**Конфигурация**

**Основы конфигурации**

Важную роль в приложении играет конфигурация, которая определяет базовые настройки приложения. Приложение ASP.NET Core может получать конфигурационные настройки из следующих источников:

* Аргументы командной строки
* Переменные среды окружения
* Объекты .NET в памяти
* Файлы (json, xml, ini)
* Azure
* Можно использовать свои кастомные источники и под них создавать провайдеры конфигурации

**Интерфейс IConfiguration**

Конфигурация приложения в ASP.NET Core представляет объект интерфейса **IConfiguration**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public interface IConfiguration  {      string this[string key] { get; set; }      IEnumerable<IConfigurationSection> GetChildren();      IChangeToken GetReloadToken();      IConfigurationSection GetSection(string key);  } |

Данный интерфейс содержит следующие компоненты:

* **this [string key]**: индексатор, через который можно получить по ключу значение параметра конфигурации. Стоит отметить, что и ключ, и значение параметра конфигурации представляет собой объект типа string
* **GetChildren()**: возвращает набор подсекций текущей секции конфигурации в виде объекта IEnumerable<IConfigurationSection>
* **GetReloadToken()**: возвращает объект IChangeToken, который применяется для отслеживания изменения конфигурации
* **GetSection(string key)**: возвращает секцию конфигурации, которая соответствует ключу key

Также конфигурация может быть представлена интерфейсом **IConfigurationRoot**, который наследуется от IConfiguration:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IConfigurationRoot : IConfiguration  {      IEnumerable<IConfigurationProvider> Providers { get; }      void Reload();  } |

* Свойство **Providers** возвращает коллекцию применяемых провайдеров конфигурации. Каждый провайдер конфигурации представляет объект IConfigurationProvider
* Метод **Reload()** перезагружает значения из всех применяемых источников конфигурации

Итак, объект IConfiguration по сути хранит все конфигурационные настройки в виде набора пар "ключ"-"значение".

**Получение данных конфигурации**

В приложении настройки конфигурации хранятся в свойстве **Configuration** объекта **WebApplication**. Соответственно через это свойство мы можем установить или получить настройки конфигурации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    // установка настроек конфигурации  app.Configuration["name"] = "Tom";  app.Configuration["age"] = "37";    app.Run(async (context) =>  {      // получение настроек конфигураци      string name = app.Configuration["name"];      string age = app.Configuration["age"];      await context.Response.WriteAsync($"{name} - {age}");  });    app.Run(); |

Для установки передаем значение по определенному ключу:

|  |  |
| --- | --- |
| 1 | app.Configuration["name"] = "Tom"; |

В данном случае в конфигурацию устанавливается элемент с ключом "name". Он получает в качестве значения строку "Tom". При этом неважно, что изначально в конфигурации нет настройки с именем "name". Если ее нет, она добавляется. Если она уже существует, ее значение переустанавливается.

И затем мы можем получить нужную настройку по ключу:

|  |  |
| --- | --- |
| 1 | string name = app.Configuration["name"]; |

Стоит обратить внимание, что в качестве значений передаются строки. Поэтому в случае с сохранением в конфигурации условного возраста по ключу "age" в качестве значения также передается строка:

|  |  |
| --- | --- |
| 1 | app.Configuration["age"] = "37"; |

**Добавление источника конфигурации**

В примере выше настройки конфигурации устанавливались по отдельности - сначала настройка "name", затем настройка "age". Однако если настроек много или если они имеют сложную структуру, гораздо проще установить их одним скопом, особенно в случае, когда настройки хранятся в файле json, xml или берутся из какого-то другого источника конфигурации. Для добавления источника конфигурации в приложении можно применять свойство **Configuration** объекта WebApplicationBuilder. Это свойство представляет класс **ConfigurationManager**, для которого определен ряд методов для добавления конфигурации.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var builder = WebApplication.CreateBuilder();    builder.Configuration.AddInMemoryCollection(new Dictionary<string, string>  {      {"name", "Tom"},      {"age", "37"}  });    var app = builder.Build();    app.Run(async (context) =>  {      // получение настроек конфигураци      string name = app.Configuration["name"];      string age = app.Configuration["age"];      await context.Response.WriteAsync($"{name} - {age}");  });    app.Run(); |

Здесь для добавления конфигурации применяется метод **AddInMemoryCollection()**. Этот метод добавляет набор настроек в виде коллекции пар ключ-значение:

|  |  |
| --- | --- |
| 1 | public static IConfigurationBuilder AddInMemoryCollection(this IConfigurationBuilder configurationBuilder, IEnumerable<KeyValuePair<string, string>> initialData) |

Как раз таким набором является стандартный словарь Dictionary<string, string>

После добавления источника конфигурации мы также можем получить настройки конфигурации через свойство app.Configuration.

