Core Layer Overview (Clean Architecture)

Core Layer Overview (Clean Architecture)

---

1. Core.Domain

Overview:

The `Core.Domain` layer represents the core business logic and domain entities of the application. Following the principles of clean architecture, this layer is the most critical as it defines the business rules, domain events, and entities that interact with other layers in the architecture. The `Core.Domain` layer is isolated from other layers, meaning it does not have dependencies on external libraries or infrastructure, ensuring a clean separation of concerns.

Structure and Files:

1. **Core.Domain.csproj**:
   - **Purpose:** The project file that defines dependencies, compilation options, and configurations specific to the domain layer.
   - **Pros:** Centralizes project-level configurations and dependencies.
   - **Cons:** Any misconfiguration can lead to build issues or incorrect dependency management.

2. **GlobalUsings.cs**:
   - **Purpose:** Contains global using directives that are available across the project.

- **Pros:** Reduces redundancy in the codebase by centralizing common namespaces.
- **Cons:** Overuse can lead to namespace conflicts or unclear code boundaries.

3. **Entities**:
   - **Overview:** Contains domain entities representing business objects.
   - **Files:**
     - **UiSchema.cs:** Defines the schema for UI elements.
     - **UiSchemaDesign.cs:** Manages the design aspects of the UI schema.
     - **ButtonControl.cs:** Represents a button control element.
     - **ComboBoxControl.cs:** Represents a combo box control element.
     - **ControlBase.cs:** Serves as the base class for all controls.
     - **DatePickerControl.cs:** Represents a date picker control element.
     - **FormControl.cs:** Manages form-related controls.
     - **RadioButtonControl.cs:** Represents a radio button control element.
     - **TextBoxControl.cs:** Represents a text box control element.
   - **Pros:** Centralizes business logic and ensures a single source of truth for domain entities.
   - **Cons:** Complex business rules can make the domain layer challenging to manage.

Key Points:

- The `Core.Domain` layer should be completely independent of other layers.
- Business rules and domain logic reside here, and they should not be affected by changes in other layers.

Core Layer Overview (Clean Architecture)

2. Core.Application

Overview:

The `Core.Application` layer handles the application logic, including use cases and application services. It orchestrates the interaction between the domain layer and the infrastructure layer. The `Core.Application` layer implements patterns like CQRS (Command Query Responsibility Segregation) and Mediator to manage interactions, which are fundamental to maintaining a clean architecture.

Structure and Files:

1. **Core.Application.csproj**:
   - **Purpose:** Project file for the application layer, managing dependencies and configurations.
   - **Pros:** Ensures application logic is separate from domain logic.
   - **Cons:** Can become complex with many dependencies.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the application layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

3. **Commands**:
   - **Overview:** Contains command classes for CQRS operations.
   - **Files:**
     - **AddCommand.cs:** Handles the logic for adding new entities.

  - **DeleteCommand.cs:** Manages the deletion of entities.
  - **UpdateCommand.cs:** Handles updating existing entities.
- **Pros:** Implements separation of concerns by isolating commands.
- **Cons:** Overhead of managing multiple commands.

4. **Controllers**:
- **Overview:** Contains API controllers.
- **Files:**
  - **BaseApiController.cs:** Base class for all API controllers, ensuring consistent behavior.
- **Pros:** Centralized place for handling API requests.
- **Cons:** Complex controllers can lead to code duplication.

5. **Helpers**:
- **Overview:** Utility classes for common tasks.
- **Files:**
  - **JsonHelper.cs:** Provides methods for JSON serialization and deserialization.
- **Pros:** Reusable utility methods.
- **Cons:** Overuse can lead to helper classes doing too much.

6. **Options**:
- **Overview:** Manages configuration options.
- **Files:**
  - **OptionsExtensions.cs:** Extensions for configuring options.
- **Pros:** Centralizes configuration logic.
- **Cons:** Mismanagement can lead to configuration errors.

7. **Queries**:
   - **Overview:** Contains query classes for CQRS operations.
   - **Files:**
     - **GetAllQuery.cs:** Handles retrieval of all entities.
   - **Pros:** Implements separation of concerns by isolating queries.
   - **Cons:** Similar to commands, can add overhead.

