# Refining District CRUD with SOLID Principles

This document outlines the step-by-step refinement of the District CRUD implementation by applying SOLID principles. Each step compares the 'Before' and 'After' states to illustrate improvements while keeping existing routes unchanged.

## Step 1: Move Database Logic to a Repository (Single Responsibility Principle - SRP)

🔴 Before: The `DistrictController` handles both request processing and database logic.

public function StoreDistrict(Request $request){  
 $validatedData = $request->validate([  
 'district\_en' => 'required|unique:districts|max:255',  
 'district\_hin' => 'required|unique:districts|max:255',  
 ]);  
  
 $data = array();  
 $data['district\_en'] = $request->district\_en;  
 $data['district\_hin'] = $request->district\_hin;  
 DB::table('districts')->insert($data);  
  
 return Redirect()->route('district')->with(['message' => 'District Inserted Successfully', 'alert-type' => 'success']);  
}

⚠️ Issues:

* - \*\*Controller does too much\*\* → It should only handle HTTP requests, not database logic.
* - \*\*Code duplication\*\* → If another controller needs to manage districts, it will have to rewrite this logic.

✅ After: Extract database logic into `DistrictRepository.php` and use it in the controller.

// app/Repositories/DistrictRepository.php  
class DistrictRepository implements DistrictRepositoryInterface  
{  
 public function store(array $data)  
 {  
 return DB::table('districts')->insert($data);  
 }  
}

Now, update `DistrictController.php` to use the repository:

public function StoreDistrict(Request $request){  
 $validatedData = $request->validate([  
 'district\_en' => 'required|unique:districts|max:255',  
 'district\_hin' => 'required|unique:districts|max:255',  
 ]);  
  
 $this->districtRepository->store([  
 'district\_en' => $request->district\_en,  
 'district\_hin' => $request->district\_hin,  
 ]);  
  
 return Redirect()->route('district')->with(['message' => 'District Inserted Successfully', 'alert-type' => 'success']);  
}

## Step 2: Use a Service Layer (Open/Closed Principle & Dependency Inversion Principle)

🔴 Before: The controller interacts directly with the repository.

✅ After: Introduce `DistrictService.php` to manage business logic.

// app/Services/DistrictService.php  
class DistrictService implements DistrictServiceInterface  
{  
 protected $districtRepository;  
  
 public function \_\_construct(DistrictRepositoryInterface $districtRepository)  
 {  
 $this->districtRepository = $districtRepository;  
 }  
  
 public function storeDistrict(array $data)  
 {  
 return $this->districtRepository->store($data);  
 }  
}

Now, `DistrictController.php` calls `DistrictService` instead of directly using the repository:

public function StoreDistrict(Request $request){  
 $validatedData = $request->validate([  
 'district\_en' => 'required|unique:districts|max:255',  
 'district\_hin' => 'required|unique:districts|max:255',  
 ]);  
  
 $this->districtService->storeDistrict([  
 'district\_en' => $request->district\_en,  
 'district\_hin' => $request->district\_hin,  
 ]);  
  
 return Redirect()->route('district')->with(['message' => 'District Inserted Successfully', 'alert-type' => 'success']);  
}

## Step 3: Enforce Interface Usage (Liskov Substitution Principle - LSP)

Ensure that `DistrictRepository` and `DistrictService` implement their respective interfaces:

✅ Define interfaces for repository and service:

// app/Repositories/DistrictRepositoryInterface.php  
interface DistrictRepositoryInterface  
{  
 public function store(array $data);  
}

// app/Services/DistrictServiceInterface.php  
interface DistrictServiceInterface  
{  
 public function storeDistrict(array $data);  
}

Now, bind interfaces in `RepositoryServiceProvider.php` for Dependency Injection:

$this->app->bind(DistrictRepositoryInterface::class, DistrictRepository::class);  
$this->app->bind(DistrictServiceInterface::class, DistrictService::class);

## Conclusion

By following these steps, we have improved the District CRUD by separating concerns, using service and repository layers, enforcing interface usage, and applying SOLID principles efficiently.