**Получение конфигурации через Dependency Injection**

Конфигурация приложения в виде объекта IConfiguration представляет один из сервисов, которые добавляются в приложение по умолчанию. Соответственно всю конфигурацию приложения мы можем получить как и любой другой сервис через механизм внедрения зависимостей. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | var builder = WebApplication.CreateBuilder();    var app = builder.Build();    // установка настроек конфигурации  app.Configuration["name"] = "Tom";  app.Configuration["age"] = "37";    // через механизм внедрения зависимостей получим сервис IConfiguration  app.Map("/", (IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

В данном случае обработчик запроса в методе app.Map() в качестве параметра appConfig получает сервис IConfiguration - по сути это тот же самый объект IConfiguration, что и app.Configuration. Подобным способом мы можем получить конфигурацию в других частях приложения, особенно там, где объект WebApplication нам недоступен.

**Нефайловые провайдеры конфигурации**

**Загрузка аргументов командной строки**

Для загрузки аргументов командной строки в конфигурацию приложения применяется провайдер **CommandLineConfigurationProvider**. Для применения этого провайдера у объекта ConfigurationManager (свойство Configuration объекта WebApplicationBuilder) вызывается метод **AddCommandLine()**, в который передаются аргументы командной строки. Но поскольку приложение по умолчанию загружает аргументы командной строки в объект конфигурации, который передается в классы приложения через dependency injection, то нам нет смысла явным образом его вызывать.

Каждый аргумент командной строки, который мы хотим использовать в качестве параметра конфигурации, должен представлять пару ключ-значение. Есть разные способы определения таких параметров. Самый распространненый - после ключа после знака равно = указывается значение, а через пробел последующие параметры:

|  |  |
| --- | --- |
| 1 | key1=value1 key2=value2 key3=value3 |

Также можно использовать другие способы:

|  |  |
| --- | --- |
| 1  2  3 | key1=value1 --key2=value2 /key3=value3  --key1 value1 /key2 value2  key1= key2=value2 |

Перед ключом можно указать префикс -- или /, тогда между ключом и значением можно опустить знак равно = и оставить просто пробел.

Также если мы хотим указать ключ, но не хотим указывать значение, то после знака равно можно не указывать значение.

**Тестирование в Visual Studio**

Для тестирования передачи параметров командной строки изменим в проекте в папке **Properties** файл **launchSettings.json**.

По умолчанию он выглядит примерно следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | {    "iisSettings": {      "windowsAuthentication": false,      "anonymousAuthentication": true,      "iisExpress": {        "applicationUrl": "[http://localhost:56234](http://localhost:56234/)",        "sslPort": 44384      }    },    "profiles": {      "HelloApp": {        "commandName": "Project",        "launchBrowser": true,        "environmentVariables": {          "ASPNETCORE\_ENVIRONMENT": "Development"        },        "applicationUrl": "[https://localhost:7256;http://localhost:5256](https://localhost:7256;http/localhost:5256)",        "dotnetRunMessages": true      },      "IIS Express": {        "commandName": "IISExpress",        "launchBrowser": true,        "environmentVariables": {          "ASPNETCORE\_ENVIRONMENT": "Development"        }      }    }  } |

Перейдем в этом файле к элементу "profiles". Этот элемент по умолчанию содержит два профиля, которые могут применяться для запуска. Один профиль называется по имени проекта (в моем случае HelloApp). Второй профиль называется "IIS". Выберем профиль, который мы используем для запуска проекта и добавим в этот элемент строку

|  |  |
| --- | --- |
| 1 | "commandLineArgs": "name=Bob age=37", |

Параметр "commandLineArgs" позволяет определить данные командной строки, которые будут передаваться приложению. В данном случае предполагается, что через командную строку будут передаваться параметр name со значением "Bob" и параметр age со значением "37". Например, я запускаю проект как консольное приложение, поэтому я изменяю профиль по имени проекта - HelloApp. Который после изменения будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | "HelloApp": {        "commandName": "Project",        "launchBrowser": true,        "commandLineArgs": "name=Bob age=37",        "environmentVariables": {          "ASPNETCORE\_ENVIRONMENT": "Development"        },        "applicationUrl": "[https://localhost:7256;http://localhost:5256](https://localhost:7256;http/localhost:5256)",        "dotnetRunMessages": true      }, |