Key Points:
- The `Core.Application` layer should not directly reference infrastructure concerns, adhering to the Dependency Inversion Principle.
- The CQRS and Mediator patterns are used to separate read and write operations, maintaining a clear separation of concerns.

---

3. Core.Application.Abstractions

Overview:
The `Core.Application.Abstractions` layer contains the abstractions, interfaces, and contracts for the application layer. This layer promotes loose coupling by providing abstractions that can be implemented by various concrete classes in other layers.

Structure and Files:

Core Layer Overview (Clean Architecture)

1. **Core.Application.Abstractions.csproj**:
   - **Purpose:** Project file for the abstractions layer.
   - **Pros:** Ensures loose coupling between application and infrastructure layers.
   - **Cons:** Can add complexity if overused.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the abstractions layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

3. **Options**:
   - **Overview:** Interfaces for configuration options.
   - **Files:**
     - **IValidateOptions.cs:** Interface for validating options.
   - **Pros:** Enforces validation across different implementations.
   - **Cons:** Adds a layer of complexity to configuration management.

Key Points:

- Abstractions are central to achieving a clean architecture, ensuring that higher-level modules are not dependent on lower-level modules but rather on abstractions.

---

4. Core.Application.Dto

Core Layer Overview (Clean Architecture)

Overview:

The `Core.Application.Dto` layer contains Data Transfer Objects (DTOs) that are used to transfer data between layers. DTOs are simple objects that carry data between processes, ensuring that only necessary information is exposed.

Structure and Files:

1. **Core.Application.Dto.csproj**:
   - **Purpose:** Project file for the DTO layer.
   - **Pros:** Decouples data representation from business logic.
   - **Cons:** Can lead to redundant code if not managed properly.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the DTO layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

3. **UiSchemaDto.cs**:
   - **Purpose:** DTO for UI schema data.
   - **Pros:** Facilitates data transfer between UI and backend.
   - **Cons:** Adds maintenance overhead for keeping DTOs up-to-date.

Key Points:

- DTOs serve as a bridge between different layers, ensuring data is transferred without exposing internal representations.

Core Layer Overview (Clean Architecture)

---

5. Core.Infrastructure

Overview:

The `Core.Infrastructure` layer is responsible for implementing the abstractions defined in the application and domain layers. This layer interacts with external systems, databases, file systems, and other resources.

Structure and Files:

1. **Core.Infrastructure.csproj**:
   - **Purpose:** Project file for the infrastructure layer.
   - **Pros:** Centralizes infrastructure-related dependencies.
   - **Cons:** Complex configurations can make it difficult to manage.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the infrastructure layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

3. **ServiceCollectionExtensions.cs**:
   - **Purpose:** Extensions for adding services to the dependency injection container.
   - **Pros:** Promotes modular service registration.

- **Cons:** Can lead to complex DI setups.

Key Points:

- The `Core.Infrastructure` layer should implement the abstractions defined in the `Core.Application.Abstractions` layer, keeping the infrastructure code isolated from the application logic.

---

6. Core.Infrastructure.AzureStorage

Overview:

The `Core.Infrastructure.AzureStorage` layer handles interactions with Azure Storage services. It provides implementations for storage queues, blobs, and other Azure storage components.

Structure and Files:

1. **Core.Infrastructure.AzureStorage.csproj**:
   - **Purpose:** Project file for Azure storage infrastructure.
   - **Pros:** Centralizes Azure storage configurations and dependencies.
   - **Cons:** Azure-specific, making it less portable.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the Azure storage infrastructure layer.
   - **Pros:** Simplifies code by removing repetitive using statements.

- **Cons:** Potential for namespace conflicts.

3. **Abstractions**:
   - **Files:**
     - **IAzureStorageQueue.cs:** Interface for Azure storage queue operations.
   - **Pros:** Promotes loose coupling for Azure storage operations.
   - **Cons:** Requires concrete implementations.

4. **Model**:
   - **Files:**
     - **MessageModel.cs:** Model representing messages in Azure storage queues.
   - **Pros:** Defines a consistent structure for messages.
   - **Cons:** Azure-specific.

5. **Options**:
   - **Files:**
     - **AzureStorageOptions.cs:** Configuration options for Azure storage.
   - **Pros:** Centralizes Azure storage configuration.
   - **Cons:** Requires proper configuration management.

