# InfoSymbolServer.Infrastructure.BackgroundJobs



## Overview

The Background Jobs in InfoSymbolServer implement periodic synchronization of trading symbols with the Binance exchange. This layer is responsible for:

- Keeping the local database in sync with the Binance exchange
- Tracking changes in symbol status and properties
- Notifying administrators about symbol changes
- Handling errors and retries during synchronization

## Architecture

The background jobs system uses a template method pattern with a base abstract class and concrete implementations for different market types. The system is built on top of Quartz.NET for job scheduling and management.

### Base Job Implementation

The `BaseBinanceSymbolSyncJob<TLogger>` serves as the foundation for all symbol synchronization jobs:

```
public abstract class BaseBinanceSymbolSyncJob<TLogger> : IJob where
TLogger : class
{
```

```
    private readonly ILogger<TLogger> _logger;
    private readonly ISymbolRepository _symbolRepository;
    private readonly IStatusRepository _statusRepository;
    private readonly IExchangeRepository _exchangeRepository;
    private readonly IUnitOfWork _unitOfWork;
    protected readonly IBinanceRestClient BinanceRestClient;
    protected readonly IEnumerable<INotificationService>
NotificationServices;
    private readonly IEmergencyNotificationService
_emergencyNotificationService;

    // constuctor

    protected abstract string ExchangeName { get; }
    protected abstract MarketType MarketType { get; }

    protected abstract Task<IEnumerable<object>> FetchSymbolsAsync();
    protected abstract bool MapSymbol(Symbol symbol, object
binanceSymbol);
    protected abstract string GetSymbolName(object binanceSymbol);

    public async Task Execute(IJobExecutionContext context)
    {
        _logger.LogInformation(
            "Starting {Exchange} symbols synchronization at {Time}",
ExchangeName, DateTime.UtcNow);

        try
        {
            // Tries to fetch exchange from database, if not exists - job
will not be executed.
            var exchange = await
_exchangeRepository.GetByNameAsync(ExchangeName);
            if (exchange == null)
            {
                _logger.LogWarning("{Exchange} exchange not found in the
database.", ExchangeName);
                return;
            }

            // Fetches symbols for this exchange from the database.
            var existingSymbols = await _symbolRepository
                .GetByFilterAsync(s => s.ExchangeId == exchange.Id &&
s.MarketType == MarketType);
            var existingSymbolsDict = existingSymbols
                .ToDictionary(s => s.SymbolName);

            // Initialization of lists for updated, delisted, etc. symbols

            // Process this market type symbols
```

```
            await ProcessSymbolsAsync(...);

            // Saves changes to the database if any changes were made
            await SaveSymbolsAsync(...);

            // Sends notifications for changes in symbols if any changes
were made
            if (updatedSymbols.Count > 0 || newSymbols.Count > 0 ||
removedSymbols.Count > 0)
            {
                foreach (var notificationService in NotificationServices)
                {
                    if (await
notificationService.AreNotificationsEnabledAsync())
                    {
                        await
notificationService.SendCombinedSymbolChangesNotificationAsync(
                            newSymbols, updatedSymbols, removedSymbols,
ExchangeName, MarketType.ToString());
                    }
                }
            }

            _logger.LogInformation(
                "{Exchange} symbol synchronization finished ...");
        }
        catch (ExchangeApiException ex)
        {
            _logger.LogError(ex, "Exchange API error during {Exchange}
symbol synchronization: {Message}",
                ExchangeName, ex.Message);
            await
_emergencyNotificationService.SendExchangeApiErrorNotificationAsync(
                ex.ExchangeName, $"Error during {ex.ExchangeName}
{ex.MarketType} symbols synchronization", ex);
        }
        // other catch blocks for handling specific exceptions
    }

    /// <summary>
    /// Process symbols for concrete market type.
    /// </summary>
    private async Task ProcessSymbolsAsync(
        Dictionary<string, Symbol> existingSymbolsDict,
        List<Symbol> updatedSymbols,
        List<Symbol> newSymbols,
        List<Symbol> removedSymbols,
        List<Status> statusChanges,
        HashSet<string> processedSymbols,
        Guid exchangeId)
```

```
    {
        try
        {
            // Fetches symbols from Binance API
            var binanceSymbols = await FetchSymbolsAsync();

            foreach (var binanceSymbol in binanceSymbols)
            {
                var symbolName = GetSymbolName(binanceSymbol);
                processedSymbols.Add(symbolName);

                if (existingSymbolsDict.TryGetValue(symbolName, out var
symbol))
                {
                    // Tries to update symbol if there are any changes and
if status is not RemovedByAdmin
                    TryUpdateSymbol(updatedSymbols, statusChanges,
binanceSymbol, symbol);
                }
                else
                {
                    // Creates new symbol entry in the database
                    CreateSymbol(newSymbols, statusChanges, exchangeId,
binanceSymbol);
                }
            }

            // Tries to set symbol as delisted if it doesn't exist in the
Binance API response
            foreach (var symbolName in existingSymbolsDict.Keys)
            {
                if (!processedSymbols.Contains(symbolName))
                    TrySetSymbolDelisted(existingSymbolsDict,
removedSymbols, statusChanges, symbolName);
            }
        }
        catch (Exception ex)
        {
            throw new ExchangeApiException(
                ExchangeName,
                MarketType.ToString(),
                $"Error during {ExchangeName} {MarketType} symbol
synchronization",
                ex);
        }
    }

    /// <summary>
    /// Tries to update symbol if it exists in the database.
    /// </summary>
```

```
    private void TryUpdateSymbol(
        List<Symbol> updatedSymbols, List<Status> statusChanges, object
binanceSymbol, Symbol symbol)
    {
        // Omitted for brevity.
    }

    /// <summary>
    /// Creates new symbol entry in the database.
    /// </summary>
    private void CreateSymbol(
        List<Symbol> newSymbols, List<Status> statusChanges, Guid
exchangeId, object binanceSymbol)
    {
        // Omitted for brevity.
    }

    /// <summary>
    /// Updates the status of delisted symbols to Delisted.
    /// </summary>
    private void TryUpdateDelistedSymbols(
        Dictionary<string, Symbol> existingSymbolsDict,
        List<Status> statusChanges,
        HashSet<string> processedSymbols,
        List<Symbol> removedSymbols)
    {
        // Omitted for brevity.
    }

    /// <summary>
    /// Saves changes to the database
    /// </summary>
    private async Task SaveSymbolsAsync(
        List<Symbol> updatedSymbols,
        List<Symbol> newSymbols,
        List<Symbol> removedSymbols,
        List<Status> statusChanges)
    {
        // Omitted for brevity.
    }
}
```

