - **Non-3NF:** `Employees [EmployeeID, DepartmentID, DepartmentName]` where `DepartmentName` depends on `DepartmentID`, not directly on `EmployeeID`.
  - **3NF Fix:** Split into:

text Collapse Wrap Cop

```text
Employees: [EmployeeID, DepartmentID]
Departments: [DepartmentID, DepartmentName]
```

➢ First Normal Form (1NF)

- **Rules**:
  - All attributes must be atomic (no multi-valued attributes).
  - Each column must have a single value.
  - No repeating groups or arrays in a single column.
- **Example (Non-1NF to 1NF)**:
  - **Non-1NF**: A table with a phones column storing multiple numbers (e.g., "123-456,789-012").
  - **1NF**: Split into a separate user_phones table.

➢ Second Normal Form (2NF)

- **Rules**:
  - Must be in 1NF.
  - All non-key attributes must depend on the entire primary key (no partial dependency).
- **Example**:
  - **Non-2NF**: A table order_details with order_id, product_id, product_name, where product_name depends only on product_id.
  - **2NF**: Move product_name to a separate products table.

➢ Third Normal Form (3NF)

- **Rules**:
  - Must be in 2NF.
  - No transitive dependencies (non-key attributes must not depend on other non-key attributes).
- **Example**:
  - **Non-3NF**: A users table with id, name, city, country, where city determines country.
  - **3NF**: Move city and country to a separate cities table.

● ER Diagrams

Entity-Relationship (ER) diagrams visually represent the database structure, showing entities, attributes, and relationships.

- **Components**:
  - **Entities**: Objects (e.g., Student, Course) represented as rectangles.
  - **Attributes**: Properties of entities (e.g., StudentID, Name) shown as ovals.
  - **Relationships**: Connections between entities (e.g., Enrolls) shown as diamonds.
  - **Cardinality**: Specifies relationship constraints (e.g., 1:1, 1:N, M:N):
    - **One-to-One** (1:1): One Student has one Passport.
    - **One-to-Many** (1:N): One Department has many Employees.
    - **Many-to-Many** (M:N): Students enroll in many Courses, resolved using a junction table (e.g., StudentCourses).

- **Example ER Diagram:**

```text
[Student] --- Enrolls ---> [Course]
| StudentID (PK)       | CourseID (PK)
| Name                 | CourseName
```

Data Types & Collation

1. Data Types

Choosing appropriate data types ensures efficient storage and querying. Common MySQL data types:

- **Numeric**:
  - **INT:** Whole numbers (e.g., id).
  - **DECIMAL(M, D)**: Fixed-point numbers (e.g., price DECIMAL(10, 2) for $1234.56).
  - **FLOAT, DOUBLE**: Floating-point numbers (less precise).
- **String**:
  - **VARCHAR(N):** Variable-length strings (e.g., name VARCHAR(255)).
  - **TEXT**: Large text (e.g., descriptions).
  - **CHAR(N)**: Fixed-length strings (e.g., country_code CHAR(2)).
- **Date/Time**:
  - **DATETIME:** Date and time (e.g., 2025-09-04 13:06:00).
  - **DATE:** Date only.
  - **TIMESTAMP:** Auto-updates on changes.
- **Boolean**: **TINYINT(1)** (0 or 1, e.g., is_active).
- **Binary**: **BLOB** for binary data (e.g., images).

2. Collation

Collation defines how string data is compared and sorted, affecting case sensitivity, accent sensitivity, and language-specific rules.

- **Examples**:
  - **utf8mb4_unicode_ci**: Case-insensitive, supports Unicode (e.g., "Apple" = "apple").

- **utf8mb4_bin**: Binary comparison, case-sensitive (e.g., "Apple" ≠ "apple").
- **latin1_swedish_ci**: Case-insensitive, Swedish-specific sorting rules.

- **Use Cases**:
  - Choose **ci** (case-insensitive) for user-facing fields like usernames.
  - Use **bin** for case-sensitive fields like passwords.
  - Match collation to the application's language and region (e.g., **utf8mb4_spanish_ci** for Spanish).

- **Setting Collation**:
  - Database level: `CREATE DATABASE mydb CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;`
  - Table/Column level: `CREATE TABLE users (name VARCHAR(50) COLLATE utf8mb4_bin);`

Common Collations (MySQL):

- `utf8mb4_unicode_ci` : Case-insensitive, supports Unicode (emojis, multi-language).
- `utf8mb4_bin` : Binary (exact match, case-sensitive).
- `latin1_swedish_ci` : Legacy, limited character support.

```sql
CREATE DATABASE ecommerce_db
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    is_active TINYINT(1) DEFAULT 1,
    profile_picture BLOB,
    country_code CHAR(2) DEFAULT 'US'
) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

4. Examples

```sql
-- Create table with collation
CREATE TABLE users (
    id INT PRIMARY KEY,
    username VARCHAR(50) COLLATE utf8mb4_general_ci
);

-- Case insensitive search
SELECT * FROM users WHERE username = 'JOHN'; -- matches 'john'

-- Case sensitive collation
ALTER TABLE users MODIFY username VARCHAR(50) COLLATE utf8mb4_bin;
SELECT * FROM users WHERE username = 'JOHN'; -- does NOT match 'john'
```

Php Laravel

SELECT/INSERT/UPDATE/DELETE • INNER/LEFT/RIGHT joins • GROUP BY/HAVING • Views • Subqueries

- ## SQL Operations

SELECT

Retrieves data from one or more tables.

- **Syntax:** `SELECT column1, column2 FROM table WHERE condition;`
- **Example:**

sql

```sql
SELECT Name, StudentID
FROM Students
WHERE DepartmentID = 1;
```

INSERT

Adds new rows to a table.

- **Syntax:** `INSERT INTO table (column1, column2) VALUES (value1, value2);`
- **Example:**

sql Collapse

```sql
INSERT INTO Students (StudentID, Name, DepartmentID)
VALUES (101, 'Alice Smith', 1);
```

UPDATE

Modifies existing rows.

- **Syntax:** `UPDATE table SET column1 = value1 WHERE condition;`
- **Example:**

sql

```sql
UPDATE Students
SET Name = 'Alice Johnson'
WHERE StudentID = 101;
```

DELETE

Removes rows from a table.

- **Syntax:** `DELETE FROM table WHERE condition;`
- **Example:**

sql

```sql
DELETE FROM Enrollments
WHERE StudentID = 101 AND CourseID = 201;
```

- ## Joins

Joins combine rows from multiple tables based on a related column.

INNER JOIN

Returns only matching rows from both tables.

- **Syntax:** `SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;`
- **Example:**

sql Collapse Wrap

```sql
SELECT s.Name, d.DepartmentName
FROM Students s
INNER JOIN Departments d ON s.DepartmentID = d.DepartmentID;
```

- Lists students and their department names, excluding students without a department.

LEFT JOIN (LEFT OUTER JOIN)

Returns all rows from the left table and matching rows from the right table (non-matches get `NULL`).

- **Syntax:** `SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;`
- **Example:**

sql Collapse Wrap

```sql
SELECT s.Name, d.DepartmentName
FROM Students s
LEFT JOIN Departments d ON s.DepartmentID = d.DepartmentID;
```

- Lists all students, including those without a department (where `DepartmentName` is `NULL`).

RIGHT JOIN (RIGHT OUTER JOIN)

Returns all rows from the right table and matching rows from the left table (non-matches get `NULL`).

- **Syntax:** `SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;`
- **Example:**

sql Collapse Wrap

```sql
SELECT s.Name, d.DepartmentName
FROM Students s
RIGHT JOIN Departments d ON s.DepartmentID = d.DepartmentID;
```

- Lists all departments, including those without students (where `Name` is `NULL`).

GROUP BY and HAVING

GROUP BY

Groups rows with the same values in specified columns into summary rows, often used with aggregate functions (**COUNT, SUM, AVG**, etc.).

- **Syntax:** `SELECT column, AGGREGATE(column) FROM table GROUP BY column;`
- **Example:**

sql Collapse

```sql
SELECT DepartmentID, COUNT(*) AS StudentCount
FROM Students
GROUP BY DepartmentID;
```

- Counts students per department.

HAVING

Filters grouped results (like **WHERE** for **GROUP BY**).

- **Syntax:** `SELECT column, AGGREGATE(column) FROM table GROUP BY column HAVING condition;`
- **Example:**

sql Collapse Wrap

```sql
SELECT DepartmentID, COUNT(*) AS StudentCount
FROM Students
GROUP BY DepartmentID
HAVING COUNT(*) > 5;
```

Views :

A view is a virtual table based on a SELECT query, stored for reuse but not physically storing data.

A saved query that acts like a virtual table.

- **Syntax**: *CREATE VIEW view_name AS SELECT ...;*

- **Example:**

```sql
CREATE VIEW StudentDepartments AS
SELECT s.Name, d.DepartmentName
FROM Students s
INNER JOIN Departments d ON s.DepartmentID = d.DepartmentID;
```

- **Usage**: Query like a table: SELECT * FROM StudentDepartments;
- **Benefits**:
  - Simplifies complex queries.
  - Enhances security by restricting access to specific columns.
  - Does not store data, so updates dynamically with the underlying tables.

4. Views

A **saved query** that acts like a virtual table.

```sql
-- Create a view
CREATE VIEW expensive_products AS
SELECT name, price
FROM products
WHERE price > 10000;

-- Use the view
SELECT * FROM expensive_products;
```

Subqueries

A subquery is a query nested within another query, used to return data for the outer query.

- **Types**:
  - **Single-row**: Returns one row (used with operators like =).
  - **Multi-row**: Returns multiple rows (used with IN, ANY, ALL).
  - **Correlated**: References the outer query, executed for each row.

- **Example (Single-row):**

sql Collapse Wrap Copy

```sql
SELECT Name
FROM Students
WHERE DepartmentID = (SELECT DepartmentID FROM Departments WHERE DepartmentName = 'Compu
```

- Finds students in the Computer Science department.

- **Example (Multi-row):**

sql Collapse Wrap

```sql
SELECT Name
FROM Students
WHERE StudentID IN (SELECT StudentID FROM Enrollments WHERE CourseID = 201);
```

- **Example (Correlated):**

sql Collapse

```sql
SELECT Name
FROM Students s
WHERE EXISTS (
    SELECT 1
    FROM Enrollments e
    WHERE e.StudentID = s.StudentID AND e.CourseID = 201
);
```

- Finds students enrolled in Course 201 (more efficient than `IN` for large datasets).

5. Subqueries

A query inside another query.

In WHERE

sql

```sql
SELECT name
FROM products
WHERE price > (SELECT AVG(price) FROM products);
```

In FROM (derived table)

sql

```sql
SELECT dept, avg_salary
FROM (
    SELECT department_id AS dept, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department_id
) t
WHERE avg_salary > 50000;
```

In SELECT

sql Copy

```sql
SELECT name,
       (SELECT COUNT(*) FROM Orders o WHERE o.customer_id = c.customer_id) AS total_orders
FROM Customers c;
```

Indexes (BTREE/Fulltext) • EXPLAIN & Query Plan • Covering indexes • Transactions & isolation • Backups (mysqldump)

Indexes (BTREE/Fulltext)

Indexes improve query performance by allowing faster data retrieval, but they increase storage and slow down write operations (**INSERT, UPDATE, DELETE**).

BTREE Index

- **Description**: A balanced tree index, organizing data in a tree structure for efficient searching, sorting, and range queries. Most common index type in relational databases (e.g., MySQL).
- **Use Case**: Suitable for exact matches, range queries (`>`, `<`, `BETWEEN`), and sorting (`ORDER BY`).
- **Syntax**:

sql Collapse Wrap Copy

```sql
CREATE INDEX idx_student_name ON Students (Name);
```

- **Example**:

sql Collapse Wrap Copy

```sql
SELECT * FROM Students WHERE Name = 'Alice Johnson';
```

  - A BTREE index on `Name` speeds up this query by reducing the rows scanned.

Fulltext Index

- **Description**: Optimized for searching text fields, supporting natural language searches, relevance ranking, and boolean queries. Common in MySQL for `TEXT` columns.
- **Use Case**: Text search in large text fields (e.g., course descriptions).
- **Syntax**:

sql Collapse Wrap Copy

```sql
CREATE FULLTEXT INDEX idx_course_desc ON Courses (Description);
```

- **Example**:

sql Collapse Wrap Copy

```sql
SELECT CourseName
FROM Courses
WHERE MATCH(Description) AGAINST('database systems' IN BOOLEAN MODE);
```

  - A FULLTEXT index enables efficient searching for "database systems" in course descriptions.

1. Indexes

Indexes = data structures that speed up lookups (like an index in a book).

Types in MySQL

- **BTREE (default)**
  - Used for `=`, `<`, `>`, `BETWEEN`, `ORDER BY`, `GROUP BY`.
  - Example: indexing `id`, `name` fields.

```sql
CREATE INDEX idx_name ON products(name);
```

- **FULLTEXT**
  - Used for searching text (`MATCH … AGAINST`).
  - Works on `CHAR`, `VARCHAR`, `TEXT`.

```sql
CREATE FULLTEXT INDEX idx_description ON products(description);

SELECT * FROM products
WHERE MATCH(description) AGAINST('gaming laptop');
```

- **EXPLAIN & Query Plan**

The EXPLAIN command in MySQL shows how a query is executed, including which indexes are used and how tables are scanned.

- **Syntax:** `EXPLAIN SELECT ...;`
- **Key Columns in Output:**
  - `id`: Query execution step.
  - `select_type`: Type of query (e.g., `SIMPLE`, `SUBQUERY`).
  - `table`: Table accessed.
  - `type`: Access method (e.g., `ALL` for full table scan, `index` for index scan, `ref` for non-unique index lookup).
  - `possible_keys`: Indexes that could be used.
  - `key`: Index actually used.
  - `rows`: Estimated rows scanned.
  - `Extra`: Additional info (e.g., `Using index`, `Using where`).

- **Example:**

sql Collapse Wrap Copy

```sql
EXPLAIN SELECT s.Name, d.DepartmentName
FROM Students s
JOIN Departments d ON s.DepartmentID = d.DepartmentID
WHERE s.Name = 'Alice Johnson';
```

- **Output (simplified):**

text Collapse Wrap Copy

```text
id | select_type | table | type | possible_keys | key            | rows | Extra
1  | SIMPLE      | s     | ref  | idx_student_name | idx_student_name | 1  | Using wh
1  | SIMPLE      | d     | eq_ref | PRIMARY       | PRIMARY        | 1  |
```

- **Interpretation:** Uses `idx_student_name` for `Students` and `PRIMARY` key for `Departments`, scanning ~1 row each.

🔹 2. EXPLAIN & Query Plan

- Shows **how MySQL executes a query** (which indexes are used, join order, etc.).

sql

```sql
EXPLAIN SELECT * FROM products WHERE name = 'Laptop';
```

Output key columns:

- `type` → access method (`ALL` = full scan, `ref` / `range` = index used, `const` = best).
- `key` → which index is used.
- `rows` → how many rows MySQL expects to scan.

👉 Always aim to avoid `ALL` (full table scans) on large tables.

Covering Indexes

A covering index satisfies a query entirely from the index without accessing the table data, improving performance.

- **How It Works**: The index includes all columns used in the **SELECT, WHERE, JOIN,** and **GROUP BY** clauses.
- **Syntax**: Create a composite index covering multiple columns.

🔹 3. Covering Indexes

- An index that contains **all the columns a query needs**.
- MySQL can answer the query **just from the index** (no table lookup).

Example:

sql

```sql
CREATE INDEX idx_products_name_price ON products(name, price);

-- This query is covered by the index
SELECT name, price FROM products WHERE name = 'Laptop';
```

- **Syntax**: Create a composite index covering multiple columns.

```sql
CREATE INDEX idx_student_dept ON Students (DepartmentID, Name);
```

- **Example**:

```sql
SELECT Name
FROM Students
WHERE DepartmentID = 1;
```

  - If `idx_student_dept` exists, MySQL can retrieve `Name` and `DepartmentID` directly from the index (`Extra: Using index` in `EXPLAIN`).
- **Benefits**:
  - Reduces disk I/O by avoiding table access.
  - Faster for read-heavy queries.
- **Trade-off**: Larger indexes increase storage and slow writes.

- # Transactions & Isolation Levels

  - ## Transactions

A transaction is a sequence of operations treated as a single unit, ensuring data consistency (**ACID** properties: ***Atomicity, Consistency, Isolation, Durability***).

- **Syntax** (MySQL):

```sql
START TRANSACTION;
INSERT INTO Students (StudentID, Name, DepartmentID) VALUES (102, 'Bob Lee', 1);
INSERT INTO Enrollments (StudentID, CourseID, EnrollmentDate) VALUES (102, 201, '2025-09-
COMMIT; -- or ROLLBACK if error
```

  - Ensures both inserts succeed or neither is applied.

- **Key Commands:**
  - `START TRANSACTION`: Begins a transaction.
  - `COMMIT`: Saves changes.
  - `ROLLBACK`: Undoes changes if an error occurs.
  - `SAVEPOINT name`: Sets a point to roll back to (e.g., `ROLLBACK TO name`).

Isolation Levels

Isolation levels control how transactions interact, balancing consistency and performance.

- **Levels (MySQL):**
  - **READ UNCOMMITTED:** Allows dirty reads (seeing uncommitted changes). Rarely used due to inconsistency risks.
  - **READ COMMITTED:** Only committed data is visible, but non-repeatable reads are possible.
  - **REPEATABLE READ:** Ensures consistent reads within a transaction (MySQL's default for `InnoDB`).
  - **SERIALIZABLE:** Strictest, preventing concurrent transactions from interfering, but slowest.

- **Syntax:**

```sql
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

- **Example:**

```sql
START TRANSACTION;
SELECT COUNT(*) FROM Students WHERE DepartmentID = 1; -- Reads 10
-- Another transaction inserts a student
SELECT COUNT(*) FROM Students WHERE DepartmentID = 1; -- Still reads 10 in REPEATABLE RE
COMMIT;
```

4. Transactions & Isolation Levels

Transactions = group of SQL statements that run as **one atomic unit** (all succeed or none succeed).

```sql
START TRANSACTION;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
UPDATE accounts SET balance = balance + 500 WHERE id = 2;
COMMIT;
```

- If error → `ROLLBACK;`

Isolation Levels

Control how concurrent transactions interact:

1. **READ UNCOMMITTED** → dirty reads allowed (fastest, least safe).
2. **READ COMMITTED** → only committed data visible.
3. **REPEATABLE READ (MySQL default)** → consistent snapshot during transaction.
4. **SERIALIZABLE** → strict, locks everything (safest, slowest).

- **Backups (mysqldump)**

mysqldump is a MySQL utility for creating logical backups (SQL scripts) of databases, tables, or schemas.

- **Syntax:**

```bash
mysqldump -u [user] -p [database] > backup.sql
```

- **Example:**

```bash
mysqldump -u root -p university > university_backup_20250904.sql
```

- Backs up the entire `university` database to a file.

Advanced Options

- Specific tables: `mysqldump -u root -p ecommerce_db users orders > tables_backup.sql`
- No data (schema only): `mysqldump -u root -p --no-data ecommerce_db > schema.sql`
- Compress: `mysqldump -u root -p ecommerce_db | gzip > backup.sql.gz`
- Restore compressed: `gunzip < backup.sql.gz | mysql -u root -p ecommerce_db`

Backup

bash Copy code

```bash
mysqldump -u root -p mydb > mydb_backup.sql
```

Backup with schema only

bash Copy code

```bash
mysqldump -u root -p --no-data mydb > schema.sql
```

Backup with data only

bash Copy code

```bash
mysqldump -u root -p --no-create-info mydb > data.sql
```

Restore

bash Copy code

```bash
mysql -u root -p mydb < mydb_backup.sql
```

Selectors & events • DOM manipulation • AJAX (jQuery & fetch) • JSON encode/decode in PHP • Client-side form validation

Selectors & Events

Selectors:

Selectors allow you to target HTML elements using JavaScript or jQuery, while **events** handle user interactions like clicks or keypresses.

Selectors

- **CSS Selectors:** Used to target elements by ID (`#id`), class (`.class`), tag, attribute, etc.
- **jQuery:** `$('selector')` (e.g., `$('#myId')`, `$('.myClass')`).
- **Vanilla JS:** `document.querySelector('selector')` or `document.getElementById('id')`.

Events

- Common events: `click`, `change`, `submit`, `mouseover`, `keyup`.
- **jQuery:** `$(selector).on('event', callback)`.
- **Vanilla JS:** `element.addEventListener('event', callback)`.

```html
<script>
    // Select by ID
    $('#myButton').on('click', function() {
        $('#output').text('Button clicked!');
    });

    // Select by class
    $('.btn').on('mouseover', function() {
        $(this).css('background-color', '#e0e0e0');
    });

    // Select by data attribute
    $('[data-action="clickMe"]').on('mouseout', function() {
        $(this).css('background-color', '');
    });
</script>
```

- **jQuery Examples:**

javascript Collapse Wrap

```javascript
// Select by ID
$('#studentForm'); // Targets <form id="studentForm">

// Select by class
$('.error'); // Targets all elements with class="error"

// Select by tag
$('input'); // Targets all <input> elements

// Select by attribute
$('input[name="studentName"]'); // Targets input with name="studentName"
```

- **Native JavaScript:**

javascript Collapse Wrap Run

```javascript
document.getElementById('studentForm'); // ID selector
document.querySelector('.error'); // CSS-style selector
document.querySelectorAll('input[name="studentName"]'); // Multiple elements
```

Events

Events trigger JavaScript/jQuery code when user actions (e.g., clicks, form submissions) occur.

Php Laravel

- jQuery Examples:

javascript

```javascript
// Click event
$('#submitBtn').click(function() {
    alert('Form submitted!');
});

// Form submission
$('#studentForm').submit(function(event) {
    event.preventDefault(); // Prevent default form submission
    console.log('Form submission intercepted');
});

// Input change
$('input[name="studentName"]').change(function() {
    console.log('Name changed to: ' + $(this).val());
});
```

Events

- Vanilla JS

js

```js
document.getElementById("submitBtn")
  .addEventListener("click", function() {
    alert("Button clicked!");
  });
```

- jQuery

js

```js
$("#submitBtn").on("click", function() {
  alert("Button clicked!");
});
```

- Native JavaScript:

javascript

```javascript
document.getElementById('submitBtn').addEventListener('click', () => {
    alert('Form submitted!');
});

document.getElementById('studentForm').addEventListener('submit', (event) => {
    event.preventDefault();
    console.log('Form submission intercepted');
});
```

- **DOM Manipulation**

DOM Manipulation involves dynamically modifying HTML elements' content, attributes, or styles using JavaScript or jQuery.

Common Operations

- **Content**: Change text (`textContent`, `.text()`), HTML (`innerHTML`, `.html()`).
- **Attributes**: Get/set attributes (`getAttribute`, `.attr()`).
- **Styles**: Modify CSS (`style`, `.css()`).
- **Add/Remove Elements**: Append (`appendChild`, `.append()`), remove (`removeChild`, `.remove()`).

```html
<script>
    $('#addButton').on('click', () => {
        const text = $('#userInput').val();
        if (typeof text === 'string' && text.trim()) {
            // Append new item
            $('#itemList').append(`<li class="item">${text}</li>`);
            // Clear input
            $('#userInput').val('');
            // Update style
            $('.item').css('color', 'blue');
        }
    });

    // Remove item on click (event delegation)
    $('#itemList').on('click', '.item', function() {
        $(this).remove();
    });
</script>
```

Php Laravel

- **Change content**

```js
document.getElementById("title").innerText = "New Title";
$("#title").text("New Title"); // jQuery
```

- **Change style**

```js
document.getElementById("title").style.color = "red";
$("#title").css("color", "red");
```

- **Add element**

```js
let div = docume
div.innerText =
document.body.appendChild(div);

$("#container").append("<div>Hello!</div>");
```

Ask ChatGPT

- **jQuery Examples:**

javascript Collapse W

```javascript
// Change text
$('#studentName').text('Alice Johnson');

// Update HTML
$('#studentList').html('<li>Alice Johnson</li><li>Bob Lee</li>');

// Add class
$('#errorMsg').addClass('error');

// Update attribute
$('input[name="studentName"]').attr('disabled', true);

// Append element
$('#studentList').append('<li>Carol White</li>');
```

- **Native JavaScript:**

javascript Collapse Wrap Run Copy

```javascript
// Change text
document.getElementById('studentName').textContent = 'Alice Johnson';

// Update HTML
document.getElementById('studentList').innerHTML = '<li>Alice Johnson</li><li>Bob Lee</l

// Add class
document.getElementById('errorMsg').classList.add('error');

// Update attribute
document.querySelector('input[name="studentName"]').setAttribute('disabled', 'true');

// Append element
const li = document.createElement('li');
li.textContent = 'Carol White';
document.getElementById('studentList').appendChild(li);
```

- ## AJAX (jQuery & fetch)

AJAX (Asynchronous JavaScript and XML) enables asynchronous communication with a server, often to fetch or send data without reloading the page.

jQuery AJAX

- Example (GET request):

javascript Collapse Wrap Run Copy

```javascript
$.ajax({
    url: 'api/students.php',
    method: 'GET',
    data: { departmentId: 1 },
    success: function(data) {
        $('#studentList').html(data.map(student => `<li>${student.name}</li>`).join(''));
    },
    error: function(xhr, status, error) {
        console.error('Error:', error);
    }
});
```

- **Example (POST request):**

javascript

```javascript
$.ajax({
    url: 'api/enroll.php',
    method: 'POST',
    data: { studentId: 101, courseId: 201 },
    success: function(response) {
        alert('Enrollment successful!');
    },
    error: function(xhr, status, error) {
        alert('Enrollment failed: ' + error);
    }
});
```

➢ Fetch API (Vanilla JS)

- Modern alternative to XMLHttpRequest, using Promises.

```javascript
const response = await fetch('getLeaveData.php');
const data = await response.json();
```

Php Laravel

```javascript
// Sending a POST request with JSON data
const response = await fetch('../components/Pending/mhme.php', {
    method: 'POST', // Specify the HTTP method
    headers: {
        'Content-Type': 'application/json', // Indicate JSON payload
    },
    body: JSON.stringify(requestData), // Convert the data to a JSON string
});

// Handle the response
if (!response.ok) {
    throw new Error(`HTTP error! Status: ${response.status}`);
}

const data = await response.json();

console.log(data); // Log the data to verify
const dutyUsed = Number(data.duty.used);
const dutyRemaining = Number(data.duty.total) - dutyUsed;
console.log(dutyRemaining);
```

```javascript
fetch(`../components/${page}.php`)
    .then(response => {
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        return response.text();
    })
    .then(data => {
        dynamicContentContainer.innerHTML = data;
        // Optionally update the browser history
        window.history.pushState({
            page: page
        }, '', `index.php?page=${page}`);
    })
    .catch(error => console.error('Error loading content:', error));
```

```php
<?php
// Get the raw POST data
$json = file_get_contents('php://input');
$data = json_decode($json, true); // Decode JSON into PHP array
```

```php
// Get JSON input
$input = json_decode(file_get_contents("php://input"), true);
```

- **Example (GET request):**

javascript Collapse Wrap R

```javascript
fetch('api/students.php?departmentId=1')
    .then(response => response.json())
    .then(data => {
        document.getElementById('studentList').innerHTML =
            data.map(student => `<li>${student.name}</li>`).join('');
    })
    .catch(error => console.error('Error:', error));
```

- **Example (POST request):**

javascript Collapse Wrap R

```javascript
fetch('api/enroll.php', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ studentId: 101, courseId: 201 })
})
    .then(response => response.json())
    .then(data => alert('Enrollment successful!'))
    .catch(error => alert('Enrollment failed: ' + error));
```

```javascript
fetch("server.php", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name: "John", age: 25 })
})
.then(res => res.json())
.then(data => console.log("Server says:", data));
```

- ## JSON Encode/Decode in PHP

PHP provides ***json_encode()*** and ***json_decode()*** to handle JSON data, commonly used in APIs and AJAX.

JSON Encode

Converts PHP arrays/objects to JSON strings.

- **Example:**

php Collapse W

```php
<?php
$students = [
    ['id' => 101, 'name' => 'Alice Johnson'],
    ['id' => 102, 'name' => 'Bob Lee']
];
header('Content-Type: application/json');
echo json_encode($students);
// Output: [{"id":101,"name":"Alice Johnson"},{"id":102,"name":"Bob Lee"}]
?>
```

JSON Decode

Converts JSON strings to PHP arrays/objects.

- **Example:**

php Collapse

```php
<?php
$json = '{"studentId":101,"courseId":201}';
$data = json_decode($json, true); // true for array, false for object
echo $data['studentId']; // Output: 101
?>
```

- ## Client-side Form Validation

Client-side form validation ensures user input is valid before sending to the server, improving UX and reducing server load.

Approaches

- **HTML5 Validation**: Built-in attributes (required, type="email", pattern).
- **JavaScript**: Custom validation logic.
- **Libraries**: Validate.js, jQuery Validation Plugin.

Php Laravel

```
<script>
    const form = document.getElementById('userForm');
    const nameInput = document.getElementById('name');
    const emailInput = document.getElementById('email');
    const nameError = document.getElementById('nameError');
    const emailError = document.getElementById('emailError');
    const resuslt = document.getElementById('result');

    form.addEventListener('submit', async (event) => {
        event.preventDefault();
        nameError.textContent = '';
        emailError.textContent = '';
        let isValid = true;

        // Custom validation
        if (!nameInput.value.trim() || nameInput.value.length < 2) {
            nameError.textContent = 'Name must be at least 2 characters.';
            isValid = false;
        }
        if (!emailInput.value.match(/^[^\s@]+@[^\s@]+\.[^\s@]+$/)) {
            emailError.textContent = 'Invalid email format.';
            isValid = false;
        }

        if (isValid) {
            try {
                const response = await fetch('api/user_create.php', {
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify({
                        name: nameInput.value,
                        email: emailInput.value
                    })
                });
                const data = await response.json();
```

Collapse Wrap Run Copy

```
$('#studentForm').submit(function(event) {
    event.preventDefault();
    let isValid = true;
    const name = $('input[name="studentName"]').val();
    const email = $('input[name="email"]').val();
    const departmentId = $('input[name="departmentId"]').val();

    // Clear previous errors
    $('.error').remove();

    // Validate name
    if (name.length < 2) {
        isValid = false;
        $('input[name="studentName"]').after('<span class="error">Name must be at least
    }

    // Validate email
    if (!/^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email)) {
        isValid = false;
        $('input[name="email"]').after('<span class="error">Invalid email</span>');
    }

    // Validate departmentId
    if (departmentId < 1) {
        isValid = false;
        $('input[name="departmentId"]').after('<span class="error">Invalid department
    }
```

Php Laravel

```
    if (isValid) {
        // Send AJAX request
        $.ajax({
            url: 'api/addStudent.php',
            method: 'POST',
            data: { name, email, departmentId },
            success: function(response) {
                alert('Student added!');
            },
            error: function() {
                alert('Error adding student');
            }
        });
    }
});
```

```
<?php
$data = json_decode(file_get_contents('php://input'), true);
$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
$stmt = $pdo->prepare('INSERT INTO Students (Name, Email, DepartmentID) VALUES (?, ?, ?)
$stmt->execute([$data['name'], $data['email'], $data['departmentId']]);
header('Content-Type: application/json');
echo json_encode(['status' => 'success']);
?>
```

5. Client-Side Form Validation

HTML5 built-in

```html
<form>
  <input type="email" required>
  <input type="password" minlength="6" required>
  <button type="submit">Submit</button>
</form>
```

Custom Validation (JS)

```js
document.getElementById("myForm").addEventListener("submit", function(e) {
  let email = document.getElementById("email").value;
  if (!email.includes("@")) {
    alert("Invalid email!");
    e.preventDefault(); // stop form submission
  }
});
```

WEEK 4

HTTP verbs/status codes • Headers • cURL GET/POST/PUT/DELETE • cURL error handling & timeouts • SSL & certificates

● HTTP Verbs/Status Codes

➢ HTTP Verbs

HTTP verbs (methods) define the action to perform on a resource in a **REST API**.

Verb	Usage Example
GET	Fetch data (idempotent) → `/users/1`
POST	Create new resource → `/users`
PUT	Update/replace resource → `/users/1`
PATCH	Partial update → `/users/1`
DELETE	Remove resource → `/users/1`

- **GET**: Retrieve data (read-only).
  - Example: `GET /api/students` - Fetch all students.
- **POST**: Create a new resource.
  - Example: `POST /api/students` - Add a new student.
- **PUT**: Update an existing resource (or create if it doesn't exist).
  - Example: `PUT /api/students/101` - Update student with ID 101.
- **DELETE**: Remove a resource.
  - Example: `DELETE /api/students/101` - Delete student with ID 101.
- **PATCH**: Partially update a resource.
  - Example: `PATCH /api/students/101` - Update specific fields (e.g., name).

➢ HTTP Status Codes

Status codes indicate the result of an HTTP request.

Standard response codes:

- **200 OK** → Success
- **201 Created** → Resource created (POST/PUT)
- **204 No Content** → Success, nothing to return
- **301/302 Redirect** → Resource moved
- **400 Bad Request** → Invalid input
- **401 Unauthorized** → Needs login/token
- **403 Forbidden** → Permission denied
- **404 Not Found** → Resource doesn't exist
- **500 Internal Server Error** → Server-side crash

- **2xx (Success):**
  - `200 OK`: Request succeeded (e.g., GET returned data).
  - `201 Created`: Resource created (e.g., POST added a student).
  - `204 No Content`: Request succeeded, no content returned (e.g., DELETE).
- **4xx (Client Error):**
  - `400 Bad Request`: Invalid request (e.g., missing parameters).
  - `401 Unauthorized`: Authentication required.
  - `403 Forbidden`: Access denied.
  - `404 Not Found`: Resource not found (e.g., invalid student ID).
- **5xx (Server Error):**
  - `500 Internal Server Error`: Server-side issue.
  - `503 Service Unavailable`: Server temporarily down.

- **1xx (Informational)**: Request received, continuing (e.g., 100 Continue).
- **2xx (Success)**:
  - 200 OK: Request succeeded.
  - 201 Created: Resource created.
  - 204 No Content: Successful, no response body.
- **3xx (Redirection)**:
  - 301 Moved Permanently: Resource moved.
  - 302 Found: Temporary redirect.
- **4xx (Client Error)**:
  - 400 Bad Request: Invalid request syntax or data.
  - 401 Unauthorized: Authentication required.
  - 403 Forbidden: Access denied.
  - 404 Not Found: Resource not found.
  - 429 Too Many Requests: Rate limit exceeded.
- **5xx (Server Error)**:
  - 500 Internal Server Error: Generic server error.
  - 503 Service Unavailable: Server temporarily down.

- **Headers:**

HTTP headers provide metadata about requests and responses, controlling behavior like caching, authentication, or content type.

