InfoSymbolServer.Application

Overview

The Application layer implements the core business logic of the InfoSymbolServer. This layer is responsible for:

- Implementing use cases through application services
- Defining Data Transfer Objects (DTOs) for client-server communication
- Validating incoming requests
- Providing error handling through custom exceptions
- Mapping between domain entities and DTOs

Key Components

Application Services

Application services implement the business logic for various operations on exchanges and symbols.

IExchangeService

Interface defining operations for managing exchanges:

```
public interface IExchangeService
{
    Task<IEnumerable<ExchangeDto>> GetAllAsync(CancellationToken
```

```
cancellationToken = default);
    Task<ExchangeDto?> GetByNameAsync(string name, CancellationToken
cancellationToken = default);
    Task<ExchangeDto> AddAsync(CreateExchangeDto exchangeDto,
CancellationToken cancellationToken = default);
    Task DeleteAsync(string name, CancellationToken cancellationToken =
default);
    IEnumerable<string> GetSupportedExchanges();
}
```

ExchangeService

Implementation of `IExchangeService` that:

- Uses repository to access data
- Validates requests through the validation pipeline
- Maps between domain entities and DTOs using AutoMapper
- Manages transactions through the unit of work pattern

Key implementation details:

```
public class ExchangeService : IExchangeService
{
    private readonly IExchangeRepository _exchangeRepository;
    private readonly IUnitOfWork _unitOfWork;
    private readonly IMapper _mapper;
    private readonly ValidationPipeline _validationPipeline;

    // Constructor with dependency injection...

    public async Task<ExchangeDto> AddAsync(CreateExchangeDto exchangeDto,
CancellationToken cancellationToken = default)
    {
        await _validationPipeline.ValidateAsync(exchangeDto,
cancellationToken);

        var exchange = _mapper.Map<Exchange>(exchangeDto);

        var existingExchange = await
_exchangeRepository.GetByNameAsync(exchange.Name, cancellationToken);
        if (existingExchange != null)
        {
            throw new ValidationException(nameof(Exchange),
exchange.Name);
        }

        if (!GetSupportedExchanges().Contains(exchange.Name))
        {
```

```
            throw new ValidationException(nameof(Exchange), $"Exchange
{exchange.Name} is not supported.");
        }

        await _exchangeRepository.AddAsync(exchange, cancellationToken);
        await _unitOfWork.SaveAsync(cancellationToken);

        return _mapper.Map<ExchangeDto>(exchange);
    }

    public IEnumerable<string> GetSupportedExchanges()
    {
        return
        [
            SupportedExchanges.BinanceSpot,
            SupportedExchanges.BinanceUsdtFutures,
            SupportedExchanges.BinanceCoinFutures
        ];
    }
}
```

ISymbolService

Interface defining operations for managing symbols:

```
public interface ISymbolService
{
    Task<IEnumerable<SymbolDto>> GetForExchangeAsync(
        string exchangeName,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default);

    Task<IEnumerable<SymbolDto>> GetActiveForExchangeAsync(
        string exchangeName,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default);

    Task<SymbolDto?> GetForExchangeByNameAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default);

    Task<SymbolDto> AddAsync(
        AddSymbolDto addSymbolDto,
        CancellationToken cancellationToken = default);

    Task<SymbolDto> DeleteAsync(
```

```
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default);

    Task<SymbolDto> RevokeDeleteAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default);
}
```

SymbolService

Implementation of `ISymbolService` that:

