**Slide 1: Title**

**Applying SOLID & SRP in Laravel**  
*An overview of structuring code for maintainability and testability.*

**Slide 2: What is SRP?**

* **Definition:**  
  A class should have only one reason to change.
* **Goal:**  
  Enhance maintainability, testability, and clarity.
* **Context:**  
  We applied SRP to a simple "Match Simulation" program that involves:
  + **Attack**
  + **Defense**
  + **Keeper (Goalkeeping)**

**Slide 3: Example That Breaks SRP**

* **Single Class Approach:**  
  All logic (attack, defense, keeper) bundled into one class.
* **File:** app/Models/Match.php
* **Issues:**
  + Multiple reasons to change
  + Harder to maintain and test
* **Code Snippet:**

php

CopyEdit

class Match {

public function simulate() {

$attackResult = $this->attack();

$defenseResult = $this->defense();

$keeperResult = $this->keeper();

return compact('attackResult', 'defenseResult', 'keeperResult');

}

protected function attack() { return "Attack executed!"; }

protected function defense() { return "Defense executed!"; }

protected function keeper() { return "Keeper saved the ball!"; }

}

**Slide 4: Example That Follows SRP**

* **Separated Responsibilities:**
  + **AttackHandler, DefenseHandler, KeeperHandler** each in their own file.
* **Central Orchestration:**  
  Match class coordinates the handlers.
* **Benefits:**
  + Each class has one responsibility.
  + Easier to maintain, test, and extend.
* **Code Structure:**

php

CopyEdit

// In AttackHandler.php

class AttackHandler {

public function executeAttack() { return "Attack executed!"; }

}

// Similar for DefenseHandler and KeeperHandler.

// In Match.php

class Match {

public function \_\_construct(AttackHandler $attack, DefenseHandler $defense, KeeperHandler $keeper) { ... }

public function simulate() {

return [

'attack' => $this->attack->executeAttack(),

'defense' => $this->defense->executeDefense(),

'keeper' => $this->keeper->executeSave(),

];

}

}

**Slide 5: Recommended Folder Structure**

CopyEdit

app/

├── Http/

│ └── Controllers/

│ └── MatchController.php

├── MatchSimulation/

│ ├── Handlers/

│ │ ├── AttackHandler.php

│ │ ├── DefenseHandler.php

│ │ └── KeeperHandler.php

│ └── Match.php

* **Controllers:**  
  Handle HTTP requests and responses.
* **Handlers (Services):**  
  Encapsulate specific business logic.
* **Match Class:**  
  Orchestrates simulation using the handlers.

**Slide 6: Understanding Controllers, Models & Handlers**

* **Controllers:**  
  Entry points for HTTP requests. They should be lean.
* **Models:**  
  Represent data and interact with the database.
* **Handlers (or Service Classes):**
  + **Purpose:** Encapsulate business logic (e.g., attack, defense, keeper logic).
  + **Benefits:**
    - Promotes Separation of Concerns
    - Enhances reusability and testability
    - Keeps controllers and models focused on their roles

**Slide 7: What If Handlers Become Controllers?**

* **Scenario:** Renaming or merging handlers as controllers.
* **Implications:**
  + **SRP Compliance:**  
    Technically, if each controller still does one thing, SRP may not be violated.
  + **Best Practices:**
    - Mixing business logic in controllers couples HTTP handling with core logic.
    - Leads to bloated controllers over time.
* **Conclusion:**  
  Although it might “work,” keeping business logic in dedicated handler classes is the cleaner, more maintainable approach.

**Slide 8: Q&A Summary**

**Q1: Why are handlers not controllers?**  
**A:** Handlers encapsulate business logic and are independent of HTTP concerns, keeping controllers lean and focused on request/response.

**Q2: What exactly are handlers?**  
**A:** They are service classes that perform specific business tasks (e.g., executing an attack in a match simulation). They improve modularity, reusability, and testability.

**Q3: If I rename handlers as controllers, does that validate SRP?**  
**A:** Renaming alone doesn’t break SRP if responsibilities remain isolated. However, merging business logic with HTTP handling is not best practice, as it can lead to bloated controllers and mixed concerns.

**Slide 9: Takeaways**

* **Adhere to SRP:**  
  Each class should have a single responsibility.
* **Separate Concerns:**  
  Use controllers for HTTP handling, models for data, and handlers for business logic.
* **Maintainability & Testability:**  
  Cleaner separation leads to code that is easier to test, extend, and maintain.
* **Best Practices:**  
  Keeping dedicated service classes (handlers) preserves the architectural integrity of your application.