Common Headers

- **Request Headers:**
  - `Content-Type`: Specifies request body format (e.g., `application/json`).
  - `Authorization`: Sends credentials (e.g., `Bearer <token>`).
  - `Accept`: Desired response format (e.g., `application/json`).
- **Response Headers:**
  - `Content-Type`: Response format (e.g., `application/json; charset=UTF-8`).
  - `Location`: URL of created resource (e.g., after `POST`).
  - `Cache-Control`: Caching behavior (e.g., `no-cache`).
  - `X-Rate-Limit`: Rate limiting info.

```php
<?php

declare(strict_types=1);

namespace App\Api;

header('Content-Type: application/json; charset=UTF-8');
header('Cache-Control: no-cache, no-store, must-revalidate');
header('X-API-Version: 1.0');

echo json_encode(['message' => 'Headers set'], JSON_THROW_ON_ERROR);
?>
```

What is a PHP Header?

The `header()` function in PHP is used to send **raw HTTP headers** to the client **before any output** is sent (i.e., before HTML or echo).

👉 Syntax:

```php
header(string $header, bool $replace = true, int $response_code = 0)
```

- `$header`: The header string (e.g., `"Location: home.php"`)
- `$replace`: Whether to replace a previous header of the same type (`true` = replace, `false` = allow multiple)
- `$response_code`: Set the HTTP response code (optional, e.g., `404`)

1. Redirecting a Page

```php
<?php
// Redirect user to home.php
header("Location: home.php");
exit; // always call exit after redirect
?>
```

2. Setting Content Type

```php
<?php
// Force the browser to treat output as JSON
header("Content-Type: application/json");
echo json_encode(["status" => "success", "msg" => "Hello World"]);
?>
```

3. Forcing File Download

```php
<?php
$file = "report.pdf";
header("Content-Type: application/octet-stream");
header("Content-Disposition: attachment; filename=\"" . basename($file) . "\"");
header("Content-Length: " . filesize($file));
readfile($file);
exit;
?>
```

5. Caching Control

```php
<?php
// Disable browser caching
header("Cache-Control: no-cache, must-revalidate");
header("Expires: Sat, 01 Jan 2000 00:00:00 GMT");
?>
```

HTTP Verbs (a.k.a. Methods)

They define the **action** the client wants the server to perform.

Verb	Purpose	Example Use
GET	Retrieve data (read-only). Should not change server state.	`GET /users` → Fetch all users
POST	Create new data.	`POST /users` → Add a new user
PUT	Replace existing resource with new data.	`PUT /users/5` → Update user with ID=5 (replace all fields)
PATCH	Partially update an existing resource.	`PATCH /users/5` → Update only the email of user 5
DELETE	Remove a resource.	`DELETE /users/5` → Delete user with ID=5

- **cURL GET/POST/PUT/DELETE**

cURL is a PHP library for making HTTP requests.

cURL is a PHP library for making HTTP requests, supporting all verbs.

1. GET

- Retrieve data from an endpoint.

```php
$ch = curl_init('http://localhost:8000/api/users.php?id=1');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_HTTPHEADER, ['Accept: application/json']);
$response = curl_exec($ch);
curl_close($ch);

echo $response ?: 'Error: ' . curl_error($ch);
```

2. POST

- Create a new resource.

```php
$data = ['name' => 'Alice Brown', 'email' => 'alice@example.com'];
$ch = curl_init('http://localhost:8000/api/users.php');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    'Content-Type: application/json',
    'Content-Length: ' . strlen(json_encode($data))
]);
$response = curl_exec($ch);
curl_close($ch);
```

3. PUT

- Update an existing resource.

Php Laravel

```php
$data = ['name' => 'Alice Updated', 'email' => 'alice.updated@example.
com'];
$ch = curl_init('http://localhost:8000/api/users.php?id=3');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'PUT');
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    'Content-Type: application/json',
    'Content-Length: ' . strlen(json_encode($data))
]);
$response = curl_exec($ch);
curl_close($ch);
```

--- PATCH ---

```php
$ch = curl_init($url);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "PATCH");
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    "Content-Type: application/json",
    "Content-Length: " . strlen($data)
]);

$response = curl_exec($ch);
curl_close($ch);
```

4. DELETE

- Remove a resource.

```php
$ch = curl_init('http://localhost:8000/api/users.php?id=3');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, 'DELETE');
curl_setopt($ch, CURLOPT_HTTPHEADER, ['Accept: application/json']);
$response = curl_exec($ch);
$status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
curl_close($ch);

echo $status === 204 ? 'User deleted.' : ($response ?: 'Error: ' .
curl_error($ch));
```

```php
$ch = curl_init("https://api.example.com/users/1");
curl_setopt($ch, CURLOPT_CUSTOMREQUEST, "DELETE");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

$response = curl_exec($ch);
curl_close($ch);

echo $response;
```

Php Laravel

```
# Get all tasks
curl http://localhost/tasks

# Get a single task
curl http://localhost/tasks/1

# Create a new task
curl -X POST -H "Content-Type: application/json" \
     -d '{"title":"Study law"}' \
     http://localhost/tasks

# Update a task
curl -X PUT -H "Content-Type: application/json" \
     -d '{"done":true}' \
     http://localhost/tasks/1

# Delete a task
curl -X DELETE http://localhost/tasks/2
```

```
C:\Aniket_PHP\Week5>curl -X POST -H "Content-Type:application/json" -d "{\"name\":\"aniket\",\"age\":20}" http://localho
st:3000/Week5/index.php
```

```
if ($method === 'POST') {
    // print_r($_POST["name"]);
    $ok = file_get_contents("php://input");
    $ok1 = json_decode($ok, true);
    print_r($ok1);
}
```

```
C:\Aniket_PHP\Week5>curl -X POST -H "Content-Type:application/json" -d "name=aniket&age=10" http://localhost:3000/Week5/
index.php
```

```
if ($method === 'POST') {
    // print_r($_POST["name"]);
    $ok = file_get_contents("php://input");
    parse_str($ok, $ok1);
    print_r($ok1);
}
```

```
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

$ok = explode("/", trim($uri, "/"));
print_r($ok);
```

```
elseif (preg_match("#^/Week5/users/(\d+)$#", $uri, $matches) && $method === 'GET') {
  // $id = $matches[1];
  $id = array_shift($matches);
```

- ## cURL Error Handling & Timeouts

Proper error handling and timeouts ensure robust cURL requests.

➢ Error Handling

- Check **curl_exec()** return value (false on failure).
- Use **curl_error()** and **curl_errno()** for error details.
- Retrieve HTTP status with **curl_getinfo($ch, CURLINFO_HTTP_CODE)**.

Php Laravel

```php
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://example.com/api/students');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);

if ($response === false) {
    echo 'cURL Error: ' . curl_error($ch);
} else {
    $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    if ($httpCode >= 400) {
        echo 'HTTP Error: ' . $httpCode;
    } else {
        echo $response;
    }
}
curl_close($ch);
?>
```

```php
$ch = curl_init("https://api.example.com/users/1");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_TIMEOUT, 10); // 10 seconds timeout

$response = curl_exec($ch);

if (curl_errno($ch)) {
    echo "cURL Error: " . curl_error($ch);
} else {
    $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    echo "HTTP Status: $httpCode\n";
    echo "Response: $response";
}

curl_close($ch);
```

➢ Timeouts

- **Connection Timeout**: Time to establish a connection (**CURLOPT_CONNECTTIMEOUT**).
- **Total Timeout**: Maximum request duration (**CURLOPT_TIMEOUT**).

```php
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://example.com/api/students');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 5); // 5 seconds to connect
curl_setopt($ch, CURLOPT_TIMEOUT, 10); // 10 seconds total
$response = curl_exec($ch);

if ($response === false && curl_errno($ch) == CURLE_OPERATION_TIMEDOUT) {
    echo 'Request timed out';
} else {
    echo $response;
}
curl_close($ch);
?>
```

- **Common cURL Errors:**
  - `CURLE_COULDNT_CONNECT` : Failed to connect to the server.
  - `CURLE_SSL_CACERT` : SSL certificate issue.
  - `CURLE_OPERATION_TIMEDOUT` : Request exceeded timeout.

```php
curl_setopt_array($ch, [
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_CONNECTTIMEOUT => 5,  // 5s max for connection
    CURLOPT_TIMEOUT => 10         // 10s max for full response
]);
```

```php
try {
    $ch = curl_init('http://localhost:8000/api/users.php?id=1');
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 5); // 5 seconds to connect
    curl_setopt($ch, CURLOPT_TIMEOUT, 10); // 10 seconds total
    curl_setopt($ch, CURLOPT_HTTPHEADER, ['Accept: application/json']);

    $response = curl_exec($ch);
    $status = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    $error = curl_error($ch);
    $errno = curl_errno($ch);
    curl_close($ch);

    if ($response === false) {
        throw new RuntimeException("cURL error ($errno): $error");
    }

    if ($status >= 400) {
        throw new RuntimeException("HTTP error: $status - " . json_decode
        ($response)->error);
    }

    echo $response;
} catch (RuntimeException $e) {
    echo 'Error: ' . htmlspecialchars($e->getMessage(), ENT_QUOTES, 'UTF-8');
}
```

- ## SSL & Certificates

SSL (Secure Sockets Layer) ensures secure communication via HTTPS. Certificates verify server identity.

- By default, cURL checks SSL certificates.

- If using **self-signed certs** (dev/test), you may get errors like `SSL certificate problem.`

When you make HTTPS requests with cURL, it checks the server's **SSL/TLS certificate** to make sure:

- The site is trusted (signed by a known Certificate Authority).
- The certificate hasn't expired or been tampered with.

Configuring PHP cURL for SSL

- Enable **CURLOPT_SSL_VERIFYPEER** and **CURLOPT_SSL_VERIFYHOST** for security.
- Provide a CA certificate bundle for verification.

```php
$ch = curl_init('https://api.example.com/users');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 2);
curl_setopt($ch, CURLOPT_CAINFO, '/path/to/cacert.pem'); // Path to CA
```

Note: Download **cacert.pem** from https://curl.se/docs/caextract.html or use system CA bundle.

cURL with SSL

- **Verify SSL:** Ensure the server's certificate is valid (default behavior).

php Collapse Wrap

```php
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true); // Verify server certificate
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 2); // Verify hostname
```

- **Custom CA Certificate** (e.g., for self-signed certificates):

php Collapse Wrap

```php
curl_setopt($ch, CURLOPT_CAINFO, '/path/to/cacert.pem'); // Path to CA certificate
```

- **Disable SSL Verification** (for testing only, not secure):

php Collapse Wrap

```php
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, false);
```

➢ **Client Certificates (for APIs that require them)**

Some APIs (like banking or payment gateways) require **your app** to present a client certificate.

Php Laravel

```php
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Path to client certificate & private key
curl_setopt($ch, CURLOPT_SSLCERT, "client_cert.pem");
curl_setopt($ch, CURLOPT_SSLKEY, "client_key.pem");

// If the cert has a password
curl_setopt($ch, CURLOPT_KEYPASSWD, "mypassword");

$response = curl_exec($ch);
```

🔹 Quick Recap

- `CURLOPT_SSL_VERIFYPEER = true` → verify SSL cert (default & recommended).
- `CURLOPT_CAINFO` → provide path to CA certificates bundle.
- `CURLOPT_SSLCERT` & `CURLOPT_SSLKEY` → use your own client certificate for secure APIs.
- `CURLOPT_SSL_VERIFYPEER = false` → only for testing ⚠️.

```php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://example.com/api/students');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, true);
curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 2);
$response = curl_exec($ch);
```

REST constraints • Routing (vanilla) • JSON responses • Pagination & filtering • Error contracts

- **REST constraints**
  REST (Representational State Transfer) isn't just "using GET/POST" — it's an **architectural style** for designing APIs. Roy Fielding (who defined REST) described **6 constraints** that make an API truly RESTful.

🔹 1. Client–Server

- Separation of **client** (frontend/UI) and **server** (backend/data).
- Client doesn't need to know how data is stored, only how to request it.
  ✅ Benefit: independent development, scalability.

👉 Example:

- Browser app = client
- PHP API = server

🔹 2. Stateless

- Each request must contain **all the info needed** (no hidden server memory about previous requests).
- Server does not store client session state between requests.
  ✅ Benefit: easier scaling, no session dependency.

👉 Example:

nstead of server remembering you're logged in:

- Client sends token in **every request**:

http

```http
GET /users/10
Authorization: Bearer <token>
```

🔹 3. Cacheable

- Responses must define if they can be cached or not (using headers).
  ✅ Benefit: improves performance & reduces server load.

👉 Example:

http

```http
Cache-Control: public, max-age=3600
```

Means → Response can be cached for 1 hour.

🔹 4. Uniform Interface

This is the **core principle** — clients and servers communicate in a **standardized way**.
It has 4 sub-rules:

1. **Resource-based URLs** → `/users/1`, `/orders/5`
2. **Use HTTP verbs properly** → GET, POST, PUT, DELETE
3. **Representations** → JSON, XML, etc.
4. **HATEOAS (Hypermedia as the Engine of Application State)** → responses can include links to related actions.

👉 Example:

```json
{
  "id": 1,
  "name": "Alice",
  "links": {
    "self": "/users/1",
    "orders": "/users/1/orders"
  }
}
```

🔹 5. Layered System

- Client doesn't know if it's talking to the **actual server** or an **intermediate** (proxy, load balancer, CDN).
  ✅ Benefit: scalability, security, performance.

👉 Example:

- Browser → CDN → API Gateway → Backend Server

🔹 6. Code on Demand (Optional)

- Server can send executable code to client (like JavaScript).
- Not widely used in APIs, more in web apps.

👉 Example:

Server sends JS code to run in browser.

🔹 1. REST Constraints (RESTful Principles)

A REST API should follow **Roy Fielding's REST constraints**:

1. **Client–Server** → separation of concerns (frontend vs backend).
2. **Stateless** → every request must contain all info (no session state stored on server).
3. **Cacheable** → responses should define if they're cacheable (via headers).
4. **Uniform Interface** → standard resource-based URLs, HTTP verbs, representations (JSON).
5. **Layered System** → client shouldn't care if it's talking to backend directly or via proxies.
6. **Code on Demand** (optional) → ability to send executable code (rare in practice).

- ## Routing (vanilla)

Routing in vanilla PHP involves mapping HTTP requests (method and URL) to specific code without a framework. This is typically done by parsing the request URI and method.

Routing = mapping URLs to actions. In **vanilla PHP** (no framework)

➢ What is Routing?

Routing decides **which code should run** based on the URL (path + method).
For example:

- ➢ `GET /users` → show all users
- ➢ `POST /users` → create a new user
- ➢ `GET /users/5` → show user with ID 5

➢ **Setup**:

- Use a web server (e.g., Apache) with **.htaccess** to route all requests to **index.php**

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php [QSA,L]
```

```
Week5 > .htaccess
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} ! - f
RewriteCond %{REQUEST_FILENAME} ! - d
RewriteRule ^ index.php [QSA,L]
```

To disable rename **.htaccess** to **.htaccess_disabled** or just stop the server

1. Basic Routing with *$_SERVER*

In vanilla PHP, you can use **$_SERVER['REQUEST_URI']** (URL path) and **$_SERVER['REQUEST_METHOD']** (HTTP verb).

Php Laravel

```php
<?php
// index.php (entry point)
header('Content-Type: application/json');
$method = $_SERVER['REQUEST_METHOD'];
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$segments = explode('/', trim($uri, '/'));

$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');

// Route: GET /api/students
if ($method === 'GET' && $segments[0] === 'api' && $segments[1] === 'students' && !isset(
    $stmt = $pdo->query('SELECT StudentID, Name, DepartmentID FROM Students');
    echo json_encode($stmt->fetchAll(PDO::FETCH_ASSOC));
}
// Route: GET /api/students/{id}
elseif ($method === 'GET' && $segments[0] === 'api' && $segments[1] === 'students' && iss
    $id = (int)$segments[2];
    $stmt = $pdo->prepare('SELECT StudentID, Name, DepartmentID FROM Students WHERE Stude
    $stmt->execute([$id]);
    $student = $stmt->fetch(PDO::FETCH_ASSOC);
    if ($student) {
        http_response_code(200);
        echo json_encode($student);
    } else {
        http_response_code(404);
        echo json_encode(['error' => 'Student not found']);
    }
}
```

```php
// Route: POST /api/students
elseif ($method === 'POST' && $segments[0] === 'api' && $segments[1] === 'students') {
    $data = json_decode(file_get_contents('php://input'), true);
    if (!isset($data['name'], $data['departmentId'])) {
        http_response_code(400);
        echo json_encode(['error' => 'Missing required fields']);
        exit;
    }
    $stmt = $pdo->prepare('INSERT INTO Students (Name, DepartmentID) VALUES (?, ?)');
    $stmt->execute([$data['name'], $data['departmentId']]);
    http_response_code(201);
    echo json_encode(['status' => 'created', 'id' => $pdo->lastInsertId()]);
}
// Route: DELETE /api/students/{id}
elseif ($method === 'DELETE' && $segments[0] === 'api' && $segments[1] === 'students' &&
    $id = (int)$segments[2];
    $stmt = $pdo->prepare('DELETE FROM Students WHERE StudentID = ?');
    $stmt->execute([$id]);
    if ($stmt->rowCount() > 0) {
        http_response_code(204);
    } else {
        http_response_code(404);
        echo json_encode(['error' => 'Student not found']);
    }
} else {
    http_response_code(404);
    echo json_encode(['error' => 'Route not found']);
}
?>
```

```php
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$method = $_SERVER['REQUEST_METHOD'];
echo "<br>Filtered URI: ";
print_r($uri);
echo "<br>";
if ($uri === '/users' && $method === 'GET') {
    echo "List all users<br>";
} elseif ($uri === '/users' && $method === 'POST') {
    echo "Create new user<br>";
} elseif (preg_match("#^/users/(\d+)$#", $uri, $matches) && $method === 'GET') {
    // $id = $matches[1];
    $id = array_shift($matches);
    // print_r($matches);
    echo "Show user with ID $id<br>";
} else {
    http_response_code(404);
    echo "404 Not Found<br>";
}


$data = ["url" => "https://example.com", "text" => "नमस्ते"];

$en = json_encode($data, JSON_PRETTY_PRINT | JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
echo $en;
print_r(json_decode($en, true));
```

2. Using a “routes table” (cleaner)

Instead of writing big if-else, define routes in an array:

```php
<?php
$routes = [
    'GET' => [
        '/users' => function () { echo "List all users"; },
        '/users/1' => function () { echo "Show user 1"; }
    ],
    'POST' => [
        '/users' => function () { echo "Create new user"; }
    ]
];

$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$method = $_SERVER['REQUEST_METHOD'];

if (isset($routes[$method][$uri])) {
    $routes[$method][$uri](); // call the function
} else {
    http_response_code(404);
    echo "404 Not Found";
}
```

3. RESTful Example (Users API)

```php
<?php
$uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$method = $_SERVER['REQUEST_METHOD'];

switch (true) {
    case $uri === '/users' && $method === 'GET':
        echo json_encode(["Alice", "Bob"]);
        break;

    case $uri === '/users' && $method === 'POST':
        echo json_encode(["message" => "User created"]);
        break;

    case preg_match("#^/users/(\d+)$#", $uri, $m) && $method === 'GET':
        echo json_encode(["id" => $m[1], "name" => "Alice"]);
        break;

    default:
        http_response_code(404);
        echo json_encode(["error" => "Not Found"]);
}
```

- **JSON Responses**

JSON is the standard format for REST API responses, ensuring interoperability.

```php
header('Content-Type: application/json');
$students = [
    ['id' => 101, 'name' => 'Alice Johnson', 'departmentId' => 1],
    ['id' => 102, 'name' => 'Bob Lee', 'departmentId' => 1]
];
http_response_code(200);
echo json_encode($students);
// Output: [{"id":101,"name":"Alice Johnson","departmentId":1},{"id
```

1. Sending a JSON Response

PHP makes this simple:

- Set the **Content-Type header**
- Use `json_encode()` to convert PHP arrays/objects into JSON

```php
<?php
header("Content-Type: application/json");

$data = [
    "status" => "success",
    "message" => "Hello, world!"
];

echo json_encode($data);
?>
```

2. Pretty Printed JSON (Readable)

```php
<?php
header("Content-Type: application/json");

$data = ["id" => 1, "name" => "Alice", "role" => "admin"];

echo json_encode($data, JSON_PRETTY_PRINT);
?>
```

🔹 3. Handling Errors with JSON

When something goes wrong, return JSON with an error code + message.

```php
<?php
header("Content-Type: application/json");

http_response_code(404);

echo json_encode([
    "error" => true,
    "message" => "User not found"
]);
?>
```

➢ **Useful Options in json_encode**

- ◆ `JSON_PRETTY_PRINT` → formatted output
- ◆ `JSON_UNESCAPED_SLASHES` → don't escape /
- ◆ `JSON_UNESCAPED_UNICODE` → keep UTF-8 characters readable

```php
<?php
$data = ["url" => "https://example.com", "text" => "नमस्ते"];

echo json_encode($data, JSON_PRETTY_PRINT | JSON_UNESCAPED_UNICODE | JSON_UNESCAPED_SLASHES);
?>
```

✅ Response:

```json
{
    "url": "https://example.com",
    "text": "नमस्ते"
}
```

- Pagination & Filtering

Pagination and filtering optimize API performance by limiting and refining data returned.

1. Why Pagination & Filtering?

➢ **Pagination** → Split large results into pages (e.g., 10 users at a time).
➢ **Filtering** → Allow clients to request only data they need (e.g., only users from India, or only active users).

✅ This avoids sending thousands of records at once → better performan

➢ **Pagination**

- **Approach**: Use query parameters like page and limit to control the number of records returned.

Common query parameters:

- `page` → which page number
- `limit` → how many results per page

👉 Example:

`GET /users?page=2&limit=5`

Php Laravel

```php
<?php
// GET /api/students?page=2&limit=10
header('Content-Type: application/json');
$page = isset($_GET['page']) ? (int)$_GET['page'] : 1;
$limit = isset($_GET['limit']) ? (int)$_GET['limit'] : 10;
$offset = ($page - 1) * $limit;

$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
$stmt = $pdo->prepare('SELECT StudentID, Name, DepartmentID FROM Students LIMIT ? OFFSET
$stmt->execute([$limit, $offset]);
$students = $stmt->fetchAll(PDO::FETCH_ASSOC);

$countStmt = $pdo->query('SELECT COUNT(*) FROM Students');
$total = $countStmt->fetchColumn();

echo json_encode([
    'data' => $students,
    'pagination' => [
        'page' => $page,
        'limit' => $limit,
        'total' => $total,
        'pages' => ceil($total / $limit)
    ]
]);
?>
```

➢ Filtering

- **Approach**: Use query parameters to filter results (e.g., departmentId, name).

Let clients filter results with query params.

👉 Example:

`GET /users?country=India&status=active`

```php
<?php
// GET /api/students?departmentId=1&name=Alice
header('Content-Type: application/json');
$departmentId = isset($_GET['departmentId']) ? (int)$_GET['departmentId'] : null;
$name = isset($_GET['name']) ? $_GET['name'] : null;

$query = 'SELECT StudentID, Name, DepartmentID FROM Students WHERE 1=1';
$params = [];
if ($departmentId) {
    $query .= ' AND DepartmentID = ?';
    $params[] = $departmentId;
}
if ($name) {
    $query .= ' AND Name LIKE ?';
    $params[] = "%$name%";
}

$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
$stmt = $pdo->prepare($query);
$stmt->execute($params);
echo json_encode(['data' => $stmt->fetchAll(PDO::FETCH_ASSOC)]);
?>
```

● Error Contracts

An error contract defines a consistent format for error responses, making it easier for clients to handle errors.

A good error response usually has:

- **code** → machine-readable error code (string/number)
- **message** → human-readable message
- **details** → optional extra info (field validation errors, stack traces, etc.)
- **status** → HTTP status code (sometimes included inside JSON too)

Consistent error format helps clients handle errors easily.

Best practice → include:

- `status` (HTTP status code)
- `error` (short error code)
- `message` (human-readable)
- `details` (optional, extra context)

```json
{
  "error": {
    "code": "USER_NOT_FOUND",
    "message": "The requested user does not exist.",
    "status": 404,
    "details": {
      "userId": 123
    }
  }
}
```

```json
{
  "status": 400,
  "error": "INVALID_INPUT",
  "message": "The 'email' field is required.",
  "details": {
    "field": "email"
  }
}
```

API keys • JWT structure & validation • OAuth2 grant types overview • Rate limiting & throttling • API docs (OpenAPI)

● API Keys

API keys are simple tokens used to authenticate API requests, often for tracking or access control.

- **How It Works**:
  - A unique key (e.g., abc123xyz) is assigned to a client.
  - The client includes the key in requests (e.g., as a header or query parameter).
  - The server validates the key against a stored list.

```php
<?php
// api/students.php
header('Content-Type: application/json');
$apiKey = $_SERVER['HTTP_X_API_KEY'] ?? '';
$validKeys = ['abc123xyz', 'def456uvw']; // Stored in DB in practice

if (!in_array($apiKey, $validKeys)) {
    http_response_code(401);
    echo json_encode(['error' => 'Invalid API key', 'code' => 'UNAUTHORIZED']);
    exit;
}

$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
$stmt = $pdo->query('SELECT StudentID, Name FROM Students');
echo json_encode(['data' => $stmt->fetchAll(PDO::FETCH_ASSOC)]);
?>
```

Php Laravel

A common pattern is:

```makefile
Authorization: Bearer YOUR_API_KEY
```

or sometimes:

```makefile
x-api-key: YOUR_API_KEY
```

It depends on the API provider.

🛠️ Example with cURL

```bash
curl -H "Authorization: Bearer 12345ABCDE" \
     https://api.example.com/data
```

🛠️ Example in PHP (cURL)

```php
<?php
$apiKey = "12345ABCDE";

$ch = curl_init("https://api.example.com/data");

curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Send key in header
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    "Authorization: Bearer $apiKey"
]);

$response = curl_exec($ch);

if ($response === false) {
    echo "cURL error: " . curl_error($ch);
} else {
    echo $response;
}

curl_close($ch);
```

🔑 Where API key comes from

- Client includes it in **header**:

  ```makefile
  Authorization: Bearer 12345ABCDE
  ```

  or

  ```makefile
  x-api-key: 12345ABCDE
  ```

- Your API (`index.php`) reads the header and verifies it against a list of valid keys.

```php
// List of valid API keys (in real apps, store in DB/env file)
$validKeys = ["12345ABCDE", "67890FGHIJ"];

// Get all request headers
$headers = getallheaders();

// Look for Authorization header
$apiKey = null;
if (isset($headers["Authorization"])) {
    // Expect format: "Bearer KEY"
    if (preg_match('/Bearer\s+(.*)$/i', $headers["Authorization"], $matches)) {
        $apiKey = $matches[1];
    }
} elseif (isset($headers["x-api-key"])) {
    $apiKey = $headers["x-api-key"];
}

// Verify
if (!$apiKey || !in_array($apiKey, $validKeys)) {
    http_response_code(401); // Unauthorized
    echo json_encode(["error" => "Invalid or missing API key"]);
    exit;
}
```

Php Laravel

```http
x-api-key: 12345abcd67890xyz
```

PHP Example (Client Side with cURL)

```php
<?php
$ch = curl_init("https://api.example.com/users");

curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_HTTPHEADER, [
    "x-api-key: 12345abcd67890xyz"
]);

$response = curl_exec($ch);
curl_close($ch);

echo $response;
?>
```

```php
<?php
header("Content-Type: application/json");

// Example: hardcoded valid keys (usually stored in DB)
$validKeys = ["12345abcd67890xyz", "abcdef123456"];

// Get key from header
$headers = getallheaders();
$apiKey = $headers['x-api-key'] ?? null;

// Check if key is valid
if (!$apiKey || !in_array($apiKey, $validKeys)) {
    http_response_code(401); // Unauthorized
    echo json_encode([
        "error" => "Invalid or missing API key"
    ]);
    exit;
}

// If valid → continue
echo json_encode([
    "message" => "Access granted",
    "data" => ["id" => 1, "name" => "Alice"]
]);
```

- ## JWT Structure & Validation

JSON Web Tokens (JWTs) are compact, self-contained tokens for secure data exchange, often used for authentication.

JWT Structure

A JWT consists of three parts, Base64-encoded and separated by dots (.):

`Header.Payload.Signature`.

JWT (JSON Web Token) is a compact, URL-safe token for authentication, consisting of three parts: **Header, Payload,** and **Signature**, **Base64-encoded** and joined by dots (**.**).

Structure

- **Header**: JSON object with algorithm (e.g., {"alg":"HS256","typ":"JWT"}).
- **Payload**: Claims (e.g., sub for user ID, exp for expiration).
- **Signature**: HMAC or RSA signature to verify integrity.

Implementation

Use `firebase/php-jwt` for JWT handling.

Install via Composer:

bash Collapse

```bash
composer require firebase/php-jwt
```

Php Laravel

1. **Header**: Metadata about the token (e.g., algorithm).

```json
{"alg": "HS256", "typ": "JWT"}
```

2. **Payload**: Claims (data) like user ID, roles, expiration.

```json
{"sub": "101", "name": "Alice", "exp": 1726147140}
```

3. **Signature**: Verifies integrity, created using a secret key.
   - Example: `HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)`.

```php
<?php
require 'vendor/autoload.php';
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

header('Content-Type: application/json');
$jwt = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
$jwt = str_replace('Bearer ', '', $jwt); // Remove "Bearer" prefix
$secret = 'your-secret-key';

try {
    $decoded = JWT::decode($jwt, new Key($secret, 'HS256'));
    $studentId = $decoded->sub;

    $pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
    $stmt = $pdo->prepare('SELECT StudentID, Name FROM Students WHERE StudentID = ?');
    $stmt->execute([$studentId]);
    $student = $stmt->fetch(PDO::FETCH_ASSOC);
```

```php
use Firebase\JWT\JWT;
use Firebase\JWT\Key;
use PDO;
use PDOException;

require 'vendor/autoload.php';

try {
    $pdo = new PDO('mysql:host=localhost;dbname=ecommerce_db;charset=utf8mb4', 'root', '', [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ]);

    $method = $_SERVER['REQUEST_METHOD'];
    $secretKey = 'your-secure-secret-key'; // Store in .env

    if ($method === 'POST' && $_SERVER['REQUEST_URI'] === '/api/auth_jwt.php/login') {
        $input = json_decode(file_get_contents('php://input'), true, 512, JSON_THROW_ON_ERROR);
        $email = $input['email'] ?? '';
        $password = $input['password'] ?? '';

        $stmt = $pdo->prepare('SELECT id, password FROM users WHERE email = :email');
        $stmt->execute(['email' => $email]);
        $user = $stmt->fetch();

        if ($user && password_verify($password, $user['password'])) {
            $payload = [
                'sub' => $user['id'],
                'iat' => time(),
                'exp' => time() + 3600 // 1 hour expiration
            ];
            $jwt = JWT::encode($payload, $secretKey, 'HS256');
            http_response_code(200);
            echo json_encode(['token' => $jwt], JSON_THROW_ON_ERROR);
```

```php
        } else {
            http_response_code(401);
            echo json_encode(['error' => 'Invalid credentials', 'code' => 'INVALID_CREDENTIALS'],
            JSON_THROW_ON_ERROR);
        }
        exit;
    }

    $authHeader = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
    if (!preg_match('/Bearer\s(\S+)/', $authHeader, $matches)) {
        http_response_code(401);
        echo json_encode(['error' => 'JWT required', 'code' => 'MISSING_JWT'],
        JSON_THROW_ON_ERROR);
        exit;
    }

    $jwt = $matches[1];
    try {
        $decoded = JWT::decode($jwt, new Key($secretKey, 'HS256'));
        $stmt = $pdo->prepare('SELECT id, name, email FROM users WHERE id = :id');
        $stmt->execute(['id' => $decoded->sub]);
        $user = $stmt->fetch();

        if ($user) {
            http_response_code(200);
            echo json_encode(['data' => $user], JSON_THROW_ON_ERROR);
        } else {
            http_response_code(401);
            echo json_encode(['error' => 'User not found', 'code' => 'INVALID_USER'],
            JSON_THROW_ON_ERROR);
        }
    } catch (\Firebase\JWT\ExpiredException $e) {
        http_response_code(401);
        echo json_encode(['error' => 'Token expired', 'code' => 'TOKEN_EXPIRED'],
        JSON_THROW_ON_ERROR);
```

```bash
# Get JWT
curl -X POST -H "Content-Type: application/json" -d '{"email":"john@example.c

# Use JWT
curl -H "Authorization: Bearer <jwt_token>" http://localhost:8000/api/auth_jw
```

PHP Example — Generating JWT

(using `firebase/php-jwt` library)

```php
<?php
require 'vendor/autoload.php';
use Firebase\JWT\JWT;

$secretKey = "mySuperSecretKey"; // keep safe!

$payload = [
    "userId" => 123,
    "email" => "alice@gmail.com",
    "role" => "admin",
    "exp" => time() + 3600 // 1 hour expiration
];

$jwt = JWT::encode($payload, $secretKey, 'HS256');

echo $jwt;
?>
```

PHP Example — Validating JWT

```php
<?php
require 'vendor/autoload.php';
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

$secretKey = "mySuperSecretKey";

// Suppose we got the JWT from client request
$headers = getallheaders();
$authHeader = $headers['Authorization'] ?? '';
$jwt = str_replace("Bearer ", "", $authHeader);

try {
    $decoded = JWT::decode($jwt, new Key($secretKey, 'HS256'));

    echo json_encode([
        "message" => "Token is valid",
        "data" => $decoded
    ]);
} catch (Exception $e) {
```

OAuth2 Grant Types Overview

What is OAuth2?

OAuth2 is a **standard way for apps to access resources on behalf of a user** — without sharing the user's password.

Example: You log in to a third-party app with **"Login with Google"**.

- That app never sees your Google password.
- Instead, Google gives it a **token** (limited access).

Main Grant Types (Flows)

OAuth2 has different "grant types" = **ways to get tokens**, depending on the situation.

1. Authorization Code Grant (most common, for web & mobile apps)

- **Used when:** You have a frontend + backend (server) app.
- **Flow:**
  1. User logs in via Google/Facebook/etc.
  2. Authorization server redirects with a **code**.
  3. Your backend exchanges the code for an **access token**.
- **Why:** Secure, because the token is only sent to your backend.

👉 Example: "Login with Google" button on a website.

2. Client Credentials Grant (machine-to-machine)

- **Used when:** No user, only servers talking to each other.
- **Flow:**
  1. App sends its **client ID + secret** to the auth server.
  2. Gets back an **access token**.
- **Why:** Good for background jobs, APIs, microservices.

👉 Example: A cron job calling a payment API.

3. Password Grant (deprecated, but worth knowing)

- **Used when:** User directly gives their username & password to the app.
- **Flow:**
  1. User enters credentials in your app.
  2. Your app exchanges them for an **access token**.
- **Why not recommended:** Insecure (app sees the password).

👉 Example: Old apps before OAuth2 best practices.

4. Implicit Grant (legacy, for SPAs)

- **Used when:** Pure frontend (Single Page Applications).
- **Flow:**
  1. User logs in.
  2. Auth server **sends token directly to browser** (no backend).
- **Why not recommended:** Token is exposed to browser/URL, less secure.

👉 Example: Angular/React app without backend (legacy way).

5. Device Code Grant (for devices with no keyboard)

- **Used when:** Smart TVs, IoT, consoles.
- **Flow:**
  1. Device shows a code on screen.
  2. User goes to a link on phone/computer and enters code.
  3. Auth server then issues token to device.

👉 Example: Logging into YouTube on a TV.