- Uses repository to access data
- Maps between domain entities and DTOs using AutoMapper

```
public class SymbolService : ISymbolService
{
    private readonly ISymbolRepository _symbolRepository;
    private readonly IExchangeRepository _exchangeRepository;
    private readonly IStatusRepository _statusRepository;
    private readonly IUnitOfWork _unitOfWork;
    private readonly IMapper _mapper;

    // Constructor with dependency injection...

    public async Task<IEnumerable<SymbolDto>> GetActiveForExchangeAsync(
        string exchangeName,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default)
    {
        var exchange = await
_exchangeRepository.GetByNameAsync(exchangeName, cancellationToken)
            ?? throw new NotFoundException(nameof(Exchange),
exchangeName);

        var symbols = await _symbolRepository.GetByFilterAsync(
            s => s.ExchangeId == exchange.Id && s.Status ==
SymbolStatus.Active,
            pageNumber,
            pageSize,
            cancellationToken);

        return _mapper.Map<IEnumerable<SymbolDto>>(symbols);
    }
```

```
    public async Task<SymbolDto> AddAsync(
        AddSymbolDto addSymbolDto,
        CancellationToken cancellationToken = default)
    {
        // Validate the DTO using the validation pipeline
        // Try to get exchange and symbol

        // If symbol exists in any state, throw a validation exception

        // Create new symbol using mapper
        var newSymbol = mapper.Map<Symbol>(addSymbolDto);
        // setting specific fields

        await symbolRepository.AddAsync(newSymbol, cancellationToken);

        // Add status record for the new symbol
        await statusRepository.AddAsync(new Status
        {
            Id = Guid.NewGuid(),
            CreatedAt = DateTime.UtcNow,
            SymbolId = newSymbol.Id,
            SymbolStatus = SymbolStatus.AddedByAdmin
        }, cancellationToken);

        await unitOfWork.SaveAsync(cancellationToken);
        return mapper.Map<SymbolDto>(newSymbol);
    }

    public async Task<SymbolDto> DeleteAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default)
    {
        // Try to get exchange and symbol

        // If symbol is already removed by admin, throw validation
exception
        if (symbol.Status == SymbolStatus.RemovedByAdmin)
        {
            throw new ValidationException(nameof(Symbol), "Symbol is
already removed by admin");
        }

        // Set status to RemovedByAdmin and update timestamp
        symbol.Status = SymbolStatus.RemovedByAdmin;
        symbol.UpdatedAt = DateTime.UtcNow;

        // Update symbol and add status change record
        await _symbolRepository.UpdateAsync(symbol, cancellationToken);
        await _statusRepository.AddAsync(new Status
```

```
        {
            Id = Guid.NewGuid(),
            CreatedAt = DateTime.UtcNow,
            SymbolId = symbol.Id,
            SymbolStatus = SymbolStatus.RemovedByAdmin
        }, cancellationToken);

        await _unitOfWork.SaveAsync(cancellationToken);
        return _mapper.Map<SymbolDto>(symbol);
    }

    public async Task<SymbolDto> RevokeDeleteAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default)
    {
        // Try to get exchange and symbol

        // If symbol is not in RemovedByAdmin status, throw validation
exception
        if (symbol.Status != SymbolStatus.RemovedByAdmin)
        {
            throw new ValidationException(nameof(Symbol), "Only symbols
with RemovedByAdmin status can be revoked");
        }

        // Set status to AddedByAdmin and update timestamp
        symbol.Status = SymbolStatus.AddedByAdmin;
        symbol.UpdatedAt = DateTime.UtcNow;

        // Update symbol and add status change record
        await _symbolRepository.UpdateAsync(symbol, cancellationToken);
        await _statusRepository.AddAsync(new Status
        {
            Id = Guid.NewGuid(),
            CreatedAt = DateTime.UtcNow,
            SymbolId = symbol.Id,
            SymbolStatus = SymbolStatus.AddedByAdmin
        }, cancellationToken);

        await _unitOfWork.SaveAsync(cancellationToken);
        return _mapper.Map<SymbolDto>(symbol);
    }
}
```

IStatusService

Interface defining operations for managing status records:

```
public interface IStatusService
{
    Task<SymbolHistoryDto> GetSymbolHistoryAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default);

    Task<IEnumerable<SymbolHistoryDto>> GetExchangeSymbolsHistoryAsync(
        string exchangeName,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default);

    Task<IEnumerable<SymbolHistoryDto>>
GetExchangeActiveSymbolsHistoryAsync(
        string exchangeName,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default);
}
```