Определим в приложении простейший код для получения значений конфигурации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var builder = WebApplication.CreateBuilder(args);  var app = builder.Build();    app.Map("/", (IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

Здесь важно отметить передачу параметров командной строки - **args** в метод WebApplication.CreateBuilder(args). Благодаря этому приложение автоматически может подхватить параметры командной строки. И после запуска мы увидим указанные значения параметров конфигурации:

**Запуск через консоль**

Используя CLI, мы можем запустить приложение из командной строки и передать ему параметры.

Итак, удалим выше определенные параметры и выполним перестроение проекта. Откроем командную строку и перейдем в консоли с помощью команды cd в папку проекта. В моем случае это папка C:\Users\Eugene\Source\Repos\CSharp\ASPNET\HelloApp\HelloApp.

Далее введем следующую команду:

dotnet run name=Tom age=35

В итоге приложение подхватит переданные данные:

**Программная симуляция аргументов командной строки**

Также мы можем на уровне кода симулировать передачу параметров командной строки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | string[] commandLineArgs = { "name=Alice", "age=29" };  // псевдопараметры командной строки  var builder = WebApplication.CreateBuilder(commandLineArgs);  var app = builder.Build();    app.Map("/", (IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

**Применение метода AddCommandLine**

Также можно было бы передать параметры командной строки через метод **AddCommandLine()**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var builder = WebApplication.CreateBuilder();  string[] commandLineArgs = { "name=Sam", "age=25" };  // псевдопараметры командной строки  builder.Configuration.AddCommandLine(commandLineArgs);  // передаем параметры в качестве конфигурации    var app = builder.Build();  app.Map("/", (IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

**Переменные среды окружения как источник конфигурации**

Для загрузки переменных среды окружения в качестве параметров конфигурации применяется провайдер **EnvironmentVariablesConfigurationProvider**. Для его использования у объекта ConfigurationManager вызывается метод **AddEnvironmentVariables()**. Однако в реальности вряд ли его придется часто использовать, так как среда ASP.NET Core уже загружает переменные среды окружения в объект конфигурации по умолчанию.

Например, получим переменную окружения "JAVA\_HOME", которая указывает на папку установки java sdk, если эта переменная определена:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    app.Map("/", (IConfiguration appConfig) => $"JAVA\_HOME: {appConfig["JAVA\_HOME"] ?? "not set"}");    app.Run(); |

Явное применение метода AddEnvironmentVariables():

|  |  |
| --- | --- |
| 1  2  3  4 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddEnvironmentVariables(); |

**Хранение конфигурации в памяти**

Провайдер **MemoryConfigurationProvider** позволяет использовать в качестве конфигурации коллекцию IEnumerable<KeyValuePair<string, string>>, которая хранит данные в виде пары ключ-значение (пример - объект Dictionary). Для добавления источника конфигурации применяется метод **AddInMemoryCollection()**, в который передается словарь конфигурационных настроек:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddInMemoryCollection(new Dictionary<string, string>  {      {"name", "Tom"},      {"age", "37"}  });    app.Map("/", (IConfiguration appConfig) =>  {      var name = appConfig["name"];      var age = appConfig["age"];      return $"{name} - {age}";  });    app.Run(); |

**Конфигурация в файлах JSON, XML и Ini**

**Конфигурация в JSON**

Как правило, для хранения конфигурации в приложении ASP.NET Core используются файлы json. Для работы с файлами json применяется провайдер **JsonConfigurationProvider**, а для загрузки конфигурации из json применяется метод расширения **AddJsonFile()**.

По умолчанию в проекте уже есть файл конфигурации json - *appsettings.json*, а также *appsettings.Development.json*, которые загружаются по умолчанию в приложении и которые мы можем использовать для хранения конфигурационных настроек.

Например, код файла appsettings.json:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | {    "Logging": {      "LogLevel": {        "Default": "Information",        "Microsoft.AspNetCore": "Warning"      }    },    "AllowedHosts": "\*"  } |

Здесь определяются настройки логгирования (элемент "Logging") и разрешенные хосты (элемент "AllowedHosts"). Одни элементы могут иметь вложенные элементы. Аналогичным образом можно задать другие необходимые настройки или удалить ранее определенные.