```php
$input = json_decode(file_get_contents('php://input'), true, 512, JSON_THROW_ON_ERROR);
$clientId = $input['client_id'] ?? '';
$clientSecret = $input['client_secret'] ?? '';

// In production, validate against a clients table
if ($clientId === 'client123' && $clientSecret === 'secret123') {
    $payload = [
        'access_token' => bin2hex(random_bytes(16)),
        'token_type' => 'Bearer',
        'expires_in' => 3600
    ];
    http_response_code(200);
    echo json_encode($payload, JSON_THROW_ON_ERROR);
} else {
    http_response_code(401);
    echo json_encode(['error' => 'Invalid client credentials', 'code' =>
    'INVALID_CLIENT'], JSON_THROW_ON_ERROR);
}
```

Rate Limiting & Throttling

What is Rate Limiting?

Rate limiting = **controlling how many requests a client can make in a certain time**.

Example:

- API allows **100 requests per minute per user**.
- If a client sends the 101st request → API returns error (usually **`429 Too Many Requests`**).

- **Why Rate Limit?**

  - Protect server from overload (DDOS, bots).
  - Control costs (if your API calls expensive services).
  - Fair usage (one client shouldn't block others).

- **What is Throttling?**

  Throttling = **slowing down requests** instead of blocking.

  Example:

  - API allows 10 requests per second.
  - If client sends 20 requests at once, server processes them slowly (queue/delay) instead of rejecting.

🔹 Common Rate Limiting Strategies

1. **Fixed Window**
   - Count requests in a fixed time window (e.g., 1 minute).
   - Simple, but can allow bursts at window edges.
2. **Sliding Window**
   - Counts requests in the *last X seconds/minutes*.
   - More accurate, smoother.
3. **Token Bucket / Leaky Bucket**
   - Each client gets a "bucket" of tokens (say 100).
   - Every request consumes 1 token.
   - Tokens refill slowly (e.g., 10 per minute).

👉 This prevents bursts but still allows some flexibility.

1. Fixed Window (simplest, but unfair at boundaries)

- **Rule:** Allow max `N` requests per time window (say 100 per minute).
- **Example:**
  - Window: `12:00:00–12:00:59` → you can make 100 requests.
  - At `12:01:00`, window resets → you get a fresh 100.
- **Problem:** If you send 100 requests at `12:00:59` and 100 more at `12:01:00`, you effectively send **200 in 2 seconds**.

2. Sliding Window (fairer than fixed)

- **Rule:** Count requests in the last `N` seconds dynamically (sliding window).
- **Example:**
  - Limit: 100 requests per minute.
  - At `12:01:30`, server looks back **60 seconds (12:00:30–12:01:30)**.
  - If you've made fewer than 100 requests in that period, request is allowed.
- **Benefit:** Smooths out the boundary problem.

- **Headers for Rate Limiting:**
  - `X-RateLimit-Limit`: Maximum requests allowed.
  - `X-RateLimit-Remaining`: Remaining requests in the window.
  - `Retry-After`: Seconds until the limit resets (for 429 responses).
  - Example:

```php
header('X-RateLimit-Limit: 100');
header('X-RateLimit-Remaining: ' . (100 - $requests));
```

Example Response Headers:

```makefile
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 20
X-RateLimit-Reset: 1668456000
```

If limit exceeded → return:

```http
429 Too Many Requests
```

- **API Docs (OpenAPI):**

OpenAPI (formerly **Swagger**) is a standard for documenting REST APIs, providing a machine-readable specification (JSON/YAML) and tools for interactive documentation.

Key Components:

- **Info**: API title, version, description.
- **Paths**: Endpoints and operations (e.g., GET /students).
- **Schemas**: Data models (e.g., Student object).
- **Security**: Authentication methods (e.g., API key, OAuth2).

🔹 What is OpenAPI?

- **OpenAPI** = a **standard format** (JSON or YAML) for describing REST APIs.
- Previously called **Swagger**.
- It defines:
  - Endpoints (`/users`, `/tasks`)
  - HTTP methods (`GET`, `POST`, `PUT`, `DELETE`)
  - Request parameters & bodies
  - Responses (success & error)
  - Authentication (API keys, JWT, OAuth2)

👉 Benefit:

- One **single source of truth** for API docs.
- Can auto-generate docs, client SDKs, and server stubs.

Example OpenAPI Spec (YAML)

```yaml
openapi: 3.0.0
info:
  title: Task API
  description: Simple API for managing tasks
  version: 1.0.0

servers:
  - url: http://localhost:3000

paths:
  /tasks:
    get:
      summary: Get all tasks
      responses:
        '200':
          description: A list of tasks
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Task'
```

```yaml
    post:
      summary: Create a new task
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/TaskInput'
      responses:
        '201':
          description: Task created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Task'
```

```yaml
  /tasks/{id}:
    get:
      summary: Get a single task
      parameters:
        - name: id
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: A single task
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Task'
        '404':
          description: Task not found
```

```yaml
  components:
    schemas:
      Task:
        type: object
        properties:
          id:
            type: integer
          title:
            type: string
          completed:
            type: boolean

      TaskInput:
        type: object
        required: [title]
        properties:
          title:
            type: string
```

How It Looks in Swagger UI

If you load the above YAML in **Swagger UI**, you get:

- ✅ Interactive documentation (with Try It Out button).
- ✅ Auto-generated forms for requests.
- ✅ Clear error codes.

Tools for OpenAPI

- **Swagger UI** → renders OpenAPI as interactive docs.
- **Swagger Editor** → edit YAML/JSON and preview instantly.
- **Swagger Codegen / OpenAPI Generator** → generate SDKs & stubs.
- **Postman** → can import/export OpenAPI specs.

Best Practices

- Always keep OpenAPI spec **in sync with code.**
- Document **error responses** (`400`, `401`, `404`, `500`).
- Add **examples** in request/response bodies.
- Document **authentication methods** (API key, JWT, OAuth2).

Choose 2 public APIs (e.g., Weather, News) • Data fetching & caching • Cron for refresh • Error retries/backoff

- **Choose 2 public APIs (e.g., Weather, News):**
  - **Open-Meteo Weather API**: A free, open-source weather API with no API key required. It provides current weather and forecasts via simple GET requests (e.g., for a location like a university campus in New York).
    Endpoint: https://api.open-meteo.com/v1/forecast?latitude=40.7128&longitude=-74.0060&current_weather=true&hourly=temperature_2m.

    https://openweathermap.org/

  - **NewsAPI**: A free news API (developer plan) that requires a free API key (sign up at newsapi.org). It fetches top headlines or search results (e.g., university-related news).
    Endpoint:
    https://newsapi.org/v2/topheadlines?country=us&category=science&apiKey=YOUR_API_KEY.

Php Laravel

1. **Weather API** → OpenWeatherMap ↗
   - Example endpoint:

```bash
GET https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY
```

2. **News API** → NewsAPI.org ↗
   - Example endpoint:

```bash
GET https://newsapi.org/v2/top-headlines?country=us&apiKey=YOUR_API_KEY
```

Both need an **API key**, which you send as a query parameter or header.

- ## Data Fetching

Data fetching involves making HTTP requests to the APIs using cURL or similar. Handle JSON responses and store raw or parsed data.

Fetching from Open-Meteo Weather API

- **Example** (Current weather for New York):

```php
<?php
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://api.open-meteo.com/v1/forecast?latitude=40.7128&l
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_TIMEOUT, 10);
$response = curl_exec($ch);
$httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);

if ($response === false || $httpCode !== 200) {
    echo 'Error fetching weather: ' . curl_error($ch);
} else {
    $data = json_decode($response, true);
    echo json_encode([
        'temperature' => $data['current_weather']['temperature'],
        'windspeed' => $data['current_weather']['windspeed']
    ]);
}
curl_close($ch);
?>
```

- **Sample Response** (parsed): `{"temperature":22.5,"windspeed":5.2}`.

```php
function fetchApi($url, $headers = []) {
    $ch = curl_init($url);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    if (!empty($headers)) {
        curl_setopt($ch, CURLOPT_HTTPHEADER, $headers);
    }

    $response = curl_exec($ch);

    if (curl_errno($ch)) {
        throw new Exception("cURL Error: " . curl_error($ch));
    }

    $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    curl_close($ch);

    if ($httpCode !== 200) {
        throw new Exception("API returned status $httpCode");
    }

    return json_decode($response, true);
}

// Example usage:
$weather = fetchApi("https://api.openweathermap.org/data/2.5/weather?q=London&appid=API_KEY");
$news    = fetchApi("https://newsapi.org/v2/top-headlines?country=us&apiKey=API_KEY");
```

- ## Caching

To avoid hitting the API every request (which costs time & rate limit), cache the response.

Caching stores API data locally (e.g., in a database) to reduce requests, improve performance, and handle rate limits. Use timestamps to expire cache (e.g., refresh weather every 30 minutes, news every hour).

Php Laravel

```php
<?php
$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
$cacheKey = 'weather_ny'; // Unique key for location

// Check cache
$stmt = $pdo->prepare('SELECT data FROM weather_cache WHERE cached_at > NOW() - INTERVAL 30 MINUTE');
$stmt->execute();
$cached = $stmt->fetchColumn();

if ($cached) {
    echo $cached; // Return cached JSON
} else {
    // Fetch fresh data
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, 'https://api.open-meteo.com/v1/forecast?latitude=40.7128&longitude=-74.0060&current_weathe
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    $response = curl_exec($ch);
    curl_close($ch);

    if ($response) {
        $data = json_encode(['temperature' => json_decode($response, true)['current_weather']['temperature']]);
        $stmt = $pdo->prepare('INSERT INTO weather_cache (data, expires_at) VALUES (?, DATE_ADD(NOW(), INTERVAL 30 MINUTE))
        $stmt->execute([$data, $data]); // Assume cacheKey as unique index
        echo $data;
    }
}
```

- **Cron for Refresh**

  Cron is a time-based job scheduler in Unix-like operating systems, including Linux, used to automate system maintenance and administration tasks, such as running scripts or commands at specific times, dates, or intervals.

  Use cron jobs (on a server) to periodically refresh cached data automatically, ensuring up-to-date info without manual intervention.

- **Cron Setup** (e.g., via `crontab -e`):
  - Refresh weather every 30 minutes: `*/30 * * * * /usr/bin/php /path/to/refresh_weather.php`
  - Refresh news every hour: `0 * * * * /usr/bin/php /path/to/refresh_news.php`

- **Best Practices:**
  - Run cron as a non-root user.
  - Log outputs (e.g., `>> /var/log/refresh.log 2>&1`).
  - Handle failures gracefully (e.g., email alerts).

```text
/to/your/project/refresh_data.php >> /var/log/cron_refresh.log 2>&1
```

Php Laravel

```php
<?php
// Fetch fresh weather and update cache
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, 'https://api.open-meteo.com/v1/forecast?latitude=40.7128&l
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
$response = curl_exec($ch);
curl_close($ch);

if ($response) {
    $pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');
    $data = json_encode(json_decode($response, true)['current_weather']);
    $stmt = $pdo->prepare('UPDATE weather_cache SET data = ?, cached_at = NOW(), expires_
    $stmt->execute([$data]);
    if ($stmt->rowCount() == 0) {
        $stmt = $pdo->prepare('INSERT INTO weather_cache (data, cached_at, expires_at) V
        $stmt->execute([$data]);
    }
    echo 'Weather refreshed successfully.';
} else {
    echo 'Failed to refresh weather.';
}
```

```
t_weather']);
, cached_at = NOW(), expires_at = DATE_ADD(NOW(), INTERVAL 30 MINUTE) WHERE id = 1');


ta, cached_at, expires_at) VALUES (?, NOW(), DATE_ADD(NOW(), INTERVAL 30 MINUTE))');
```

- **In Windows this can be done using the task Scheduler:**

1. Create a **script.bat** file and include the **php.exe** path and **cron.php** file path.

cron.php | script.bat | shellScript.vbs | errors.log .\

script.bat

```
"C:\xampp\php\php.exe" -f"C:\Aniket_PHP\cron.php"
```

2. Now Create a **ShellScript.vbs** file and include the **script.bat** path.

cron.php | script.bat | shellScript.vbs | errors.log .\ | Running E

shellScript.vbs

```
Set WinScriptHost = CreateObject("WScript.Shell")
WinscriptHost.Run Chr(34) & "C:\Aniket_PHP\script.bat" & Chr(34), 0
Set WindScriptHost = Nothing
```

3. Open Task Scheduler ->Create Task-> Give a NAME -> GO TO -> Triggers -> New -> (Select as you please) **[Enable]** -> GO TO -> Actions -> NEW -> SELECT OUR (**shellScript.vbs**) file -> OK -> CONDITIONS-> **Disable**[Start the task only if the Computer is on AC Power]

Php Laravel

```php
<?php
date_default_timezone_set("Asia/Kolkata");
try {
    try {
        $pdo = new PDO("mysql:host=localhost;dbname=ajax;charset=utf8mb4", "root", "");
    } catch (PDOException $e) {
        error_log(
            "\n PDO::Exception Connection failed on : " . date("l dS M Y h:i:s")
                . " error : " . $e->getMessage() . " code: " . $e->getCode(),
            3,
            __DIR__ . "/errors.log"
        );
    }

    $smt = $pdo->prepare("INSERT INTO user(`username`,`password`) VALUES('ABC','112121212')");
    if ($smt->execute()) {
        $id = $pdo->lastInsertId() ?? null;
        // echo "Inserted";
        error_log("\n Attempted Insert SuccessFull at index: $id at: " . date("l dS M Y h:i:s"), 3, __DIR__ . "/errors.log");
    } else {
        error_log(
            "\n Attempted Insert Failed at: " . date("l dS M Y h:i:s")
                . "error" . $smt->errorInfo(),
            3,
            __DIR__ . "/errors.log"
        );
    }
} catch (PDOException $e) {
    error_log(
        "\n PDO::Exception on : " . date("l dS M Y h:i:s")
            . " error : " . $e->getMessage() . " code: " . $e->getCode(),
        3,
        __DIR__ . "/errors.log"
    );
}
```

- **Error Retries and Exponential Backoff**

Sometimes APIs fail (network issue, 500 error). Best practice = **retry with backoff.**

Error Retries involve attempting a failed operation multiple times before giving up, while **Exponential Backoff** increases the delay between retries to avoid overwhelming the server during temporary issues (e.g., network timeouts or rate limits).

Implement retries with exponential backoff for transient errors (e.g., network issues, rate limits). Retry up to 3 times, doubling wait time (e.g., 1s, 2s, 4s).

Collapse Wrap Copy

```php
<?php
function fetchWithRetry($url, $maxRetries = 3) {
    $attempt = 0;
    while ($attempt < $maxRetries) {
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        curl_setopt($ch, CURLOPT_TIMEOUT, 10);
        $response = curl_exec($ch);
        $httpCode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
        $error = curl_error($ch);
        curl_close($ch);

        if ($response !== false && $httpCode === 200) {
            return $response; // Success
        }

        $attempt++;
        if ($attempt < $maxRetries) {
            $wait = pow(2, $attempt - 1); // Exponential backoff: 1s, 2s, 4s
            echo "Retry $attempt after $wait seconds (Error: " . ($error ?: "HTTP $httpC
            sleep($wait);
        } else {
            throw new Exception("Failed after $maxRetries attempts: " . ($error ?: "HTTP
        }
    }
}
```

```php
// Usage for Weather
try {
    $weatherData = fetchWithRetry('https://api.open-meteo.com/v1/forecast?latitude=40.71
    // Cache and process $weatherData
    echo json_decode($weatherData, true)['current_weather']['temperature'];
} catch (Exception $e) {
    echo 'Weather fetch failed: ' . $e->getMessage();
    // Fallback to cache or default
}
```

```php
function fetchWithRetry($url, $maxRetries = 3) {
    $attempt = 0;
    $wait = 1;

    while ($attempt < $maxRetries) {
        try {
            return fetchApi($url);
        } catch (Exception $e) {
            $attempt++;
            if ($attempt >= $maxRetries) {
                throw $e; // rethrow last error
            }
            sleep($wait);   // wait before retry
            $wait *= 2;     // exponential backoff (1s, 2s, 4s...)
        }
    }
}
```

Search & filtering • Pagination • Export CSV/JSON • Dockerize local run (optional)

- ## Search & Filtering

Search & Filtering allows clients to narrow down results using multiple criteria (exact matches, ranges, text search, etc.).

Search and filtering allow clients to query specific data using parameters (e.g., name, departmentId) to narrow results.

- **Implementation**:
  - Use query parameters for filtering (e.g., ?name=Alice&departmentId=1).
  - Support partial matches for search (e.g., LIKE for names).
  - Sanitize inputs to prevent SQL injection.

```php
<?php
// api/students.php
header('Content-Type: application/json');
$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');

// Build query with filters
$query = 'SELECT s.StudentID, s.Name, s.Email, d.DepartmentName
          FROM Students s
          JOIN Departments d ON s.DepartmentID = d.DepartmentID
          WHERE 1=1';
$params = [];

// Search by name (partial match)
if (isset($_GET['name']) && !empty($_GET['name'])) {
    $query .= ' AND s.Name LIKE ?';
    $params[] = '%' . $_GET['name'] . '%';
}

// Filter by department
if (isset($_GET['departmentId']) && is_numeric($_GET['departmentId'])) {
    $query .= ' AND s.DepartmentID = ?';
    $params[] = (int)$_GET['departmentId'];
}

// Execute query
$stmt = $pdo->prepare($query);
$stmt->execute($params);
$students = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Request Example: `GET /api/students?name=Ali&departmentId=1`

- ## Pagination

Pagination limits the number of results returned, improving performance and usability.

Implementation:

- Use **page** and **limit** query parameters.
- Include metadata (e.g., total records, pages) in responses.
- Use SQL **LIMIT** and **OFFSET** for pagination.

```php
<?php
// api/students.php
header('Content-Type: application/json');
$pdo = new PDO('mysql:host=localhost;dbname=university', 'root', '');

$page = isset($_GET['page']) ? max(1, (int)$_GET['page']) : 1;
$limit = isset($_GET['limit']) ? max(1, min(100, (int)$_GET['limit'])) : 10;
$offset = ($page - 1) * $limit;

// Build query with filters
$query = 'SELECT s.StudentID, s.Name, s.Email, d.DepartmentName
          FROM Students s
          JOIN Departments d ON s.DepartmentID = d.DepartmentID
          WHERE 1=1';
$params = [];

if (isset($_GET['name']) && !empty($_GET['name'])) {
    $query .= ' AND s.Name LIKE ?';
    $params[] = '%' . $_GET['name'] . '%';
}
if (isset($_GET['departmentId']) && is_numeric($_GET['departmentId'])) {
    $query .= ' AND s.DepartmentID = ?';
    $params[] = (int)$_GET['departmentId'];
}
```

```php
// Count total records
$countQuery = 'SELECT COUNT(*) FROM Students s WHERE 1=1' . (isset($_GET['name']) ? ' AN
$countStmt = $pdo->prepare($countQuery);
$countStmt->execute($params);
$total = $countStmt->fetchColumn();

// Add pagination to query
$query .= ' LIMIT ? OFFSET ?';
$params[] = $limit;
$params[] = $offset;

$stmt = $pdo->prepare($query);
$stmt->execute($params);
$students = $stmt->fetchAll(PDO::FETCH_ASSOC);

echo json_encode([
    'data' => $students,
    'pagination' => [
        'page' => $page,
        'limit' => $limit,
        'total' => $total,
        'pages' => ceil($total / $limit)
    ]
]);
?>
```

```
GET /api/students?page=2&limit=5&name=Ali
```

- ## Export CSV/JSON

Allow clients to export data in CSV or JSON format for analysis or reporting.

Implementation:

- Add a **format** query parameter (e.g., **?format=csv** or **?format=json**).
- For CSV, stream headers and data to avoid memory issues.
- For JSON, reuse the existing response format.

```php
// Set headers for JSON download
header('Content-Type: application/json; charset=utf-8');
header('Content-Disposition: attachment; filename="' . $safeFilename . '"');
header('Cache-Control: no-cache, no-store, must-revalidate');
```

1. JSON Export (default in APIs)

JSON is standard for REST APIs. You already do this with `header("Content-Type: application/json")`.

```php
<?php
header("Content-Type: application/json");

$data = [
    ["id" => 1, "name" => "Alice", "email" => "alice@example.com"],
    ["id" => 2, "name" => "Bob", "email" => "bob@example.com"],
];

echo json_encode($data, JSON_PRETTY_PRINT);
```

2. CSV Export

CSV (Comma-Separated Values) is useful for Excel/Sheets.

```php
<?php
header("Content-Type: text/csv");
header("Content-Disposition: attachment; filename=\"users.csv\"");

$data = [
    ["id" => 1, "name" => "Alice", "email" => "alice@example.com"],
    ["id" => 2, "name" => "Bob", "email" => "bob@example.com"],
];

$output = fopen("php://output", "w");

// Write header row
fputcsv($output, ["ID", "Name", "Email"]);

// Write data rows
foreach ($data as $row) {
    fputcsv($output, $row);
}

fclose($output);
```

Php Laravel

```php
// Export based on format
if ($format === 'csv') {
    header('Content-Type: text/csv');
    header('Content-Disposition: attachment; filename="students.csv"');
    $output = fopen('php://output', 'w');
    fputcsv($output, ['StudentID', 'Name', 'Email', 'DepartmentName']); // Headers
    foreach ($students as $student) {
        fputcsv($output, $student);
    }
    fclose($output);
} else {
    header('Content-Type: application/json');
    $countStmt = $pdo->prepare('SELECT COUNT(*) FROM Students s WHERE 1=1' . (isset($_GE
    $countStmt->execute(array_slice($params, 0, -2));
    $total = $countStmt->fetchColumn();

    echo json_encode([
        'data' => $students,
        'pagination' => [
            'page' => $page,
            'limit' => $limit,
            'total' => $total,
            'pages' => ceil($total / $limit)
        ]
    ]);
}
?>
```

- **Request Examples:**
  - JSON: `GET /api/students?page=1&limit=5&format=json`
    - Returns JSON as shown above.
  - CSV: `GET /api/students?page=1&limit=5&format=csv`

- **Dockerize Local Run (Optional)**

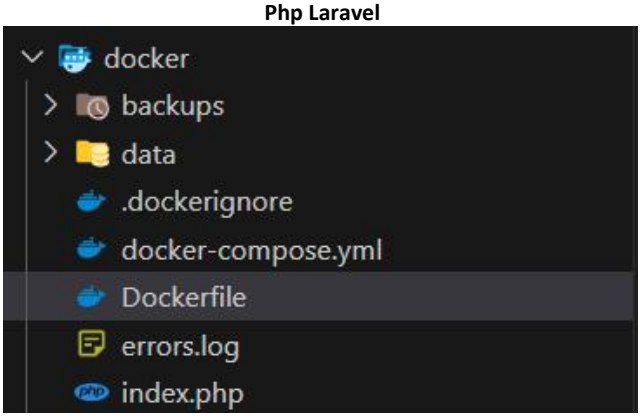
Docker containerizes the API, database, and dependencies for consistent local development.

Docker is a **platform** (or tool) that helps you build, run, and manage applications inside **containers**.

A **container** is like a lightweight, portable box that contains everything your application needs to run:

- the code
- libraries
- dependencies
- configurations

Php Laravel



kerignore | docker-compose.yml | Dockerfile | cron.php | script.bat

Dockerfile > ...

```
# Use official PHP with Apache
FROM php:8.2-apache

# Install PHP extensions required for MySQL
RUN docker-php-ext-install mysqli pdo pdo_mysql

# Enable Apache mod_rewrite (good for frameworks like Laravel/CodeIgniter)
RUN a2enmod rewrite

# Copy composer from the composer image (fast & convenient)
COPY --from=composer:2 /usr/bin/composer /usr/bin/composer

# Install PHP dependencies
# RUN composer install --no-dev --optimize-autoloader

# Set working directory
WORKDIR /var/www/html

# Copy your application source
COPY . /var/www/html/

# Set permissions
# RUN chown -R www-data:www-data /var/www/html \
#     && chmod -R 755 /var/www/html

# Expose Apache port
EXPOSE 80
```

.dockerignore | docker-compose.yml | Dockerfile

docker > docker-compose.yml

```
# version: "3.8"
services:
  app:
    build: .
    container_name: php_app
    volumes:
      - .:/var/www/html
    ports:
      - "3000:80"
    depends_on:
      - db
    environment:
      DOCKER_ENV: "1"

  db:
    image: mysql:8.0
    container_name: mysql_db
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: mydb
      MYSQL_USER: myuser
      MYSQL_PASSWORD: mypass
    volumes:
      - ./data/mysql:/var/lib/mysql
    ports:
      - "3306:3306"

  phpmyadmin:
    image: phpmyadmin:latest
    container_name: phpmyadmin
    restart: always
    ports:
      - "8081:80"
    environment:
      PMA_HOST: db
      PMA_PORT: 3306
      MYSQL_USER: myuser
      MYSQL_PASSWORD: mypass
      MYSQL_ROOT_PASSWORD: rootpass
    depends_on:
      - db

  backup:
    image: mysql:8.0
    container_name: mysql_backup
    depends_on:
      - db
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
    volumes:
      - ./backups:/backups
    entrypoint: >
      sh -c 'mysqldump -h db -u root -prootpass mydb > /backups/mydb.sql'
```

.dockerignore | doc

docker > .dockerignore

```
data/
error.log
mydb.sql
```

Php Laravel

```
docker > index.php > ...
<?php
date_default_timezone_set("Asia/Kolkata");

$host = getenv('DOCKER_ENV') === '1' ? 'db' : '127.0.0.1';
// print_r($host);
try {
    try {
        $pdo = new PDO(
            "mysql:host=$host;port=3306;dbname=mydb;charset=utf8mb4",
            "root",
            "rootpass"
        );

        // $pdo = new PDO("mysql:host=db;dbname=mydb;charset=utf8mb4", "root", "rootpass");


        //  $pdo = new PDO(
        //      "mysql:host=127.0.0.1;port=3306;dbname=mydb;charset=utf8mb4",        //PHP on host machine → host=127.0.0.1 (or lo
        //      "root",
        //      "rootpass"
        // );
        error_log("\n login success at: " . date("l dS M Y h:i:s"), 3, __DIR__ . "/errors.log");
    } catch (PDOException $e) {
        error_log(
            "\n PDO::Exception Connection failed on : " . date("l dS M Y h:i:s")
                . " error : " . $e->getMessage() . " code: " . $e->getCode(),
            3,
            __DIR__ . "/errors.log"
        );
    }
```

- **To start the docker Container:**

```
PS C:\Aniket_PHP\docker> docker-compose up -d
```

```
docker-compose up -d --build
```

```
PS C:\Aniket_PHP\docker> docker-compose ps
NAME          IMAGE                COMMAND
mysql_db      mysql:8.0            "docker-entrypoint.s…"
php_app       docker-app           "docker-php-entrypoi…"
phpmyadmin    phpmyadmin:latest    "/docker-entrypoint.…"
```

```
PS C:\Aniket_PHP\docker> docker ps
CONTAINER ID   IMAGE                COMMAND
1ebd07314663   phpmyadmin:latest    "/docker-entrypoint.…"
6c409eae2b5d   docker-app           "docker-php-entrypoi…"
4860d2c86792   mysql:8.0            "docker-entrypoint.s…"
PS C:\Aniket_PHP\docker>
```

- **To stop the docker container:**

```
PS C:\Aniket_PHP\docker> docker-compose down
```

- **Accessing the Mysql through CLI:**

```
docker exec -it mysql_db mysqldump -u root -p mydb > mydb.sql
```

Here the "***mysql_dump***" is the mysql service container name in ***docker-compose.json***

```
PS C:\Aniket_PHP\docker> docker-compose exec db mysql -u root -prootpass mydb -e "select * from users"
```

Here "***db***" is ***service name*** of the mysql.

🚀 **Run the Backup**

Now run:

```sh
docker-compose up backup
```

After running "**docker-compose up backup**" a backup file will be created in backups filder named "**mydb.sql**"

- **How it works:**
  - **App (PHP + Apache)** → http://`localhost`:3000
  - **phpMyAdmin** → http://`localhost`:8081
  - **MySQL** → port 3306 (so you can connect with tools like MySQL Workbench)

- **Database stored persistently in ./data/mysql**

```yaml
# MySQL Database
mysql:
  image: mysql:8.0
  container_name: mysql_db
  ports:
    - "3306:3306"
  environment:
    - MYSQL_ROOT_PASSWORD=root_password
    - MYSQL_DATABASE=app_db
    - MYSQL_USER=app_user
    - MYSQL_PASSWORD=secret_password
  volumes:
    - mysql_data:/var/lib/mysql
    - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  networks:
    - app_network

# Redis for caching and rate limiting
redis:
  image: redis:7-alpine
  container_name: redis_cache
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  networks:
    - app_network
```

```yaml
volumes:
  mysql_data:
  redis_data:

networks:
  app_network:
    driver: bridge
```

Here:

The line **./init.sql:/docker-entrypoint-initdb.d/init.sql** in **docker-compose.yml** is a volume mount that tells Docker to copy a local file (**init.sql**) into the MySQL container's initialization directory.

How It Works

When you run `docker-compose up`, Docker:

1. **Starts the MySQL container** (the `db` service)
2. **Checks if the database is initialized** (first run only)
3. **Copies your** `init.sql` **file** into `/docker-entrypoint-initdb.d/init.sql` inside the container
4. **Executes the SQL commands** in alphabetical order
5. **Creates the database, tables, and inserts sample data** as defined in your `init.sql`

The line - db-data:/var/lib/mysql ensures that your data is persistently stored and automatically restored when you restart the container.

How Persistent Storage Works

What is a Docker Volume?

- A **Docker volume** is a storage mechanism that exists **outside** of the container lifecycle
- When you define `db-data:/var/lib/mysql`, Docker creates a **named volume** called `db-data`
- This volume stores MySQL data files **independently** of the container

Project setup (Laravel + Sail/Vite) • Folder structure & lifecycle • Config & env • Routing basics • Controllers

- **Project setup (Laravel + Sail/Vite):**

1. Install Laravel (via Composer)

If you already have **Composer** installed:

bash Copy code

```bash
composer create-project laravel/laravel my-app
cd my-app
```

This creates a fresh Laravel project in `my-app/`.

We can directly run the laravel project locally using the "**php artisan serve**". You can see ur application live on **localhost.**

To Dockerize:

2. Laravel Sail (Dockerized Environment)

Sail = Laravel's built-in Docker environment (PHP, MySQL, Redis, etc.).

Install Sail

bash Copy code

```bash
composer require laravel/sail --dev
```

Publish Sail config:

bash Copy code

```bash
php artisan sail:install
```

Choose services (e.g., MySQL, Redis, Meilisearch, etc.).

This will generate a `docker-compose.yml`.

"**composer require laravel/sail --dev**" is not required only the second command.

Step 2: Set Up Laravel Sail (Docker for Local Dev)

Sail uses Docker to run your app in containers (like mini virtual machines) so everything works the same on any computer.

- Assuming Docker is installed and running, add Sail to your project:

```text
composer require laravel/sail --dev
```

- Install Sail's files:

```text
php artisan sail:install
```

- Start Sail (this might take a bit the first time as it downloads stuff):

```text
./vendor/bin/sail up
```

"./vendor/bin/sail up" won't work as it requires **windows subsystem for linux(WSL)** and **docker** so we need to install linux subsytem on windows.

Open terminal:

1. **wsl --install** OR **wsl --install -d ubuntu**
2. **wsl -l -v** :to list all the subsystem
3. **wsl --set-default ubuntu**
4. **wsl -d ubuntu** OR **wsl** :to start all the instances
5. **wsl --terminate ubuntu :** to start only ubuntu
6. **wsl --shutdown** : to stop all the services

After installing wsl and docker start both of them and verify using the "**wsl -l -v**". and run this command.

Run migrations:

```bash
./vendor/bin/sail artisan migrate
```

4. Vite (Frontend Assets)

Laravel 9+ uses **Vite** (instead of Mix) for CSS/JS bundling.

Install dependencies:

```bash
./vendor/bin/sail npm install
```

Run dev server:

```bash
./vendor/bin/sail npm run dev
```

Now run "**./vendor/bin/sail up -d**" this will start the docker container and the laravel project will run independently as a docker container on "**https://localhost**" in case of port error make changes in **compose.yml** file

To close the application "**./vendor/bin/sail down**"

Note:

To avoid writing "**./vendor/bin/sail**" again and again we can use alias.

In linux terminal:

- **alias sail="bash vendor/bin/sail"**

Now no need to write ./vendor/bin/sail just use sail like "**sail up -d**" and "**sail down**"

Global Installation (Rarely used):

Step 1: Install Laravel

- Open your terminal (command prompt).
- Run this to install the Laravel installer (a helper tool):

text Collapse Wrap Copy

```
composer global require laravel/installer
```

- Create a new project called "myapp" (change the name if you want):

text Collapse Wrap Copy

```
laravel new myapp
```

- Go into the project folder:

text Collapse Wrap Copy

```
cd myapp
```

This creates a basic Laravel app. laravel.com

- **Folder structure & lifecycle:**

Folder Structure

Laravel's folders are organized to keep things tidy. Here's a simple breakdown of the main ones:

- **app/**: Your app's core code. Subfolders like:
  - **Http/**: Controllers (handle requests) and Middleware (filters requests).
  - **Models/**: Database models (like User).
  - **Providers/**: Service providers (set up app features).
- **bootstrap/**: Starts the app (don't touch much).
- **config/**: Settings files (like app.php for timezone).
- **database/**: Migrations (create tables), seeds (sample data).
- **public/**: Public files like index.php (entry point), CSS/JS.
- **resources/**: Views (HTML templates), raw CSS/JS.
- **routes/**: Route files like web.php (define URLs).
- **storage/**: Logs, caches, uploaded files.
- **tests/**: For writing tests.
- **vendor/**: Downloaded packages (don't edit).

You can add folders as needed, but this is the default. laravel.com

Php Laravel

```
university-management/
├── app/                    # Core application code
│   ├── Http/               # HTTP request handling
│   │   ├── Controllers/    # Controllers
│   │   ├── Middleware/     # Request middleware
│   │   └── Requests/       # Form request validation
│   ├── Models/             # Eloquent models
│   └── Providers/          # Service providers
├── bootstrap/              # Application bootstrap
├── config/                 # Configuration files
├── database/               # Database migrations/seeds
│   ├── migrations/         # Database schema
│   └── seeders/            # Database seeders
├── public/                 # Web server document root
├── resources/              # Frontend assets
│   ├── css/                # CSS files
│   ├── js/                 # JavaScript files
│   └── views/              # Blade templates
├── routes/                 # Route definitions
│   ├── web.php             # Web routes
│   ├── api.php             # API routes
│   └── console.php         # Artisan commands
├── storage/                # File storage/logs
├── tests/                  # Test files
├── vendor/                 # Composer dependencies
├── .env                    # Environment variables
├── artisan                 # Laravel CLI
├── composer.json           # PHP dependencies
└── vite.config.js          # Vite configuration
```

- Application Lifecycle:

Request Lifecycle (How a Request Works)

Think of it like a journey:

1. User visits a URL → Goes to public/index.php.
2. Index.php loads the app from bootstrap/app.php.
3. The HTTP Kernel (in app/Http/Kernel.php) starts: Sets up error handling, logging.
4. Service Providers (in bootstrap/providers.php) load features like database, routes.
5. Router checks routes (in routes/web.php) and sends request to a controller or function.
6. Controller runs code (e.g., gets data from DB) and returns a response (like a view).
7. Response goes back through middleware (e.g., checks login) and to the user.