StatusService

Implementation of `IStatusService` that:

- Uses repository to access data
- Maps between domain entities and DTOs using AutoMapper

```
public class StatusService : IStatusService
{
    private readonly IStatusRepository _statusRepository;
    private readonly ISymbolRepository _symbolRepository;
    private readonly IExchangeRepository _exchangeRepository;
    private readonly IMapper _mapper;

    // Constructor with dependency injection...

    public async Task<SymbolHistoryDto> GetSymbolHistoryAsync(
        string symbolName,
        string exchangeName,
        CancellationToken cancellationToken = default)
    {
        var exchange = await _exchangeRepository
            .GetByNameAsync(exchangeName, cancellationToken)
            ?? throw new NotFoundException(nameof(Exchange),
exchangeName);

        var symbol = await _symbolRepository
```

```
            .GetSingleByFilterAsync(s =>
                s.SymbolName == symbolName && s.ExchangeId == exchange.Id,
cancellationToken)
            ?? throw new NotFoundException($"Symbol {symbolName} on
exchange {exchangeName}");

        var statuses = await _statusRepository.GetByFilterAsync(
            s => s.SymbolId == symbol.Id, null, null, cancellationToken);

        return new SymbolHistoryDto
        {
            SymbolName = symbolName,
            History = _mapper.Map<IEnumerable<StatusDto>>(statuses)
        };
    }
}
```

INotificationSettingsService

Interface defining operations for managing notification settings:

```
public interface INotificationSettingsService
{
    Task<NotificationSettingsDto> GetAsync();
    Task<NotificationSettingsDto>
UpdateAsync(UpdateNotificationSettingsDto updateDto);
}
```

NotificationSettingsService

Implementation of `INotificationSettingsService` that:

- Uses repository to access data
- Maps between domain entities and DTOs using AutoMapper
- Provides default settings when none exist
- Validates notification configuration status using `INotificationValidationService`

```
public class NotificationSettingsService : INotificationSettingsService
{
    private readonly INotificationSettingsRepository _repository;
    private readonly IMapper _mapper;
    private readonly INotificationValidationService _validationService;

    public NotificationSettingsService(
        INotificationSettingsRepository repository,
        IMapper mapper,
        INotificationValidationService validationService)
```

```
    {
        _repository = repository;
        _mapper = mapper;
        _validationService = validationService;
    }

    public async Task<NotificationSettingsDto> GetAsync()
    {
        var settings = await _repository.GetAsync();

        // If settings don't exist yet, create default settings
        settings ??= new NotificationSettings
        {
            IsTelegramEnabled = true,
            IsEmailEnabled = true
        };

        var dto = _mapper.Map<NotificationSettingsDto>(settings);

        // Add configuration status information
        dto = dto with
        {
            IsTelegramConfigured = _validationService.IsTelegramValid(),
            IsEmailConfigured = _validationService.IsEmailValid()
        };

        return dto;
    }

    public async Task<NotificationSettingsDto>
UpdateAsync(UpdateNotificationSettingsDto updateDto)
    {
        // Validate if user can enable notifications
        ValidateNotificationSettings(updateDto);

        var settingsToUpdate = _mapper.Map<NotificationSettings>
(updateDto);
        var updatedSettings = await
_repository.UpdateAsync(settingsToUpdate);

        var dto = _mapper.Map<NotificationSettingsDto>(updatedSettings);

        // Add configuration status information
        dto = dto with
        {
            IsTelegramConfigured = _validationService.IsTelegramValid(),
            IsEmailConfigured = _validationService.IsEmailValid()
        };

        return dto;
```

```
    }

    private void
ValidateNotificationSettings(UpdateNotificationSettingsDto updateDto)
    {
        var errors = new Dictionary<string, string>();

        // If tries to enable Telegram notifications and they are not
properly configured, add an error
        if (updateDto.IsTelegramEnabled &&
!_validationService.IsTelegramValid())
        {
            errors.Add(
                nameof(updateDto.IsTelegramEnabled),
                "Cannot enable Telegram notifications because they are not
properly configured");
        }

        // If tries to enable Email notifications and they are not
properly configured, add an error
        if (updateDto.IsEmailEnabled &&
!_validationService.IsEmailValid())
        {
            errors.Add(
                nameof(updateDto.IsEmailEnabled),
                "Cannot enable Email notifications because they are not
properly configured");
        }

        if (errors.Count != 0)
        {
            throw new ValidationException(errors);
        }
    }
}
```