Однако в данном случае для примера возьмем новый файл json. Итак, добавим в проект новый файл **config.json**:

И определим в нем следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4 | {    "person": "Tom",    "company": "Microsoft"  } |

Здесь задается два элемента с ключами "person" и "company". Используем эти настройки в приложении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("config.json");    app.Map("/", (IConfiguration appConfig) => $"{appConfig["person"]} - {appConfig["company"]}");    app.Run(); |

Для установки конфигурации из json-файла название файла передается в метод AddJsonFile().

При определении настроек в файле json нам надо учитывать, что они должны иметь уникальные ключи. Но при этом мы можем использовать для конфигурации более чем одного файла:

|  |  |
| --- | --- |
| 1  2  3 | builder.Configuration                  .AddJsonFile("config.json")                  .AddJsonFile("otherconfig.json"); |

И если во втором файле есть настройки, которые имеют тот же ключ, что и настройки первого файла, то происходит переопределение настроек: настройки из второго файла заменяют настройки первого.

Но json может хранить также более сложные по составу объекты, например:

|  |  |
| --- | --- |
| 1  2  3  4 | {    "person": {"profile": {"name": "Tomas", "email":  "tom@gmail.com"}},    "company": { "name": "Microsoft"}  } |

И чтобы обратиться к этой настройке, нам надо использовать знак двоеточия для обращения к иерархии настроек:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("config.json");    app.Map("/", (IConfiguration appConfig) =>  {      var personName = appConfig["person:profile:name"];      var companyName = appConfig["company:name"];      return $"{personName} - {companyName}";  });    app.Run(); |

**Конфигурация в XML**

За использование конфигурации в XML-файле отвечает провайдер **XmlConfigurationProvider**. Для загрузки xml-файла применяется метод расширения **AddXmlFile()**.

Итак, добавим в проект новый xml-файл, который назовем **config.xml**. Затем изменим его код на следующий:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <?xml version="1.0" encoding="utf-8" ?>  <configuration>    <person>Tom</person>    <company>Microsoft</company>  </configuration> |

Здесь определены два элемента person и company, которые представляют конфигурационные настройки.

Обратите внимание, что у файла xml в свойствах должно быть выставлено копирование при компиляции в выходную папку приложения:

Теперь используем этот файл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddXmlFile("config.xml");    app.Map("/", (IConfiguration appConfig) => $"{appConfig["person"]} - {appConfig["company"]}");    app.Run(); |

Если у нас файл конфигурации имеет разные уровни вложенности, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <?xml version="1.0" encoding="utf-8" ?>  <configuration>    <person>        <profile>            <name>Tomas</name>            <email>toma@gmail.com</email>        </profile>    </person>    <company>        <name>Microsoft</name>    </company>  </configuration> |

то мы можем обращаться к этим уровням также, как и в файле json:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | app.Map("/", (IConfiguration appConfig) =>  {      var personName = appConfig["person:profile:name"];      var companyName = appConfig["company:name"];      return $"{personName} - {companyName}";  }); |

**Конфигурация в ini-файлах**

Для работы с конфигурацией INI применяется провайдер **IniConfigurationProvider**. А для загрузки конфигурации из INI-файла нам надо использовать метод расширения **AddIniFile()**.

Например, добавим в проект текстовый файл и переименуем его в **config.ini**.

Определим в этом файле следующее содержимое:

|  |  |
| --- | --- |
| 1  2 | person="Bob"  company="Microsoft" |

Подключим конфигурацию из этотого файла в приложении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddIniFile("config.ini");    app.Map("/", (IConfiguration appConfig) => $"{appConfig["person"]} - {appConfig["company"]}");    app.Run(); |

**Конфигурация по умолчанию и объединение конфигурации**

Применяемый для создания приложения объект **WebApplicationBuilder** уже предоставляет некоторую конфигурацию по умолчанию. Посмотрим, что представляет подключаемая по умолчанию конфигурация:

Эта конфигурация загружается из разных источников в следующем порядке:

* Сначала в качестве провайдера конфигурации устанавливается объект класса **ChainedConfigurationProvider**, который добавляет объект IConfiguration в качестве источника конфигурации и который фактически соединяет все далее применяемые провайдеры конфигурации в одну цепочку.
* Далее загружается конфигурация из файла **appsettings.json**
* Затем загружается конфигурация из файла **appsettings.[Environment].json**, где [Environment] передает название окружения, например, **appsettings.Production.json** и **appsettings.Development.json**, благодаря чему мы можем задать конфигурацию для разных состояний проекта
* Если проект запущен в окружении с именем "Development", загружается конфигурация App secrets (секретные настройки)
* Затем загружаются переменные среды
* В конце загружаются аргументы командной строки

Как мы видим, объект конфигурации может использовать одновременно сразу несколько источников и провайдеров конфигурации. Все подключаемые источники конфигурации считываются в том порядке, в котором они были добавлены. Если разные источники конфигурации содержат одинаковые ключи, то используется значение, последнего добавленного источника конфигурации. Например, в данном случае значения из appsettings.Develoment.json перекроют значения с теми же ключами из appsettings.json.

И в своем приложении мы можем также объединять конфигурационные настройки из разных источников:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration      .AddJsonFile("config.json")      .AddXmlFile("config.xml")      .AddIniFile("config.ini")      .AddInMemoryCollection(new Dictionary<string, string>  {      { "name", "Sam"},      { "age", "32"}  }); ;    app.Map("/", (IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

**Анализ конфигурации**

Для работы с конфигурацией интерфейс IConfiguration определяет следующие методы:

* **GetSection(name)**: возвращает объект IConfiguration, который представляет только определенную секцию name
* **GetChildren()**: возвращает все подсекции текущего объекта конфигурации в виде набора объектов IConfiguration
* **GetReloadToken()**: возвращает токен - объект IChangeToken, который используется для уведомления при изменении конфигурации
* **GetConnectionString(name)**: эквивалентен вызову GetSection("ConnectionStrings")[name] и предназначается непосредственно для работы со строками подключения к различным базам даных
* **[key]**: индексатор, который позволяет получить по определенному ключу key хранящееся значение

Например, если у нас есть следующая конфигурация в файле config.json:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | {    "ConnectionStrings": {      "DefaultConnection": "Main database",      "UsersContext": "Users database"    }  } |

То есть мы можем с помощью метода GetSection() получить отдельные секции и их значения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("config.json");    app.Map("/", (IConfiguration appConfig) =>  {      IConfigurationSection connStrings = appConfig.GetSection("ConnectionStrings");      string defaultConnection = connStrings.GetSection("DefaultConnection").Value;        return defaultConnection ;  });    app.Run(); |

Каждая отдельная секция представляет объект **IConfigurationSection**. Если секция содержит другие секции, то также можем вызвать у ней метод GetSection(). Если же секция содержит только значение, то оно доступно через свойство Value.

Также мы могли бы использовать один вызов GetSection(), передав ему полный путь к нужной секции:

|  |  |
| --- | --- |
| 1 | string defaultConnection = appConfig.GetSection("ConnectionStrings:DefaultConnection").Value; |

Вложенные секции от родительских отделяются двоеточием.

Также мы могли бы получить нужное значение, используя индексаторы:

|  |  |
| --- | --- |
| 1 | string defaultConnection = appConfig["ConnectionStrings:DefaultConnection"]; |

Ну и кроме того, для работы непосредственно с секцией "ConnectionStrings" нам доступен метод GetConnectionString():

|  |  |
| --- | --- |
| 1 | string con = appConfig.GetConnectionString("DefaultConnection"); |

Используя выше рассмотренные методы, мы можем провести анализ всего файла конфигурации. Например, пусть в проекте определен следующий конфигурационный файл **project.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | {    "projectConfig": {      "dependencies": {        "Microsoft.Extensions.Configuration": "1.0.0",        "Microsoft.Extensions.Configuration.Json": "1.0.0",        "Microsoft.NETCore.App": {          "version": "1.0.1",          "type": "platform"        },        "Microsoft.AspNetCore.Diagnostics": "1.0.0",          "Microsoft.AspNetCore.Server.IISIntegration": "1.0.0",        "Microsoft.AspNetCore.Server.Kestrel": "1.0.1",        "Microsoft.Extensions.Logging.Console": "1.0.0"      },        "tools": {        "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"      },        "frameworks": {        "netcoreapp1.0": {          "imports": [            "dotnet5.6",            "portable-net45+win8"          ]        }      },        "buildOptions": {        "emitEntryPoint": true,        "preserveCompilationContext": true      },        "runtimeOptions": {        "configProperties": {          "System.GC.Server": true        }      },        "publishOptions": {        "include": [          "wwwroot",          "web.config"        ]      },        "scripts": {        "postpublish": [ "dotnet publish-iis --publish-folder %publish:OutputPath% --framework %publish:FullTargetFramework%" ]      }    }  } |