Simple: Request in → Setup → Route → Controller → Response out. laravel.com

Application Lifecycle

1. **Entry Point:** `public/index.php` receives HTTP request
2. **Bootstrap:** Loads `bootstrap/app.php` and autoloader
3. **Kernel:** `app/Http/Kernel.php` processes middleware
4. **Routing:** `routes/web.php` or `routes/api.php` matches route
5. **Controller:** Executes controller action
6. **Response:** Returns view, JSON, or redirect
7. **Middleware:** Post-processing (response middleware)

Diagram (Simplified):

```pgsql
Client Request
      ↓
public/index.php
      ↓
Bootstrap Application
      ↓
HTTP Kernel (Middleware)
      ↓
Router (web.php / api.php)
      ↓
Controller / Closure
      ↓
Response (JSON / View)
      ↓
Client
```

- ## Config & env:

Configuration Files

- In config/ folder, files like app.php set things like app name, timezone.
- Access them in code: `config('app.name')` (returns your app name).
- Cache for speed (production only): `php artisan config:cache`. Clear with `php artisan config:clear`.

.env File

- This is for secrets and env-specific settings (like local vs production).
- Copy .env.example to .env.
- Example: Set `APP_NAME=MyApp`, `DB_CONNECTION=mysql`, `DB_HOST=127.0.0.1`.
- In config files, use `env('APP_NAME', 'Default')` to get values (second is fallback).
- Never commit .env to Git (it's in .gitignore) – it has passwords!
- For debug: Set `APP_DEBUG=true` in .env to see errors (false in production).

Use `php artisan key:generate` to set APP_KEY in .env for security. laravel.com

1. What is `.env` in Laravel?

- `.env` is the **environment configuration file**.
- Stores settings that **change per environment**: local, staging, production.
- Examples: database credentials, API keys, debug mode, mail server.

> **Never commit `.env` to Git** — it contains sensitive info.

Example `.env` file

```env
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:randomgeneratedkey
APP_DEBUG=true
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=mysql
DB_PORT=3306
DB_DATABASE=my_app
DB_USERNAME=sail
DB_PASSWORD=password

CACHE_DRIVER=file
QUEUE_CONNECTION=sync
```

2. Accessing Env Variables in Laravel

Use the `env()` **helper** in `config` files:

```php
// config/database.php
'mysql' => [
    'driver' => 'mysql',
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
],
```

- If the env variable is missing, Laravel uses the **default value** (second argument).

🔹 3. Config Files (`config/`)

- Laravel stores all app configs in the `config/` folder.
- Examples:
  - `app.php` → app name, timezone, locale
  - `database.php` → DB connections
  - `mail.php` → mail settings
  - `cache.php` → cache drivers
- You can **access config anywhere** using the `config()` helper:

```php
echo config('app.name');       // Laravel
echo config('database.mysql.host'); // mysql
```

- **Routing basics:**

🔹 1. What is Routing?

Routing is **mapping URLs to code**. When a user visits a URL, Laravel checks the **route definitions** and runs the corresponding **controller or closure**.

- Browser → `/users` → Controller returns response.
- API → `/api/tasks` → Returns JSON.

🔹 2. Route Files

- `routes/web.php` → Routes for **browser / web pages**
- `routes/api.php` → Routes for **APIs** (usually return JSON)
- `routes/console.php` → Routes for **Artisan commands**
- `routes/channels.php` → Routes for **broadcasting channels**

> Routes in `api.php` are **prefixed with** `/api` automatically and usually **stateless**.

3. Basic Routes

Closure-based route

```php
// routes/web.php
Route::get('/', function () {
    return 'Hello, Laravel!';
});
```

Route returning JSON (API)

```php
// routes/api.php
Route::get('/tasks', function () {
    return response()->json([
        ['id' => 1, 'title' => 'Buy milk'],
        ['id' => 2, 'title' => 'Finish report']
    ]);
});
```

4. Route with Parameters

- **Required parameter**

```php
Route::get('/users/{id}', function ($id) {
    return "User ID: $id";
});
```

- **Optional parameter**

```php
Route::get('/users/{name?}', function ($name = 'Guest') {
    return "Hello, $name";
});
```

- Optional: /user or /user/John.

```text
Route::get('/user/{name?}', function ($name = 'Guest') {
    return 'Hello ' . $name;
});
```

- Constrain: ID must be number.

text

```
Route::get('/user/{id}', function ($id) {
    return 'User ' . $id;
})->whereNumber('id');
```

7. Route to Controller

Instead of closure, call **controller method**:

```php
Route::get('/tasks', [App\Http\Controllers\TaskController::class, 'index']);
```

- `TaskController@index` handles the request.
- Cleaner & easier to maintain for large apps.

Old Laravel <=7

new Laravel >=8

8. HTTP Verbs in Routing

Laravel supports all HTTP verbs:

```php
Route::get('/tasks', 'TaskController@index');     // Read
Route::post('/tasks', 'TaskController@store');    // Create
Route::put('/tasks/{id}', 'TaskController@update'); // Update
Route::delete('/tasks/{id}', 'TaskController@destroy'); // Delete
```

Route Examples

```php
Route::get('/hello', fn() => "Hello World");
Route::post('/users', [UserController::class, 'store']);
Route::put('/users/{id}', [UserController::class, 'update']);
Route::delete('/users/{id}', [UserController::class, 'destroy']);
```

Route Groups:

Example (without group)

```php
Route::get('/admin/dashboard', [AdminController::class, 'dashboard']);
Route::get('/admin/users', [AdminController::class, 'users']);
Route::get('/admin/settings', [AdminController::class, 'settings']);
```

Notice how `/admin/` repeats every time.

Example (with group)

```php
Route::prefix('admin')->group(function () {
    Route::get('/dashboard', [AdminController::class, 'dashboard']);
    Route::get('/users', [AdminController::class, 'users']);
    Route::get('/settings', [AdminController::class, 'settings']);
});
```

Middleware

✅ What is Middleware?

Middleware is like a **security guard** that checks your request **before it reaches your controller**.

Think of it like:

"Are you logged in?"
"Do you have admin access?"
"Are you making too many requests?"

🧠 Common Middleware in Laravel

Middleware	What it does
`auth`	Allows only logged-in users
`guest`	Allows only visitors (not logged-in users)
`throttle`	Limits request rate (for APIs)
`verified`	Ensures email is verified

💡 Example: Applying Middleware to a single route

```php
Route::get('/dashboard', [UserController::class, 'dashboard'])
    ->middleware('auth');
```

👉 This means only **logged-in users** can visit `/dashboard`.

💡 Example: Applying Middleware to a route group

```php
Route::middleware(['auth'])->group(function () {
    Route::get('/profile', [UserController::class, 'profile']);
    Route::get('/settings', [UserController::class, 'settings']);
});
```

Routing to a Controller:

🧩 Step 1: Create a Controller

You can create one using Artisan command:

```bash
php artisan make:controller UserController
```

This creates:

```swift
app/Http/Controllers/UserController.php
```

Step 2: Add a method in the controller

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    // This method will handle the /users route
    public function index()
    {
        $users = [
            ['id' => 1, 'name' => 'Aniket'],
            ['id' => 2, 'name' => 'Rahul'],
        ];

        return response()->json($users);
    }

    // Another method for single user
    public function show($id)
    {
        return response()->json(['id' => $id, 'name' => 'Aniket']);
    }
}
```

🧭 Step 3: Define Routes (web.php or api.php)

Now, in `routes/web.php` or `routes/api.php`, define routes that point to this controller.

```php
use App\Http\Controllers\UserController;

// Route for listing all users
Route::get('/users', [UserController::class, 'index']);

// Route for showing single user
Route::get('/users/{id}', [UserController::class, 'show']);
```

🧩 Step 5: Use RESTful Resource Routes (optional but powerful)

Instead of writing each route manually, Laravel can generate **all CRUD routes** automatically.

```bash
php artisan make:controller UserController --resource
```

Now, inside `routes/web.php`:

```php
Route::resource('users', UserController::class);
```

✅ This single line gives you 7 routes:

HTTP Method	URL	Controller Method	Purpose
GET	/users	index	Show all users
GET	/users/create	create	Show create form
POST	/users	store	Save user
GET	/users/{id}	show	Show single user
GET	/users/{id}/edit	edit	Show edit form
PUT/PATCH	/users/{id}	update	Update user
DELETE	/users/{id}	destroy	Delete user

You can run:

bash Copy co

```bash
php artisan route:list
```

- **Controllers:**

Controllers group route logic into classes.

🔹 1. What is a Controller?

- A **class** that groups related route handling logic.
- Keeps routes file **clean** (no long closures).
- Works with **Models**, **Requests**, and **Responses**.

🟢 What is a Controller?

Think of a controller as a **traffic manager** for your Laravel app.

- A **user makes a request** (like visiting `/users` in the browser).
- Laravel's **routes** decide *which controller* should handle it.
- The **controller** does the work (talks to the database, processes logic).
- Then it returns a **response** (like an HTML view or JSON data).

2. Creating a Controller

Using Artisan

bash

```bash
./vendor/bin/sail artisan make:controller TaskController
```

This creates:

swift

```swift
app/Http/Controllers/TaskController.php
```

```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TaskController extends Controller
{
    // Show all tasks
    public function index()
    {
        $tasks = [
            ['id' => 1, 'title' => 'Buy milk'],
            ['id' => 2, 'title' => 'Finish report']
        ];

        return response()->json($tasks);
    }

    // Show single task
    public function show($id)
    {
        return response()->json(['id' => $id, 'title' => 'Sample Task']);
    }
}
```

4. Routing to Controller

php

```php
// routes/api.php
use App\Http\Controllers\TaskController;

Route::get('/tasks', [TaskController::class, 'index']);      // GET /tasks
Route::get('/tasks/{id}', [TaskController::class, 'show']);  // GET /tasks/1
```

5. RESTful Resource Controller

Laravel can auto-generate **all CRUD methods**:

```bash
./vendor/bin/sail artisan make:controller TaskController --resource
```

This generates methods:

Method	Purpose	Route Example
index	List all tasks	GET /tasks
create	Show form to create task	GET /tasks/create
store	Store new task	POST /tasks
show	Show single task	GET /tasks/{id}
edit	Show form to edit task	GET /tasks/{id}/edit
update	Update task	PUT/PATCH /tasks/{id}
destroy	Delete task	DELETE /tasks/{id}

```bash
php artisan make:controller UserController --resource
```

Now, inside `routes/web.php`:

```php
Route::resource('users', UserController::class);
```

You can run:

```bash
php artisan route:list
```

6. Controller with Request Validation

```php
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|string|max:255',
        'completed' => 'boolean'
    ]);

    // Save task (example)
    $task = [
        'id' => rand(3, 100),
        'title' => $validated['title'],
        'completed' => $validated['completed'] ?? false
    ];

    return response()->json($task, 201);
}
```

- `Request` → captures user input
- `validate()` → ensures data is valid before stori

Php Laravel

```php
<?php

namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    public function index()
    {
        $users = User::all(); // fetch all users

        return view('users', ['users' => $users]);
    }
}
```

d) Create Blade View

`resources/views/users.blade.php`:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Users List</title>
</head>
<body>
    <h1>Users</h1>
    <ul>
        @foreach($users as $user)
            <li>{{ $user->name }} - {{ $user->email }}</li>
        @endforeach
    </ul>
</body>
</html>
```

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    public function store(Request $request)
    {
        // ✅ Step 1: Validate the request data
        $validated = $request->validate([
            'name' => 'required|string|max:255',   // must be text and not empty
            'email' => 'required|email|unique:users,email', // must be valid & unique
            'age' => 'nullable|integer|min:18' // optional, but must be 18+
        ]);

        // ✅ Step 2: If validation passes, continue
        // (you could save to DB here)
        return response()->json([
            'message' => 'User data is valid!',
            'data' => $validated
        ], 201);
    }
}
```

🧭 Step 3: Add a Route

In your `routes/api.php`:

```php
use App\Http\Controllers\UserController;

Route::post('/users', [UserController::class, 'store']);
```

🧪 Step 4: Test It

Send a **POST** request (using Postman or a form) to:

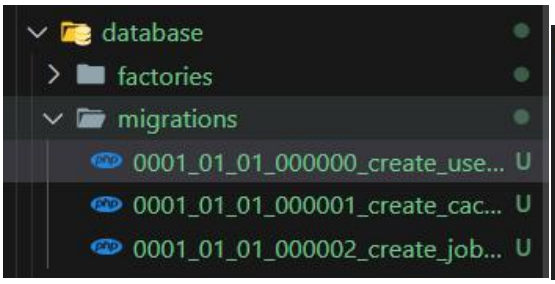
`http://localhost:8000/api/users`

✅ Example JSON Body:

```json
{
  "name": "Aniket",
  "email": "aniket@example.com",
  "age": 22
}
```

- **Model:** tables

Create a table in **PHPMYADMIN** or in **database/migrate/files**




```php
public function up(): void
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
```

```bash
php artisan migrate
```

1. Create a Model:

Run this command:

```bash
php artisan make:model Trail
```

This will create `app/Models/Trail.php`.

Then open `Trail.php` and make sure it looks like this:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Trail extends Model
{
    use HasFactory;

    // Specify table name if it doesn't follow Laravel naming convention
    protected $table = 'trail';

    // If you don't have created_at and updated_at columns
    public $timestamps = false;

    // Optional: specify which fields are mass assignable
    protected $fillable = ['name'];
}
```

Open `app/Models/Userss.php` and make it look like this:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Userss extends Model
{
    // Specify table name since it's not the default 'userss' -> 'userss'
    protected $table = 'userss';

    // Specify the primary key since it's 'user_id' not 'id'
    protected $primaryKey = 'user_id';

    // If your table doesn't have timestamps (created_at, updated_at)
    public $timestamps = false;

    // If you want to allow mass assignment
    protected $fillable = ['name'];
}
```

2: Create a Controller:

Create a controller that will fetch data:

```bash
php artisan make:controller TrailController
```

Php Laravel

Then in `app/Http/Controllers/TrailController.php`:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Trail;

class TrailController extends Controller
{
    public function index()
    {
        // Fetch all rows from 'trail' table
        $trails = Trail::all();

        // Return as JSON (for API) or pass to a Blade view
        return response()->json($trails);
        // OR to a view:
        // return view('trail.index', ['trails' => $trails]);
    }
}
```

3 Step 3: Define a Route

Open `routes/web.php` and add:

```php
use App\Http\Controllers\TrailController;

Route::get('/trails', [TrailController::class, 'index']);
```

4 Step 4: Test in Browser

Run the dev server:

```bash
php artisan serve
```

Then visit:

```arduino
http://127.0.0.1:8000/trails
```

5 Optional: Display in Blade View

If you want to show it in a HTML table instead of JSON:

1. Create a view `resources/views/trail/index.blade.php`:

```html
<h1>Trails</h1>
<ul>
    @foreach($trails as $trail)
        <li>{{ $trail->id }} - {{ $trail->name }}</li>
    @endforeach
</ul>
```

2. Change controller return:

```php
return view('trail.index', ['trails' => $trails]);
```

In Laravel, **Eloquent automatically assumes your table name** based on the **model name**:

- Model name: `Trial`
- Default table name: `trials` (plural, lowercase)

So if you don't specify `$table`, Laravel tries to query:

```sql
SELECT * FROM trials
```

...but your actual table is named `trial` (singular).

Blade layouts/components • Blade directives • Forms & CSRF • Form Request Validation • Localization (intro)

1. Blade Layouts/Components

Blade lets you create reusable layouts (like a master template) and components (small reusable UI bits, like buttons).

🧩 What is Blade?

Blade is a simple templating engine that allows you to write PHP inside HTML **without messy syntax**.

Example:

blade Copy code

```
<h1>Hello, {{ $name }}</h1>
```

If `$name = "Aniket"`, it prints:

Copy code

```
Hello, Aniket
```

🎯 Why use Blade Layouts & Components?

Because they help you **avoid repeating the same HTML** (like header, navbar, footer) across multiple pages.

✅ With **layouts**, you define a master structure.
✅ With **components**, you create small reusable pieces (like buttons or cards).

Step 1: Create a Layout

- Make a layout file: `resources/views/layouts/app.blade.php`

blade Collapse Wrap Copy

```blade
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>@yield('title', 'My App')</title>
    @vite(['resources/css/app.css', 'resources/js/app.js']) <!-- From Vite -->
    <style> body { font-family: Arial; margin: 20px; } </style>
</head>
<body>
    <header>
        <h1>My Laravel App</h1>
        <nav>
            <a href="/">Home</a> | <a href="/hello">Hello</a>
        </nav>
    </header>

    <main>
        @yield('content') <!-- Child pages put content here -->
    </main>

    <footer>
        <p>&copy; 2025 My App</p>
    </footer>
</body>
</html>
```

- `@yield('title')` pulls a title from child views (default 'My App'). `@yield('content')` is where page-specific stuff goes.

Step 2: Use the Layout in a View

- Create a view: `resources/views/welcome.blade.php`

blade Collapse Wrap Copy

```blade
@extends('layouts.app') <!-- Use the layout -->

@section('title', 'Welcome Page') <!-- Set title -->

@section('content')
    <h2>Hello, Fresher!</h2>
    <p>This is my first Blade view. {{ now()->format('Y') }} year.</p> <!-- Blade express
@endsection
```

Step 3: Route to the View

- In `routes/web.php`:

php Collapse

```php
Route::get('/', function () {
    return view('welcome'); // Returns resources/views/welcome.blade.php
});
```

🧩 Part 2: Blade Components (Reusable Pieces)

Components are **mini-templates** for small, repeating sections like buttons, cards, alerts, etc.

🧠 1. Create a Component

Run:

bash

```bash
php artisan make:component Alert
```

This creates:

swift

```swift
app/View/Components/Alert.php
resources/views/components/alert.blade.php
```

Php Laravel

- Edit the component view (`resources/views/components/alert.blade.php`):

blade Collapse Wrap Copy

```blade
<div style="background: {{ $type === 'error' ? 'red' : 'green' }}; color: white; padding
    {{ $slot }} <!-- Slot is the content passed in -->
</div>
```

- Edit the class (`app/View/Components/Alert.php`) if needed (for logic, but simple for now).

- Use in a view (e.g., welcome.blade.php):

blade Collapse Wrap Copy

```blade
@section('content')
    <x-alert type="success">This is a success message!</x-alert> <!-- x- means component
    <x-alert type="error">Error here!</x-alert>
@endsection
```

Anonymous Component (for one-offs, place in `resources/views/components/card.blade.php`):

blade Collapse Wrap

```blade
<div class="bg-white p-6 rounded shadow">
    {{ $slot }}
</div>
```

Usage:

blade Collapse Wrap

```blade
<x-card>
    <h2>Student Details</h2>
    <p>Name: {{ $student->name }}</p>
</x-card>
```

Php Laravel

```
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ config('app.name', 'University Management') }} - @yield('title')</title>
    @vite(['resources/css/app.css', 'resources/js/app.js']) <!-- Vite assets -->
    @stack('styles') <!-- For page-specific CSS -->
</head>
<body class="bg-gray-100 font-sans">
    <header class="bg-blue-600 text-white p-4">
        <nav class="container mx-auto flex justify-between">
            <a href="{{ route('dashboard') }}" class="text-xl font-bold">University App</a
            <div>
                <a href="{{ route('students.index') }}" class="mx-2">Students</a>
                <a href="{{ route('departments.index') }}" class="mx-2">Departments</a>
                @auth
                    <form method="POST" action="{{ route('logout') }}" class="inline">
                        @csrf
                        <button type="submit" class="mx-2">Logout</button>
                    </form>
                @else
                    <a href="{{ route('login') }}" class="mx-2">Login</a>
                @endauth
            </div>
        </nav>
    </header>

    <main class="container mx-auto mt-6 px-4">
        @yield('content')
    </main>

    <footer class="bg-gray-800 text-white p-4 mt-10">
        <div class="container mx-auto text-center">
            &copy; {{ date('Y') }} University Management. All rights reserved.
        </div>
    </footer>

    @stack('scripts') <!-- For page-specific JS -->
```

Welcome.blade.php

```
@extends('layouts.app')

@section('title', 'Students List')

@push('styles')
    <link href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2/dist/tailwind.min.css" rel="st
@endpush

@section('content')
    <div class="bg-white p-6 rounded shadow">
```

```
        {{ $students->links() }} <!-- Pagination links -->
    </div>
@endsection

@push('scripts')
    <script>
        console.log('Students page loaded');
    </script>
@endpush
```

🧩 `{{ route('students.index') }}` — What It Actually Means

That line refers to a **named route** in Laravel — not directly to a file or folder.

So:

> `students.index` = the **name** of a route, not the view or controller itself.

⚙️ Where the Name Comes From

When you define a route using `Route::resource()`, Laravel **automatically generates** a set of routes and names.

Example:

```php
Route::resource('students', StudentController::class);
```

This one line creates **7 routes automatically**, all with names like this:

HTTP Method	URL	Controller Method	Route Name
GET	/students	index()	students.index
GET	/students/create	create()	students.create
POST	/students	store()	students.store
GET	/students/{id}	show()	students.show
GET	/students/{id}/edit	edit()	students.edit
PUT/PATCH	/students/{id}	update()	students.update
DELETE	/students/{id}	destroy()	students.destroy

```php
Route::get('/dashboard', [DashboardController::class, 'index'])->name('dashboard');

Route::resource('students', StudentController::class);
Route::resource('departments', DepartmentController::class);

Route::get('/login', [AuthController::class, 'showLoginForm'])->name('login');
Route::post('/logout', [AuthController::class, 'logout'])->name('logout');
```

- **Blade directives:**

Php Laravel

These are **special** `@` **commands** in Blade that make your HTML templates dynamic, clean, and safe.

🧩 1. Output Directives

Directive	Description	Example
`{{ }}`	Echo escaped data (safe)	`{{ $user->name }}`
`{!! !!}`	Echo unescaped HTML (⚠️ use carefully)	`{!! $post->content !!}`
`@{{ }}`	Escape Blade when using in JS/Vue	`@{{ variable }}`

⚙️ 2. Conditional Directives

`@if`, `@elseif`, `@else`, `@endif`

```blade
@if($age >= 18)
  <p>Adult</p>
@elseif($age >= 13)
  <p>Teenager</p>
@else
  <p>Child</p>
@endif
```

`@unless` (opposite of `@if`)

```blade
@unless($user->isAdmin())
  <p>You are not an admin.</p>
@endunless
```

`@isset` and `@empty`

```blade
@isset($name)
  <p>Name: {{ $name }}</p>
@endisset

@empty($cart)
  <p>Your cart is empty!</p>
@endempty
```

🔁 3. Loop Directives

`@for`, `@foreach`, `@forelse`, `@while`

```blade
@for($i = 0; $i < 3; $i++)
  <p>Count: {{ $i }}</p>
@endfor
```

```blade
@foreach($users as $user)
  <li>{{ $user->name }}</li>
@endforeach
```

`@forelse` — handles empty arrays gracefully

```blade
@forelse($products as $product)
  <p>{{ $product->name }}</p>
@empty
  <p>No products available.</p>
@endforelse
```

Special Loop Variables

Inside loops, Blade provides `$loop`:

```blade
@foreach($items as $item)
  @if($loop->first) <b>First item!</b> @endif
  {{ $item }}
  @if($loop->last) <b>Last item!</b> @endif
@endforeach
```

🧠 Common `$loop` Properties

Property	Description	Example
`$loop->index`	Current loop index (starts from 0)	0, 1, 2
`$loop->iteration`	Current iteration (starts from 1)	1, 2, 3
`$loop->remaining`	How many iterations are left	
`$loop->count`	Total number of items in loop	
`$loop->first`	`true` on the first iteration	
`$loop->last`	`true` on the last iteration	
`$loop->even` / `$loop->odd`	`true` if current iteration is even/odd	

Php Laravel

🧠 4. Include / Extend Views

`@include` — insert another view

```blade
@include('partials.navbar')
```

(looks for `resources/views/partials/navbar.blade.php`)

`@includeIf`, `@includeWhen`

```blade
@includeIf('partials.footer')
@includeWhen($user->isAdmin(), 'partials.admin-menu')
```

You can pass data:

```blade
@include('partials.user', ['name' => 'Aniket'])
```

🎛️ 5. Switch / Case

```blade
@switch($role)
  @case('admin')
    <p>Welcome, Admin!</p>
    @break

  @case('user')
    <p>Welcome, User!</p>
    @break

  @default
    <p>Unknown Role</p>
@endswitch
```

🔐 6. Authentication Directives

```blade
@auth
  <p>Welcome back, {{ auth()->user()->name }}!</p>
@endauth

@guest
  <p>Please log in.</p>
@endguest
```

📄 7. CSRF and Method Spoofing

CSRF token

```blade
<form method="POST">
  @csrf
</form>
```

PUT / DELETE in forms

HTML forms only support GET/POST, so Laravel "spoofs" others:

```blade
<form method="POST">
  @method('PUT')
  @csrf
</form>
```

🧩 11. Environment Checks

Sometimes you want code to run only in local/dev mode:

```blade
@env('local')
    <p>Running in local environment</p>
@endenv
```

🧩 12. PHP Blocks

If you need raw PHP inside Blade:

```blade
@php
    $year = date('Y');
@endphp

<p>Current Year: {{ $year }}</p>
```

🧩 8. Comments

Blade comments **won't appear in HTML output:**

blade

```
{{-- This is a Blade comment --}}
```

- ## Forms & CSRF Protection

🔒 3. Forms & CSRF Protection

Every POST form must include a **CSRF token** (Cross-Site Request Forgery protection). Laravel automatically checks it.

Example:

blade

```
<form action="/users" method="POST">
    @csrf
    <input type="text" name="name">
    <button type="submit">Submit</button>
</form>
```

If you forget `@csrf`, Laravel throws a **419 Page Expired** error.

```
<!-- resources/views/register.blade.php -->
<!DOCTYPE html>
<html>
<head>
    <title>Register Form</title>
</head>
<body>

    <h2>Register</h2>

    <form method="POST" action="{{ route('register.submit') }}">
        @csrf  <!-- ✅ Very important for security -->

        <label>Name:</label>
        <input type="text" name="name"><br><br>

        <label>Email:</label>
        <input type="email" name="email"><br><br>

        <button type="submit">Submit</button>
    </form>

</body>
</html>
```

⚙️ 3️⃣ Add a Route

In your `routes/web.php`:

php Copy code

```
use App\Http\Controllers\RegisterController;

Route::get('/register', [RegisterController::class, 'showForm'])->name('register.form');
Route::post('/register', [RegisterController::class, 'submitForm'])->name('register.submit');
```

🎮 4️⃣ Create a Controller

Run:

```bash
php artisan make:controller RegisterController
```

Then open `app/Http/Controllers/RegisterController.php` and add:

```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class RegisterController extends Controller
{
    public function showForm()
    {
        return view('register');
    }

    public function submitForm(Request $request)
    {
        // Access form data
        $name = $request->input('name');
        $email = $request->input('email');

        // Just show what was submitted
        return "Name: $name, Email: $email";
    }
}
```

- **Form Request Validation**

You can validate input directly in your controller, but it's better to use Form Request Classes.

🪄 Step 1: Create a Form Request

Run this Artisan command:

```bash
php artisan make:request RegisterRequest
```

This creates a file:

`app/Http/Requests/RegisterRequest.php`

Php Laravel

🧠 Step 2: Define Validation Rules

Open `RegisterRequest.php` and edit it:

```php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class RegisterRequest extends FormRequest
{
    // 🟢 Who can make this request?
    public function authorize(): bool
    {
        return true; // true = anyone can, false = deny
    }

    // 🧩 Define your validation rules here
    public function rules(): array
    {
        return [
            'name' => 'required|min:3',
            'email' => 'required|email|unique:users,email',
            'password' => 'required|min:6|confirmed',
        ];
    }

    // (optional) Custom messages
    public function messages(): array
    {
        return [
            'name.required' => 'Please enter your name!',
            'email.unique' => 'This email is already taken.',
        ];
    }
}
```

⚙️ Step 3: Use It in Your Controller

Instead of `Request $request`, use your new request class:

```php
use App\Http\Requests\RegisterRequest;

class RegisterController extends Controller
{
    public function store(RegisterRequest $request)
    {
        // If validation fails, Laravel automatically returns a JSON or redirects with errors.
        // If validation passes:
        $validated = $request->validated();

        // Now safely use the validated data
        return response()->json([
            'message' => 'Registration successful!',
            'data' => $validated
        ]);
    }
}
```

📦 Step 4: Add a Route

```php
Route::post('/register', [RegisterController::class, 'store']);
```

🧠 Step 5: Example Request (Postman or Form)

POST /register

Body (JSON):

```json
{
  "name": "Aniket",
  "email": "aniket@example.com",
  "password": "secret",
  "password_confirmation": "secret"
}
```

- Localization (Intro)

🌐 What is Localization?

Localization means making your application display text, dates, messages, etc. in **different languages** depending on the user's preference.

For example:

- English: "Welcome"
- Hindi: "स्वागत है"
- French: "Bienvenue"

Laravel makes this super easy with its **translation system**.

Php Laravel

🧱 1️⃣ Where Translations Live

All your translations are stored in:

```bash
resources/lang/
```

Inside this folder, you can have **subfolders for each language code**, like:

```bash
resources/lang/
├── en/
│   └── messages.php
├── hi/
│   └── messages.php
└── fr/
    └── messages.php
```

🧩 2️⃣ Create a Translation File

Example: `resources/lang/en/messages.php`

```php
<?php

return [
    'welcome' => 'Welcome to our website!',
    'goodbye' => 'Goodbye, see you soon!',
];
```

Now make another for Hindi:

`resources/lang/hi/messages.php`

```php
<?php

return [
    'welcome' => 'हमारी वेबसाइट पर आपका स्वागत है!',
    'goodbye' => 'अलविदा, फिर मिलेंगे!',
];
```

🧠 3️⃣ Use Translations in Blade

You can use the `__()` helper or the `@lang` directive.

```blade
<h1>{{ __('messages.welcome') }}</h1>
<p>@lang('messages.goodbye')</p>
```

✅ Output will depend on the **current language (locale)**.

⚙️ 4️⃣ Set the App Locale

You set the default language in `config/app.php`:

```php
'locale' => 'en',
```

If you change it to:

```php
'locale' => 'hi',
```

🔁 5️⃣ Switch Language Dynamically

You can change locale at runtime — for example, with a dropdown.

```php
use Illuminate\Support\Facades\App;

Route::get('/set/{lang}', function ($lang) {
    App::setLocale($lang);
    return view('welcome');
});
```

If you visit `/set/hi`, the language changes to Hindi temporarily.

Quick Switcher

- Add to layout:

```blade
<a href="?ism="?lang=en">English</a> | <a href="?lang=es">Español</a>
```

- In controller: `app()->setLocale(request('lang')); session(['locale' => request('lang')]);`

7️⃣ Using Placeholders in Translations

You can insert variables in messages easily:

```php
// resources/lang/en/messages.php
return [
    'greeting' => 'Hello, :name!',
];
```

Then in Blade:

```blade
{{ __('messages.greeting', ['name' => 'Aniket']) }}
```

Output:

```
Hello, Aniket!
```

Migrations & schema builder • Seeders & Factories • Faker • Eloquent models • Mass assignment & casts

- **Migrations & schema builder:**

Migrations are like **version control for your database**. They let you define tables and columns in PHP instead of writing raw SQL.

🧠 2️⃣ Create a Migration

You create one using Artisan:

```bash
php artisan make:migration create_users_table
```

This creates a file in:

```bash
database/migrations/2025_10_13_000000_create_users_table.php
```

Php Laravel

Migration file example

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void {
        Schema::create('users', function (Blueprint $table) {
            $table->id(); // auto-increment PK
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->timestamps(); // created_at + updated_at
        });
    }

    public function down(): void {
        Schema::dropIfExists('users');
    }
};
```

3 Migration File Structure

A migration file looks like this:

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();                // Auto-increment primary key
            $table->string('name');      // VARCHAR(255)
            $table->string('email')->unique(); // Unique column
            $table->timestamps();        // created_at & updated_at
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('users'); // Reverse migration
    }
};
```

4 Run the Migration

To apply migrations (create tables):

```bash
php artisan migrate
```

To undo the last migration:

```bash
php artisan migrate:rollback
```

To refresh all migrations (drop all tables and recreate):

```bash
php artisan migrate:refresh
```

5 Schema Builder Basics

Laravel provides the **Schema Builder** via the `Schema` and `Blueprint` classes.

You use it inside migrations to define your table structure.

Example — Add a Column

```bash
php artisan make:migration add_age_to_users_table
```

Php Laravel

Migration file:

```php
public function up(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->integer('age')->nullable();
    });
}

public function down(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('age');
    });
}
```

```php
Schema::create('sessions', function (Blueprint $table) {
    $table->string('id')->primary();
    $table->foreignId('user_id')->nullable()->index();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->longText('payload');
    $table->integer('last_activity')->index();
});
```

Common Column Types

Type	Example	Description
`string()`	`$table->string('name', 100)`	VARCHAR(100)
`text()`	`$table->text('bio')`	Long text
`integer()`	`$table->integer('age')`	INT
`bigInteger()`	`$table->bigInteger('views')`	BIGINT
`boolean()`	`$table->boolean('is_active')`	true/false
`date()`	`$table->date('birth_date')`	YYYY-MM-DD
`timestamp()`	`$table->timestamp('created_at')`	DateTime
`float()`	`$table->float('price', 8, 2)`	Decimal number
`enum()`	`$table->enum('status', ['active', 'inactive'])`	Enum type

🔑 7️⃣ Primary & Foreign Keys

Primary Key (auto)

```php
$table->id(); // same as bigIncrements('id')
```

Foreign Key (relation)

```php
$table->unsignedBigInteger('user_id');
$table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
```

🟢 This means:
"`user_id` in this table refers to `id` in the `users` table — delete related rows if user is deleted."

🧩 8️⃣ Default Columns

Laravel automatically adds:

```php
$table->timestamps();  // created_at, updated_at
$table->softDeletes(); // deleted_at (for soft delete feature)
```

🧰 1 0 Useful Artisan Commands

Command	Description
`php artisan migrate`	Run all pending migrations
`php artisan migrate:rollback`	Undo the last batch
`php artisan migrate:reset`	Undo all migrations
`php artisan migrate:refresh`	Reset & re-run all migrations
`php artisan migrate:fresh`	Drop all tables & re-run migrations
`php artisan make:migration`	Create a new migration file

- Seeders & Factories

Seeders = populate database with **sample/dummy data**.
Factories + Faker = generate realistic dummy data.

🎯 1️⃣ What Are Seeders & Factories?

