InfoSymbolServer.Infrastructure.DataAccess

Overview

The Data Access layer of InfoSymbolServer implements the persistence concerns of the application. This layer is responsible for:

- Defining the database context and entity configurations
- Implementing repository interfaces defined in the Domain layer
- Managing database connections and transactions
- Providing a clean abstraction for data access operations

Key Components

ApplicationDbContext

The `ApplicationDbContext` serves as the central point for database interactions using Entity Framework Core:

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Exchange> Exchanges { get; set; }
    public DbSet<Symbol> Symbols { get; set; }
    public DbSet<Status> Statuses { get; set; }
    public DbSet<NotificationSettings> NotificationSettings { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
```

```
        modelBuilder.ApplyConfigurationsFromAssembly(AssemblyReference.Assembly);
        base.OnModelCreating(modelBuilder);
    }
}
```

The context is configured to use PostgreSQL as the database provider and applies entity configurations from the assembly.

Entity Configurations

Entity configurations define the database schema and constraints for each entity type:

ExchangeConfiguration

```
public class ExchangeConfiguration : IEntityTypeConfiguration<Exchange>
{
    public void Configure(EntityTypeBuilder<Exchange> builder)
    {
        builder.HasKey(e => e.Id);

        builder.Property(e => e.Name)
            .IsRequired();

        builder.Property(e => e.CreatedAt)
            .IsRequired();

        builder.HasMany(e => e.Symbols)
            .WithOne(s => s.Exchange)
            .HasForeignKey(s => s.ExchangeId);

        builder.HasIndex(e => e.Name)
            .IsUnique();
    }
}
```

SymbolConfiguration

```
public class SymbolConfiguration : IEntityTypeConfiguration<Symbol>
{
    public void Configure(EntityTypeBuilder<Symbol> builder)
    {
        builder.HasKey(s => s.Id);

        builder.Property(s => s.SymbolName)
            .IsRequired();
```

```
        builder.Property(s => s.MarketType)
            .IsRequired()
            .HasConversion(new EnumToStringConverter<MarketType>());

        builder.Property(s => s.BaseAsset)
            .IsRequired();

        // other properties

        builder.HasIndex(s => new { s.ExchangeId, s.SymbolName,
s.MarketType })
            .IsUnique();
    }
}
```

StatusConfiguration

```
public class StatusConfiguration : IEntityTypeConfiguration<Status>
{
    public void Configure(EntityTypeBuilder<Status> builder)
    {
        builder.HasKey(s => s.Id);

        builder.Property(s => s.CreatedAt)
            .IsRequired();

        builder.Property(s => s.SymbolStatus)
            .IsRequired()
            .HasConversion(new EnumToStringConverter<SymbolStatus>());

        builder.HasOne(s => s.Symbol)
            .WithMany()
            .HasForeignKey(s => s.SymbolId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

NotificationSettingsConfiguration

```
public class NotificationSettingsConfiguration :
IEntityTypeConfiguration<NotificationSettings>
{
    public void Configure(EntityTypeBuilder<NotificationSettings> builder)
    {
        builder.HasKey(ns => ns.Id);

        builder.Property(ns => ns.IsTelegramEnabled)
```

```
            .IsRequired()
            .HasDefaultValue(true);

        builder.Property(ns => ns.IsEmailEnabled)
            .IsRequired()
            .HasDefaultValue(true);

        // Seed initial data - notifications enabled by default
        builder.HasData(
            new NotificationSettings
            {
                Id = Guid.Parse("5a23149e-79cc-4fed-8533-c3b4415c2cdb"),
                IsTelegramEnabled = true,
                IsEmailEnabled = true
            }
        );
    }
}
```

> **Note**
>
> `EnumToStringConverter` is used for enum properties to store them as strings in the database.

Repository Implementations

The Data Access layer provides concrete implementations of the domain repository interfaces.

