## Problems Found

### 1. Form Action Mismatch in Delete Confirmation

**Assigned to: Miguel**

**Issue Description:** The delete confirmation form in Delete.cshtml had an incorrect action attribute that didn't match the controller action method name.

**SOLID** **Principle Violation:** While not directly a SOLID violation, this represents poor consistency and could lead to runtime errors.

**Impact:**

* Delete functionality would fail
* Poor user experience
* Potential security issues

### 2. Booking Controller Not Using Repository Pattern

**Assigned to: Thomas**

**Issue Description:** The BookingController was directly using ApplicationDbContext instead of utilizing the repository pattern through dependency injection.

**SOLID** **Principle Violations:**

* **Dependency Inversion Principle (DIP):** High-level modules (Controller) depending on low-level modules (DbContext) instead of abstractions
* **Single Responsibility Principle (SRP):** Controller handling both web concerns and data access logic

### 3. Repository Not Implementing Interface

**Assigned to: Heraj**

**Issue Description:** The BookingRepository class was not implementing an interface, making it difficult to create unit tests and violating dependency inversion principles.

**SOLID** **Principle Violations:**

* **Dependency Inversion Principle (DIP):** No abstraction layer between high-level and low-level modules
* **Open/Closed Principle (OCP):** Difficult to extend or modify behavior without changing existing code

**Impact:**

* Tight coupling between components
* Difficult to unit test
* Poor maintainability and extensibility

## Fixes Applied

### 1. Delete Form Action Fix

**Implemented by: Miguel**

**Solution:** Updated the form action in Delete.cshtml to correctly reference the controller action method.

**Changes Made:** Updated the form action attribute from "DeleteConfirmed" to "Delete" to match the controller action method name.

**Result:**

* Delete functionality now works correctly
* Consistent naming convention followed
* Improved user experience

### 2. Implement Repository Pattern in Controller

**Implemented by: Thomas**

**Solution:** Modified the BookingController to use dependency injection with the repository pattern instead of directly accessing the DbContext.

**Changes Made:** Modified the BookingController to use dependency injection with the repository pattern instead of directly accessing the DbContext. The controller now receives IBookingService through its constructor and uses repository methods for all data operations.

**Benefits:**

* Follows Dependency Inversion Principle
* Improved testability
* Better separation of concerns
* Reduced coupling

### 3. Create Repository Interface and Implementation

**Implemented by: Heraj**

**Solution:** Created IBookingRepository interface and ensured BookingRepository implements it properly.

**Interface Created:** The interface defines standard CRUD operations including Add, Update, Delete, GetById and GetAll methods.

**Implementation:** The BookingRepository class implements all interface methods and receives ApplicationDbContext through dependency injection.

**Dependency Injection Registration:** The interface and implementation were registered in Program.cs using AddScoped to ensure proper dependency injection throughout the application.

**Benefits:**

* Enables proper dependency injection
* Supports unit testing with mocking
* Follows SOLID principles
* Improves code maintainability

## Testing Strategy

We implemented a comprehensive three-layer testing strategy covering:

1. **Model Layer Testing**
2. **Service/Repository Layer Testing**
3. **Controller Layer Testing**

### 1. Model Layer Tests

**Implemented by: Eishi**

**Purpose:** Validate that model properties respect data annotations and business rules.

**Test Examples:**

* Required field validation
* Date range validation
* String length validation
* Custom validation attributes

**Sample Test Approach:** Tests validate that model properties respect data annotations by creating model instances with invalid data and checking that validation fails appropriately.

### 2. Repository Layer Tests

**Implemented by: Oscar**

**Purpose:** Test CRUD operations and data access logic using mocked dependencies.

**Testing Framework:** NUnit with Moq for mocking

**Test Categories:**

* Create operations
* Read operations (single and multiple)
* Update operations
* Delete operations
* Error handling scenarios

**Sample Test Approach:** Tests use in-memory databases or mocked dependencies to verify CRUD operations work correctly. Each test creates a repository instance, performs an operation, and validates the expected outcome.

### 3. Controller Layer Tests

**Implemented by: Miguel**

**Purpose:** Test HTTP responses, model validation, and proper view/redirect returns.

**Test Categories:**

* Action method returns types
* Model binding and validation
* Redirect behaviors
* View data validation
* Error handling

**Sample Test Approach:** Tests use mocked repository dependencies to isolate controller logic. Tests verify that controllers return appropriate ViewResults, RedirectResults, or error responses based on different scenarios.