Concept	Purpose
Seeder	Inserts sample data into the database (manually or via code).
Factory	Automatically generates **fake data** using Faker library.
Together	Used to populate DB quickly for testing or demo purposes.

2 Seeder (Manual Data)

You can create a seeder using Artisan:

```bash
php artisan make:seeder UserSeeder
```

This creates:

```bash
database/seeders/UserSeeder.php
```

Example:

```php
namespace Database\Seeders;

use Illuminate\Database\Seeder;
use App\Models\User;

class UserSeeder extends Seeder
{
    public function run(): void
    {
        User::create([
            'name' => 'Aniket',
            'email' => 'aniket@example.com',
            'password' => bcrypt('password'),
        ]);
    }
}
```

Now register this seeder in:

```bash
database/seeders/DatabaseSeeder.php
```

```php
public function run(): void
{
    $this->call(UserSeeder::class);
}
```

Run it:

```bash
php artisan db:seed
```

✅ This will insert one user record into your `users` table.

If not registered in **Databaseseeder.php** you can use:

Run seeder

```bash
php artisan db:seed --class=UserSeeder
```

⚙️ 3️⃣ Factory (Auto Fake Data)

Factories use **Faker** to create random but realistic data.

Create a factory:

```bash
php artisan make:factory UserFactory --model=User
```

This creates:

```bash
database/factories/UserFactory.php
```

Example:

```php
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'password' => bcrypt('password'),
        ];
    }
}
```

🧪 4️⃣ Using Factory in a Seeder

You can use a factory to quickly generate many users:

```php
public function run(): void
{
    \App\Models\User::factory(10)->create(); // Creates 10 fake users
}
```

Run:

```bash
php artisan db:seed
```

Now your `users` table will have 10 random records.

5 Seeder + Factory Combo Example

Let's say you have **students** and **departments** tables.

DepartmentFactory

```php
public function definition(): array
{
    return [
        'name' => fake()->randomElement(['IT', 'Science', 'Arts', 'Commerce']),
    ];
}
```

StudentFactory

```php
public function definition(): array
{
    return [
        'name' => fake()->name(),
        'email' => fake()->unique()->safeEmail(),
        'department_id' => \App\Models\Department::factory(), // creates a related department
    ];
}
```

Seeder

```php
public function run(): void
{
    \App\Models\Student::factory(20)->create(); // Creates 20 students + departments
}
```

Your `students` table likely has a **foreign key** like:

```php
department_id -> departments.id
```

So if you just use `randomElement(['IT', 'Science', ...])`, you're generating a **string**, not a valid `department_id`.

That means your seeder would fail (or the relationship would break) because there's **no matching department** in the `departments` table.

✅ Why `DepartmentFactory` is needed

When you use:

```php
'department_id' => \App\Models\Department::factory(),
```

Laravel does this automatically:

1. It **creates a new department record** in the `departments` table (e.g., "IT" or "Arts").
2. It **uses that department's ID** as the `department_id` in your student record.

So both tables stay consistent and the **foreign key relationship is valid**.

📄 6️⃣ Common Artisan Commands

Command	Description
`php artisan make:seeder NameSeeder`	Create a seeder file
`php artisan make:factory ModelFactory --model=Model`	Create a factory file
`php artisan db:seed`	Run all seeders
`php artisan db:seed --class=UserSeeder`	Run a specific seeder
`php artisan migrate:fresh --seed`	Recreate DB & run all seeders

🧩 7️⃣ Factory Shortcuts

Code	Meaning
`Model::factory()->create()`	Create 1 fake record
`Model::factory(5)->create()`	Create 5 fake records
`Model::factory()->make()`	Create (but not insert into DB)
`Model::factory()->create(['name' => 'Aniket'])`	Override values

- **Faker**

🎯 1️⃣ What is Faker?

Faker is a PHP library that creates *real-looking* fake data — like names, emails, company names, phone numbers, etc.

✅ Commonly used in:

- **Factories** (Laravel built-in)
- **Testing**
- **Database seeding**

🧰 2️⃣ Faker Is Built Into Laravel

In Laravel 8+, Faker is **already included** by default.
You can use it in:

- **Factories** → via `fake()` helper
- **Anywhere else** → via dependency injection or helper functions

💡 **In simple words:**

> Faker is Laravel's *data generator friend* — it gives you names, emails, and other random info instantly for testing or seeding.

✨ 3️⃣ Example in a Factory

Example: `database/factories/UserFactory.php`

```php
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class UserFactory extends Factory
{
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'password' => bcrypt('password'),
            'address' => fake()->address(),
            'phone' => fake()->phoneNumber(),
        ];
    }
}
```

✅ `fake()` automatically gives you a Faker instance.

⚙️ 4️⃣ Run With Seeder

```php
public function run(): void
{
    \App\Models\User::factory(10)->create();
}
```

This will generate **10 random users** with unique names, emails, etc.

5 Common Faker Methods

Method	Example Output	Description
`name()`	"John Doe"	Random name
`firstName()`	"Amit"	Random first name
`lastName()`	"Patel"	Random last name
`email()`	"john.doe@example.org"	Random email
`safeEmail()`	"alex@domain.test"	Non-real (safe) email
`phoneNumber()`	"+91-9876543210"	Random phone number
`address()`	"123 Main Street, Pune"	Random address
`city()`	"Mumbai"	Random city
`state()`	"Maharashtra"	Random state
`country()`	"India"	Random country
`text(200)`	"Lorem ipsum dolor sit amet..."	Random text
`date()`	"2025-10-13"	Random date
`company()`	"Infosys Ltd."	Random company name
`jobTitle()`	"Software Engineer"	Random job title
`numberBetween(1, 100)`	47	Random number in range
`randomElement(['A', 'B', 'C'])`	"B"	Pick random element
`uuid()`	"550e8400-e29b-41d4-a716-446655440000"	Unique identifier

6 Using Faker Anywhere (Optional)

If you want to use Faker outside of factories (for example, in a controller or tinker):

```php
use Faker\Factory as Faker;

$faker = Faker::create();

echo $faker->name();
echo $faker->email();
echo $faker->city();
```

- Eloquent models

Php Laravel

🎯 1️⃣ What is Eloquent?

Eloquent is Laravel's built-in **ORM (Object Relational Mapper)**.
It helps you interact with your database **using PHP objects instead of SQL queries**.

✅ You don't need to write:

sql Copy code

```sql
SELECT * FROM users WHERE id = 1;
```

Instead, you can just write:

php Copy code

```php
$user = User::find(1);
```

Laravel automatically does the SQL for you behind the scenes.

🧱 2️⃣ Creating a Model

Use Artisan to make a model:

bash

```bash
php artisan make:model User
```

This creates:

swift

```swift
app/Models/User.php
```

Example:

php

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $fillable = ['name', 'email', 'password'];
}
```

Basic usage

php

```php
// Retrieve all users
$users = User::all();

// Find a user
$user = User::find(1);

// Create a new user
User::create([
    'name' => 'Alice',
    'email' => 'alice@example.com',
    'password' => bcrypt('secret')
]);
```

⚙️ 3️⃣ Model Structure

Let's break that down:

Property	Purpose	Example
`$fillable`	Allows mass assignment	`['name', 'email']`
`$table`	(optional) specify table name	`protected $table = 'students';`
`$primaryKey`	Change primary key	`protected $primaryKey = 'student_id';`
`$timestamps`	Disable timestamps	`public $timestamps = false;`
`$casts`	Cast fields to types	`['is_active' => 'boolean']`

🧩 4️⃣ CRUD with Eloquent

✅ Create (Insert)

```php
$user = new User();
$user->name = 'Aniket';
$user->email = 'aniket@example.com';
$user->password = bcrypt('123456');
$user->save();
```

OR shorter way (mass assignment):

```php
User::create([
    'name' => 'Aniket',
    'email' => 'aniket@example.com',
    'password' => bcrypt('123456')
]);
```

📖 Read (Select)

Method	Description	Example
`all()`	Get all records	`User::all()`
`find($id)`	Find by primary key	`User::find(1)`
`where()`	Filter results	`User::where('name', 'Aniket')->get()`
`first()`	Get the first record	`User::where('email', 'aniket@example.com')->first()`
`pluck()`	Get single column values	`User::pluck('name')`

Php Laravel

Update

```php
$user = User::find(1);
$user->name = 'Aniket Raikar';
$user->save();
```

OR:

```php
User::where('id', 1)->update(['name' => 'Aniket Raikar']);
```

Delete

```php
$user = User::find(1);
$user->delete();
```

OR:

```php
User::destroy(1);
```

5 Example — Students Model

Let's say you have a `students` table:

id	name	email	age

Model

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Student extends Model
{
    protected $fillable = ['name', 'email', 'age'];
}
```

Controller

```php
use App\Models\Student;

class StudentController extends Controller
{
    public function index()
    {
        $students = Student::all();
        return view('students.index', compact('students'));
    }

    public function store(Request $request)
    {
        Student::create($request->all());
        return redirect()->back()->with('success', 'Student added!');
    }
}
```

6 Query Builder vs Eloquent

Feature	Eloquent	Query Builder
Returns	Model objects	Arrays
Syntax	`User::where('age', '>', 20)->get()`	`DB::table('users')->where('age', '>', 20)->get()`
Relations	Yes	No
Easy CRUD	✅	❌

- **Mass assignment & casts**

Mass Assignment

- Protects against assigning unwanted fields from request input.

```php
protected $fillable = ['name', 'email', 'password']; // allowed fields
protected $guarded = ['id']; // forbidden fields
```

- Only `$fillable` attributes can be assigned via `User::create($request->all())`.

Php Laravel

Casts

- Automatically convert database columns to PHP types.

```php
protected $casts = [
    'email_verified_at' => 'datetime',
    'is_active' => 'boolean',
    'options' => 'array',   // JSON column to PHP array
];
```

Example:

```php
$user = User::find(1);
$user->is_active; // returns true/false instead of 0/1
```

⚙️ 1️⃣ What Is Mass Assignment?

Mass assignment = **inserting or updating multiple fields at once** instead of one-by-one.

For example 👇

Without mass assignment:

```php
$user = new User();
$user->name = 'Aniket';
$user->email = 'aniket@example.com';
$user->password = bcrypt('123456');
$user->save();
```

With mass assignment:

```php
User::create([
    'name' => 'Aniket',
    'email' => 'aniket@example.com',
    'password' => bcrypt('123456'),
]);
```

🧱 2️⃣ Why Do We Need Protection?

Imagine a user sends unexpected data from a form, like:

```json
{
  "name": "Aniket",
  "email": "aniket@example.com",
  "is_admin": true
}
```

If your model has an `is_admin` column — this could let users **give themselves admin access** 😱

To prevent this, Laravel protects models from **mass assignment vulnerabilities**.

🧰 3️⃣ `$fillable` Property (Allowed Fields)

You explicitly tell Laravel which fields can be filled using mass assignment.

Example — `app/Models/User.php`:

```php
class User extends Model
{
    protected $fillable = ['name', 'email', 'password'];
}
```

✅ Now only these fields can be assigned using:

```php
User::create($request->all());
```

If a field like `is_admin` is sent, Laravel ignores it.

❌ `$guarded` Property (Blocked Fields)

Alternative way — tell Laravel which fields **NOT** to fill.

Example:

```php
class User extends Model
{
    protected $guarded = ['is_admin'];
}
```

✅ This means all fields can be mass assigned **except** `is_admin`.

🧩 4️⃣ Example — Student Model

```php
class Student extends Model
{
    protected $fillable = ['name', 'email', 'age', 'department'];
}
```

Now in your controller:

```php
Student::create($request->all());
```

Laravel will only insert columns listed in `$fillable`.

🔄 5️⃣ What Are Casts?

Casts automatically convert your DB values into specific PHP types when retrieved, and vice versa when stored.

Example:

```php
class User extends Model
{
    protected $casts = [
        'email_verified_at' => 'datetime',
        'is_active' => 'boolean',
        'points' => 'integer',
    ];
}
```

✅ So when you do:

```php
$user = User::find(1);
```

- `$user->email_verified_at` → Carbon (date) object
- `$user->is_active` → `true` or `false` (boolean)
- `$user->points` → integer

🧠 6️⃣ Common Cast Types

Cast Type	Converts Into	Example
`integer`	PHP int	`['age' => 'integer']`
`float`	PHP float	`['price' => 'float']`
`boolean`	true/false	`['is_active' => 'boolean']`
`string`	PHP string	`['phone' => 'string']`
`datetime`	Carbon object	`['created_at' => 'datetime']`
`array`	PHP array	`['settings' => 'array']`
`json`	JSON data	`['meta' => 'json']`
`encrypted`	Auto encrypted/decrypted	`['secret' => 'encrypted']`

Relationships (1-1, 1-n, n-n, morph) • Query builder vs Eloquent • Eager loading • Scopes & accessors/mutators • Pagination

- **Relationships (1-1, 1-n, n-n, morph) :**

🧩 1️⃣ **ONE-TO-ONE (1:1)**

Meaning: One record in a table is linked to exactly one record in another.

🧠 Example:

- A **User** has **one Profile**.
- A **Profile** belongs to **one User**.

Models

User.php

```php
class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class);
    }
}
```

Profile.php

```php
class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Usage

```php
$user = User::find(1);
echo $user->profile->bio; // Access related profile

$profile = Profile::find(1);
echo $profile->user->name; // Access related user
```

✅ Automatically runs:

```sql
SELECT * FROM profiles WHERE user_id = 1;
```

🧱 2️⃣ ONE-TO-MANY (1:N)

Meaning: One record has **many related records**.

🧠 Example:

- A **Department** has many **Students**.
- Each **Student** belongs to one **Department**.

Department.php

```php
class Department extends Model
{
    public function students()
    {
        return $this->hasMany(Student::class);
    }
}
```

Student.php

```php
class Student extends Model
{
    public function department()
    {
        return $this->belongsTo(Department::class);
    }
}
```

Usage

```php
$department = Department::find(1);
foreach ($department->students as $student) {
    echo $student->name;
}

$student = Student::find(5);
echo $student->department->name;
```

🔗 3️⃣ MANY-TO-MANY (N:N)

Meaning: Records from both tables can have multiple connections.

🧠 Example:

- A **Student** can join many **Courses**.
- A **Course** can have many **Students**.

We need a **pivot table** for this (a middle table).

Php Laravel

Models

`Student.php`

```php
class Student extends Model
{
    public function courses()
    {
        return $this->belongsToMany(Course::class);
    }
}
```

`Course.php`

```php
class Course extends Model
{
    public function students()
    {
        return $this->belongsToMany(Student::class);
    }
}
```

Usage

```php
$student = Student::find(1);
$student->courses()->attach(2); // Add relation
$student->courses()->detach(2); // Remove relation
$student->courses()->sync([1,2,3]); // Keep only these

foreach ($student->courses as $course) {
    echo $course->name;
}
```

✅ Laravel automatically uses pivot table `course_student` (alphabetical order).

If table name differs, specify it:

```php
return $this->belongsToMany(Course::class, 'student_course');
```

🧩 3 `$student->courses()->attach(2);`

- **attach()** adds a new row to the pivot table.
- Here you're linking student `#1` with course `#2`.

So in `course_student`, a new record is added:

student_id	course_id
1	2

If you call `attach()` again with the same course, it'll cause a duplicate unless you handle it.

🧩 4 `$student->courses()->detach(2);`

- **detach()** removes that relationship from the pivot table.
- So it deletes the row where `student_id = 1` and `course_id = 2`.

After this, student 1 is **no longer enrolled** in course 2.

🧩 5 `$student->courses()->sync([1,2,3]);`

- **sync()** is smarter — it ensures the pivot table matches **exactly** what you pass in.
- It:
  - Adds new links for courses `[1,2,3]` (if not already linked)
  - Removes all **other** existing course links not in `[1,2,3]`

✅ So after running this, the student will be enrolled only in courses with IDs 1, 2, and 3.

🌀 4 POLYMORPHIC (MORPH) RELATIONSHIPS

Meaning: One model can belong to **multiple other models using a single table**.

🧠 Example:

- Both **Post** and **Video** can have **Comments**.
  Instead of separate `post_comments` & `video_comments`, we use **one** `comments` **table**.

Database Tables

posts	videos	comments
id	id	id
title	title	commentable_id
		commentable_type
		body

Models

`Comment.php`

```php
class Comment extends Model
{
    public function commentable()
    {
        return $this->morphTo();
    }
}
```

`Post.php`

```php
class Post extends Model
{
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Php Laravel

Video.php

```php
class Video extends Model
{
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Usage

```php
$post = Post::find(1);
$post->comments()->create(['body' => 'Nice post!']);

$video = Video::find(1);
$video->comments()->create(['body' => 'Great video!']);

$comment = Comment::find(1);
echo $comment->commentable->title; // Works for both post/video
```

✅ Laravel automatically saves:

```text
commentable_id = 1
commentable_type = App\Models\Post
```

4️⃣ Using it in Code

➕ Create comments on posts and videos

Copy code

```php
$post = Post::create(['title' => 'First Post', 'body' => 'This is my blog post.']);
$video = Video::create(['title' => 'Laravel Tutorial', 'url' => 'https://youtu.be/abcd1234']);

$post->comments()->create(['body' => 'Great post!']);
$video->comments()->create(['body' => 'Awesome video!']);
```

📥 Fetch comments

Copy code

```php
foreach ($post->comments as $comment) {
    echo $comment->body;
}

foreach ($video->comments as $comment) {
    echo $comment->body;
}
```

- **Query builder vs Eloquent:**

🧩 1️⃣ What are they?

🛠️ Query Builder

- A **lower-level** database query interface.
- Gives **more control** over SQL queries.
- Doesn't use Models (it works directly with tables).
- Returns **arrays** or **stdClass objects**, not Eloquent Models.

✅ Think of it as **"writing SQL in PHP style"**.

Example:

```php
$users = DB::table('users')->where('active', 1)->get();
```

💎 Eloquent ORM (Object-Relational Mapping)

- A **higher-level**, object-oriented layer built on top of Query Builder.
- Uses **Models** that represent database tables.
- Lets you work with **objects** instead of raw data.
- Automatically manages **relationships**, **timestamps**, **casting**, etc.

✅ Think of it as **"working with data as PHP objects"**.

Example:

```php
$users = User::where('active', 1)->get();
```

⚖️ 2️⃣ Comparison Table

Feature	Query Builder	Eloquent ORM
Uses Models	❌ No	✅ Yes
Return Type	Array/Object	Model Instances
Ease of Use	Manual	Simple & expressive
Performance	Slightly faster	Slightly slower
Relationships	Must join manually	Built-in (`hasMany`, `belongsTo`, etc.)
Best for	Complex joins, heavy queries	CRUD operations, relationships
Example Use Case	Analytics, reporting	CRUD, APIs, web apps

💻 3️⃣ Example Comparison

Let's say we have a `users` table with columns:
`id, name, email, is_active`

🧱 Using Query Builder

```php
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->select('name', 'email')
            ->where('is_active', 1)
            ->orderBy('name')
            ->get();

foreach ($users as $user) {
    echo $user->name . " - " . $user->email . "<br>";
}
```

✅ SQL equivalent:

```sql
SELECT name, email FROM users WHERE is_active = 1 ORDER BY name;
```

💎 Using Eloquent ORM

```php
$users = User::where('is_active', 1)
            ->orderBy('name')
            ->get(['name', 'email']);

foreach ($users as $user) {
    echo $user->name . " - " . $user->email . "<br>";
}
```

Php Laravel

🧮 4️⃣ Relationships Example

Query Builder ❌ (Manual Join)

```php
$students = DB::table('students')
    ->join('departments', 'students.department_id', '=', 'departments.id')
    ->select('students.name', 'departments.name as department')
    ->get();
```

Eloquent ✅ (Automatic Relation)

```php
$students = Student::with('department')->get();

foreach ($students as $student) {
    echo $student->name . ' - ' . $student->department->name;
}
```

Yes — in **Eloquent**, relationships like `with('department')` only work **if you've defined that relationship** inside the respective model classes **beforehand**.

```php
class Student extends Model
{
    // Each student belongs to one department
    public function department()
    {
        return $this->belongsTo(Department::class);
    }
}
```

```php
class Department extends Model
{
    // A department has many students
    public function students()
    {
        return $this->hasMany(Student::class);
    }
}
```

🧠 5️⃣ When to Use What

Situation	Use
Need full model features (create, update, delete, relationships)	🟢 **Eloquent**
Just fetching raw data (reports, analytics, aggregates)	🟡 **Query Builder**
Complex joins or performance-critical queries	🔵 **Query Builder**
API or CRUD endpoints	🟢 **Eloquent**

📘 6️⃣ Bonus Tip: You Can Combine Them!

Sometimes you can even **mix both**:

```php
$users = User::where('is_active', 1)
            ->selectRaw('count(*) as total, role_id')
            ->groupBy('role_id')
            ->get();
```

⚡ Bonus: You Can Mix Both!

You can use Query Builder **inside** an Eloquent model:

```php
$users = User::where('active', 1)
            ->whereRaw('YEAR(created_at) = ?', [2025])
            ->get();
```

What Are They?

Feature	Query Builder	Eloquent ORM
Definition	A simple, fluent interface to build and run SQL queries directly.	An Object-Relational Mapper (ORM) that represents each database table as a **Model class**.
Output	Returns plain PHP objects (`stdClass`) or arrays.	Returns full **Model objects** (with relationships, methods, accessors, etc.).
Speed	Slightly faster (less overhead).	Slightly slower (adds model layer).
Complexity	Simple for raw queries.	Easier for relational data, but higher abstraction.

1 Query Builder Example

```php
use Illuminate\Support\Facades\DB;

// Get all users
$users = DB::table('users')->get();

// Insert a new user
DB::table('users')->insert([
    'name' => 'Aniket',
    'email' => 'aniket@example.com',
]);

// Update
DB::table('users')
    ->where('id', 1)
    ->update(['name' => 'Updated Name']);

// Delete
DB::table('users')->where('id', 2)->delete();
```

2 Eloquent Example

```php
use App\Models\User;

// Get all users
$users = User::all();

// Insert
User::create([
    'name' => 'Aniket',
    'email' => 'aniket@example.com',
]);

// Update
$user = User::find(1);
$user->name = 'Updated Name';
$user->save();

// Delete
User::destroy(2);
```

- Eager Loading

What is Eager Loading?

When you fetch data using Eloquent, it can also load the **related data** from other tables.

By default, Eloquent uses **lazy loading** — meaning:

> It only loads related data when you access it.

This can cause **N+1 query problem** (too many SQL queries).

Php Laravel

⚠️ The Problem — Lazy Loading (default)

Example:

You have two models:

- `Department` → has many `Student`s
- `Student` → belongs to `Department`

```php
$departments = Department::all();

foreach ($departments as $department) {
    echo $department->name;
    echo $department->students->count(); // triggers a query every time
}
```

✅ Problem:

If you have 10 departments, Laravel runs:

```rust
1 query for departments
+ 10 queries for students
= 11 total queries ❌
```

The Fix — Eager Loading

With **eager loading**, you tell Laravel to **load related data in advance** — in a *single joined query*.

Example

```php
$departments = Department::with('students')->get();

foreach ($departments as $department) {
    echo $department->name;
    echo $department->students->count(); // no extra queries
}
```

✅ Queries executed:

```rust
1 for departments
1 for students (for all departments)
= 2 total ✅
```

💡 Example: How It Looks in SQL

Without eager loading:

```sql
SELECT * FROM users;
SELECT * FROM posts WHERE user_id = 1;
SELECT * FROM posts WHERE user_id = 2;
SELECT * FROM posts WHERE user_id = 3;
...
```

With eager loading:

```sql
SELECT * FROM users;
SELECT * FROM posts WHERE user_id IN (1,2,3,...);
```

⚙️ Step 4: When You Loop

```php
foreach ($users as $user) {
    echo $user->posts->count();
}
```

This doesn't hit the database again —

because `posts` are **already loaded in memory** from the eager load.

Php Laravel

🧱 Syntax

Basic Eager Loading

```php
$users = User::with('posts')->get();
```

Multiple Relationships

```php
$users = User::with(['posts', 'roles', 'profile'])->get();
```

Nested Eager Loading

```php
$users = User::with('posts.comments')->get();
// Loads users → their posts → each post's comments
```

🧱 1️⃣ Multiple Eager Loading

```php
$users = User::with(['posts', 'roles', 'profile'])->get();
```

🔍 SQL generated (roughly)

```sql
SELECT * FROM users;
SELECT * FROM posts WHERE user_id IN (1, 2, 3, ...);
SELECT * FROM role_user WHERE user_id IN (1, 2, 3, ...);
SELECT * FROM roles WHERE id IN (...);
SELECT * FROM profiles WHERE user_id IN (1, 2, 3, ...);
```

🧱 2️⃣ Nested Eager Loading

```php
$users = User::with('posts.comments')->get();
```

🔍 SQL generated (roughly)

```sql
SELECT * FROM users;
SELECT * FROM posts WHERE user_id IN (1, 2, 3, ...);
SELECT * FROM comments WHERE post_id IN (10, 11, 12, ...);
```

💎 Example — Real Case

Let's say:

Models

```php
class Department extends Model {
    public function students() {
        return $this->hasMany(Student::class);
    }
}

class Student extends Model {
    public function department() {
        return $this->belongsTo(Department::class);
    }
}
```

Controller

```php
public function index() {
    // ✅ Eager loading students with departments
    $departments = Department::with('students')->get();

    foreach ($departments as $department) {
        echo "<h3>{$department->name}</h3>";
        foreach ($department->students as $student) {
            echo $student->name . "<br>";
        }
    }
}
```

🎯 Conditional Eager Loading

You can even apply filters to eager-loaded relations:

```php
$departments = Department::with(['students' => function($query) {
    $query->where('active', 1);
}])->get();
```

⚙️ Lazy Eager Loading

If you already have results but forgot to load relations:

```php
$departments = Department::all();
$departments->load('students');
```

This fetches the students *afterwards* in one query.

📄 Summary

Type	Description	Example	Queries
Lazy Loading	Loads when accessed	`$user->posts`	Many
Eager Loading	Loads ahead of time	`User::with('posts')->get()`	Few
Lazy Eager Loading	Loads later	`$users->load('posts')`	Few

💬 In Simple Words:

> Eager Loading = “Tell Laravel to fetch related data in advance, not one by one.”

- **Scopes & accessors/mutators:**

🧠 In simple words:

> Scopes = saved filters or query conditions that you can reuse easily.

🧩 1️⃣ SCOPES

👉 What it means

Scopes help you **reuse common query conditions** easily.

Example — Instead of writing this everywhere:

```php
User::where('is_active', 1)->get();
```

You can define a **scope** once in your model and then just call:

```php
User::active()->get();
```

🧠 Types of Scopes

There are **two** types:

1. **Local Scopes**
2. **Global Scopes**

🔹 Local Scope Example

In your model (`User.php`)

```php
class User extends Model
{
    // Define a scope method starting with "scope"
    public function scopeActive($query)
    {
        return $query->where('is_active', 1);
    }

    public function scopeOfRole($query, $role)
    {
        return $query->where('role', $role);
    }
}
```

Usage:

```php
$activeUsers = User::active()->get();

$admins = User::active()->ofRole('admin')->get();
```

✅ Laravel automatically knows that `scopeActive()` corresponds to the method `active()`.

🔹 Global Scope Example

Use this if you want every query on a model to have a condition by default.

Example: Show only active users everywhere.

Create a Scope Class (optional)

```php
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class ActiveScope implements Scope
{
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('is_active', 1);
    }
}
```

🪄 Step 1: Create a Scope Class

Run this Artisan command:

```bash
php artisan make:scope PublishedScope
```

> (If your Laravel version doesn't support this shortcut, you can create it manually under `app/Scopes/PublishedScope.php`.)

Register in Model

```php
class User extends Model
{
    protected static function booted()
    {
        static::addGlobalScope(new ActiveScope);
    }
}
```

Now:

```php
User::all(); // automatically adds WHERE is_active = 1
```

If you want to ignore it:

```php
User::withoutGlobalScope(ActiveScope::class)->get();
```

4. Scopes

Custom query methods on a model.

```php
// User.php
public function scopeActive($query) {
    return $query->where('is_active', 1);
}

// Usage
$users = User::active()->get();
```

- You can also pass parameters:

```php
public function scopeOfAge($query, $age) {
    return $query->where('age', '>=', $age);
}

User::ofAge(18)->get();
```

🧩 2️⃣ ACCESSORS & MUTATORS

These let you **modify model attributes** (fields) when reading or writing data.

🔹 Accessors (get...)

Used when **getting** a value.

Example: You want the first letter of a user's name to always be capitalized.

```php
class User extends Model
{
    public function getNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

Usage:

```php
$user = User::find(1);
echo $user->name; // "Aniket" (even if saved as "aniket")
```

🔹 Mutators (set...)

Used when **saving** a value (writing to database).

Example: You want to store all names in lowercase.

```php
class User extends Model
{
    public function setNameAttribute($value)
    {
        $this->attributes['name'] = strtolower($value);
    }
}
```

Usage:

```php
$user = new User;
$user->name = 'ANIKET';
$user->save();

// stored in DB as "aniket"
```

Php Laravel

Combined Example

```php
class User extends Model
{
    // Accessor
    public function getEmailAttribute($value)
    {
        return strtolower($value);
    }

    // Mutator
    public function setPasswordAttribute($value)
    {
        $this->attributes['password'] = bcrypt($value);
    }

    // Scope
    public function scopeActive($query)
    {
        return $query->where('is_active', 1);
    }
}
```

Usage:

```php
// Accessor Example
$user = User::find(1);
echo $user->email; // always lowercase

// Mutator Example
$user->password = 'mypassword'; // automatically encrypted

// Scope Example
$active = User::active()->get();
```

Newer Laravel Style (since 9.x+)

You can now use the `Attribute` class for both getter & setter:

```php
use Illuminate\Database\Eloquent\Casts\Attribute;

class User extends Model
{
    protected function name(): Attribute
    {
        return Attribute::make(
            get: fn ($value) => ucfirst($value),
            set: fn ($value) => strtolower($value),
        );
    }
}
```

In simple words:

- "Scopes = reusable query filters"
- "Accessors = format when reading"
- "Mutators = format when saving"

- **Pagination:**

1 What is Pagination?

When you show a lot of records (say 500 students), you don't want to load all at once — it's slow and heavy. Instead, you show **10 or 15 per page**, and let users click "Next" or "Previous".

Laravel makes this **super easy** — no manual `LIMIT` or `OFFSET` SQL required.

2 How to Use Pagination

Example using Eloquent

```php
$students = Student::paginate(10); // 10 per page
```

This automatically handles:

- SQL query with limit and offset
- Current page detection
- HTML page links

Example using Query Builder

```php
$students = DB::table('students')->paginate(10);
```

Both ways return a `Paginator` **object** — not just a collection.

3 Displaying Pagination Links (in Blade)

In your Blade view:

```blade
<table>
@foreach ($students as $student)
    <tr>
        <td>{{ $student->name }}</td>
        <td>{{ $student->email }}</td>
    </tr>
@endforeach
</table>

{{ $students->links() }}
```

✅ Laravel automatically generates pagination links like:

```mathematica
« Previous  1 2 3 4 5  Next »
```

5 Useful Variations

Change Page Size

```php
$students = Student::paginate(5); // 5 per page
```

Custom Page Name

```php
$students = Student::paginate(10, ['*'], 'p'); // uses ?p=2 instead of ?page=2
```

Get Pagination Info

```php
$students->currentPage(); // current page number
$students->lastPage();    // total pages
$students->perPage();     // items per page
$students->total();       // total records
```

6 Simple Pagination (faster)

If you only need **Next / Previous** (no page numbers):

```php
$students = Student::simplePaginate(10);
```

Generates:

```mathematica
« Previous  Next »
```

Uses less memory because it doesn't count total records.