ExchangeRepository

Implements the `IExchangeRepository` interface from the Domain layer:

```
public class ExchangeRepository : IExchangeRepository
{
    private readonly ApplicationDbContext _context;

    public ExchangeRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<Exchange>> GetAllAsync(CancellationToken
cancellationToken = default)
    {
        return await _context.Exchanges
            .OrderBy(e => e.Name)
```

```
            .ToListAsync(cancellationToken: cancellationToken);
    }

    // other methods
}
```

SymbolRepository

Implements the `ISymbolRepository` interface from the Domain layer:

```
public class SymbolRepository : ISymbolRepository
{
    private readonly ApplicationDbContext _context;

    public SymbolRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<Symbol>> GetByFilterAsync(
        Expression<Func<Symbol, bool>> filter,
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default)
    {
        IQueryable<Symbol> query = _context.Symbols
            .Where(filter)
            .OrderBy(s => s.SymbolName);

        if (pageNumber.HasValue && pageSize.HasValue)
        {
            query = query
                .Skip((pageNumber.Value - 1) * pageSize.Value)
                .Take(pageSize.Value);
        }

        return await query.ToListAsync(cancellationToken);
    }

    // other methods
}
```

StatusRepository

Implements the `IStatusRepository` interface from the Domain layer:

```
public class StatusRepository : IStatusRepository
{
```

```
    private readonly ApplicationDbContext _context;

    public StatusRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<IEnumerable<Status>> GetAllAsync(
        int? pageNumber = null,
        int? pageSize = null,
        CancellationToken cancellationToken = default)
    {
        IQueryable<Status> query = _context.Statuses
            .Include(s => s.Symbol)
            .OrderByDescending(s => s.CreatedAt);

        if (pageNumber.HasValue && pageSize.HasValue)
        {
            query = query
                .Skip((pageNumber.Value - 1) * pageSize.Value)
                .Take(pageSize.Value);
        }

        return await query.ToListAsync(cancellationToken);
    }

    // other methods
}
```

NotificationSettingsRepository

Implements the `INotificationSettingsRepository` interface from the Domain layer:

```
public class NotificationSettingsRepository :
INotificationSettingsRepository
{
    private readonly ApplicationDbContext _context;

    public NotificationSettingsRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<NotificationSettings?> GetAsync()
    {
        // Always return the first record, as we only have one settings
record in the table
        return await _context.NotificationSettings.FirstOrDefaultAsync();
    }
```

```
    public async Task<NotificationSettings>
UpdateAsync(NotificationSettings settings)
    {
        var existingSettings = await GetAsync();
        if (existingSettings == null)
        {
            _context.NotificationSettings.Add(settings);
        }
        else
        {
            existingSettings.IsTelegramEnabled =
settings.IsTelegramEnabled;
            existingSettings.IsEmailEnabled = settings.IsEmailEnabled;
            _context.NotificationSettings.Update(existingSettings);
        }

        await _context.SaveChangesAsync();

        return existingSettings ?? settings;
    }
}
```

UnitOfWork

Implements the `IUnitOfWork` interface from the Domain layer:

```
public class UnitOfWork : IUnitOfWork
{
    private readonly ApplicationDbContext _context;

    public UnitOfWork(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task SaveAsync(CancellationToken cancellationToken =
default)
    {
        await _context.SaveChangesAsync(cancellationToken);
    }
}
```

> **Note**

`UnitOfWork.SaveAsync()` should be called after all data modification operations to ensure changes are persisted to the database.

Database Schema

The Data Access layer generates migrations that create the following PostgreSQL database schema:

Exchanges Table

- `id` (UUID): Primary key
- `name` (TEXT): Exchange name (unique)
- `created_at` (TIMESTAMP): When the exchange was created