Проанализируем и выведем все его содержимое в браузере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("project.json");    app.Map("/", (IConfiguration appConfig) => GetSectionContent(appConfig.GetSection("projectConfig")));    app.Run();    string GetSectionContent(IConfiguration configSection)  {      System.Text.StringBuilder contentBuilder = new();      foreach (var section in configSection.GetChildren())      {          contentBuilder.Append($"\"{section.Key}\":");          if (section.Value == null)          {              string subSectionContent = GetSectionContent(section);              contentBuilder.Append($"{{\n{subSectionContent}}},\n");          }          else          {              contentBuilder.Append($"\"{section.Value}\",\n");          }      }      return contentBuilder.ToString();  } |

**Создание провайдера конфгурации**

Фреймворк ASP.NET Core по умолчанию предоставляет богатый функционал для работы с конфигурацией - для работы с json, xml и так далее. Однако, в каких-то ситуациях этого может быть недостаточно. И в этом случае мы можем определить свои источники и провайдеры конфигурации.

Создание конфигурации вовлекает три компонента:

**IConfigurationSource** (определяет источник конфигурации),

**ConfigurationProvider** (сам провайдер конфигурации)

и некий класс, который добавляет метод расширения к объекту IConfiguration.

Допустим, мы хотим хранить конфигурацию в простом текстовом файле. И для этого добавим новый класс, который назовем **TextConfigurationProvider**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class TextConfigurationProvider : ConfigurationProvider  {      public string FilePath { get; set; }      public TextConfigurationProvider(string path)      {          FilePath = path;      }      public override void Load()      {          var data = new Dictionary<string, string>(StringComparer.OrdinalIgnoreCase);          using (StreamReader textReader = new StreamReader(FilePath))          {              string? line;              while ((line = textReader.ReadLine()) != null)              {                  string key = line.Trim();                  string? value = textReader.ReadLine() ?? "";                  data.Add(key, value);              }          }          Data = data;      }  } |

Класс TextConfigurationProvider будет представлять провайдер конфигурации и поэтому должен быть унаследован от класса ConfigurationProvider.

В этом классе с помощью StreamReader происходит считывание текстового файла и добавление данных в словарь data. Для загрузки данных переопределяется метод Load(), определенный в базовом классе.

В данном случае мы предполагаем, что на первой строке будет располагаться ключ, а на следующей строке значение. И после считывания эти ключ и значение будут попадать в словарь data.

После считывания словарь data присваивается свойству **Data**, которое унаследовано от ConfigurationProvider. Это свойство как раз и хранит все те конфигурационные настройки, которые потом используются в программе.

И чтобы получить путь к файлу, он передается через параметр конструктора.

Далее нам надо обращаться к этому провайдеру. Для этого определим класс источника конфигурации, который назовем **TextConfigurationSource**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class TextConfigurationSource : IConfigurationSource  {      public string FilePath { get; }      public TextConfigurationSource(string filename)      {          FilePath = filename;      }      public IConfigurationProvider Build(IConfigurationBuilder builder)      {          // получаем полный путь для файла          string filePath = builder.GetFileProvider().GetFileInfo(FilePath).PhysicalPath;          return new TextConfigurationProvider(filePath);      }  } |

Источник конфигурации должен реализовать интерфейс **IConfigurationSource**, и в частности, его метод Build. В этот метод в качестве параметра передается строитель конфигурации.

В данном случае этот объект нам позволяет получить полный путь к текстовому файлу. Краткое название файла (относительный путь) передается в класс источника через конструктор и хранится в свойстве FilePath. После создания полного пути к файлу этот путь передается в конструктор TextConfigurationProvider.

