API Infrastructure Comprehensive Guide

Table of Contents

---

1. Presentation Layer

1.1 Core

- **Location**: `Presentation/core`
- **Components**:
    - `ApiBase`: Sets up the fundamental configurations and bootstrapping of the API.
    - `StartupHelpers`: Contains helper methods for configurations that might not be directly part of the startup sequence but are essential for the application's functionality.

```
public class StartupHelpers
{
    public static void
ConfigureAdditionalServices(IServiceCollection services)
    {
        // This method is used for adding additional services or
configurations
        // that are required by the application.
    }
}
```

Nd.GradGate.Kernel

StartupIoC

The `StartupIoC` class is crucial for dependency injection, ensuring that applications and actions are registered and available throughout the application.

- **Configuration**:

```
public class StartupIoC
{
    public static void Configure(IServiceCollection services)
    {
        // Here, we're registering applications and actions with the
service container.
        // Using Singleton or Scoped as per the lifecycle
requirements.
        services.AddSingleton<IMyApplication, MyApplication>();
        services.AddSingleton<IMyAction, MyAction>();
        // Additional dependencies can be added here.
    }
}
```

RepositoriesIoC

`RepositoriesIoC` is focused exclusively on the registration of repository classes which interact with the database. Typically, repositories are registered with transient or scoped lifetimes.

- **Configuration**:

```
public class RepositoriesIoC
{
    public static void Configure(IServiceCollection services)
    {
        // Register repositories as Transient, meaning a new instance
will be created
        // each time a repository is required.
        services.AddTransient<IMyRepository, MyRepository>();
        // Add more repositories as needed.
    }
}
```

Controllers

Controllers in an API act as the entry point for handling HTTP requests and returning responses. Each controller should be focused on a single application area (separation of concerns).

- **HTTP Methods and Parameter Bindings**:

```
[ApiController]
[Route("[controller]")]
public class MyController : ControllerBase
{
    private readonly IMyApplication _myApplication;

    public MyController(IMyApplication myApplication)
    {
```

```
        // Dependency injection is used here to get an instance of
IMyApplication.
        _myApplication = myApplication;
    }

    [HttpGet("{id}")]
    public ActionResult<MyDto> Get(int id)
    {
        // An example of using [HttpGet] attribute. The method handles
GET requests
        // and uses "from route" binding to get the 'id' parameter.
        var result = _myApplication.GetMyEntity(id);
        return Ok(result);
    }

    [HttpPost]
    public IActionResult Post([FromBody] MyDto myDto)
    {
        // This POST method uses [FromBody] to receive data. It's
useful for
        // receiving complex data types like objects.
        _myApplication.AddMyEntity(myDto);
        return Ok();
    }
}
```

2. Application Layer

The Application Layer is where business logic and operations are defined. Here, we define applications that handle various business operations and actions that represent the individual steps or processes within an operation.

Example Structure

- **Folder**: `NameApplication`

```
// INameApplication.cs
public interface INameApplication
{
    // Interface methods define the contract for the application
logic.
    MyEntity GetMyEntity(int id);
    void AddMyEntity(MyDto myDto);
}

// NameApplication.cs
public class NameApplication : INameApplication
{
    private readonly INameAction _nameAction;

    public NameApplication(INameAction nameAction)
```

```
    {
        // The application uses actions to perform its operations.
        _nameAction = nameAction;
    }

    public MyEntity GetMyEntity(int id)
    {
        // Implementing the method defined in the interface.
        return _nameAction.ExecuteAction(id);
    }

    public void AddMyEntity(MyDto myDto)
    {
        // Another example method that might use a different action.
    }
}
```

4. Domain Layer

Domain Models

In the Domain Layer, we define the entities that represent the business models. These entities should be a direct representation of how the business data is structured and stored. They typically map to database tables but may include additional logic or methods relevant to the business domain.

- **Example**: User model representation.

```
public class User : IEntity<User, Guid>
{
    public Guid ID { get; set; }
    // Additional properties and business logic specific to the User
entity.
}
```

5. Infrastructure Layer

Nd.GradKernel.Core.DataAccess

This part of the infrastructure layer contains classes and methods for accessing and manipulating the database. It includes the database context, repositories for handling CRUD operations, and mappings between domain models and database tables.

Context and Repositories

The database context is typically configured with Entity Framework Core to facilitate database operations. Repositories are then used to encapsulate the logic needed to access data sources.

- **Repository Implementation**:

```
public class MyRepository : IMyRepository,
IAbstractRepository<MyEntity>
{
    private readonly MyDbContext _context;

    public MyRepository(MyDbContext context)
    {
        _context = context;
    }

    public MyEntity SaveOrUpdate(MyEntity entity)
    {
        // Implement the SaveOrUpdate logic, using _context to
interact with the database.
        return entity;
    }

    // Implement other CRUD methods.
}
```

Actions

Actions in the infrastructure layer are responsible for performing specific operations, typically involving database interactions.

- **Example**: An action for getting a particular entity.

```
public class MyAction : IMyAction
{
    private readonly IMyRepository _myRepository;

    public MyAction(IMyRepository myRepository)
    {
        _myRepository = myRepository;
    }

    public MyEntity ExecuteAction(int id)
    {
        // The action interacts with the repository to fetch or update
data.
        return _myRepository.GetById(id);
    }
}
```

Handling Saves and Updates

When saving or updating data, the typical workflow involves receiving DTOs (Data Transfer Objects) from the application layer, converting these DTOs to domain entities, and then persisting these changes to the database through repositories.

- **DTO to Domain Conversion**: DTOs serve as a simpler or different view of a domain entity, often tailored for a specific operation or use case. Before saving or updating, it's crucial to map the data from DTOs to domain entities.

- **Save/Update Logic**:

```
public class MyAction : IMyAction
{
    private readonly IMyRepository _myRepository;

    public MyAction(IMyRepository myRepository)
    {
        _myRepository = myRepository;
    }

    public void SaveOrUpdateEntity(MyDto myDto)
    {
        // Convert DTO to Domain Object
        var myEntity = new MyEntity
        {
            // Assume MyEntity has properties ID, Name, and Value that
correspond to MyDto
            ID = myDto.ID,
            Name = myDto.Name,
            Value = myDto.Value
        };

        // Use repository method to save or update the entity in the
database
        _myRepository.SaveOrUpdate(myEntity);
    }
}
```

Importance of Using DTOs

DTOs are crucial in the context of APIs for several reasons:

- **Encapsulation**: They help encapsulate the data and separate what is sent/received over the network from the internal domain model, leading to more secure and maintainable code.
- **Flexibility**: Allows changing the internal domain model without necessarily impacting the external API contracts.
- **Performance Optimization**: DTOs can be optimized to carry only the data needed for a particular operation, thus potentially reducing the payload size and improving performance.

Feel free to extend this guide with specific examples and explanations as per your organization's API architecture and coding practices.