7️⃣ Example Controller + View

Controller

```php
public function index()
{
    $students = Student::paginate(10);
    return view('students.index', compact('students'));
}
```

View (`resources/views/students/index.blade.php`)

```blade
<h2>Student List</h2>

<table>
    <tr>
        <th>Name</th>
        <th>Email</th>
    </tr>
    @foreach ($students as $student)
        <tr>
            <td>{{ $student->name }}</td>
            <td>{{ $student->email }}</td>
        </tr>
    @endforeach
</table>

{{ $students->links() }}
```

✅ That's it! Pagination works automatically with URLs like:

```bash
/students?page=1
/students?page=2
```

🧱 8 API Pagination (JSON format)

If you're building an API:

```php
return Student::paginate(10);
```

✅ Laravel returns JSON automatically:

```json
{
  "current_page": 1,
  "data": [
    { "id": 1, "name": "John" },
    { "id": 2, "name": "Jane" }
  ],
  "first_page_url": "http://app.test/students?page=1",
  "next_page_url": "http://app.test/students?page=2",
  "last_page": 5,
  "per_page": 10,
  "total": 50
}
```

➢ Build 'students' CRUD views with Blade & FormRequest:

⚙️ Step 1: Create Model, Migration & Controller

Run:

```bash
php artisan make:model Student -mcr
```

This will create:

File	Purpose
`app/Models/Student.php`	Model
`database/migrations/..._create_students_table.php`	Migration
`app/Http/Controllers/StudentController.php`	Controller
`resources/views/students/`	(we'll create views here)

Php Laravel

🧩 Step 2: Migration

Open `database/migrations/..._create_students_table.php` and define columns:

```php
public function up(): void
{
    Schema::create('students', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->integer('age')->nullable();
        $table->timestamps();
    });
}
```

Run:

```bash
php artisan migrate
```

🧠 Step 3: Model (`app/Models/Student.php`)

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Student extends Model
{
    use HasFactory;

    protected $fillable = ['name', 'email', 'age'];
}
```

🪶 Step 4: FormRequest for Validation

Run:

```bash
php artisan make:request StudentRequest
```

Then edit `app/Http/Requests/StudentRequest.php`:

```php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class StudentRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'name'  => 'required|string|max:255',
            'email' => 'required|email|unique:students,email,' . $this->id,
            'age'   => 'nullable|integer|min:1|max:100',
        ];
    }
```

Php Laravel

🧭 Step 5: Controller (`app/Http/Controllers/StudentController.php`)

php Copy code

```php
namespace App\Http\Controllers;

use App\Models\Student;
use App\Http\Requests\StudentRequest;

class StudentController extends Controller
{
    public function index()
    {
        $students = Student::paginate(5);
        return view('students.index', compact('students'));
    }

    public function create()
    {
        return view('students.create');
    }

    public function store(StudentRequest $request)
    {
        Student::create($request->validated());
        return redirect()->route('students.ind )->with('success', 'Student created successfully!
    }
```

```php
    public function edit(Student $student)
    {
        return view('students.edit', compact('student'));
    }

    public function update(StudentRequest $request, Student $student)
    {
        $student->update($request->validated());
        return redirect()->route('students.index')->with('success', 'Student updated successfully!
    }

    public function destroy(Student $student)
    {
        $student->delete();
        return redirect()->route('students.index')->with('success', 'Student deleted successfully!
    }
}
```

🧩 Step 6: Routes (`routes/web.php`)

php Copy code

```php
use App\Http\Controllers\StudentController;

Route::resource('students', StudentController::class);
```

🎨 Step 7: Blade Views (CRUD)

Make folder:

bash Copy code

```bash
resources/views/students/
```

```php
return Redirect::route('profile.edit')->with('status', 'profile-updated');
```

Php Laravel

index.blade.php

blade

Copy code

```blade
@extends('layouts.app')

@section('content')
<div class="container">
    <h2>Students</h2>
    <a href="{{ route('students.create') }}" class="btn btn-primary mb-3">Add Student</a>

    @if (session('success'))
        <div class="alert alert-success">{{ session('success') }}</div>
    @endif

    <table class="table table-bordered">
        <tr>
            <th>ID</th><th>Name</th><th>Email</th><th>Age</th><th>Actions</th>
        </tr>
        @foreach($students as $student)
        <tr>
            <td>{{ $student->id }}</td>
            <td>{{ $student->name }}</td>
            <td>{{ $student->email }}</td>
            <td>{{ $student->age }}</td>
            <td>
                <a href="{{ route('students.edit', $student) }}" class="btn btn-warning btn-sm">Ed
                <form action="{{ route('studen  destroy', $student) }}" method="POST" style="disp
                    @csrf @method('DELETE')
```

```blade
                    <button class="btn btn-danger btn-sm">Delete</button>
                </form>
            </td>
        </tr>
        @endforeach
    </table>

    {{ $students->links() }}
</div>
@endsection
```

create.blade.php

blade

```blade
@extends('layouts.app')

@section('content')
<div class="container">
    <h2>Add Student</h2>

    <form method="POST" action="{{ route('students.store') }}">
        @csrf
        @include('students.form')
        <button class="btn btn-success mt-2">Save</button>
    </form>
</div>
@endsection
```

edit.blade.php

blade

```blade
@extends('layouts.app')

@section('content')
<div class="container">
    <h2>Edit Student</h2>

    <form method="POST" action="{{ route('students.update', $student) }}">
        @csrf @method('PUT')
        @include('students.form')
        <button class="btn btn-success mt-2">Update</button>
    </form>
</div>
@endsection
```

form.blade.php

blade

Copy code

```blade
<div class="form-group mb-2">
    <label>Name</label>
    <input type="text" name="name" class="form-control" value="{{ old('name', $student->name ?? ''
    @error('name') <small class="text-danger">{{ $message }}</small> @enderror
</div>

<div class="form-group mb-2">
    <label>Email</label>
    <input type="email" name="email" class="form-control" value="{{ old('email', $student->email ?
    @error('email') <small class="text-danger">{{ $message }}</small> @enderror
</div>

<div class="form-group mb-2">
    <label>Age</label>
    <input type="number" name="age" class="form-control" value="{{ old('age', $student->age ?? '')
    @error('age') <small class="text-danger">{{ $message }}</small> @enderror
</div>
```

Php Laravel

form.blade.php

blade Copy code

```blade
="form-group mb-2">
>Name</label>
 type="text" name="name" class="form-control" value="{{ old('name', $student->name ?? '') }}">
('name') <small class="text-danger">{{ $message }}</small> @enderror


="form-group mb-2">
>Email</label>
 type="email" name="email" class="form-control" value="{{ old('email', $student->email ?? '') }}">
('email') <small class="text-danger">{{ $message }}</small> @enderror


="form-group mb-2">
>Age</label>
 type="number" name="age" class="form-control" value="{{ old('age', $student->age ?? '') }}">
('age') <small class="text-danger">{{ $message }}</small> @enderror
```

✅ **Done!**

Now visit:

arduino

```
http://127.0.0.1:8000/students
```

You'll see a complete CRUD interface with:

- Pagination
- Validation (via FormRequest)
- Separate reusable form
- Flash messages

Action	HTTP Verb	URL	Route Name
index	GET	/students	students.index
create	GET	/students/create	students.create
store	POST	/students	students.store
show	GET	/students/{student}	students.show
edit	GET	/students/{student}/edit	students.edit
update	PUT/PATCH	/students/{student}	students.update
destroy	DELETE	/students/{student}	students.destroy

API routes & resource controllers • API Resources/Transformers • Pagination & filtering • Versioning (v1) • Error handling

➢ API routes & resource controllers:

✅ How to bring back `api.php` support manually in Laravel 12

Even though the provider is gone, you can still create your own "mini" one.

Step 1 — Create a new provider

Run this artisan command:

```bash
php artisan make:provider RouteServiceProvider
```

This will generate:

```swift
app/Providers/RouteServiceProvider.php
```

Step 2 — Add back route group registration

Open the newly created file and replace its contents with this:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Route;
use Illuminate\Foundation\Support\Providers\RouteServiceProvider as ServiceProvider;

class RouteServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        $this->routes(function () {
            // Default web routes
            Route::middleware('web')
                ->group(base_path('routes/web.php'));

            // 👇 Add your API routes manually
            Route::prefix('api')
                ->middleware('api')
                ->group(base_path('routes/api.php'));
        });
    }
}
```

Php Laravel

Step 4 — Create your API routes file

Create:

```bash
routes/api.php
```

Example content:

```php
<?php

use Illuminate\Support\Facades\Route;

Route::get('/hello', fn() => response()->json(['msg' => 'Hello from API!']));
```

Step 5 — Test

Run:

```bash
php artisan route:list
```

✅ You should now see:

```bash
GET|HEAD  api/hello
```

Open in browser:

```arduino
http://127.0.0.1:8000/api/hello
```

1. API Routes & Resource Controllers

In Laravel, API routes live in `routes/api.php`.

Basic API Route

```php
use App\Http\Controllers\UserController;

Route::get('/users', [UserController::class, 'index']);
```

Resource Controller

- Provides all CRUD endpoints with one line.

```bash
php artisan make:controller UserController --api
```

- Routes:

```php
Route::apiResource('users', UserController::class);
```

- Generated endpoints:

HTTP	URL	Action
GET	/users	index
POST	/users	store
GET	/users/{id}	show
PUT/PATCH	/users/{id}	update
DELETE	/users/{id}	destroy

`routes/api.php`

This file is automatically included in your project.

Example:

```php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\Api\StudentController;

Route::get('/students', [StudentController::class, 'index']);
```

When you run the Laravel app, this route becomes available at:

```arduino
http://127.0.0.1:8000/api/students
```

✅ Notice: Laravel automatically **prefixes** `api/` to all routes in `api.php`.

Php Laravel

2 What are Resource Controllers?

A **Resource Controller** automatically handles **CRUD operations** for a model (Create, Read, Update, Delete).

So instead of writing multiple routes like this:

```php
Route::get('/students', [StudentController::class, 'index']);
Route::post('/students', [StudentController::class, 'store']);
Route::get('/students/{id}', [StudentController::class, 'show']);
Route::put('/students/{id}', [StudentController::class, 'update']);
Route::delete('/students/{id}', [StudentController::class, 'destroy']);
```

You can simply write:

```php
Route::apiResource('students', StudentController::class);
```

✅ Laravel will automatically create **all 5 routes** for you.

3 Commands to Create Controller

To create a resource controller:

For web (with views)

```bash
php artisan make:controller StudentController --resource
```

For API only (no `create` / `edit` methods)

```bash
php artisan make:controller Api/StudentController --api
```

4 Example: Resource Controller

```php
namespace App\Http\Controllers\Api;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Student;

class StudentController extends Controller
{
    // GET /api/students
    public function index()
    {
        return Student::all();
    }

    // POST /api/students
    public function store(Request $request)
    {
        $validated = $request->validate([
            'name' => 'required',
            'email' => 'required|email|unique:students'
        ]);

        $student = Student::create($validated);
        return response()->json($student, 201);
    }

    // GET /api/students/{id}
    public function show($id)
    {
        return Student::findOrFail($id);
    }

    // PUT /api/students/{id}
    public function update(Request $request, $id)
    {
        $student = Student::findOrFail($id);
        $student->update($request->all());
        return response()->json($student, 200);
    }

    // DELETE /api/students/{id}
    public function destroy($id)
    {
        Student::destroy($id);
        return response()->json(null, 204);
    }
}
```

5 Generated Routes

Run this command to see them:

```bash
php artisan route:list
```

It will show something like:

Method	URI	Action
GET	/api/students	index
POST	/api/students	store
GET	/api/students/{student}	show
PUT/PATCH	/api/students/{student}	update
DELETE	/api/students/{student}	destroy

In simple words:

- "`api.php` = defines REST API routes"
- "`apiResource()` = automatically builds CRUD endpoints"
- "`Controller` = contains logic for each endpoint"

- **API Resources/Transformers**

1 What are API Resources?

API Resources (introduced in Laravel 5.5) are a way to **transform your Eloquent models into JSON responses** — in a clean, reusable, and consistent format.

2. API Resources / Transformers

- Used to **format JSON responses** consistently.

```bash
php artisan make:resource UserResource
```

Example: `UserResource.php`

```php
namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class UserResource extends JsonResource {
    public function toArray($request) {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at->toDateTimeString(),
        ];
    }
}
```

Using in Controller

```php
use App\Http\Resources\UserResource;

public function index() {
    $users = User::paginate(10);
    return UserResource::collection($users);
}
```

✅ This ensures only selected fields are exposed to clients.

⚙️ 2️⃣ Creating a Resource

Run the command:

```bash
php artisan make:resource StudentResource
```

This creates:

```swift
app/Http/Resources/StudentResource.php
```

🧠 4️⃣ Using the Resource in Controller

For single record:

```php
use App\Http\Resources\StudentResource;

public function show($id)
{
    $student = Student::findOrFail($id);
    return new StudentResource($student);
}
```

For multiple records:

```php
public function index()
{
    return StudentResource::collection(Student::all());
}
```

5 Adding Extra Info (Meta Data)

You can include **extra info** like status or messages:

```php
public function index()
{
    $students = Student::all();
    return StudentResource::collection($students)
        ->additional([
            'meta' => [
                'success' => true,
                'total' => $students->count()
            ]
        ]);
}
```

- Pagination & Filtering:

2 Basic Pagination

Laravel makes pagination super easy.

```php
public function index()
{
    // Get 10 students per page
    $students = Student::paginate(10);
    return StudentResource::collection($students);
}
```

Php Laravel

⚙️ 3️⃣ Custom Per Page

You can allow the client to control the page size:

```php
public function index(Request $request)
{
    $perPage = $request->get('per_page', 10); // default 10
    $students = Student::paginate($perPage);
    return StudentResource::collection($students);
}
```

Now you can call:

```bash
GET /api/students?per_page=5&page=2
```

🔍 4️⃣ Filtering (Search or Conditions)

Filtering means the client can request **specific data**, e.g. "students whose name contains 'anik'".

Example:

```php
public function index(Request $request)
{
    $query = Student::query();

    // Filter by name
    if ($request->has('name')) {
        $query->where('name', 'like', '%' . $request->name . '%');
    }

    // Filter by email
    if ($request->has('email')) {
        $query->where('email', $request->email);
    }

    // Apply pagination
    $students = $query->paginate(10);

    return StudentResource::collection($students);
}
```

🔹 Example URLs:

API Call	Description
`/api/students`	Get all students
`/api/students?name=ani`	Filter by name
`/api/students?email=ani@test.com`	Filter by email
`/api/students?name=ani&page=2`	Filter + paginate

⚡ 5️⃣ Sorting (Optional)

You can add sorting as well:

```php
if ($request->has('sort_by')) {
    $sortOrder = $request->get('sort_order', 'asc');
    $query->orderBy($request->sort_by, $sortOrder);
}
```

Now you can call:

```pgsql
GET /api/students?sort_by=name&sort_order=desc
```

- **API Versioning (v1)**

🔹 4. API Versioning (v1)

Versioning helps maintain multiple versions of your API.

Route prefix

```php
Route::prefix('v1')->group(function () {
    Route::apiResource('users', UserController::class);
});
```

- Endpoint becomes:

```bash
GET /api/v1/users
```

- Future version: `/api/v2/users` with updated fields or behavior.

🧩 1️⃣ What is API Versioning?

Over time, your API changes — you might:

- Add new fields
- Change response format
- Remove old endpoints

If users (like mobile apps) already use your old API, you **can't break** them.

✅ **Solution:** Version your API, so both old and new versions can coexist.

Example:

Version	Endpoint	Description
v1	`/api/v1/students`	Original API
v2	`/api/v2/students`	New version with improvements

So the client decides which version to use.

⚙️ 2️⃣ Folder Structure (Best Practice)

In Laravel, you can keep versioned controllers like this:

```markdown
app/
└── Http/
    └── Controllers/
        └── Api/
            ├── V1/
            │   └── StudentController.php
            └── V2/
                └── StudentController.php
```

Each version can have different logic or response formats.

🧱 3️⃣ Example: routes/api.php

```php
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\Api\V1\StudentController as StudentV1;
use App\Http\Controllers\Api\V2\StudentController as StudentV2;

// Version 1
Route::prefix('v1')->group(function () {
    Route::apiResource('students', StudentV1::class);
});

// Version 2
Route::prefix('v2')->group(function () {
    Route::apiResource('students', StudentV2::class);
});
```

Now you can access both:

```bash
http://127.0.0.1:8000/api/v1/students
http://127.0.0.1:8000/api/v2/students
```

4 Example: V1 Controller

app/Http/Controllers/Api/V1/StudentController.php

```php
namespace App\Http\Controllers\Api\V1;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Student;

class StudentController extends Controller
{
    public function index()
    {
        // Basic response
        return Student::all(['id', 'name', 'email']);
    }
}
```

5 Example: V2 Controller

app/Http/Controllers/Api/V2/StudentController.php

```php
namespace App\Http\Controllers\Api\V2;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Student;

class StudentController extends Controller
{
    public function index()
    {
        // Improved response (extra field or formatted)
        return Student::select('id', 'name as full_name', 'email')
                    ->get()
                    ->map(function ($student) {
                        $student->version = 'v2';
                        return $student;
                    });
    }
}
```

- **Error Handling:**

5. Error Handling

- Use `try/catch` in controllers or handle globally in `app/Exceptions/Handler.php`.

Example: Controller

```php
public function show($id) {
    try {
        $user = User::findOrFail($id);
        return new UserResource($user);
    } catch (\Illuminate\Database\Eloquent\ModelNotFoundException $e) {
        return response()->json([
            'error' => 'User not found'
        ], 404);
    }
}
```

1 Default Error Handling in Laravel

Laravel comes with a built-in error handler in:

```swift
app/Exceptions/Handler.php
```

This handles:

- Validation errors
- Model not found errors
- Authentication errors
- Server errors

Php Laravel

Example: Validation Error

```php
$request->validate([
    'name' => 'required',
    'email' => 'required|email|unique:students',
]);
```

If a field is missing:

```json
{
  "message": "The email field is required.",
  "errors": {
    "email": ["The email field is required."]
  }
}
```

✅ Laravel automatically returns **JSON response** if it's an API request.

Example: Model Not Found

Copy code

```php
$student = Student::findOrFail(10);
```

If `id=10` doesn't exist, Laravel throws a `ModelNotFoundException` → automatically returns **404 JSON** for API routes.

Copy code

```json
{
  "message": "No query results for model [App\\Models\\Student] 10"
}
```

2 Custom Error Handling

You can customize errors in `Handler.php`.

Copy

```php
public function render($request, Throwable $exception)
{
    if ($request->is('api/*')) {
        if ($exception instanceof \Illuminate\Database\Eloquent\ModelNotFoundException) {
            return response()->json([
                'message' => 'Resource not found'
            ], 404);
        }

        if ($exception instanceof \Illuminate\Validation\ValidationException) {
            return response()->json([
                'message' => $exception->getMessage(),
                'errors' => $exception->errors()
            ], 422);
        }

        return response()->json([
            'message' => 'Server Error'
        ], 500);
    }

    return parent::render($request, $exception);
}
```

🧱 3️⃣ HTTP Status Codes

When handling errors, always return the **correct HTTP status code**:

Situation	HTTP Code	Example
Validation Error	422	Missing field
Resource Not Found	404	Model not found
Unauthorized	401	Not logged in
Forbidden	403	Access denied
Server Error	500	Something unexpected

Example:

```php
return response()->json(['message' => 'Unauthorized'], 401);
```

⚡ 4️⃣ Try-Catch for Specific Errors

Sometimes you want to catch an exception manually:

```php
try {
    $student = Student::findOrFail($id);
} catch (\Illuminate\Database\Eloquent\ModelNotFoundException $e) {
    return response()->json([
        'message' => 'Student not found'
    ], 404);
}
```

🔹 5️⃣ Example: API Controller with Error Handling

```php
namespace App\Http\Controllers\Api\V1;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Models\Student;
use App\Http\Resources\StudentResource;

class StudentController extends Controller
{
    public function show($id)
    {
        try {
            $student = Student::findOrFail($id);
            return new StudentResource($student);
        } catch (\Illuminate\Database\Eloquent\ModelNotFoundException $e) {
            return response()->json([
                'message' => 'Student not found'
            ], 404);
        } catch (\Exception $e) {
            return response()->json([
                'message' => 'Server error'
            ], 500);
        }
    }
}
```

Laravel Breeze + Sanctum • Login/registration & password reset • Policies & Gates • Roles/permissions (Spatie) • Rate limiting

- ## Laravel Breeze + Sanctum:

Laravel Breeze

> A simple starter kit for authentication (login/register/forgot password), available for Blade, Vue, React, or API.

🧩 1️⃣ What is Laravel Breeze?

👉 **Laravel Breeze** is a **starter kit** that gives you ready-made **authentication features** like:

- ✅ Login
- ✅ Registration
- ✅ Password reset
- ✅ Email verification
- ✅ Logout

💡 In simple words:

> **Breeze** helps you quickly get login/register functionality working in Laravel — either for **web** (Blade) or **API + SPA** (React/Vue).

⚙️ 2️⃣ Installing Laravel Breeze

Make sure you already have a Laravel project set up.

Then install Breeze using Composer:

```bash
composer require laravel/breeze --dev
```

Then run:

```bash
php artisan breeze:install
```

🧱 3️⃣ Choose Your Stack

When you run `php artisan breeze:install`, Laravel will ask:

```css
Which Breeze stack would you like to install?
  [0] blade
  [1] react
  [2] vue
  [3] api
```

Options:

Stack	Description
Blade	Simple login/register using Blade templates
React / Vue	For SPAs (frontend JS frameworks)
API	For building mobile apps or API-only backends

⚡ 4️⃣ Example: Installing Blade Version

```bash
php artisan breeze:install blade
npm install
npm run dev
php artisan migrate
```

Now open:

👉 `http://127.0.0.1:8000/register`

👉 `http://127.0.0.1:8000/login`

🎉 You'll see ready-made forms for login and register.

In-case of **Npm Error**: Download the latest version **22** and switch on it.

1. Install nvm for Windows ↗
2. Then run:

```bash
nvm install 22
nvm use 22
node -v
```

✅ That switches Node instantly.

🔐 5️⃣ Example: Installing API Version (for mobile/backend)

```bash
php artisan breeze:install api
php artisan migrate
```

Now you'll get routes like:

Method	Endpoint	Description
POST	`/register`	Register new user
POST	`/login`	Login and get token
POST	`/logout`	Logout
GET	`/user`	Get current user info

You can check routes using:

```bash
php artisan route:list
```

🧩 1️⃣ What is Laravel Sanctum?

Laravel Sanctum is a lightweight authentication system for:

- ✅ Single Page Applications (SPA)
- ✅ Mobile Apps
- ✅ Token-based APIs

It allows users to:

- Login → receive a **token**
- Use that token in headers → to access protected API routes

💡 In simple words:

> **Sanctum** lets your API know "who is making the request" using a **secure token**, instead of sessions or cookies.

Laravel Sanctum

> Provides **API token authentication** for SPAs and mobile apps.

It's already included when you install Breeze with `--api`.
If separate:

```bash
composer require laravel/sanctum
php artisan install:sanctum
php artisan migrate
```

🧱 3️⃣ Add Sanctum Middleware

In `app/Http/Kernel.php`, ensure this line exists:

```php
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

Php Laravel

🔐 4️⃣ Generate Token on Login

Example Login API:

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

public function login(Request $request)
{
    $credentials = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if (!Auth::attempt($credentials)) {
        return response()->json(['message' => 'Invalid credentials'], 401);
    }

    $user = Auth::user();

    // Create token
    $token = $user->createToken('api_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'token' => $token
    ]);
}
```

Usage

Register

```bash
POST /api/register
{
  "name": "John",
  "email": "john@example.com",
  "password": "password",
  "password_confirmation": "password"
}
```

Login

```bash
POST /api/login
{
  "email": "john@example.com",
  "password": "password"
}
```

Returns:

```json
{
  "token": "1|fJ7hZ3vNqpZQJj4sI..."
}
```

Use this token in Authorization header:

```makefile
Authorization: Bearer <token>
```

Protected Route Example

```php
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});
```

🔒 6 Protecting Routes with Sanctum

In `routes/api.php`:

```php
use Illuminate\Support\Facades\Route;

Route::middleware('auth:sanctum')->group(function () {
    Route::get('/profile', function (Request $request) {
        return $request->user();
    });

    Route::apiResource('students', StudentController::class);
});
```

🧠 7 Logging Out

To delete a token and log out:

```php
public function logout(Request $request)
{
    $request->user()->currentAccessToken()->delete();
    return response()->json(['message' => 'Logged out']);
}
```

This removes the token from the database → user can't access APIs anymore.

🧠 7 Folder Structure (Blade Version)

```lua
resources/
 ├── views/
 │   ├── auth/          <-- login/register/reset views
 │   ├── components/    <-- Blade components
 │   └── layouts/       <-- app layout (navbar, etc.)
 ├── js/
 │   └── app.js         <-- front-end logic
```

- **Login/registration & password reset:**

Php Laravel

1 Overview

Laravel provides built-in authentication features.

If you've installed **Laravel Breeze** (recommended for beginners), it already includes:

- Registration
- Login / Logout
- Password Reset
- Email Verification (optional)
- Works with **Sanctum** tokens for APIs

2 Registration (Sign Up)

Example Route: `routes/api.php`

```php
use App\Http\Controllers\Auth\RegisteredUserController;

Route::post('/register', [RegisteredUserController::class, 'store']);
```

Controller: `app/Http/Controllers/Auth/RegisteredUserController.php`

This comes with Laravel Breeze.

```php
public function store(Request $request)
{
    $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|unique:users',
        'password' => 'required|string|confirmed|min:8',
    ]);

    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);

    // Create Sanctum token
    $token = $user->createToken('api_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'token' => $token
    ], 201);
}
```

Example Request (POST `/register`)

```json
{
  "name": "Aniket",
  "email": "aniket@test.com",
  "password": "password",
  "password_confirmation": "password"
}
```

Example Response

```json
{
  "user": {
    "id": 1,
    "name": "Aniket",
    "email": "aniket@test.com"
  },
  "token": "1|abcdefg12345..."
}
```

3 Login (Sign In)

Route

```php
use App\Http\Controllers\Auth\AuthenticatedSessionController;

Route::post('/login', [AuthenticatedSessionController::class, 'store']);
```

Php Laravel

⚙️ Controller: `AuthenticatedSessionController`

```php
public function store(Request $request)
{
    $credentials = $request->validate([
        'email' => 'required|string|email',
        'password' => 'required|string',
    ]);

    if (!Auth::attempt($credentials)) {
        return response()->json(['message' => 'Invalid credentials'], 401);
    }

    $user = Auth::user();

    $token = $user->createToken('api_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'token' => $token
    ]);
}
```

✅ Example Request (POST `/login`)

```json
{
  "email": "aniket@test.com",
  "password": "password"
}
```

✅ Example Response

```json
{
  "user": {
    "id": 1,
    "name": "Aniket",
    "email": "aniket@test.com"
  },
  "token": "1|QzX1m2n..."
}
```

🚪 4️⃣ Logout

📁 Route

```php
Route::post('/logout', [AuthenticatedSessionController::class, 'destroy'])
    ->middleware('auth:sanctum');
```

⚙️ Controller

```php
public function destroy(Request $request)
{
    $request->user()->currentAccessToken()->delete();
    return response()->json(['message' => 'Logged out']);
}
```

🔁 5️⃣ Password Reset (Forgot Password)

Laravel handles this through notifications and mail.

📁 Step 1 — Routes

Add these routes (already included with Breeze):

```php
use App\Http\Controllers\Auth\PasswordResetLinkController;
use App\Http\Controllers\Auth\NewPasswordController;

Route::post('/forgot-password', [PasswordResetLinkController::class, 'store']);
Route::post('/reset-password', [NewPasswordController::class, 'store']);
```

📩 Step 2 — Forgot Password Controller

```php
public function store(Request $request)
{
    $request->validate(['email' => 'required|email']);

    $status = Password::sendResetLink(
        $request->only('email')
    );

    return $status === Password::RESET_LINK_SENT
                ? response()->json(['message' => __($status)])
                : response()->json(['message' => __($status)], 400);
}
```

Php Laravel

Step 3 — Reset Password Controller

```php
public function store(Request $request)
{
    $request->validate([
        'token' => 'required',
        'email' => 'required|email',
        'password' => 'required|min:8|confirmed',
    ]);

    $status = Password::reset(
        $request->only('email', 'password', 'password_confirmation', 'token'),
        function ($user) use ($request) {
            $user->forceFill([
                'password' => Hash::make($request->password),
            ])->save();
        }
    );

    return $status === Password::PASSWORD_RESET
                ? response()->json(['message' => __($status)])
                : response()->json(['message' => __($status)], 400);
}
```

✅ This updates the password in the database once the token is verified.

6 Example API Flow Summary

Step	Endpoint	Description
`/register`	Register user	Creates new user + token
`/login`	Login user	Returns token
`/logout`	Logout	Deletes token
`/forgot-password`	Send reset link	Sends email
`/reset-password`	Update password	Sets new password

7 Environment Setup

To make password reset work, configure your mail settings in `.env`:

```ini
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=youremail@gmail.com
MAIL_PASSWORD=yourpassword
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=youremail@gmail.com
MAIL_FROM_NAME="${APP_NAME}"
```

Now, password reset emails will be sent automatically using Laravel's mail system.

2 For Gmail (most common case)

If you're using **Gmail**, the correct config is:

```env
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=your_gmail_address@gmail.com
MAIL_PASSWORD=your_app_password
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=your_gmail_address@gmail.com
MAIL_FROM_NAME="${APP_NAME}"
```

⚠️ **IMPORTANT: You can't use your normal Gmail password!**

Google **blocks "less secure app" logins.**
You must use an **App Password** instead.

Steps to get your Gmail App Password:

1. Go to https://myaccount.google.com ↗
2. Enable **2-Step Verification** for your account.
3. Go to:
   Security → App passwords
4. Select **App: Mail, Device: Other (Custom)** → type "Laravel"
5. Click **Generate**
6. Copy the **16-character password** (e.g. `abcd efgh ijkl mnop`)
7. Paste it in your `.env`:

```ini
MAIL_PASSWORD=abcd efgh ijkl mnop
```

✅ That's your SMTP password — not your regular Gmail password!

🧩 3️⃣ How to test it

Run:

```nginx
php artisan tinker
```

Then inside Tinker:

```php
Mail::raw('Hello from Laravel', function ($message) {
    $message->to('your_email@gmail.com')->subject('Test Email');
});
```

send an OTP rather than a link For ForgotPassword:

⚙️ Step 1: Create migration for storing OTPs

Run:

```bash
php artisan make:migration create_password_otps_table
```

Then edit it:

```php
Schema::create('password_otps', function (Blueprint $table) {
    $table->id();
    $table->string('email');
    $table->string('otp');
    $table->timestamp('expires_at');
    $table->timestamps();
});
```

Run:

```bash
php artisan migrate
```

Php Laravel

⚙️ Step 2: Create a model

```bash
php artisan make:model PasswordOtp
```

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class PasswordOtp extends Model
{
    protected $fillable = ['email', 'otp', 'expires_at'];
}
```

⚙️ Step 3: Forgot Password Controller — generate & email OTP

Edit or create a controller (for example):

```bash
php artisan make:controller Auth/OtpPasswordController
```

Then in `OtpPasswordController.php`:

```php
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Models\PasswordOtp;
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;
use Carbon\Carbon;

class OtpPasswordController extends Controller
{
    public function sendOtp(Request $request)
    {
        $request->validate(['email' => 'required|email|exists:users,email']);

        $otp = rand(100000, 999999); // 6-digit OTP

        PasswordOtp::updateOrCreate(
            ['email' => $request->email],
            ['otp' => $otp, 'expires_at' => Carbon::now()->addMinutes(10)]
        );
```

Php Laravel

```php
        Mail::raw("Your password reset OTP is: $otp", function ($message) use ($request) {
            $message->to($request->email)
                    ->subject('Your Password Reset OTP');
        });

        return back()->with('status', 'OTP sent to your email!');
    }
}
```

Step 4: Verify OTP and Reset Password

Add another method:

```php
use Illuminate\Support\Facades\Hash;

public function verifyOtp(Request $request)
{
    $request->validate([
        'email' => 'required|email',
        'otp' => 'required',
        'password' => 'required|confirmed|min:6',
    ]);

    $record = PasswordOtp::where('email', $request->email)
        ->where('otp', $request->otp)
        ->where('expires_at', '>', Carbon::now())
        ->first();
```

```php
    if (!$record) {
        return back()->withErrors(['otp' => 'Invalid or expired OTP']);
    }

    User::where('email', $request->email)->update([
        'password' => Hash::make($request->password),
    ]);

    $record->delete(); // remove used OTP

    return redirect()->route('login')->with('status', 'Password updated successfully!');
}
```

Step 5: Add routes

In `routes/web.php` (or `routes/api.php` for API):

```php
use App\Http\Controllers\Auth\OtpPasswordController;

Route::post('/forgot-password-otp', [OtpPasswordController::class, 'sendOtp'])->name('password.otp
Route::post('/reset-password-otp', [OtpPasswordController::class, 'verifyOtp'])->name('password.ot
```

```php
Mail::raw("hello aniket singh", function ($message) {
    $message->to("aniketsingh52777@gmail.com")->subject("invitation");
});
```

```env
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
# MAIL_HOST=127.0.0.1
MAIL_PORT=587
MAIL_USERNAME=aniketsingh42242@gmail.com
MAIL_PASSWORD=gborobdfmhbohmqd
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=aniketsingh42242@gmail.coms
MAIL_FROM_NAME="Aniket Singh"
```

- **Policies & Gates**

Php Laravel

1 Gates

Gates are **closures** that determine if a user can perform a given action.
They're useful for **simple, single-action authorization checks**.

Example: Only allow admins to delete posts

Step 1: Define a Gate
In `app/Providers/AuthServiceProvider.php`:

```php
use Illuminate\Support\Facades\Gate;

public function boot(): void
{
    $this->registerPolicies();

    Gate::define('delete-post', function ($user, $post) {
        return $user->is_admin; // only admin can delete
    });
}
```

Step 2: Use the Gate
In a controller:

```php
use Illuminate\Support\Facades\Gate;

public function destroy(Post $post)
{
    if (Gate::denies('delete-post', $post)) {
        abort(403, 'You are not authorized to delete this post.');
    }

    $post->delete();
    return redirect()->back()->with('success', 'Post deleted.');
}
```

Or in Blade:

```blade
@can('delete-post', $post)
    <form action="{{ route('posts.destroy', $post) }}" method="POST">
        @csrf
        @method('DELETE')
        <button type="submit">Delete</button>
    </form>
@endcan
```

📁 Example: `AuthServiceProvider.php`

```php
use Illuminate\Support\Facades\Gate;
use App\Models\User;

public function boot()
{
    Gate::define('view-admin-dashboard', function (User $user) {
        return $user->is_admin === 1;
    });
}
```

🧠 Using the Gate in Controller

```php
public function dashboard()
{
    if (Gate::denies('view-admin-dashboard')) {
        abort(403, 'You are not allowed to view this page.');
    }

    return view('admin.dashboard');
}
```

Php Laravel

OR simpler:

```php
Gate::authorize('view-admin-dashboard');
```

If the user isn't admin → Laravel automatically throws a `403 Forbidden`.

2 Policies

Policies are classes that group authorization logic for a **specific model**. They're cleaner for apps with many permissions on models.

Example: PostPolicy

Step 1: Generate a Policy

```bash
php artisan make:policy PostPolicy --model=Post
```

This creates `app/Policies/PostPolicy.php`.

Step 2: Define methods

```php
public function update(User $user, Post $post)
{
    return $user->id === $post->user_id; // only owner can update
}

public function delete(User $user, Post $post)
{
    return $user->is_admin; // only admin can delete
}
```

Step 3: Register the Policy

In `app/Providers/AuthServiceProvider.php`:

```php
protected $policies = [
    Post::class => PostPolicy::class,
];
```

Step 4: Use the Policy

In controller:

```php
public function update(Request $request, Post $post)
{
    $this->authorize('update', $post); // throws 403 if unauthorized

    $post->update($request->all());
    return redirect()->back();
}
```

In Blade:

```blade
@can('update', $post)
    <a href="{{ route('posts.edit', $post) }}">Edit</a>
@endcan
```

Php Laravel

3 Other Variants

Method	Description
`Gate::allows('ability', $model)`	Returns `true` or `false`
`Gate::denies('ability', $model)`	Returns `true` if the user **cannot** perform the action
`$this->authorize('ability', $model)`	Throws `403` if unauthorized
`@can('ability', $model)`	Blade directive, hides/shows content

1 Basic Syntax

```php
use Illuminate\Support\Facades\Gate;

if (Gate::allows('delete-post', $post)) {
    // The current user can delete this post
} else {
    // Not authorized
}
```

- `'delete-post'` → the **ability** you defined in a Gate or Policy.
- `$post` → optional parameter passed to the Gate/Policy.

Returns:

- `true` → user is allowed
- `false` → user is not allowed

⚙️ 2 Two ways Laravel handles Authorization

1. **Gates** — simple, closure-based checks
2. **Policies** — organized, model-specific rules

🔒 GATES (simple checks)

Think of a **Gate** like a "yes/no rule".

🗂️ Example: `AuthServiceProvider.php`

```php
use Illuminate\Support\Facades\Gate;
use App\Models\User;

public function boot()
{
    Gate::define('view-admin-dashboard', function (User $user) {
        return $user->is_admin === 1;
    });
}
```

Here we define a rule:

✅ Only users where `is_admin == 1` can view the admin dashboard.

🧠 Using the Gate in Controller

```php
public function dashboard()
{
    if (Gate::denies('view-admin-dashboard')) {
        abort(403, 'You are not allowed to view this page.');
    }

    return view('admin.dashboard');
}
```

OR simpler:

```php
Gate::authorize('view-admin-dashboard');
```

If the user isn't admin → Laravel automatically throws a `403 Forbidden`.

🧱 POLICIES (model-specific rules)

Policies are like **classes that group Gates together** for a model.

They are cleaner when you have many permissions (e.g. Post create/edit/delete).

🧩 1️⃣ Create a Policy

```bash
php artisan make:policy PostPolicy --model=Post
```

This creates:

```swift
app/Policies/PostPolicy.php
```

2️⃣ Check the base controller

Open:

`app/Http/Controllers/Controller.php`

It must include the following:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Foundation\Auth\Access\AuthorizesRequests;
use Illuminate\Foundation\Validation\ValidatesRequests;
use Illuminate\Routing\Controller as BaseController;

class Controller extends BaseController
{
    use AuthorizesRequests, ValidatesRequests;
}
```

Php Laravel

⚙️ 2️⃣ Define Rules in Policy

Example: `app/Policies/PostPolicy.php`

```php
namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    public function view(User $user, Post $post)
    {
        // Anyone can view
        return true;
    }

    public function update(User $user, Post $post)
    {
        // Only the owner can update
        return $user->id === $post->user_id;
    }

    public function delete(User $user, Post $post)
    {
        // Only admin or owner can delete
        return $user->is_admin || $user->id === $post->user_id;
    }
}
```

🧩 3️⃣ Register Policy

Open `app/Providers/AuthServiceProvider.php` and map it:

```php
protected $policies = [
    Post::class => PostPolicy::class,
];
```

🧠 4️⃣ Use Policy in Controller

```php
public function update(Request $request, Post $post)
{
    $this->authorize('update', $post);

    $post->update($request->all());
    return response()->json(['message' => 'Post updated!']);
}
```

If the user is not allowed → Laravel automatically returns **403 Forbidden**.