The base job implements the core synchronization workflow:

1. **Initialization**
   - Logs the start of synchronization
   - Retrieves the exchange from the database
   - Fetches existing symbols for the market type

2. **Symbol Processing**
   - Compares existing symbols with those from the exchange
   - Identifies new, updated, and removed symbols
   - Tracks status changes
   - Maintains a list of processed symbols
3. **Database Updates**
   - Saves all changes in a single transaction
   - Updates existing symbols
   - Adds new symbols
   - Marks removed symbols as delisted
4. **Notifications**
   - Sends notifications about symbol changes
   - Handles errors with emergency notifications
5. **Error Handling**
   - Catches and logs exchange API errors
   - Handles database errors
   - Sends emergency notifications for critical failures

## Concrete Implementation

The `BinanceSpotSymbolSyncJob` is a concrete implementation of the base job for spot market symbols:

```
[DisallowConcurrentExecution]
public class BinanceSpotSymbolSyncJob :
BaseBinanceSymbolSyncJob<BinanceSpotSymbolSyncJob>
{
    public BinanceSpotSymbolSyncJob(
        ILogger<BinanceSpotSymbolSyncJob> logger,
        ISymbolRepository symbolRepository,
        IStatusRepository statusRepository,
        IExchangeRepository exchangeRepository,
        IUnitOfWork unitOfWork,
        IBinanceRestClient binanceRestClient,
        IEnumerable<INotificationService> notificationServices,
        IEmergencyNotificationService emergencyNotificationService)
        : base(logger, symbolRepository, statusRepository,
exchangeRepository, unitOfWork,
               binanceRestClient, notificationServices,
emergencyNotificationService)
    {
    }

    protected override string ExchangeName => "BinanceSpot";
    protected override MarketType MarketType => MarketType.Spot;
```

```
    protected override async Task<IEnumerable<object>> FetchSymbolsAsync()
    {
        var response = await
BinanceRestClient.SpotApi.ExchangeData.GetExchangeInfoAsync();
        return response.Data.Symbols;
    }

    protected override bool MapSymbol(Symbol symbol, object binanceSymbol)
    {
        return BinanceSymbolMapper.MapSpotSymbol(symbol,
(BinanceSymbol)binanceSymbol);
    }

    protected override string GetSymbolName(object binanceSymbol)
    {
        return ((BinanceSymbol)binanceSymbol).Name;
    }
}
```

This implementation:

- Is decorated with `[DisallowConcurrentExecution]` to prevent multiple instances from running simultaneously
- Specifies the exchange name as "BinanceSpot" and market type as `MarketType.Spot`
- Implements the abstract methods to fetch and map symbols from the Binance Spot API endpoint
- Uses the `BinanceSymbolMapper` to convert Binance-specific fields to domain model properties

> **Note**
>
> Implementations for USDT Futures and Coin Futures jobs omitted.