Data Transfer Objects (DTOs)

DTOs are used to transfer data between the Application layer and the Presentation layer.

Exchange DTOs

The application defines two main exchange DTOs:

ExchangeDto: Used for reading exchange data

```
public record ExchangeDto
{
    public Guid Id { get; init; }
```

```
    public string Name { get; init; } = null!;
    public DateTime CreatedAt { get; init; }
}
```

CreateExchangeDto: Used for creating new exchanges

```
public record CreateExchangeDto
{
    public string Name { get; init; } = null!;
}
```

Symbol DTOs

SymbolDto: Used for reading symbol data

```
public record SymbolDto
{
    public Guid Id { get; init; }
    public Guid ExchangeId { get; init; }
    public string SymbolName { get; init; } = null!;
    public MarketType MarketType { get; init; }
    public string BaseAsset { get; init; } = null!;
    public string QuoteAsset { get; init; } = null!;
    public SymbolStatus Status { get; init; }
    public int PricePrecision { get; init; }
    public int QuantityPrecision { get; init; }
    public ContractType? ContractType { get; init; }
    public DateTime? DeliveryDate { get; init; }
    public string? MarginAsset { get; init; }
    public decimal MinQuantity { get; init; }
    public decimal MinNotional { get; init; }
    public decimal MaxQuantity { get; init; }
    public DateTime UpdatedAt { get; init; }
}
```

AddSymbolDto: Used for adding or updating symbols

```
public record AddSymbolDto
{
    public string ExchangeName { get; set; } = null!;
    public string SymbolName { get; init; } = null!;
    public MarketType MarketType { get; init; }
    public string? BaseAsset { get; init; }
    public string? QuoteAsset { get; init; }
    public int? PricePrecision { get; init; }
    public int? QuantityPrecision { get; init; }
    public ContractType? ContractType { get; init; }
```

```
    public DateTime? DeliveryDate { get; init; }
    public string? MarginAsset { get; init; }
    public decimal? MinQuantity { get; init; }
    public decimal? MinNotional { get; init; }
    public decimal? MaxQuantity { get; init; }
}
```

Status DTOs

StatusDto: Used for reading status data

```
public record StatusDto
{
    public DateTime UpdatedAt { get; init; }
    public SymbolStatus SymbolStatus { get; init; }
}
```

SymbolHistoryDto: Used for reading symbol history

```
public record SymbolHistoryDto
{
    public string SymbolName { get; init; } = null!;
    public IEnumerable<StatusDto> History { get; init; } = [];
}
```

NotificationSettings DTOs

NotificationSettingsDto: Used for reading notification settings data

```
public record NotificationSettingsDto
{
    public bool IsTelegramEnabled { get; init; }
    public bool IsEmailEnabled { get; init; }
    public bool IsTelegramConfigured { get; init; }
    public bool IsEmailConfigured { get; init; }
}
```

UpdateNotificationSettingsDto: Used for updating notification settings

```
public record UpdateNotificationSettingsDto
{
    public bool IsTelegramEnabled { get; init; }
    public bool IsEmailEnabled { get; init; }
}
```

Validation

The Application layer implements request validation using FluentValidation.