5 Use in Blade Views

```blade
@can('update', $post)
    <button>Edit Post</button>
@endcan

@cannot('delete', $post)
    <p>You cannot delete this post.</p>
@endcannot
```

In Simple Words

- **Gate** = a single rule (e.g. "is admin?")
- **Policy** = a collection of rules tied to a model (e.g. "can edit post?")

- **Roles/permissions (Spatie):**

1 What is Spatie Laravel Permission?

Spatie Laravel Permission is a popular package that lets you:

- Assign **roles** to users (`admin`, `editor`, `subscriber`)
- Assign **permissions** to users or roles (`edit-post`, `delete-post`)
- Check these roles/permissions easily in code or Blade

✅ It works perfectly with **Breeze + Sanctum APIs**.

2 Install the Package

```bash
composer require spatie/laravel-permission
```

Publish the config & migration:

```bash
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"
php artisan migrate
```

In case of length error change the default string size form **191** to **125**

By setting:

```php
Schema::defaultStringLength(125);
```

This will create:

- `config/permission.php`
- migrations for `roles`, `permissions`, and pivot tables.

This creates the following tables:

- `roles`
- `permissions`
- `model_has_roles`
- `model_has_permissions`
- `role_has_permissions`

🧱 3️⃣ Add the Trait to User Model

```php
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable
{
    use HasRoles;

    // other code...
}
```

✅ Now each user can have roles and permissions.

🔐 4️⃣ Creating Roles & Permissions

You can do it via **tinker**, seeders, or controller.

Example: Seeder

```php
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

Role::create(['name' => 'admin']);
Role::create(['name' => 'editor']);

Permission::create(['name' => 'edit post']);
Permission::create(['name' => 'delete post']);
Permission::create(['name' => 'publish post']);
```

Assign Permissions to Roles

```php
$role = Role::findByName('editor');
$role->givePermissionTo('edit post');

$role = Role::findByName('admin');
$role->givePermissionTo(Permission::all());
```

Assign Roles to Users

```php
$user = User::find(1);
$user->assignRole('admin');  // User is now admin

$user2 = User::find(2);
$user2->assignRole('editor'); // User is editor
```

4 Assign Role/Permission to Users

```php
$user = User::find(1);

// Assign role
$user->assignRole('admin');

// Assign individual permission
$user->givePermissionTo('delete posts');

// Check role
if ($user->hasRole('admin')) { ... }

// Check permission
if ($user->can('edit posts')) { ... }
```

5 Check Roles / Permissions in Code

```php
if ($user->hasRole('admin')) {
    // user is admin
}

if ($user->can('edit post')) {
    // user can edit post
}
```

Controller Example

```php
public function update(Request $request, Post $post)
{
    if (!auth()->user()->can('edit post')) {
        return response()->json(['message' => 'Forbidden'], 403);
    }

    $post->update($request->all());
    return response()->json(['message' => 'Post updated!']);
}
```

To use It in **web.php** as middleware we need to make some changes in **bootstrap/app.php:**

Php Laravel

👉 Inside this `withMiddleware()` closure, **add the aliases** like this:

```php
use Spatie\Permission\Middlewares\RoleMiddleware;
use Spatie\Permission\Middlewares\PermissionMiddleware;
use Spatie\Permission\Middlewares\RoleOrPermissionMiddleware;

->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'role' => RoleMiddleware::class,
        'permission' => PermissionMiddleware::class,
        'role_or_permission' => RoleOrPermissionMiddleware::class,
    ]);
})
```

6 Middleware

Spatie provides **middleware** so you can protect routes easily:

```php
Route::group(['middleware' => ['role:admin']], function () {
    Route::delete('/posts/{post}', [PostController::class, 'destroy']);
});

Route::group(['middleware' => ['permission:edit post']], function () {
    Route::put('/posts/{post}', [PostController::class, 'update']);
});
```

```php
Route::get('/test', [UserController::class, "index"])->middleware("role:admin");
// Route::group(['middleware' => ['permission:View Post']], function () {
//     Route::get('/test2', [UserController::class, 'index']);
// });
Route::middleware("role:Super admin")->group(function () {
    Route::get('/test2', [UserController::class, 'index']);
});
```

```php
$user->assignRole('admin');
// $user->removeRole('super admin');
$user->givePermissionTo('Delete Post');
// $user->revokePermissionTo('Delete Post');
```

Php Laravel

7 Blade Example

```blade
@role('admin')
    <button>Delete Post</button>
@endrole

@can('edit post')
    <button>Edit Post</button>
@endcan
```

In simple words:

> **Spatie Permission = Ready-made role & permission management.**
> Instead of writing custom policies for every tiny action, you can just assign roles/permissions to users and check them easily anywhere (Controller, API, Blade).

3 Creating Roles and Permissions

You can do this in a seeder, Tinker, or controller.

Example:

```php
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

// Create permissions
Permission::create(['name' => 'edit posts']);
Permission::create(['name' => 'delete posts']);
Permission::create(['name' => 'publish posts']);

// Create roles and assign permissions
$roleAdmin = Role::create(['name' => 'admin']);
$roleAdmin->givePermissionTo(Permission::all());

$roleEditor = Role::create(['name' => 'editor']);
$roleEditor->givePermissionTo(['edit posts', 'publish posts']);
```

8 Quick Summary Table

Concept	Method	Example
Assign role to user	`$user->assignRole('admin')`	admin, editor
Assign permission to role	`$role->givePermissionTo('edit post')`	edit-post permission
Check role	`$user->hasRole('admin')`	true/false
Check permission	`$user->can('edit post')`	true/false
Middleware	Protect routes	`['middleware' => ['role:admin']]`
Blade directives	Show buttons	`@role`, `@can`

- **Rate limiting:**

🧩 1️⃣ What is Rate Limiting?

Rate limiting means:

> "Limit how many requests a user can make to your application in a given time."

Example:

- Allow 60 requests per minute
- If the user sends the 61st request → they get a **429 Too Many Requests** error.

✅ Prevents abuse (like spamming your API or login brute-force attacks).

⚙️ 2️⃣ How Laravel Handles Rate Limiting

Laravel uses middleware called:

```php
'throttle'
```

This middleware checks how many requests a user (or IP) has made within a time frame.

🧩 2️⃣ Basic Example — Throttle Middleware on Routes

Example:

```php
Route::middleware(['throttle:5,1'])->group(function () {
    Route::get('/login-attempt', function () {
        return 'Trying to log in...';
    });
});
```

🔍 Meaning:

- `5,1` = allow **5 requests per 1 minute** per user/IP.

🧠 3️⃣ You Can Apply It to a Single Route Too

```php
Route::get('/api/data', function () {
    return ['data' => 'some data'];
})->middleware('throttle:10,2');
```

✅ Allows **10 requests every 2 minutes** per unique IP/user.

⚡ 4️⃣ Defining Custom Rate Limiters (Recommended Way)

You can define **named limiters** in:

```swift
app/Providers/RouteServiceProvider.php
```

Example:

```php
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;

public function boot(): void
{
    RateLimiter::for('api', function ($request) {
        return Limit::perMinute(10)->by($request->user()?->id ?: $request->ip());
    });
}
```

Now you can apply it to routes like this:

```php
Route::middleware(['throttle:api'])->group(function () {
    Route::get('/posts', [PostController::class, 'index']);
});
```

✅ This allows **10 requests per minute per user (or IP)** for all `/posts` routes.

5 Rate Limiting by User, IP, or Custom Key

You can change how you identify the "client":

```php
RateLimiter::for('login', function ($request) {
    return [
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

This will rate limit based on **email** — useful for login attempts.

```php
RateLimiter::for('login', function ($request) {
    return Limit::perMinute(3)->by($request->input('email'));
});
```

```php
Route::middleware(['throttle:login'])->group(function () {
    Route::get('login', [AuthenticatedSessionController::class, 'create'])
        ->name('login');
});
```

```php
Route::middleware(["throttle:6,1", "role:Super admin"])->group(function () {
    Route::get('/test2', [UserController::class, 'index']);
});
```

Default Rule

```php
RateLimiter::for('api', function (Request $request) {
    return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
});
```

7 Advanced Example — Dynamic Limits

You can even define different limits for admins and normal users:

```php
RateLimiter::for('api', function ($request) {
    return $request->user()?->isAdmin()
        ? Limit::none()
        : Limit::perMinute(20)->by($request->user()?->id ?: $request->ip());
});
```

- ✅ Admins = unlimited
- ✅ Normal users = 20 per minute

🔐 6️⃣ Different Limits per Route

You can use custom throttle rules in routes:

```php
Route::middleware(['throttle:10,1'])->group(function () {
    Route::get('/profile', [UserController::class, 'profile']); // 10 req/min
});

Route::middleware(['throttle:100,1'])->group(function () {
    Route::get('/products', [ProductController::class, 'index']); // 100 req/min
});
```

Syntax:

`throttle:<max_requests>,<minutes>`

Example:

`throttle:10,1` → 10 requests per minute.

🧩 7️⃣ Named Limiters (Advanced)

You can define custom limiters for specific features.

```php
RateLimiter::for('login', function (Request $request) {
    return Limit::perMinute(5)->by($request->ip());
});
```

Then apply it to routes:

```php
Route::post('/login', [AuthController::class, 'login'])->middleware('throttle:login');
```

📊 8️⃣ What Happens When Limit Exceeds?

If you exceed your rate limit, Laravel automatically returns:

HTTP 429 — Too Many Requests

Example Response:

```json
{
  "message": "Too Many Attempts."
}
```

It also sends headers:

```makefile
Retry-After: 30
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 0
```

Queues (Redis) & Workers • Jobs & dispatch • Events & Listeners • Notifications (mail/db) • Task Scheduling

- ## Queues (Redis) & Workers :

⚙️ 1️⃣ What Are Queues?

Queues let you **run time-consuming tasks in the background**, instead of during the main HTTP request.

✅ Example:
Instead of making users wait for:

- Emails to send
- Reports to generate
- Notifications to push
- Image uploads to process

You **push the job to a queue**, and a **worker** will handle it later (asynchronously).

⚙️ 2️⃣ How It Works

Here's the flow 👇

1️⃣ Controller adds a **Job** to the queue.
2️⃣ The job is stored in the **queue system (Redis, Database, etc.)**.
3️⃣ A **Worker** listens for jobs and runs them one by one.

⚡ 2️⃣ Flow Overview

Here's the full **Queue + Worker flow**:

```scss
[ Controller / Event / Listener ]
        ↓
    Job Dispatched
        ↓
   Stored in Queue (Redis)
        ↓
 Worker (background process)
        ↓
 Executes Job (sends email, etc.)
```

🧠 Visual Example

```arduino
User clicks "Register"
   ↓
Job: SendWelcomeEmail → added to queue
   ↓
Queue Worker picks the job
   ↓
Email sent in background
```

🧠 3️⃣ Setup: Configure Queue Driver

By default, Laravel uses the **sync** driver (runs jobs immediately).

In `.env`, change this:

```env
QUEUE_CONNECTION=redis
```

Now it will push jobs to **Redis**.

🧱 3️⃣ Supported Queue Drivers

Laravel supports several queue drivers out of the box:

Driver	Description
`sync`	Executes jobs instantly (no background queue)
`database`	Stores jobs in a DB table
`redis`	Stores jobs in Redis (fast, best for production)
`beanstalkd`	Another message queue server
`sqs`	AWS Simple Queue Service

🧩 Step 1: Add Redis Service to Sail

Open your project's file:

```
docker-compose.yml
```

```yaml
  redis:
    image: 'redis:alpine'
    ports:
      - '6379:6379'
    networks:
      - sail

networks:
  sail:
    driver: bridge

volumes:
  sailmysql:
    driver: local
```

👉 Note: You can replace the `redis` version (e.g., `redis:7-alpine`) if needed.

🧩 Step 2: Rebuild Sail Containers

After saving the file, rebuild your Sail environment:

```bash
./vendor/bin/sail build --no-cache
./vendor/bin/sail up -d
```

This downloads and starts the Redis container.

🧩 Step 3: Check if Redis is Running

Run:

```bash
./vendor/bin/sail ps
```

To verify Redis inside the container:

```bash
./vendor/bin/sail exec redis redis-cli ping
```

If it returns:

```nginx
PONG
```

Step 4: Update Laravel Config

In your `.env` file:

```env
QUEUE_CONNECTION=redis
CACHE_DRIVER=redis
SESSION_DRIVER=redis
```

And make sure `config/database.php` has Redis configured (by default it does):

```php
'redis' => [
    'client' => env('REDIS_CLIENT', 'phpredis'),
    'default' => [
        'host' => env('REDIS_HOST', 'redis'),
        'password' => env('REDIS_PASSWORD', null),
        'port' => env('REDIS_PORT', 6379),
        'database' => env('REDIS_DB', 0),
    ],
],
```

Change the redis host from **127.0.0.1** to **redis** in the **.env** file as well

⚠️ Important:

> In Docker Sail, the host name should be `redis`, not `127.0.0.1`, because containers communicate via network aliases.

Then install the **PHP Redis extension:**

```bash
composer require predis/predis
```

🧩 Step 3: Create a Job

```bash
php artisan make:job SendWelcomeEmail
```

This creates:

`app/Jobs/SendWelcomeEmail.php`

```php
<?php

namespace App\Jobs;

use Illuminate\Contracts\Queue\ShouldQueue;
use App\Mail\WelcomeMail;
use Illuminate\Support\Facades\Mail;
use Illuminate\Foundation\Queue\Queueable;

class SendWelcomeEmail implements ShouldQueue
{
    use Queueable;
    public $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function handle(): void
    {
        Mail::to($this->user->email)->send(new WelcomeMail($this->user));
    }
}
```

if **WelcomeMail** doesn't exist in **App\Mail** the Create it:

If it **doesn't exist**, create it using Artisan:

```bash
php artisan make:mail WelcomeMail --markdown=emails.welcome
```

Php Laravel

```php
namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class WelcomeMail extends Mailable
{
    use Queueable, SerializesModels;

    public $user;

    public function __construct($user)
    {
        $this->user = $user;
    }

    public function build()
    {
        return $this->markdown('emails.welcome')
                    ->subject('Welcome!')
                    ->with(['user' => $this->user]);
    }
}
```

🧩 Step 4: Dispatch the Job

In your controller:

```php
use App\Jobs\SendWelcomeEmail;
use App\Models\User;

public function register(Request $request)
{
    $user = User::create($request->all());

    // Dispatch job to queue
    SendWelcomeEmail::dispatch($user);

    return response()->json(['message' => 'User created successfully']);
}
```

🧩 Step 5: Run the Queue Worker

In one terminal tab, start the worker:

```bash
php artisan queue:work
```

This listens for new jobs in Redis and executes them.

You can also use:

```bash
php artisan queue:listen
```

...but `queue:work` is faster and preferred for production.

Note: to run the redis in sail use the "**./vendor/bin/sail artisan queue:work** "

Php Laravel

🧩 Step 6: Retry / Failed Jobs

If something fails, Laravel can track failed jobs.

1. Create the failed_jobs table:

```bash
php artisan queue:failed-table
php artisan migrate
```

2. Check failed jobs:

```bash
php artisan queue:failed
```

3. Retry failed jobs:

```bash
php artisan queue:retry all
```

4. Clear failed jobs:

```bash
php artisan queue:flush
```

🧩 Optional: Delayed Jobs

You can delay a job like this:

```php
SendWelcomeEmail::dispatch($user)->delay(now()->addMinutes(5));
```

Method-2:

🔌 4️⃣ Install & Configure Redis

Install Redis on your machine (if not already).

For Windows:

- Use Docker or `memurai` (Redis-compatible)
- Or use **Laragon / WSL2** which supports Redis

Then install the PHP package:

```bash
composer require predis/predis
```

✅ Step 2: Set `.env`

```env
QUEUE_CONNECTION=redis
CACHE_DRIVER=redis
SESSION_DRIVER=redis
```

✅ Step 3: Start Redis server

If you're using Laravel Sail (Docker):

```bash
./vendor/bin/sail up -d redis
```

If local, install Redis and start it:

```bash
redis-server
```

🧱 5️⃣ Create a Job

Use Artisan to generate a new job:

```bash
php artisan make:job SendWelcomeEmail
```

This creates:

`app/Jobs/SendWelcomeEmail.php`

Example Job Code

```php
namespace App\Jobs;

use App\Mail\WelcomeMail;
use Illuminate\Support\Facades\Mail;

class SendWelcomeEmail extends Job
{
    public $user;

    public function __construct($user)
    {
        $this->user = $user;  // store user data
    }

    public function handle()
    {
        Mail::to($this->user->email)->send(new WelcomeMail($this->user));
    }
}
```

Php Laravel

🚀 7️⃣ Dispatching a Job

You can dispatch the job anywhere (e.g., in a controller):

```php
use App\Jobs\SendWelcomeEmail;
use App\Models\User;

public function store(Request $request)
{
    $user = User::create($request->all());
    SendWelcomeEmail::dispatch($user);

    return "User created and welcome email queued!";
}
```

⚙️ 6️⃣ Dispatch the Job

In your controller:

```php
use App\Jobs\SendWelcomeEmail;

public function register(Request $request)
{
    $user = User::create($request->all());

    // Push job into the queue
    SendWelcomeEmail::dispatch($user);

    return response()->json(['message' => 'User registered successfully!']);
}
```

🧰 8️⃣ Run the Worker

This command starts the background worker:

```bash
php artisan queue:work
```

Now Laravel will constantly check Redis for new jobs, and process them automatically.

✅ Output example:

```yaml
[2025-10-22 11:30:10] Processing: App\Jobs\SendWelcomeEmail
[2025-10-22 11:30:11] Processed:  App\Jobs\SendWelcomeEmail
```

🔄 9️⃣ Queue Monitoring Commands

Command	Purpose
`php artisan queue:work`	Run worker
`php artisan queue:listen`	Same as above but restarts on code changes
`php artisan queue:failed`	List failed jobs
`php artisan queue:retry {id}`	Retry a failed job
`php artisan queue:flush`	Delete all failed jobs

⚠️ 1️⃣0️⃣ Handling Failed Jobs

In `.env`, set up a table for failed jobs:

```env
QUEUE_CONNECTION=redis
QUEUE_FAILED_DRIVER=database
```

Then create the table:

```bash
php artisan queue:failed-table
php artisan migrate
```

Now, if a job fails, Laravel stores it in `failed_jobs` table — useful for debugging or retrying.

Php Laravel

🧩 1️⃣1️⃣ Delayed Jobs Example

You can delay execution like this:

```php
SendWelcomeEmail::dispatch($user)->delay(now()->addMinutes(5));
```

🔁 8️⃣ Queue Retry & Failed Jobs

If a job fails, Laravel can retry automatically.

Retry failed jobs:

```bash
php artisan queue:retry all
```

List failed jobs:

```bash
php artisan queue:failed
```

Delete failed jobs:

```bash
php artisan queue:flush
```

You can also create a **failed_jobs** table:

```bash
php artisan queue:failed-table
php artisan migrate
```

Method-3:Database

⚙️ 1️⃣ Set Queue Driver to Database

Open your `.env` file and change:

```env
QUEUE_CONNECTION=database
```

⚙️ 2️⃣ Create the Jobs Table

Laravel stores queued jobs in a database table named `jobs`.

Run these commands:

```bash
php artisan queue:table
php artisan migrate
```

✅ This will create a table called `jobs` with columns like:

- `id`
- `queue`
- `payload`
- `attempts`
- `available_at`
- `created_at`

⚙️ 3️⃣ (Optional) Create Failed Jobs Table

To store jobs that fail:

```bash
php artisan queue:failed-table
php artisan migrate
```

This creates the `failed_jobs` table.

Now rest remains the same create a job and dispatch it using a controller.

⚙️ 6️⃣ View Jobs in the Database

Now, your job will be inserted into the `jobs` table.

You can check with:

```sql
SELECT * FROM jobs;
```

The job will stay there until it's processed.

⚙️ 7️⃣ Start the Worker

Run this command to process queued jobs:

```bash
php artisan queue:work
```

Laravel will pick up jobs from the `jobs` table and execute them.

✅ Summary

Step	Command	Description
1	`QUEUE_CONNECTION=database`	Use database queue
2	`php artisan queue:table && php artisan migrate`	Create jobs table
3	`php artisan queue:failed-table && php artisan migrate`	Track failed jobs
4	`php artisan make:job JobName`	Create a job
5	`JobName::dispatch()`	Dispatch a job
6	`php artisan queue:work`	Process jobs

- **Jobs & dispatch:**

🧠 1️⃣ What Is a "Job"?

A **Job** is a class that contains code you want to run later — either **in the background** (using a queue) or **immediately**.

💡 Example:

You want to send an email after user registration.

That email sending process can be written inside a **Job** class.

🧱 2️⃣ Create a Job

Run the Artisan command:

bash

```bash
php artisan make:job SendWelcomeEmail
```

This will create a file at:

swift

```swift
app/Jobs/SendWelcomeEmail.php
```

⚙️ 3️⃣ Example Job Code

php

```php
<?php

namespace App\Jobs;

use App\Mail\WelcomeMail;
use Illuminate\Support\Facades\Mail;

class SendWelcomeEmail implements ShouldQueue
{
    public $user;

    // Accept data from controller
    public function __construct($user)
    {
        $this->user = $user;
    }

    // This method runs when the job is processed
    public function handle()
    {
        Mail::to($this->user->email)->send(new WelcomeMail($this->user));
    }
}
```

🧩 Key points:

- `implements ShouldQueue` → tells Laravel to push this job to a **queue**.
- `handle()` → contains the actual logic to run.
- `$this->user` → the data passed when dispatching.

🚀 4️⃣ Dispatch the Job

In your controller, after a user registers:

```php
use App\Jobs\SendWelcomeEmail;

public function register(Request $request)
{
    $user = User::create($request->all());

    // Dispatch the job
    SendWelcomeEmail::dispatch($user);

    return response()->json(['message' => 'User registered successfully!']);
}
```

✅ This line:

```php
SendWelcomeEmail::dispatch($user);
```

means → "Add a new job to the queue to send a welcome email for this user."

🔁 5️⃣ Run the Job (Worker)

If your queue driver is `sync` (default), the job runs immediately.

Check your `.env`:

```env
QUEUE_CONNECTION=sync
```

✅ If you change it to `redis` or `database`, it will go to background queue:

```env
QUEUE_CONNECTION=redis
```

Then start the worker:

```bash
php artisan queue:work
```

You'll see:

```yaml
[2025-10-22 11:30:00] Processing: App\Jobs\SendWelcomeEmail
[2025-10-22 11:30:01] Processed:  App\Jobs\SendWelcomeEmail
```

⏳ 6️⃣ Delay a Job

You can delay execution:

```php
SendWelcomeEmail::dispatch($user)->delay(now()->addMinutes(5));
```

✅ This waits 5 minutes before sending.

8 Retrying & Failed Jobs

If your job fails (e.g., mail server down), Laravel will automatically retry a few times.

You can configure:

```bash
php artisan queue:failed
php artisan queue:retry all
```

You can also add a `failed()` method in your Job:

```php
public function failed(\Exception $exception)
{
    Log::error('Failed to send welcome email: '.$exception->getMessage());
}
```

9 In Simple Words

> A **Job** = task to be done (like sending email).
> `dispatch()` = tell Laravel "go run this job (now or later)".
> `queue:work` = background worker that actually runs it.

Quick Summary

Term	Meaning	Example
Job	A class that defines a background task	`SendWelcomeEmail`
Dispatch	Add job to queue	`SendWelcomeEmail::dispatch($user)`
Queue Driver	Where jobs are stored	`sync`, `database`, `redis`
Worker	Runs queued jobs	`php artisan queue:work`
Delay	Run job later	`->delay(now()->addMinutes(5))`
Failed Jobs	Jobs that error out	`queue:failed`

- **Notifications (mail/db):**

1 What Are Notifications?

> Laravel **Notifications** let you send alerts or updates to users through different "channels" — like **email**, **database**, **Slack**, or **SMS**.

Examples:

- Send "Welcome" email when a user registers
- Store "New Comment" notification in the database
- Notify admin when a new order is placed

Php Laravel

2 Create a Notification

Run this command:

```bash
php artisan make:notification WelcomeNotification
```

This creates:

```swift
app/Notifications/WelcomeNotification.php
```

3 Structure of a Notification

Here's what it looks like (simplified):

```php
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\MailMessage;

class WelcomeNotification extends Notification
{
    use Queueable;

    public function __construct(public $user) {}

    // Channels (mail, database, sms, etc.)
    public function via($notifiable)
    {
        return ['mail', 'database'];
    }

    // Email version
    public function toMail($notifiable)
    {
        return (new MailMessage)
                    ->subject('Welcome to our app!')
                    ->greeting('Hello '.$this->user->name.'!')
                    ->line('Thank you for registering.')
                    ->action('Visit Dashboard', url('/dashboard'))
                    ->line('We're happy to have you!');
    }

    // Database version
    public function toDatabase($notifiable)
    {
        return [
            'message' => 'Welcome, '.$this->user->name.'! Thanks for joining.',
            'user_id' => $this->user->id,
        ];
    }
}
```

4 Sending the Notification

In your controller:

```php
use App\Notifications\WelcomeNotification;
use Illuminate\Support\Facades\Notification;

public function register(Request $request)
{
    $user = User::create($request->all());

    // Method 1: Notify that specific user
    $user->notify(new WelcomeNotification($user));

    // Method 2: Send to multiple users
    // Notification::send($users, new WelcomeNotification($user));

    return response()->json(['message' => 'User registered & notified!']);
}
```

✅ Laravel automatically sends an **email** and stores the **database record**.

Php Laravel

5 Database Notifications Setup

To use the **database** channel, you must create a notifications table:

bash

Copy code

```bash
php artisan notifications:table
php artisan migrate
```

This creates a `notifications` table with JSON data.

Now each record looks like this:

id	type	notifiable_type	notifiable_id	data	read_at
1	App\Notifications\WelcomeNotification	App\Models\User	1	{"message": "Welcome, Aniket!"}	NULL

6 View Notifications

You can fetch notifications like this:

```php
$user = auth()->user();

// All notifications
$user->notifications;

// Unread notifications
$user->unreadNotifications;

// Mark one as read
$user->unreadNotifications->first()->markAsRead();
```

5 Retrieve notifications

```php
$user->notifications;        // all notifications
$user->unreadNotifications;  // only unread ones
$user->readNotifications;    // only read ones
```

7 Mail Notification Example

If your `.env` has mail configured:

```env
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=your_email@gmail.com
MAIL_PASSWORD=your_password
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=your_email@gmail.com
MAIL_FROM_NAME="My App"
```

Then Laravel will automatically send the email via `toMail()`.

8 Queued Notifications (Optional)

To send in background (faster for users):

```php
use Illuminate\Contracts\Queue\ShouldQueue;

class WelcomeNotification extends Notification implements ShouldQueue
{
    // ...
}
```

✅ This pushes notifications into the **queue** (Redis, Database, etc.).

Php Laravel

🎨 9️⃣ Blade Example: Show Notifications in Dashboard

```blade
<h3>Notifications</h3>

<ul>
@foreach (auth()->user()->unreadNotifications as $notification)
    <li>{{ $notification->data['message'] }}</li>
@endforeach
</ul>
```

✅ Displays all unread notifications stored in the database.

🧩 STEP 1 — Create notifications table

Run:

```bash
php artisan notifications:table
php artisan migrate
```

✅ This creates a `notifications` table for storing DB notifications.

🧱 STEP 2 — Create a Notification class

Run:

```bash
php artisan make:notification WelcomeNotification
```

It will create:

```swift
app/Notifications/WelcomeNotification.php
```

✍️ STEP 3 — Define how it should be sent

Open `app/Notifications/WelcomeNotification.php` and modify like this:

```php
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;

class WelcomeNotification extends Notification
{
    use Queueable;

    public $user;

    /**
     * Create a new notification instance.
     */
    public function __construct($user)
    {
        $this->user = $user;
    }

    /**
     * Define channels (mail + database).
     */
    public function via($notifiable)
    {
        return ['mail', 'database'];
    }

    /**
     * Email notification content.
     */
    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->subject('Welcome to Our App 🎉')
            ->greeting('Hello ' . $this->user->name . '!')
            ->line('Thanks for registering on our website.')
            ->action('Visit Dashboard', url('/dashboard'))
            ->line('We're glad to have you with us!');
    }

    /**
     * Database notification content.
     */
    public function toDatabase($notifiable)
    {
        return [
            'message' => 'Welcome ' . $this->user->name . '! Thanks for joining us.',
            'user_id' => $this->user->id,
        ];
    }
}
```

⚙️ STEP 4 — Trigger notification

In your `User` model (`app/Models/User.php`), make sure it uses the `Notifiable` trait:

```php
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;
}
```

Then, in a controller (like after registration):

```php
use App\Models\User;
use App\Notifications\WelcomeNotification;

public function store(Request $request)
{
    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => bcrypt($request->password),
    ]);

    $user->notify(new WelcomeNotification($user));

    return "Notification sent!";
}
```

📬 STEP 5 — Email setup (optional)

In your `.env` file:

```env
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=youremail@gmail.com
MAIL_PASSWORD=yourapppassword
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=youremail@gmail.com
MAIL_FROM_NAME="My Laravel App"
```

You must enable “App Passwords” if using Gmail.

💾 STEP 6 — View notifications (DB)

You can now fetch notifications like this:

```php
$user = User::find(1);

foreach ($user->notifications as $notification) {
    echo $notification->data['message'] . '<br>';
}
```

Or only unread ones:

```php
$user->unreadNotifications;
```

You can mark them as read:

```php
$user->unreadNotifications->markAsRead();
```

✅ Summary

Step	Command / Concept	Purpose
1	`php artisan notifications:table`	Creates notifications migration
2	`php artisan migrate`	Creates notifications table
3	`php artisan make:notification`	Generates a notification class
4	`$user->notify(new WelcomeNotification($user))`	Sends mail + stores DB entry
5	`$user->notifications`	Retrieve user's notifications

- **Task Scheduling:**

🧠 1️⃣ What is Task Scheduling?

> **Task Scheduling** = telling Laravel to **run specific code automatically at certain times** — hourly, daily, weekly, etc.

Instead of writing multiple `cron` entries manually, Laravel gives you **one easy place** to define all scheduled tasks in PHP.

💡 Examples:

- Send daily reports every morning
- Delete old log files every week
- Auto-refresh cache every hour

Php Laravel

⚙️ 2 The Schedule File

All scheduled tasks go inside:

```arduino
app/Console/Kernel.php
```

You'll find a `schedule()` method like this:

```php
protected function schedule(Schedule $schedule)
{
    // Define your scheduled tasks here
}
```

app/Console/Kernel.php

```php
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * Define the application's command schedule.
     */
    protected function schedule(Schedule $schedule): void
    {
        // Example: $schedule->command('emails:send')->daily();
    }

    /**
     * Register the commands for the application.
     */
    protected function commands(): void
    {
        $this->load(__DIR__.'/Commands');

        require base_path('routes/console.php');
    }
}
```

Php Laravel

3 Basic Example

Let's say you want to run a job every minute that logs something:

```php
protected function schedule(Schedule $schedule)
{
    $schedule->call(function () {
        \Log::info('Task running at: ' . now());
    })->everyMinute();
}
```

✅ This will add a log entry every minute.

---

4 Run the Scheduler

In your terminal, run:

```bash
php artisan schedule:run
```

This will check if any scheduled tasks are due to run **at that moment**.

5 Real Cron Setup (in server)

You don't have to call `schedule:run` manually every minute.
Instead, you set **one cron job** on your server:

```bash
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

✅ This runs every minute, and Laravel handles *all your tasks* inside `Kernel.php`.

6 Common Schedule Methods

Method	Description
`->everyMinute()`	Run every minute
`->hourly()`	Every hour
`->daily()`	Every day at midnight
`->dailyAt('13:00')`	Every day at 1 PM
`->weekly()`	Every week
`->monthly()`	Every month
`->yearly()`	Every year
`->mondays()`	Every Monday
`->fridays()->at('18:00')`	Every Friday at 6 PM

🧱 7️⃣ Run Commands or Jobs Automatically

You can schedule Artisan commands or Jobs too:

🔹 Run Artisan Command

```php
$schedule->command('emails:send')->daily();
```

(You can create custom commands with `php artisan make:command SendEmai`

🔹 Run a Job

```php
use App\Jobs\SendReportJob;

$schedule->job(new SendReportJob)->dailyAt('09:00');
```

🧰 9️⃣ Conditions & Prevent Overlap

You can prevent the same task from running twice (useful if one takes long):

```php
$schedule->command('emails:send')
        ->withoutOverlapping();
```

You can also add conditions:

```php
$schedule->call(function () {
    // Only on weekdays
})->weekdays();
```

📃 Quick Summary

Concept	Meaning	Example
Schedule File	Where you define tasks	`app/Console/Kernel.php`
Command	Runs artisan commands	`$schedule->command('emails:send')`
Callback	Run custom PHP code	`$schedule->call(function() {...})`
Job	Dispatch queued job	`$schedule->job(new MyJob)`
Cron Setup	Runs Laravel scheduler every minute	`php artisan schedule:run`
Frequency	Time interval	`daily()`, `hourly()`, etc.

Example2:

🎯 GOAL

Run a task **every minute** that writes a message to your Laravel log file (`storage/logs/laravel.log`).

🧠 What is Task Scheduling?

Task Scheduling in Laravel allows you to **automate repetitive tasks**, like:

- Sending daily emails
- Cleaning old logs
- Updating reports
- Running database backups

You define these tasks **in code**, and Laravel runs them **automatically** using the `scheduler`.

⚙️ Step 1: Open the Schedule Configuration

If you're using the **new Laravel (v11 or 12)** structure,
you won't have `app/Console/Kernel.php` anymore.
Now, scheduling is done inside `routes/console.php`.

So open:

```bash
routes/console.php
```

📝 Step 2: Add a Simple Scheduled Task

Example — log a message every minute:

```php
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    \Log::info('Scheduler is working! - ' . now());
})->everyMinute();
```

That's it!
Now Laravel will run this closure every minute.

🧩 Step 3: Run the Scheduler

Run this command **once** in the background:

```bash
php artisan schedule:work
```

Or, if you're using Linux cron:

```bash
* * * * * cd /path-to-your-project && php artisan schedule:run >> /dev/null 2>&1
```

✅ This tells your OS to run Laravel's scheduler every minute.

