

# Fl.Configuration.Extensions

A .NET library that provides functional programming extensions for Microsoft's Configuration system, offering type-safe configuration binding with functional programming patterns.

## Features

- **Type-Safe Configuration Binding:** Automatically maps configuration sections to strongly-typed objects
- **Functional Programming Support:** Integration with LanguageExt Option monad for null-safe operations
- **Convention-Based:** Uses type name as configuration section key
- **Dual APIs:** Both strict (throws on missing) and safe (returns Option) configuration retrieval
- **Multi-Framework Support:** Targets .NET 8.0, 9.0, and 10.0

## Installation

```
dotnet add package Fl.Configuration.Extensions
```

## Usage

### Configuration Class Setup

```
public record DatabaseConfig
{
    public string ConnectionString { get; init; }
    public int Timeout { get; init; }
}
```

### Configuration File (appsettings.json)

```
{
    "DatabaseConfig": {
        "ConnectionString": "Server=localhost;Database=MyDb;",
        "Timeout": 30
    }
}
```

### Required Configuration (Strict)

Use `GetRequiredConfiguration<T>()` when the configuration section must be present:

```
var dbConfig = configuration.GetRequiredConfiguration<DatabaseConfig>();
// Throws KeyNotFoundException if "DatabaseConfig" section is missing
Console.WriteLine($"Connection: {dbConfig.ConnectionString}");
```

## Optional Configuration (Safe)

Use `GetConfiguration<T>()` for optional configuration sections:

```
var optionalConfig = configuration.GetConfiguration<DatabaseConfig>();

// Functional pattern matching:
optionalConfig.Match(
    /* Some: */ config => {
        Console.WriteLine($"Connection: {config.ConnectionString}");
        // Use the configuration
    },
    /* None: */ () => {
        Console.WriteLine("Configuration not found, using defaults");
        // Handle missing configuration gracefully
    }
);

// Or check if present:
if (optionalConfig.IsSome)
{
    var config = optionalConfig.ValueUnsafe();
    // Use config...
}
```

## API Reference

### Extension Methods

#### `GetRequiredConfiguration<T>()`

```
public static T GetRequiredConfiguration<T>(this IConfiguration configuration) where
T : class, new()
```

- **Returns:** Instance of type `T` populated from configuration
- **Throws:** `KeyNotFoundException` if the configuration section is missing
- **Section Key:** Uses `typeof(T).Name` as the configuration section key

## GetConfiguration<T>()

```
public static Option<T> GetConfiguration<T>(this IConfiguration configuration) where  
T : class, new()
```

- **Returns:** Option<T> - Some<T> if found, None if missing
- **Section Key:** Uses `typeof(T).Name` as the configuration section key
- **Safe:** Never throws exceptions, use functional pattern matching

## Dependencies

- Microsoft.Extensions.Configuration ( $\geq 10.0.3$ )
- Microsoft.Extensions.Configuration.Abstractions ( $\geq 10.0.3$ )
- Microsoft.Extensions.Configuration.Binder ( $\geq 10.0.3$ )
- Fl.Functional.Utils ( $\geq 0.1.0$ )
- LanguageExt: Functional programming library for Option monad

## Target Frameworks

- .NET 8.0
- .NET 9.0
- .NET 10.0

## Ideal Use Cases

1. **Microservices Architecture:** Clean, type-safe configuration binding
2. **Functional Programming:** Integration with functional codebases using LanguageExt
3. **Configuration Validation:** Early detection of missing required configuration
4. **Optional Feature Configuration:** Safe handling of optional configuration sections
5. **Strongly-Typed Configuration:** Automatic mapping without manual binding code

## Benefits

- **Type Safety:** Compile-time guarantees for configuration structure
- **Functional Programming:** Option monad prevents null reference exceptions
- **Convention-Based:** No magic strings, uses type names as section keys
- **Clean API:** Simple, expressive methods for common configuration patterns
- **Multi-Framework:** Supports latest .NET versions

# Namespace Fl.Configuration.Extensions

## Classes

[ConfigurationExtensions](#)