6. **Services**:
   - **Files:**
     - **AzureStorageQueueService.cs:** Implementation of storage queue services.
   - **Pros:** Provides concrete implementations for Azure storage interactions.
   - **Cons:** Azure-specific, less portable.

Key Points:

- The Azure Storage implementations are tightly coupled to Azure services, which could pose challenges if the application needs to migrate to another cloud provider.

---

7. Core.Infrastructure.EventGrid

Overview:

The `Core.Infrastructure.EventGrid` layer handles interactions with Azure Event Grid. It provides implementations for publishing and subscribing to events within the application.

Structure and Files:

1. **Core.Infrastructure.EventGrid.csproj**:
   - **Purpose:** Project file for Event Grid infrastructure.
   - **Pros:** Centralizes Event Grid configurations and dependencies.
   - **Cons:** Azure-specific, making it less portable.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the Event Grid infrastructure layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

Core Layer Overview (Clean Architecture)

3. **Abstractions**:
   - **Files:**
     - **IEventGrid.cs:** Interface for Event Grid operations.
     - **IEventGridService.cs:** Interface for Event Grid service operations.
     - **IEventService.cs:** Interface for event service operations.
     - **IPublisher.cs:** Interface for event publishing.
     - **ISubscriber.cs:** Interface for event subscribing.
   - **Pros:** Promotes loose coupling for event grid operations.
   - **Cons:** Requires concrete implementations.

4. **Constants**:
   - **Files:**
     - **EventGridEventType.cs:** Constants representing event types in Event Grid.
     - **EventType.cs:** General event type constants.
     - **EventTypeHeader.cs:** Constants for event type headers.
   - **Pros:** Centralizes event-related constants.
   - **Cons:** Adds maintenance overhead for updating constants.

5. **Dto**:
   - **Files:**
     - **PublisherResponseDto.cs:** DTO representing the response from a publisher.
   - **Pros:** Facilitates data transfer between publisher and subscriber.
   - **Cons:** Adds maintenance overhead for keeping DTOs up-to-date.

6. **Enums**:

Core Layer Overview (Clean Architecture)

- **Files:**
  - **EventGridMessageType.cs:** Enum representing message types in Event Grid.
- **Pros:** Provides a clear set of options for message types.
- **Cons:** Requires updates when new message types are introduced.

7. **Models**:
   - **Files:**
     - **GridEvent.cs:** Model representing an event in Event Grid.
   - **Pros:** Defines a consistent structure for events.
   - **Cons:** Azure-specific.

8. **Services**:
   - **Files:**
     - **EventGridService.cs:** Implementation of Event Grid service operations.
     - **EventService.cs:** Implementation of general event service operations.
     - **PublisherService.cs:** Implementation of event publishing services.
     - **SubscriberService.cs:** Implementation of event subscribing services.
   - **Pros:** Provides concrete implementations for event grid interactions.
   - **Cons:** Azure-specific, less portable.

Key Points:

- The Event Grid layer is specialized for handling event-driven interactions, making it highly efficient for distributed systems but tightly coupled to Azure.

Core Layer Overview (Clean Architecture)

8. Core.Infrastructure.SqlServer

Overview:

The `Core.Infrastructure.SqlServer` layer is responsible for interactions with SQL Server databases.

This includes configurations, connections, and executing database operations.

Structure and Files:

1. **Core.Infrastructure.SqlServer.csproj**:
   - **Purpose:** Project file for SQL Server infrastructure.
   - **Pros:** Centralizes SQL Server configurations and dependencies.
   - **Cons:** SQL Server-specific, less portable.

2. **GlobalUsings.cs**:
   - **Purpose:** Global using directives for the SQL Server infrastructure layer.
   - **Pros:** Simplifies code by removing repetitive using statements.
   - **Cons:** Potential for namespace conflicts.

3. **PersistenceOptions.cs**:
   - **Purpose:** Configuration options for persistence in SQL Server.
   - **Pros:** Centralizes SQL Server persistence configuration.
   - **Cons:** Requires proper configuration management.

Key Points:

Core Layer Overview (Clean Architecture)

- The SQL Server layer should be implemented in a way that allows easy migration to other relational databases if needed.