💡 Step 4: Example — Custom Command Scheduling

Let's make a custom command that sends a message.

1️⃣ Create the command:

```bash
php artisan make:command GreetUser
```

2️⃣ In `app/Console/Commands/GreetUser.php`:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class GreetUser extends Command
{
    protected $signature = 'app:greet-user';
    protected $description = 'Greet the user in the log';

    public function handle()
    {
        \Log::info('Hello from Task Scheduler! ' . now());
        $this->info('Greeting logged successfully!');
    }
}
```

3️⃣ Schedule the command (inside `routes/console.php`):

```php
use Illuminate\Support\Facades\Schedule;

Schedule::command('app:greet-user')->hourly();
```

4️⃣ Test it manually:

```bash
php artisan app:greet-user
```

Then check your `storage/logs/laravel.log` —
you'll see:

```yaml
[2025-10-28 15:12:05] local.INFO: Hello from Task Scheduler! 2025-10-28 15:12:05
```

Php Laravel

🧩 Example 1 — Schedule a Controller Method

You **can't directly call** a controller from `Kernel` or `console.php`, but you can trigger an **Artisan command** or a **route call** that points to a controller.

✅ Example using `Artisan::call()`:

```php
use Illuminate\Support\Facades\Artisan;
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    Artisan::call('route:call', [
        'uri' => '/send-daily-report'
    ]);
})->daily();
```

✅ Or simpler — call controller logic directly:

```php
use App\Http\Controllers\ReportController;
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    (new ReportController)->sendDailyReport();
})->daily();
```

🧩 Example 2 — Schedule a Job

Jobs are the **best practice** for scheduling.

1️⃣ Create a Job:

```bash
php artisan make:job SendDailyReportJob
```

2️⃣ Add logic in `app/Jobs/SendDailyReportJob.php`:

```php
namespace App\Jobs;

use App\Models\User;
use App\Notifications\DailyReportNotification;

class SendDailyReportJob extends Job
{
    public function handle()
    {
        foreach (User::all() as $user) {
            $user->notify(new DailyReportNotification());
        }
    }
}
```

3️⃣ Schedule it in `app/Console/Kernel.php` (or `routes/console.php`):

```php
use App\Jobs\SendDailyReportJob;

Schedule::job(new SendDailyReportJob)->dailyAt('09:00');
```

✅ This runs every day at 9 AM and sends notifications to all users.

🧩 Example 3 — Schedule a Notification

You can send a notification directly too:

```php
use App\Models\User;
use App\Notifications\DailyReportNotification;
use Illuminate\Support\Facades\Schedule;

Schedule::call(function () {
    $user = User::first();
    $user->notify(new DailyReportNotification());
})->everyMinute(); // or daily(), hourly(), etc.
```

PHPUnit/Pest • Feature/API tests • Factories in tests • Debugging (telescope/logging) • CI with GitHub Actions (lint+tests)

- **PHPUnit/Pest:**

🧠 1 What is Testing?

> **Testing** means checking that your code works as expected — automatically.

Instead of clicking buttons and checking manually, you write small test scripts that:

- ✅ run your app's code
- ✅ verify results
- ✅ tell you immediately if something breaks after an update

⚙️ 2 What Are PHPUnit & Pest?

Tool	Description
PHPUnit	Default testing framework included with Laravel (older, but powerful).
Pest	A newer, cleaner, and easier testing framework that sits on top of PHPUnit.

👉 Both run your tests from the `tests/` folder.

🧩 3 Folder Structure

When you create a new Laravel project, you'll see:

```kotlin
tests/
 ├── Feature/
 ├── Unit/
phpunit.xml
```

🔹 Feature Tests

These test **whole routes, controllers, or APIs** (e.g., "user can register successfully")

🔹 Unit Tests

These test **individual functions or classes** (e.g., "discount is calculated correctly")

Incase Of error:

🧩 The error

```pgsql
SQLSTATE[HY000] [1049] Unknown database 'testing'
```

🥇 Option 1: Create the `testing` database manually

Open your MySQL client or phpMyAdmin and run:

```sql
CREATE DATABASE testing;
```

Note: The name of the Test File shouls end with "**Test.php**"

By default, PHPUnit (which Laravel uses under the hood) **only runs test files and methods that follow specific naming patterns.**

✅ PHPUnit naming rules

- The **file name must end with** `Test.php`
- The **test method** must start with `test` or have the `#[Test]` attribute.

So your file name should be something like:

```
CalculateTest.php
```

not just `calculate.php`.

🧱 4️⃣ Example: PHPUnit Test

Let's test if the home route works 👇

Create a test:

```bash
php artisan make:test HomePageTest
```

File: `tests/Feature/HomePageTest.php`

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;

class HomePageTest extends TestCase
{
    public function test_home_page_loads_successfully()
    {
        $response = $this->get('/');   // simulate GET request

        $response->assertStatus(200);  // check HTTP 200 OK
    }
}
```

✅ Run the test:

```bash
php artisan test
```

Output:

```arduino
PASS  Tests\Feature\HomePageTest
✓ home page loads successfully
```

⚙️ 5️⃣ Example: Testing a Controller (POST request)

Let's test registration:

```php
public function test_user_can_register()
{
    $response = $this->post('/register', [
        'name' => 'John',
        'email' => 'john@example.com',
        'password' => 'secret123',
        'password_confirmation' => 'secret123'
    ]);

    $response->assertStatus(302); // redirect after success
    $this->assertDatabaseHas('users', ['email' => 'john@example.com']);
}
```

✅ This test checks both:

- HTTP response (redirect)
- Database entry (user created)

6 Example: Unit Test

Create:

```bash
php artisan make:test CalculatorTest --unit
```

File: `tests/Unit/CalculatorTest.php`

```php
<?php

namespace Tests\Unit;

use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase
{
    public function test_addition_works()
    {
        $sum = 2 + 3;
        $this->assertEquals(5, $sum);
    }
}
```

Run:

```bash
php artisan test
```

Output:

```
✓ addition works
```

7 Assertions (Checks)

Common PHPUnit assertions 👇

Assertion	Meaning	Example
`$this->assertEquals(a, b)`	Check equality	`assertEquals(5, 2+3)`
`$this->assertTrue(condition)`	Check true	`assertTrue($user->active)`
`$response->assertStatus(200)`	HTTP 200 OK	
`$response->assertSee('Welcome')`	Text appears in response	
`$this->assertDatabaseHas(table, data)`	DB record exists	

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Models\User;
use Illuminate\Foundation\Testing\RefreshDatabase;

class UserTest extends TestCase
{
    use RefreshDatabase;

    /** @test */
    public function user_can_be_created()
    {
        $user = User::factory()->create([
            'name' => 'Aniket',
            'email' => 'aniket@example.com'
        ]);

        $this->assertDatabaseHas('users', [
            'email' => 'aniket@example.com',
        ]);
    }
}
```

Example 3 — Testing a Controller (Feature Test)

Let's test a route like `/students`.

```php
public function test_students_page_returns_view()
{
    $response = $this->get('/students');
    $response->assertStatus(200);
    $response->assertViewIs('students.index');
}
```

🪄 8️⃣ Pest — Cleaner Version of PHPUnit

Install Pest:

```bash
composer require pestphp/pest --dev
php artisan pest:install
```

Now tests look like this 👇

File: `tests/Feature/HomePageTest.php`

```php
it('loads the home page', function () {
    $this->get('/')->assertStatus(200);
});
```

✅ No class syntax, no `public function` — just plain English!

🧰 9️⃣ Run Tests

Both PHPUnit and Pest use the same command:

```bash
php artisan test
```

Or specifically for Pest:

```bash
./vendor/bin/pest
```

- **Feature/API tests:**

🧠 1️⃣ What Are Feature & API Tests?

In Laravel, tests are divided mainly into two types:

Type	Purpose	Example
Unit Tests	Test small pieces of code (functions, models)	Checking a discount calculation
Feature (API) Tests	Test how multiple parts work together — routes, controllers, middleware, DB, etc.	Testing user registration or login

So when you test an **API endpoint**, it's a **Feature Test** — because it touches your routes, controllers, validation, and database.

🧩 2️⃣ Creating a Feature (API) Test

You can create a new test file with:

```bash
php artisan make:test UserApiTest
```

This creates:

`tests/Feature/UserApiTest.php`

Php Laravel

3 Example 1: Test a GET API

Let's say you have this route in `routes/api.php`:

```php
Route::get('/users', [UserController::class, 'index']);
```

And in `UserController`:

```php
public function index()
{
    return response()->json(User::all());
}
```

Now write a **Feature Test**:

```php
<?php

namespace Tests\Feature;

use Tests\TestCase;
use App\Models\User;

class UserApiTest extends TestCase
{
    public function test_get_users_api_returns_data()
    {
        // create dummy users in the database
        User::factory()->count(3)->create();

        // call the API
        $response = $this->getJson('/api/users');

        // check that the response is successful
        $response->assertStatus(200);

        // check that response has 3 users
        $response->assertJsonCount(3);
    }
}
```

✅ This test:

- creates 3 fake users using a factory
- hits `/api/users`
- verifies HTTP 200 OK
- verifies JSON contains 3 users

4 Example 2: Test POST (User Registration)

If your route is:

```php
Route::post('/register', [AuthController::class, 'register']);
```

and your controller:

```php
public function register(Request $request)
{
    $validated = $request->validate([
        'name' => 'required',
        'email' => 'required|email|unique:users',
        'password' => 'required|min:6|confirmed'
    ]);

    $user = User::create([
        'name' => $validated['name'],
        'email' => $validated['email'],
        'password' => bcrypt($validated['password']),
    ]);

    return response()->json(['user' => $user], 201);
}
```

Test file:

```php
public function test_user_can_register()
{
    $response = $this->postJson('/api/register', [
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'password' => 'secret123',
        'password_confirmation' => 'secret123'
    ]);

    $response->assertStatus(201);
    $this->assertDatabaseHas('users', ['email' => 'john@example.com']);
}
```

✅ Laravel's built-in test helpers (`postJson`, `getJson`, etc.) make it easy to simulate API requests.

❌ 5️⃣ Example 3: Test Validation Error

If you forget to send an email:

```php
public function test_registration_fails_without_email()
{
    $response = $this->postJson('/api/register', [
        'name' => 'John Doe',
        'password' => 'secret123',
        'password_confirmation' => 'secret123'
    ]);

    $response->assertStatus(422);
    $response->assertJsonValidationErrors(['email']);
}
```

🧪 6️⃣ Useful Test Helpers

Helper	Description
`$this->get('/api/users')`	Send GET request
`$this->post('/api/register', $data)`	Send POST request
`$this->getJson()` / `$this->postJson()`	Send JSON requests
`$response->assertStatus(200)`	Assert HTTP response code
`$response->assertJson([...])`	Match JSON structure
`$this->assertDatabaseHas('table', ['field'=>'value'])`	Check DB
`$this->assertDatabaseMissing()`	Ensure data doesn't exist

⚡ 7️⃣ Running Tests

Run all tests:

```bash
php artisan test
```

Run specific test file:

```bash
php artisan test --filter=UserApiTest
```

Run specific method:

```bash
php artisan test --filter=test_user_can_register
```

- **Factories in tests:**

🧠 1️⃣ What are Factories?

A **factory** in Laravel lets you quickly create **dummy data** (users, posts, etc.) for testing or seeding. It uses the **Faker** library behind the scenes to generate random names, emails, titles, etc.

So instead of writing:

```php
User::create([
    'name' => 'John Doe',
    'email' => 'john@example.com',
    'password' => bcrypt('secret123'),
]);
```

You can just do:

```php
User::factory()->create();
```

✅ Laravel automatically fills all fields with **realistic fake data**.

Php Laravel

🧩 2️⃣ Where Are Factories Stored?

Factories live in

```bash
database/factories/
```

Example:

`database/factories/UserFactory.php`

⚙️ 3️⃣ Example: User Factory

When you create a new model with a factory:

```bash
php artisan make:model User -mf
```

> The `-mf` flag = **migration + factory.**

Now open the generated file:

`database/factories/UserFactory.php`

Example:

```php
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    public function definition()
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => bcrypt('password'),
            'remember_token' => Str::random(10),
        ];
    }
}
```

🧪 4️⃣ Using Factories in Tests

Example test:

`tests/Feature/UserApiTest.php`

```php
use Tests\TestCase;
use App\Models\User;

class UserApiTest extends TestCase
{
    public function test_get_all_users()
    {
        // create 5 fake users in DB
        User::factory()->count(5)->create();

        // call the API
        $response = $this->getJson('/api/users');

        // assert API returned 5 users
        $response->assertStatus(200);
        $response->assertJsonCount(5);
    }
}
```

🔢 5️⃣ Factory Methods Cheat Sheet

Method	What It Does	Example
`User::factory()->make()`	Create an object (not saved)	Use for testing logic
`User::factory()->create()`	Create + save in DB	Use in feature tests
`User::factory()->count(3)->create()`	Create multiple	3 users
`User::factory()->create(['name' => 'Aniket'])`	Override attributes	Custom name
`User::factory()->hasPosts(5)`	Create related models	1 user + 5 posts

6 Example with Relationships

Let's say you have a **Post** model related to **User**:

```php
// PostFactory.php
class PostFactory extends Factory
{
    public function definition()
    {
        return [
            'title' => fake()->sentence(),
            'body' => fake()->paragraph(),
            'user_id' => User::factory(), // automatically create a user
        ];
    }
}
```

In a test:

```php
public function test_user_posts()
{
    $post = Post::factory()->create();

    $this->assertDatabaseHas('posts', [
        'id' => $post->id,
        'user_id' => $post->user_id,
    ]);
}
```

✅ This creates both a post and its author automatically.

- **Debugging (telescope/logging):**

1 What Is Debugging?

Debugging means **finding and fixing errors** in your Laravel app.
Laravel provides two main ways to debug:

Tool	Purpose
Logging	Record messages in `storage/logs/laravel.log`
Telescope	A powerful dashboard to watch requests, jobs, queries, mail, etc. live in your browser

2 Laravel Logging (Simple & Always Available)

✅ Example 1: Write to Log

Use the **Log facade**:

```php
use Illuminate\Support\Facades\Log;

Log::info('User registered successfully');
Log::warning('Something looks odd...');
Log::error('Failed to send email!');
```

These go into:

```pgsql
storage/logs/laravel.log
```

Open it and you'll see:

```pgsql
[2025-10-22 14:00:00] local.INFO: User registered successfully
[2025-10-22 14:01:00] local.ERROR: Failed to send email!
```

✅ Example 2: Log With Context (Extra Info)

```php
Log::info('New user registered', ['email' => $user->email]);
```

Log output:

```pgsql
[2025-10-22] local.INFO: New user registered {"email":"john@example.com"}
```

✅ Example 3: Log Errors in `try...catch`

```php
try {
    $user = User::findOrFail($id);
} catch (\Exception $e) {
    Log::error('User not found', ['error' => $e->getMessage()]);
}
```

3 Logging Levels

Laravel supports PSR-3 log levels:

Level	Use for
`emergency`	App totally unusable
`alert`	Must fix immediately
`critical`	Critical conditions
`error`	Runtime errors
`warning`	Non-critical issue
`notice`	Normal but important
`info`	General info
`debug`	Detailed debug info

Excellent question — this line in your `.env` file: 👇

```env
LOG_CHANNEL=stack
LOG_LEVEL=debug
```

directly controls **how and where Laravel logs messages**, and **how detailed those logs are**.
Let's break down both parts clearly 👇

🧩 1. `LOG_CHANNEL=stack`

➤ **What is a "log channel"?**

A **log channel** tells Laravel *where to write logs*.
Examples of channels:

- `single` → writes all logs to one file (`storage/logs/laravel.log`)
- `daily` → creates a new log file every day
- `slack` → sends logs to a Slack channel
- `syslog`, `errorlog` → system-level logs
- `stack` → combines multiple channels (for example, log to both file **and** Slack)

Php Laravel

So:

```env
LOG_LEVEL=debug
```

means Laravel will log **everything** — from errors to small debugging details.

If you change it to:

```env
LOG_LEVEL=error
```

then only serious errors (error, critical, alert, emergency) will appear — no debug/info/warning logs.

5 Laravel Telescope — Developer Dashboard

Laravel Telescope is a real-time monitoring tool.
It shows everything Laravel does under the hood — requests, queries, exceptions, jobs, mails, logs, etc.

Install Telescope

```bash
composer require laravel/telescope
```

Publish and migrate:

```bash
php artisan telescope:install
php artisan migrate
```

Run your app, then open:

```arduino
http://127.0.0.1:8000/telescope
```

6 What You Can See in Telescope

Section	Description
Requests	All HTTP requests (method, route, response time)
Commands	Artisan commands run
Jobs	Queued jobs executed
Exceptions	Errors and stack traces
Logs	All `Log::info`, `Log::error`, etc.
Queries	SQL queries executed
Mail	Sent emails (content preview!)
Notifications	Sent database/email notifications
Cache	Cache hits/misses
Events	Dispatched events

Example:

- When you open a page, Telescope records the route, controller, queries, and response time.
- If something breaks, you can open "Exceptions" and see the full trace.

Php Laravel

🔒 7️⃣ Restrict Telescope Access (for security)

By default, Telescope is only available in `local` environment.

If you need it elsewhere, edit:

`app/Providers/TelescopeServiceProvider.php`

```php
protected function gate()
{
    Gate::define('viewTelescope', function ($user) {
        return in_array($user->email, [
            'admin@example.com',
        ]);
    });
}
```

🧰 8️⃣ Real-World Example

```php
public function store(Request $request)
{
    Log::info('Creating new post', ['title' => $request->title]);

    try {
        $post = Post::create($request->all());
        Log::info('Post created successfully', ['id' => $post->id]);
    } catch (\Exception $e) {
        Log::error('Failed to create post', ['error' => $e->getMessage()]);
    }

    return response()->json(['message' => 'Post processed']);
}
```

Then in Telescope, you'll see:

- Request → `/api/posts`
- Logs → info + error messages
- Database → insert query executed
- Time taken, etc.

📋 9️⃣ Summary

Tool	What It Does	Where to See
Log Facade	Write info/errors manually	`storage/logs/laravel.log`
Telescope	Visual dashboard of all app activity	`/telescope`
log levels	Control severity (`debug`, `error`, etc.)	`.env → LOG_LEVEL`
tail logs	View live log updates	Terminal command

💬 In Simple Words

> **Logging** = record what's happening.
> **Telescope** = watch it live in a dashboard.
> Together, they make finding and fixing bugs 10× easier.

- CI with GitHub Actions (lint+tests):

Php Laravel

🧠 1️⃣ What is CI (Continuous Integration)?

Continuous Integration (CI) means that every time you push code to GitHub:

- Your code is automatically **checked**, **tested**, and **validated**.
- This ensures your app always stays bug-free and consistent.

In Laravel, the most common CI tasks are:

✅ **Linting** (code style check)
✅ **Testing** (run PHPUnit or Pest tests)

And GitHub Actions is the tool that runs these automatically on GitHub's servers.

🧩 2️⃣ Basic Idea

Whenever you push or make a pull request →
GitHub runs a workflow file `.github/workflows/tests.yml`

That workflow will:

1. Install PHP and dependencies
2. Set up your Laravel environment
3. Run tests automatically
4. Report success/failure ✅❌

⚙️ 3️⃣ Create Workflow File

Create a new file in your repo:

bash

```bash
.github/workflows/laravel-ci.yml
```

.github > workflows > laravel-ci.yml

```yaml
name: Laravel CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  laravel-tests:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_ROOT_PASSWORD: root
          MYSQL_DATABASE: laravel
        ports:
          - 3306:3306
        options: >-
          --health-cmd="mysqladmin ping -h localhost"
          --health-interval=10s
          --health-timeout=5s
          --health-retries=3

    steps:
      - name: 📥 Checkout code
        uses: actions/checkout@v4

      - name: 🧱 Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.2'
          extensions: mbstring, bcmath, intl, pdo_mysql
          coverage: none

      - name: 📦 Install dependencies
        run: composer install --prefer-dist --no-interaction

      - name: 🔧 Copy .env
        run: cp .env.example .env

      - name: 🔑 Generate app key
        run: php artisan key:generate

      - name: 🗃️ Run migrations
        env:
          DB_CONNECTION: mysql
          DB_DATABASE: laravel
          DB_USERNAME: root
          DB_PASSWORD: root
          DB_HOST: 127.0.0.1
        run: php artisan migrate --force

      - name: ✅ Run Tests
        env:
          DB_CONNECTION: mysql
          DB_DATABASE: laravel
          DB_USERNAME: root
          DB_PASSWORD: root
          DB_HOST: 127.0.0.1
        run: php artisan test

        # 6️⃣ Run lint (Laravel Pint)
      - name: Run Laravel Pint
        run: vendor/bin/pint --test
```

🧪 4️⃣ What This Does

Step	What Happens
`Checkout code`	Downloads your repo
`Setup PHP`	Installs PHP 8.2
`Install dependencies`	Runs `composer install`
`Copy .env`	Creates `.env` file
`Generate app key`	Needed for Laravel
`Run migrations`	Sets up a fresh DB
`Run tests`	Executes PHPUnit/Pest tests

🧹 4. Optional — Add Laravel Pint (Code Linter)

Install it locally:

```bash
composer require laravel/pint --dev
```

Check code style manually:

```bash
vendor/bin/pint --test
```

Or auto-fix:

```bash
vendor/bin/pint
```

🧰 6️⃣ Example PHPUnit Test

`tests/Feature/ExampleTest.php`

```php
public function test_homepage_returns_ok()
{
    $response = $this->get('/');
    $response->assertStatus(200);
}
```

When you push, GitHub will:

- Set up Laravel
- Migrate DB
- Run this test

✅ If OK → CI passes
❌ If not → CI fails

Php Laravel

7️⃣ View Results on GitHub

Go to your repo → **Actions tab**

You'll see:

- ✅ Green (Passed)
- ❌ Red (Failed)
- 🟡 Running (In progress)

Click to view detailed logs — you can see migrations, artisan commands, test output, etc.

Explanation:

⚙️ Workflow Triggers

```yaml
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
```

- **What it does:**
  Defines **when** this workflow should run.
- It runs automatically:
  - Whenever someone **pushes** new commits to the `main` branch.
  - Whenever someone **opens a pull request** targeting the `main` branch.
- So this ensures all code merged into main passes tests.

🧱 Job Definition

```yaml
jobs:
  laravel-tests:
    runs-on: ubuntu-latest
```

- `jobs:` — defines one or more jobs GitHub Actions will run.
- `laravel-tests:` — name of the job (you can have multiple jobs if needed).
- `runs-on: ubuntu-latest` —
  This tells GitHub Actions to use a **fresh virtual machine** with **Ubuntu Linux** as the operating system.
  It will install PHP, Composer, MySQL, etc., inside this environment.

🧰 Services Section (for MySQL)

```yaml
services:
  mysql:
    image: mysql:8.0
    env:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: laravel
    ports:
      - 3306:3306
    options: >-
      --health-cmd="mysqladmin ping -h localhost"
      --health-interval=10s
      --health-timeout=5s
      --health-retries=3
```

Let's break this down:

- **Purpose:**
  Sets up a **MySQL database service** running **inside a Docker container** for your tests to use.

Php Laravel

Line by line:

- `mysql:` — the name of this service (you can reference it as a hostname like `mysql` or `127.0.0.1`).
- `image: mysql:8.0` — tells GitHub to use the official **MySQL 8.0 Docker image**.
- `env:` — environment variables passed to that container.
  - `MYSQL_ROOT_PASSWORD: root` → sets MySQL root password.
  - `MYSQL_DATABASE: laravel` → automatically creates a database named "laravel".
- `ports: - 3306:3306` → maps MySQL's internal port to the same port on the runner, so your Laravel app can connect.
- `options:` → health check configuration:
  - `--health-cmd="mysqladmin ping -h localhost"` checks if MySQL is ready.
  - `--health-interval=10s` → run that check every 10 seconds.
  - `--health-timeout=5s` → fail check if it takes longer than 5s.
  - `--health-retries=3` → consider it unhealthy after 3 failed checks.

✅ This ensures your workflow **waits until MySQL is fully up** before running tests.

📋 Steps Section

This is the **heart of the CI job** — a series of actions and shell commands that set up, configure, and test your Laravel app.

1️⃣ Checkout your code

yaml Copy code

```yaml
    - name: 📥 Checkout code
      uses: actions/checkout@v4
```

- **What it does:**
  Downloads your repository's code into the GitHub runner (so it can run commands on your files).
- `actions/checkout@v4` is an official GitHub Action for that.

2️⃣ Setup PHP environment

yaml

```yaml
    - name: 🧱 Setup PHP
      uses: shivammathur/setup-php@v2
      with:
        php-version: '8.2'
        extensions: mbstring, bcmath, intl, pdo_mysql
        coverage: none
```

- **What it does:**
  Installs **PHP 8.2** and any needed extensions on the Ubuntu runner.

Explanation of each part:

- `uses: shivammathur/setup-php@v2` → a popular GitHub Action that installs PHP.
- `php-version: '8.2'` → specifies PHP version.
- `extensions:` → lists PHP extensions required by Laravel:
  - `mbstring` → used for string manipulation (multibyte).
  - `bcmath` → used for precise math operations.
  - `intl` → used for date/locale formatting.
  - `pdo_mysql` → enables PHP to connect to MySQL using PDO.
- `coverage: none` → disables code coverage tools (speeds up tests).

3 Install Composer dependencies

```yaml
    - name: 📦 Install dependencies
      run: composer install --prefer-dist --no-interaction
```

- **What it does:**
  Runs `composer install` to install all required PHP packages (from `composer.json`).
- `--prefer-dist` → downloads packages as zipped distributions (faster).
- `--no-interaction` → runs non-interactively (no prompts).

4 Copy .env file

```yaml
    - name: 🔧 Copy .env
      run: cp .env.example .env
```

- **What it does:**
  Creates a `.env` file from `.env.example`.
- The `.env` file contains Laravel's configuration (like DB credentials, mail, cache, etc.)
- Since this is a fresh environment, it doesn't exist — so this step is essential.

5 Generate Laravel app key

```yaml
    - name: 🔑 Generate app key
      run: php artisan key:generate
```

- **What it does:**
  Runs the Artisan command to generate a random `APP_KEY` inside `.env`.
- This key is used by Laravel for encryption (sessions, password resets, etc.)

Php Laravel

6 Run Migrations

```yaml
    - name: 🗃️ Run migrations
      env:
        DB_CONNECTION: mysql
        DB_DATABASE: laravel
        DB_USERNAME: root
        DB_PASSWORD: root
        DB_HOST: 127.0.0.1
      run: php artisan migrate --force
```

- **What it does:**
  Runs all your **database migrations** to create tables for testing.

Breakdown:

- The `env:` block temporarily sets database environment variables for this step.
  - So Laravel knows how to connect to the MySQL container.
- `php artisan migrate --force` runs migrations without confirmation prompts.
- The database `laravel` is created earlier in the MySQL container setup.

✅ This prepares a clean database for your tests.

7 Run Tests

```yaml
    - name: ✅ Run Tests
      env:
        DB_CONNECTION: mysql
        DB_DATABASE: laravel
        DB_USERNAME: root
        DB_PASSWORD: root
        DB_HOST: 127.0.0.1
      run: php artisan test
```

- **What it does:**
  Runs your **PHPUnit / Pest** test suite using Laravel's built-in test runner (`php artisan test`).
- The same database credentials are passed here.
- If tests fail, this step (and thus the whole workflow) will fail ❌.

✅ This is where your actual CI validation happens.

📄 Summary

Step	Purpose
`on:`	Defines when to trigger CI (on push/pull to main).
`runs-on:`	Sets up Ubuntu VM.
`services:`	Launches a MySQL container for the database.
`checkout`	Downloads your repo's code.
`setup-php`	Installs PHP 8.2 with necessary extensions.
`composer install`	Installs Laravel dependencies.
`cp .env.example .env`	Creates environment config.
`php artisan key:generate`	Sets Laravel encryption key.
`php artisan migrate`	Builds database schema.
`php artisan test`	Runs automated tests.

- Protect API with Sanctum with example:

🧱 **Example: "User + Posts API" protected by Sanctum**

⚙️ **1. Install Sanctum**

If not already installed:

```bash
composer require laravel/sanctum
```

Then publish and migrate:

```bash
php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"
php artisan migrate
```

🧩 **2. Enable Sanctum Middleware**

Open `app/Http/Kernel.php`, and ensure this inside your `api` **middleware group**:

```php
'api' => [
    \Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
    'throttle:api',
    \Illuminate\Routing\Middleware\SubstituteBindings::class,
],
```

👤 **3. Add Sanctum Trait in User Model**

In `app/Models/User.php`:

```php
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
}
```

Php Laravel

🔑 4. Create Authentication Routes (Token-based)

In `routes/api.php`:

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Models\User;

// Register
Route::post('/register', function (Request $request) {
    $validated = $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|unique:users',
        'password' => 'required|string|min:8',
    ]);

    $user = User::create([
        'name' => $validated['name'],
        'email' => $validated['email'],
        'password' => bcrypt($validated['password']),
    ]);

    $token = $user->createToken('api_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'token' => $token,
    ]);
});

// Login
Route::post('/login', function (Request $request) {
    $user = User::where('email', $request->email)->first();

    if (! $user || ! Hash::check($request->password, $user->password)) {
        return response()->json(['message' => 'Invalid credentials'], 401);
    }

    $token = $user->createToken('api_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'token' => $token,
    ]);
});
```

🧱 5. Protect Routes with `auth:sanctum`

Now, any route inside this middleware group requires a **valid Sanctum token**.

```php
Route::middleware('auth:sanctum')->group(function () {
    Route::get('/user', function (Request $request) {
        return $request->user();
    });

    Route::get('/posts', function () {
        return [
            ['title' => 'First Post', 'body' => 'Hello Sanctum!'],
            ['title' => 'Second Post', 'body' => 'Protected route demo'],
        ];
    });

    Route::post('/logout', function (Request $request) {
        $request->user()->currentAccessToken()->delete();
        return response()->json(['message' => 'Logged out']);
    });
});
```

🧪 6. Test Using `curl` or Postman

🔹 **Step 1 — Register**

```bash
curl -X POST http://127.0.0.1:8000/api/register ^
-H "Accept: application/json" ^
-H "Content-Type: application/json" ^
-d "{\"name\":\"Aniket\",\"email\":\"aniket@example.com\",\"password\":\"11111111\"}"
```

✅ Response:

```json
{
  "user": {
    "id": 1,
    "name": "Aniket",
    "email": "aniket@example.com"
  },
  "token": "1|AbCdEfGhIjKlMnOpQrSt..."
}
```

🔹 **Step 2 — Access Protected Route**

Copy the token and send it as a **Bearer token**:

```bash
curl -H "Authorization: Bearer 1|AbCdEfGhIjKlMnOpQrSt..." ^
http://127.0.0.1:8000/api/posts
```

✅ Response:

```json
[
  { "title": "First Post", "body": "Hello Sanctum!" },
  { "title": "Second Post", "body": "Protected route demo" }
]
```

⚡ 7. How Sanctum Protects It

- Each token is tied to a specific user.
- Laravel automatically checks for the Bearer token in the `Authorization` header.
- The middleware `auth:sanctum` ensures that only valid tokens can access the route.
- Revoked tokens can't be reused.

Step 2 — Create Post

Copy your token, then run:

```bash
curl -X POST http://127.0.0.1:8000/api/posts ^
-H "Accept: application/json" ^
-H "Authorization: Bearer 1|AbCdEfGhIjKlMn..." ^
-H "Content-Type: application/json" ^
-d "{\"title\":\"My First Post\",\"body\":\"This is a protected post example.\"}"
```

✅ Response:

```json
{
  "message": "Post created successfully",
  "post": {
    "id": 1,
    "user_id": 1,
    "title": "My First Post",
    "body": "This is a protected post example.",
    "created_at": "2025-10-30T10:45:21.000000Z"
  }
}
```

- **Soft Delete:**

In Laravel, **Soft Deletes** allow you to "delete" records **without actually removing them from the database.**
Instead of being deleted, a timestamp is added to the deleted_at column — so you can restore or query them later.

🧱 Step 1: Add `deleted_at` Column to Your Table

In your **migration file**, add a `softDeletes()` column:

```php
Schema::table('users', function (Blueprint $table) {
    $table->softDeletes(); // adds a 'deleted_at' TIMESTAMP nullable column
});
```

Then run:

```bash
php artisan migrate
```

⚙️ Step 2: Add the `SoftDeletes` Trait in Your Model

In your Eloquent model (e.g., `User.php`):

```php
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class User extends Model
{
    use SoftDeletes;

    protected $dates = ['deleted_at']; // (optional in Laravel 9+)
}
```

Php Laravel

🧩 Step 3: Use Soft Delete in Code

➤ Soft delete a record:

```php
$user = User::find(1);
$user->delete(); // sets deleted_at timestamp
```

➤ Get only non-deleted records (default):

```php
$users = User::all(); // automatically excludes soft-deleted ones
```

➤ Get *only* soft-deleted records:

```php
$trashed = User::onlyTrashed()->get();
```

➤ Get *all* records including soft-deleted:

```php
$all = User::withTrashed()->get();
```

➤ Restore a soft-deleted record:

```php
$user = User::withTrashed()->find(1);
$user->restore(); // sets deleted_at to null
```

➤ Permanently delete a record:

```php
$user = User::onlyTrashed()->find(1);
$user->forceDelete(); // actually removes it from DB
```

Login:

how can i make sure that if once login if i visit login page again it should redirect me to '/myacccount' page

✅ Option 2: Use a middleware (cleaner, reusable way)

You can use Laravel's built-in `guest` middleware to restrict routes to only non-authenticated users.

Example in your `routes/web.php`:

```php
Route::middleware('guest')->group(function () {
    Route::get('/accounts/login', [LoginController::class, 'showLoginForm'])->name('login');
    Route::post('/accounts/login', [LoginController::class, 'login']);
});
```

Then your "my account" routes can use the opposite:

```php
Route::middleware('auth')->group(function () {
    Route::get('/myaccount', [AccountController::class, 'index']);
});
```

How it works:

- The `guest` middleware automatically redirects authenticated users to `/home` by default.
- You can customize that redirect path easily by setting it in your `LoginController`:

```php
public function __construct()
{
    $this->middleware('guest')->except('logout');
}

protected $redirectTo = '/myaccount';
```

or globally in `app/Http/Middleware/RedirectIfAuthenticated.php`:

```php
public function handle($request, Closure $next, ...$guards)
{
    if (Auth::check()) {
        return redirect('/myaccount'); // customize destination
    }

    return $next($request);
}
```

Login with remember me:

```html
<input type="checkbox" name="remember"> Remember Me
```

Then modify your login code like this:

```php
public function login(Request $request)
{
    $credentials = $request->only('email', 'password');
    $remember = $request->has('remember'); // true if checkbox is checked

    if (!auth()->attempt($credentials, $remember)) {
        return back()->withErrors([
            'message' => 'The Email ID or password is incorrect, please try again.'
        ]);
    }

    return redirect()->to('/myaccount');
}
```