# InfoSymbolServer.Domain



## Overview

The Domain layer is the core of the InfoSymbolServer application, containing business entities, enumerations, and repository interfaces. This layer is independent of external concerns and defines the fundamental business rules and data structures used throughout the application.

## Domain Models

### Exchange

The `Exchange` class represents a trading platform or its market (like BinanceSpot, BinanceUsdtFutures, etc.) and stores its key properties.

```
public class Exchange
{
    public Guid Id { get; set; }
    public string Name { get; set; } = null!;
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public ICollection<Symbol> Symbols { get; set; } = [];
}
```

Properties:

| Property | Type | Description |
|---|---|---|
| Id | Guid | Unique identifier for the exchange |
| Name | string | Name of the exchange (e.g., "Binance") |
| CreatedAt | DateTime | Date and time when the exchange was created |
| Symbols | ICollection | Navigation property to all symbols available on this exchange |

## Symbol

The `Symbol` class represents a trading instrument on an exchange (e.g., BTC/USDT pair).

```
public class Symbol
{
    public Guid Id { get; set; }
    public Guid ExchangeId { get; set; }
    public string SymbolName { get; set; } = string.Empty;
    public MarketType MarketType { get; set; }
    public string BaseAsset { get; set; } = string.Empty;
    public string QuoteAsset { get; set; } = string.Empty;
    public SymbolStatus Status { get; set; }
    public int PricePrecision { get; set; }
    public int QuantityPrecision { get; set; }
    public ContractType? ContractType { get; set; }
    public DateTime? DeliveryDate { get; set; }
    public string? MarginAsset { get; set; }
    public decimal MinQuantity { get; set; }
    public decimal MinNotional { get; set; }
    public decimal MaxQuantity { get; set; }
    public DateTime UpdatedAt { get; set; }
    public Exchange? Exchange { get; set; }
}
```

Properties:

| Property | Type | Description |
|---|---|---|
| Id | Guid | Unique identifier for the symbol |
| ExchangeId | Guid | Foreign key to the associated exchange |
| SymbolName | string | Trading pair name (e.g., "BTCUSDT") |
| MarketType | MarketType | Type of market (Spot, UsdtFutures, CoinFutures) |
| BaseAsset | string | Base asset in the trading pair (e.g., "BTC" in "BTCUSDT") |

| Property | Type | Description |
|---|---|---|
| QuoteAsset | string | Quote asset in the trading pair (e.g., "USDT" in "BTCUSDT") |
| Status | SymbolStatus | Current trading status |
| PricePrecision | int | Number of decimal places allowed for price |
| QuantityPrecision | int | Number of decimal places allowed for quantity |
| ContractType | ContractType? | Type of contract for futures/options |
| DeliveryDate | DateTime? | Expiration date for futures contracts |
| MarginAsset | string? | Asset used for margin (e.g., "USDT" for USDT-margined futures) |
| MinQuantity | decimal | Minimum quantity allowed for orders |
| MinNotional | decimal | Minimum order value (price × quantity) |
| MaxQuantity | decimal | Maximum quantity allowed for orders |
| UpdatedAt | DateTime | Date and time when the symbol was updated |
| Exchange | Exchange? | Navigation property to the parent exchange |

> **Note**
>
> An Exchange can have many Symbols (one-to-many relationship).
> Each Symbol belongs to exactly one Exchange.

## Status

The `Status` class represents a record of status changes for symbols.

```
public class Status
{
    public Guid Id { get; set; }
    public DateTime CreatedAt { get; set; }
    public SymbolStatus SymbolStatus { get; set; }
    public Guid SymbolId { get; set; }
    public Symbol? Symbol { get; set; }
}
```

Properties:

| Property | Type | Description |
|---|---|---|
| Id | Guid | Unique identifier for the status change record |
| CreatedAt | DateTime | Created at |

| Property | Type | Description |
|---|---|---|
| SymbolStatus | SymbolStatus | New status of the symbol |
| SymbolId | Guid | Foreign key to the associated symbol |
| Symbol | Symbol? | Navigation property to related symbol |

### NotificationSettings

The `NotificationSettings` class represents notification settings for the application.

```
public class NotificationSettings
{
    public Guid Id { get; set; }
    public bool IsTelegramEnabled { get; set; }
    public bool IsEmailEnabled { get; set; }
}
```

Properties:

| Property | Type | Description |
|---|---|---|
| Id | Guid | Unique identifier for the notification settings |
| IsTelegramEnabled | bool | Whether Telegram notifications are enabled |
| IsEmailEnabled | bool | Whether Email notifications are enabled |

## Enumerations

### MarketType

Defines the types of markets available for trading:

```
public enum MarketType
{
    Spot,            // Immediate delivery markets
    UsdtFutures,     // USDT-margined futures contracts
    CoinFutures      // Coin-margined futures contracts
}
```

### ContractType

Defines the types of contracts for trading instruments:

```
public enum ContractType
{
```

```
    Spot,            // Spot trading with immediate settlement
    Perpetual,       // Futures with no expiration date
    CurrentQuarter,  // Futures expiring this quarter
    NextQuarter      // Futures expiring next quarter
}
```