ValidationPipeline

Central service for validating request DTOs. It allows not to instantiate validator in-place every time it's needed to validate a DTO. Instead, it registered as a service and allows to inject it everywhere where needed and validate every type of DTO, if there is a validator defined for it's type.

```
public class ValidationPipeline
{
    private readonly IServiceProvider _serviceProvider;

    public ValidationPipeline(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    public void Validate<TDto>(TDto dto) where TDto : class
    {
        var validatorType =
typeof(IValidator<>).MakeGenericType(typeof(TDto));

        if (serviceProvider.GetService(validatorType) is IValidator
validator)
        {
            var context = new ValidationContext<TDto>(dto);
            var result = validator.Validate(context);

            if (!result.IsValid)
            {
                var errors = result.Errors
                    .GroupBy(e => e.PropertyName)
                    .ToDictionary(
                        g => g.Key,
                        g => g.Select(e => e.ErrorMessage).ToArray());

                throw new ValidationException(errors);
            }
        }
    }

    // Also has asynchronous alternative
}
```

Validators

Individual validators for DTOs using FluentValidation:

CreateExchangeDtoValidator: Validates exchange creation requests

```
public class CreateExchangeDtoValidator :
AbstractValidator<CreateExchangeDto>
{
    public CreateExchangeDtoValidator()
    {
        RuleFor(x => x.Name)
            .NotEmpty().WithMessage("Exchange name is required")
            .MaximumLength(100).WithMessage("Exchange name must not exceed
100 characters");
    }
}
```

Object Mapping

The Application layer uses `AutoMapper` for mapping between entities and DTOs:

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        // Exchange mappings
        CreateMap<Exchange, ExchangeDto>();
        CreateMap<CreateExchangeDto, Exchange>();

        // Symbol mappings
        CreateMap<Symbol, SymbolDto>();

        // Status mappings
        CreateMap<Status, StatusDto>()
            .ForMember(dest => dest.UpdatedAt, opt => opt.MapFrom(src =>
src.CreatedAt));

        // NotificationSettings mappings
        CreateMap<NotificationSettings, NotificationSettingsDto>();
        CreateMap<UpdateNotificationSettingsDto, NotificationSettings>();
    }
}
```

Custom Exceptions

The Application layer defines custom exceptions to provide meaningful error responses:

NotFoundException: Thrown when a requested entity is not found

```
public class NotFoundException : Exception
{
    public NotFoundException()
        : base("The requested resource was not found.")
    {
    }

    public NotFoundException(string message)
        : base(message)
    {
    }

    public NotFoundException(string name, object key)
        : base($"Entity \"{name}\" with identifier {key} was not found.")
    {
    }
}
```

ValidationException: Thrown when validation fails

```
public class ValidationException : Exception
{
    public IDictionary<string, string[]> Errors { get; }

    public ValidationException()
        : base("One or more validation failures have occurred.")
    {
        Errors = new Dictionary<string, string[]>();
    }

    public ValidationException(IDictionary<string, string[]> errors)
        : this()
    {
        Errors = errors ?? new Dictionary<string, string[]>();
    }
}
```

Dependency Registration

Provides an extension method for registering its services:

```
public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddApplication(this
IServiceCollection services)
    {
        // Registers services implementations.
```

```
        services.AddScoped<ISymbolService, SymbolService>();
        services.AddScoped<IExchangeService, ExchangeService>();
        services.AddScoped<IStatusService, StatusService>();
        services.AddScoped<INotificationSettingsService,
NotificationSettingsService>();

        // Configures validation.
        services.AddValidatorsFromAssembly(AssemblyReference.Assembly);
        services.AddScoped<ValidationPipeline>();

        // Configures AutoMapper to user MappingProfile that defined
earlier.
        services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());

        services.AddProblemDetails();

        return services;
    }
}
```

> **Note**
>
> ASP.NET Problem Details are configured for standardized error responses

Application Layer Workflow

1. **Request Reception**: The Presentation layer receives client requests and converts them to DTOs
2. **Validation**: DTOs are validated using FluentValidation through the ValidationPipeline
3. **Service Processing**: Application services process validated requests
4. **Data Access**: Services use domain repositories to perform data operations
5. **Domain Entity Mapping**: Results are mapped from domain entities to DTOs
6. **Response Return**: DTOs are returned to the Presentation layer