## Symbol Mapping

The `BinanceSymbolMapper` class handles the conversion between Binance API models and domain entities:

```
public static class BinanceSymbolMapper
{
    public static bool MapSpotSymbol(Symbol target, BinanceSymbol source)
    {
        var updated = false;

        if (target.SymbolName != source.Name)
```

```
        {
            target.SymbolName = source.Name;
            updated = true;
        }
        if (target.MarketType != MarketType.Spot)
        {
            target.MarketType = MarketType.Spot;
            updated = true;
        }

        // other mappings

        var priceFilter = source.PriceFilter;
        var lotSizeFilter = source.LotSizeFilter;
        var notionalFilter = source.NotionalFilter;

        var pricePrecision = priceFilter != null ?
CalculatePrecision(priceFilter.TickSize) : 0;
        var quantityPrecision = lotSizeFilter != null ?
CalculatePrecision(lotSizeFilter.StepSize) : 0;

        if (target.PricePrecision != pricePrecision)
        {
            target.PricePrecision = pricePrecision;
            updated = true;
        }
        if (target.QuantityPrecision != quantityPrecision)
        {
            target.QuantityPrecision = quantityPrecision;
            updated = true;
        }

        // other filter mappings

        return updated;
    }

    // helper methods
}
```

The mapper:

- Converts Binance-specific fields to domain model properties
- Handles different property names and types
- Returns a boolean indicating if any properties were changed
- Maintains consistency across different market types

# Job Configuration

The background jobs are configured using Quartz.NET in the `ServiceCollectionExtensions` class:

```
private static void ConfigureQuarts(this IServiceCollection services,
IConfiguration configuration)
{
    services.AddQuartz(options =>
    {
        // Configures PostgreSQL as persistent jobs storage
        options.UsePersistentStore(configure =>
        {

configure.UsePostgres(configuration.GetConnectionString("DefaultConnection
")!);
            configure.UseNewtonsoftJsonSerializer();
        });

        var schedule = configuration.GetValue<string>
("BinanceSymbolSyncJobSchedule")!;

        // Register Binance symbol sync jobs
        RegisterSymbolSyncJob<BinanceSpotSymbolSyncJob>(
            options, configuration, schedule);
        RegisterSymbolSyncJob<BinanceUsdtFuturesSymbolSyncJob>(
            options, configuration, schedule);
        RegisterSymbolSyncJob<BinanceCoinFuturesSymbolSyncJob>(
            options, configuration, schedule);
    });

    services.AddQuartzHostedService(options =>
    {
        options.WaitForJobsToComplete = true;
    });
}
```

Each job is registered using a helper method:

```
private static void RegisterSymbolSyncJob<TJob>(
    IServiceCollectionQuartzConfigurator options,
    IConfiguration configuration,
    string schedule) where TJob : IJob
{
    var jobKey = JobKey.Create(typeof(TJob).Name);

    options
        .AddJob<TJob>(jobBuilder => jobBuilder
            .WithIdentity(jobKey)
            .DisallowConcurrentExecution()
```

```
            .RequestRecovery()
        )
        .AddTrigger(triggerBuilder => triggerBuilder
            .ForJob(jobKey)
            .WithIdentity($"{jobKey.Name}-Trigger")
            .WithCronSchedule(schedule, cronBuilder =>

cronBuilder.WithMisfireHandlingInstructionFireAndProceed())
        );
}
```

Key features of the job configuration:

- Uses PostgreSQL for persistent job storage
- Prevents concurrent execution of the same job
- Enables job recovery in case of failures
- Uses a common schedule for all symbol sync jobs
- Configures misfire handling to proceed with execution

# Error Handling

The background jobs implement comprehensive error handling:

1. **Exchange API Errors**
   - Catches `ExchangeApiException`
   - Logs the error with details
   - Sends emergency notification to administrators
2. **Database Errors**
   - Catches `DatabaseException`
   - Logs the error with context
   - Sends emergency notification with operation details
3. **System Errors**
   - Catches unexpected exceptions
   - Logs the error with stack trace
   - Sends emergency notification with component information

# Notification Integration

The background jobs integrate with the notification system to alert administrators about:

1. **Symbol Changes**
   - New symbols added to the exchange
   - Existing symbols updated with new properties
   - Symbols removed or delisted from the exchange

2. **System Issues**
   - Exchange API failures
   - Database errors
   - Unexpected system errors

# Configuration

The background jobs are configured in `appsettings.json`:

```
{
  "BinanceSymbolSyncJobSchedule": "0 0/15 * * * ?"  // Runs every 15
minutes
}
```

The schedule uses a cron expression to define when the jobs should run. The example above runs the jobs every 15 minutes.

# Dependency Registration

All background job services are registered in the DI container through the `AddInfrastructure` extension method:

```
public static IServiceCollection AddInfrastructure(
    this IServiceCollection services,
    IConfiguration configuration,
    IHostEnvironment environment)
{
    // Background job configuration
    services.ConfigureQuarts(configuration);

    // Register Binance API client
    services.AddBinance();

    return services;
}
```