İçindekiler

1. Introduction 1

2. System Overview 1

3. Requirements 1

3.1 Functional Requirements 1

3.2 Non-Functional Requirements 2

4. Design 2

4.1. Architecture 2

4.2. Key Classes and Interfaces 2

5. Implementation Plan 4

6. Conclusion 4

1. Introduction

This document provides an business analysis of the configuration management system to be implemented. The goal is to create a dynamic configuration library that allows accessing and updating configuration settings without requiring application restarts. The settings are stored in a database and are refreshed periodically.

2. System Overview

The system comprises three main components:

1. **Configuration Library (DLL)**: Provides dynamic access to configuration settings.
2. **Data Access Layer**: Manages database interactions.
3. **User Interface**: Allows users to view and update configuration settings.

3. Requirements

3.1 Functional Requirements

I. Configuration Library Initialization:
   - Must initialize with three parameters:
     - `applicationName`: The name of the application using the library.
     - `connectionString`: Database connection information.
     - `refreshTimerIntervalInMs`: Interval in milliseconds for refreshing configuration settings.

II. Configuration Retrieval:
   - Must provide a method `T GetValue<T>(string key)` to fetch configuration values.
   - Only active configurations (`IsActive = 1`) should be returned.

III. Dynamic Updates:
   - Must periodically refresh configuration settings from the database.

- Should handle new configuration entries and updates without requiring application restarts.

IV. Error Handling:
- Must continue operating with the last known good configuration if the database becomes unavailable.

V. User Interface:
- Provide a web interface for viewing, adding, and updating configuration settings.
- Allow filtering configurations by name.

3.2 Non-Functional Requirements

I. Performance:
- The library should have minimal performance overhead.
- Configuration refresh should be efficient to avoid performance degradation.

II. Scalability:
- The system should handle a large number of configuration settings and high request rates.

III. Security:
- Secure database access and protect sensitive configuration data.
- Ensure only authorized users can access and modify configuration settings.

IV. Reliability:
- Ensure high availability and fault tolerance in configuration retrieval and updates.

4. Design

4.1. Architecture

The system follows a layered architecture with the following layers:

I. Business Layer:
- Manages configuration logic and periodic refresh operations.
- Contains the `ConfigurationReader` class.

II. Data Access Layer:
- Handles database operations using the repository pattern.
- Includes generic repository interfaces and implementations.

III. Presentation Layer:
- Provides a web interface for managing configurations.
- Uses ASP.NET Core for the web application.

4.2. Key Classes and Interfaces

ConfigurationReader

public class ConfigurationReader : IConfigurationService

```
{
   private readonly string _applicationName;
   private readonly int _refreshTimerInterval;
   private readonly string _connectionString;
   private readonly IConfigurationEntityRepository _configurationEntityRepository;
   private Dictionary<string, ConfigurationEntity> _configurationDictionary;
   public ConfigurationReader(string applicationName, int refreshTimerInterval, string connectionString)
   {
      _applicationName = applicationName;
      _refreshTimerInterval = refreshTimerInterval;
      _connectionString = connectionString;
   }
   public IConfigurationEntityRepository configurationEntityRepository => _configurationEntityRepository;
   public async Task TAddAsync(ConfigurationEntity entity)
   {
      entity.IsActive = true; //yeni oluşturulan entity'lerin default olarak aktif olmasını sağlıyorum.
      await _configurationEntityRepository.AddAsync(entity);
```

```
    }
    public IEnumerable<ConfigurationEntity> TFindPredicate(Expression<Func<ConfigurationEntity, bool>> predicate)
    {
        return _configurationEntityRepository.FindPredicate(predicate);
    }
    public IEnumerable<ConfigurationEntity> TGetAll()
    {
        return _configurationEntityRepository.GetAll().Where(c => c.IsActive != false).ToList(); //aktif olanları getiriyorum.
    }
    public ConfigurationEntity TGetById(int id)
    {
        return _configurationEntityRepository.GetById(id);
    }
    public T TGetValue<T>(string key)
    {
        return _configurationEntityRepository.GetValue<T>(key);
    }
    public async Task TUpdateAsync(ConfigurationEntity entity)
    {
        entity.IsActive = true; //güncellenen entity'lerin default olarak aktif olmasını sağlıyorum.
        await _configurationEntityRepository.UpdateAsync(entity);
    }
}
```

GenericRepository

```
public class GenericRepository<T> : IGenericRepository<T> where T : class
{
    private readonly AppDbContext _context;

    public GenericRepository(AppDbContext context)
    {
        _context = context;
    }
    public async Task AddAsync(T entity)
    {
        await _context.AddAsync(entity);
        _context.SaveChanges();
    }
    public IEnumerable<T> FindPredicate(Expression<Func<T, bool>> predicate)
    {
        return _context.Set<T>().Where(predicate);
    }
    public IEnumerable<T> GetAll()
    {
        return _context.Set<T>().ToList();
    }
    public T GetById(int id)
    {
        return _context.Set<T>().Find(id);
    }
    public async Task UpdateAsync(T entity)
    {
        _context.Update(entity);
        await _context.SaveChangesAsync();
    }
}
```

5. Implementation Plan

I. Setup the Project:
   - Create a new .NET 8 solution.
   - Create class library projects for Business, DataAccess, and Entities.
   - Add necessary NuGet packages (e.g., Microsoft.EntityFrameworkCore).

II. Define Entities:
   - Create `ConfigurationEntity` class in the Entities project.

III. Data Access Layer:
   - Implement IGenericRepository and GenericRepository.

IV. Business Layer:
   - Implement ConfigurationReader class.

V. Web Application:
   - Create an ASP.NET Core project.
   - Implement necessary controllers and views for managing configurations.

VI. Configuration and Dependency Injection:
   - Configure services and dependency injection in `Program.cs`.

VII. Testing:
   - Write unit tests for the Business and Data Access layers.
   - Ensure all requirements are met.

VIII. Documentation:
   - Document the code and provide usage examples.

IX. Deployment:
   - Deploy the web application to a server.

6. Conclusion

This document outlines the analysis, requirements, design, and implementation plan for the configuration management system. The proposed system ensures dynamic and efficient management of configuration settings with minimal impact on application performance and reliability.