## SymbolStatus

Defines the possible trading statuses for a symbol:

```
public enum SymbolStatus
{
    // The symbol is actively trading on the exchange without any
restrictions.
    Active,

    // Trading for this symbol has been temporarily suspended for a finite
period (not permanently).
    Suspended,

    // The symbol is in a pre-trading phase, preceding the active trading
stage.
    PreLaunch,

    // The symbol has been permanently removed from the exchange.
    Delisted,

    // Relevant for time-limited instruments, such as quarterly futures,
    // indicating the instrument has reached its expiration date.
    Expired,

    // This status may precede the Expired status for certain instruments,
    // typically indicating the instrument is in its settlement phase.
    Settling,

    // The symbol was manually added by an administrator and has not yet
    // been synchronized with the exchange.
    AddedByAdmin,

    // The symbol was removed by an administrator and is no longer being
    // synchronized with the exchange, although it may still be trading
there.
    RemovedByAdmin
}
```

# Repository Interfaces

The domain layer defines repository interfaces that abstract data access operations. These interfaces follow the Repository pattern, which isolates domain objects from the details of database access code. The concrete implementations of these interfaces are provided in the Infrastructure layer.

## IExchangeRepository

Defines operations for accessing and manipulating `Exchange` entities:

```
public interface IExchangeRepository
{
    Task<IEnumerable<Exchange>> GetAllAsync(CancellationToken
cancellationToken = default);
    Task<IEnumerable<Exchange>> GetByFilterAsync(Expression<Func<Exchange,
bool>> filter, CancellationToken cancellationToken = default);
    Task<Exchange?> GetByNameAsync(string name, CancellationToken
cancellationToken = default);
    Task AddAsync(Exchange exchange, CancellationToken cancellationToken =
default);
    Task UpdateAsync(Exchange exchange, CancellationToken
cancellationToken = default);
    Task DeleteAsync(Exchange exchange, CancellationToken
cancellationToken = default);
}
```

This interface provides:

- Retrieval of all exchanges with pagination support
- Filtering exchanges based on custom expressions
- Lookup by name for specific exchange retrieval
- Operations for adding, updating, and removing exchanges from the system

The actual data access logic and Entity Framework integration will be implemented in the `ExchangeRepository` class in the Infrastructure layer.

## ISymbolRepository

Defines operations for accessing and manipulating `Symbol` entities:

```
public interface ISymbolRepository
{
    Task<IEnumerable<Symbol>> GetByFilterAsync(
        Expression<Func<Symbol, bool>> filter, int? pageNumber = null,
int? pageSize = null, CancellationToken cancellationToken = default);

    Task<Symbol?> GetSingleByFilterAsync(
        Expression<Func<Symbol, bool>> filter, CancellationToken
```

```
cancellationToken = default);

    Task AddAsync(Symbol symbol, CancellationToken cancellationToken =
default);

    Task AddRangeAsync(IEnumerable<Symbol> symbols, CancellationToken
cancellationToken = default);

    Task UpdateAsync(Symbol symbol, CancellationToken cancellationToken =
default);

    Task UpdateRangeAsync(IEnumerable<Symbol> symbols, CancellationToken
cancellationToken = default);

    Task DeleteAsync(Symbol symbol, CancellationToken cancellationToken =
default);
}
```

This interface provides:

- Retrieval of all trading symbols with pagination support
- Filtering symbols based on custom expressions
- Specialized method to get all symbols for a particular exchange
- Lookup by ID for specific symbol retrieval
- Operations for adding, updating, and removing symbols from the system

The actual implementation will be provided in the `SymbolRepository` class in the Infrastructure layer.

## IStatusRepository

Defines operations for accessing and manipulating `Status` entities:

```
public interface IStatusRepository
{
    Task<IEnumerable<Status>> GetAllAsync(
        int? pageNumber = null, int? pageSize = null, CancellationToken
cancellationToken = default);

    Task<IEnumerable<Status>> GetByFilterAsync(
        Expression<Func<Status, bool>> filter, int? pageNumber = null,
int? pageSize = null, CancellationToken cancellationToken = default);

    Task<Status?> GetSingleByFilterAsync(
        Expression<Func<Status, bool>> filter, CancellationToken
cancellationToken = default);

    Task AddAsync(Status status, CancellationToken cancellationToken =
```

```
default);

    Task AddRangeAsync(IEnumerable<Status> statuses, CancellationToken
cancellationToken = default);
}
```

## INotificationSettingsRepository

Defines operations for accessing and manipulating `NotificationSettings` entities:

```
public interface INotificationSettingsRepository
{
    Task<NotificationSettings?> GetAsync();
    Task<NotificationSettings> UpdateAsync(NotificationSettings settings);
}
```

This interface provides:

- Retrieval of the current notification settings
- Update operation for modifying notification settings

The actual implementation will be provided in the `NotificationSettingsRepository` class in the Infrastructure layer.

## IUnitOfWork

Manages the transaction scope for database operations:

```
public interface IUnitOfWork
{
    Task SaveAsync(CancellationToken cancellationToken = default);
}
```

The concrete implementation in the Infrastructure layer (typically `UnitOfWork` class) will handle the actual database transaction management using Entity Framework's `SaveChangesAsync` method.