Чтобы задействовать функционал источника конфигурации создадим вспомогательный класс **TextConfigurationExtensions**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public static class TextConfigurationExtensions  {      public static IConfigurationBuilder AddTextFile(          this IConfigurationBuilder builder, string path)      {          if (builder == null)          {              throw new ArgumentNullException(nameof(builder));          }          if (string.IsNullOrEmpty(path))          {              throw new ArgumentException("Путь к файлу не указан");          }            var source = new TextConfigurationSource(path);          builder.Add(source);          return builder;      }  } |

Этот класс определяет для объекта IConfigurationBuilder метод расширения **AddTextFile()**, в котором создается источник конфигурации TextConfigurationSource, который затем добавляется к строителю конфигурации:

|  |  |
| --- | --- |
| 1  2 | var source = new TextConfigurationSource(path);  builder.Add(source); |

Теперь добавим в проект сам файл конфигурации. Пусть он будет называться **config.txt** и будет иметь следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4 | name  Tom  age  33 |

То есть на первой строке идет ключ настройки, а на второй ее значение и так далее.

Таким образом, у нас получится следующая структура проекта:

И в конце изменим код файла **Program.cs** для использования конфигурации из текстового файла:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddTextFile("config.txt");    app.Map("/",(IConfiguration appConfig) => $"{appConfig["name"]} - {appConfig["age"]}");    app.Run(); |

И при обращении к приложению будут использованы настройки из текстового файла:

**Проекция конфигурации на классы**

Фреймворк ASP.NET Core позволяет проецировать конфигурационные настройки на классы C#.

Например, определим в проекте новый файл **person.json**, который будет хранить данные пользователя:

|  |  |
| --- | --- |
| 1  2  3  4 | {    "name": "Tom",    "age": "22"  } |

Под эти данные определим в проекте класс **Person**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; } = 0;  } |

Теперь свяжем конфигурацию из файла **person.json** с объектом класса **Person**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("person.json");  var tom = new Person();  app.Configuration.Bind(tom);    // связываем конфигурацию с объектом tom    app.Run(async (context) => await context.Response.WriteAsync($"{tom.Name} - {tom.Age}"));    app.Run();      public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; } = 0;  } |

Ключевой момент заключается в применении метода **Bind**:

|  |  |
| --- | --- |
| 1  2 | var tom = new Person();  app.Configuration.Bind(tom); |

Для объекта IConfiguration определен метод **Bind()**, который в качестве параметра принимает объект, который надо связать с данными. Стоит отметить, что между конфигурацией в json и классом Person имеется соответствие по названию свойств, благодаря чему может осуществляться связка (регистр в данном случае роли не играет).

В качестве альтернативы методу Bind мы могли бы использовать метод **Get<T>()**, который возвращает объект созданного класса:

|  |  |
| --- | --- |
| 1 | Person tom = app.Configuration.Get<Person>(); |

В этом случае нам не надо предварительно создавать объект класса Person.

Подобным образом можно выполнять привязку при получении конфигурации через механизм Dependency Injection:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("person.json");    app.Map("/", (IConfiguration appConfig) =>  {      var tom = appConfig.Get<Person>();  // связываем конфигурацию с объектом tom      return $"{tom.Name} - {tom.Age}";  });  app.Run();    public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; } = 0;  } |

**Привязка сложных объектов**

Рассмотрим привязку более сложных по структуре данных. Определим следующий файл **person.json**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | {    "age": "28",    "name": "Tom",    "languages": [      "English",      "German",      "Spanish"    ],    "company": {      "title": "Microsoft",      "country": "USA"    }  } |

Для представления этих данных в коде C# определим следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; }      public List<string> Languages { get; set; } = new();      public Company? Company { get; set; }  }  public class Company  {      public string Title { get; set; } = "";      public string Country { get; set; } = "";  } |

Теперь выполним в приложении привязку из конфигурации json в объекты классов C#:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("person.json");      var tom = new Person();  app.Configuration.Bind(tom);    app.Run(async (context) =>  {      context.Response.ContentType = "text/html; charset=utf-8";      string name = $"<p>Name: {tom.Name}</p>";      string age = $"<p>Age: {tom.Age}</p>";      string company = $"<p>Company: {tom.Company?.Title}</p>";      string langs = "<p>Languages:</p><ul>";      foreach (var lang in tom.Languages)      {          langs += $"<li><p>{lang}</p></li>";      }      langs += "</ul>";        await context.Response.WriteAsync($"{name}{age}{company}{langs}");  });  app.Run();    public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; }      public List<string> Languages { get; set; } = new();      public Company? Company { get; set; }  }  public class Company  {      public string Title { get; set; } = "";      public string Country { get; set; } = "";  } |

**Привязка конфигурации из xml**

Возьмем выше определенные классы Person и Company. И добавим в проект файл **person.xml**, который будет содержать аналогичные данные:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | <?xml version="1.0" encoding="utf-8" ?>  <person>    <name>Tom</name>    <age>35</age>    <languages name="0">English</languages>    <languages name="1">German</languages>    <languages name="2">Chinese</languages>    <company>      <title>Microsoft</title>      <country>USA</country>    </company>  </person> |

Обратите внимание на установку в файле xml массивов - они имеют атрибут name, который определяет условный индекс.