Symbols Table

- `id` (UUID): Primary key
- `exchange_id` (UUID): Foreign key to Exchanges
- `symbol_name` (TEXT): Trading pair name
- `market_type` (TEXT): Market type (stored as string)
- `base_asset` (TEXT): Base asset code
- `quote_asset` (TEXT): Quote asset code
- `status` (TEXT): Trading status (stored as string)
- `price_precision` (INTEGER): Number of decimal places allowed for price
- `quantity_precision` (INTEGER): Number of decimal places allowed for quantity
- `contract_type` (TEXT): Type of contract (null for spot, otherwise stored as string)
- `delivery_date` (TIMESTAMP): Expiration date for futures contracts (nullable)
- `margin_asset` (TEXT): Asset used for margin (nullable)
- `min_quantity` (DECIMAL): Minimum quantity allowed for orders
- `min_notional` (DECIMAL): Minimum order value (price × quantity)
- `max_quantity` (DECIMAL): Maximum quantity allowed for orders

Statuses Table

- `id` (UUID): Primary key
- `symbol_id` (UUID): Foreign key to Symbols
- `symbol_status` (TEXT): New symbol status (stored as string)
- `created_at` (TIMESTAMP): When the status was created

NotificationSettings Table

- `id` (UUID): Primary key

- `is_telegram_enabled` (BOOLEAN): Whether Telegram notifications are enabled (defaults to true)
- `is_email_enabled` (BOOLEAN): Whether Email notifications are enabled (defaults to true)

Design-Time Factory

The `ApplicationDbContextDesignTimeFactory` is used to create an instance of the `ApplicationDbContext` for Entity Framework Core migrations:

```
public class ApplicationDbContextDesignTimeFactory :
IDesignTimeDbContextFactory<ApplicationDbContext>
{
    public ApplicationDbContext CreateDbContext(string[] args)
    {
        var configuration = new ConfigurationBuilder()
            .SetBasePath(Path.Combine(Directory.GetCurrentDirectory(),
$"../{nameof(InfoSymbolServer)}"))
            .AddJsonFile("appsettings.json")
            .Build();

        var connectionString =
configuration.GetConnectionString("DefaultConnection");

        var optionsBuilder = new
DbContextOptionsBuilder<ApplicationDbContext>();
        optionsBuilder
            .UseNpgsql(connectionString, b =>

b.MigrationsAssembly(AssemblyReference.Assembly.GetName().Name))
            .UseSnakeCaseNamingConvention();

        return new ApplicationDbContext(optionsBuilder.Options);
    }
}
```

This factory allows Entity Framework Core to create migrations without requiring a running application.

Configuration

Database Connection

The application uses PostgreSQL as its database, with connection details configured in `appsettings.json`:

```
"ConnectionStrings": {
  "DefaultConnection":
"Host=db;Database=infosymboldb;Username=postgres;Password=postgres;Include
ErrorDetail=true"
}
```

The connection string is retrieved in several places:

1. In `ServiceCollectionExtensions` for configuring the `ApplicationDbContext`:

```
options.UseNpgsql(
  configuration.GetConnectionString("DefaultConnection"),
  b => b.MigrationsAssembly(AssemblyReference.Assembly))
```

2. In `ApplicationDbContextDesignTimeFactory` for EF Core migrations:

```
var connectionString =
configuration.GetConnectionString("DefaultConnection");
```

Dependency Registration

All data access services are registered in the DI container through the `AddInfrastructure` extension method:

```
public static IServiceCollection AddInfrastructure(
    this IServiceCollection services,
    IConfiguration configuration,
    IHostEnvironment environment)
{
    // DbContext configuration
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseNpgsql(
            configuration.GetConnectionString("DefaultConnection"),
            b => b.MigrationsAssembly(AssemblyReference.Assembly))
    );

    // Repository registrations
    services.AddScoped<IExchangeRepository, ExchangeRepository>();
    services.AddScoped<ISymbolRepository, SymbolRepository>();
    services.AddScoped<IStatusRepository, StatusRepository>();
    services.AddScoped<INotificationSettingsRepository,
NotificationSettingsRepository>();
    services.AddScoped<IUnitOfWork, UnitOfWork>();
```

```
    return services;
}
```

By centralizing all data access registrations, the application startup remains clean and focused on composition rather than implementation details.