Применим конфигурацию из выше определенного файла xml в приложении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddXmlFile("person.xml");    var tom = new Person();  app.Configuration.Bind(tom);    app.Run(async (context) =>  {      context.Response.ContentType = "text/html; charset=utf-8";      string name = $"<p>Name: {tom.Name}</p>";      string age = $"<p>Age: {tom.Age}</p>";      string company = $"<p>Company: {tom.Company?.Title}</p>";      string langs = "<p>Languages:</p><ul>";      foreach (var lang in tom.Languages)      {          langs += $"<li><p>{lang}</p></li>";      }      langs += "</ul>";        await context.Response.WriteAsync($"{name}{age}{company}{langs}");  });  app.Run();    public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; }      public List<string> Languages { get; set; } = new();      public Company? Company { get; set; }  }  public class Company  {      public string Title { get; set; } = "";      public string Country { get; set; } = "";  } |

Как можно заметить из кода, меняется только подключение файла с json на xml, а весь остальной код остается прежним.

**Привязка секций конфигурации**

В примерах выше выполнялась привязка корневого объекта конфигурации, однако также можно осуществлять привязку отдельных секций. Например, выше в файле json и xml была определена секция company, которая хранит компанию пользователя. Выполним привязку отдельно этой секции к объекту класса Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var builder = WebApplication.CreateBuilder();  var app = builder.Build();    builder.Configuration.AddJsonFile("person.json");    Company company = app.Configuration.GetSection("company").Get<Company>();    app.Run(async (context) =>  {      await context.Response.WriteAsync($"{company.Title} - {company.Country}");  });  app.Run();    public class Company  {      public string Title { get; set; } = "";      public string Country { get; set; } = "";  } |

С помощью метода GetSection() получаем нужную нам секцию конфигурации и затем также можно вызвать методы Bind или Get и выполнить привязку.

**Передача конфигурации через IOptions**

Фреймворк ASP.NET Core реализует паттерн **Options**, который позволяет передавать конфигурацию не просто как набор настроек в виде пар ключ-значение, а как объекты определенных классов.

Для применения этого паттерна в приложении у объекта IServiceCollection, который представляет коллекцию сервисов приложения, определен метод **Configure()**:

|  |  |
| --- | --- |
| 1  2  3  4 | public static IServiceCollection Configure<TOptions>(this IServiceCollection services, IConfiguration config) where TOptions : class  public static IServiceCollection Configure<TOptions>(this IServiceCollection services, IConfiguration config, Action<BinderOptions> configureBinder) where TOptions : class  public static IServiceCollection Configure<TOptions>(this IServiceCollection services, string name, IConfiguration config) where TOptions : class  public static IServiceCollection Configure<TOptions>(this IServiceCollection services, string name, IConfiguration config, Action<BinderOptions> configureBinder) |

Этот метод реализован как метод расширения для типа IServiceCollection. И все версии метода типизируются типом, объект которого надо передавать через механизм внедрения зависимостей. И также все версии метода принимают в качестве одного из параметров объект конфигурации, на основе которой будет создаваться объект TOptions.

Допустим, у нас в проекте определен файл конфигурации **person.json** со следующим содержимым:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | {    "age": "37",    "name": "Tom",    "languages": [      "English",      "German",      "Spanish"    ],    "company": {      "title": "Microsoft",      "country": "USA"    }  } |

Данный файл по сути описывает одного пользователя. Элемент name сопоставляется с именем пользвателя, age - с возрастом, languages представляет языки, которыми владеет пользователь, а элемент company - компания, в которой пользователь работает. И мы хотим использовать эти данные в приложении как целостный объект. Для этого добавим вначале в проект класс **Person**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Person  {      public string Name { get; set; } = "";      public int Age { get; set; }      public List<string> Languages { get; set; } = new();      public Company? Company { get; set; }  }  public class Company  {      public string Title { get; set; } = "";      public string Country { get; set; } = "";  } |

Для представления компании пользователя определен дополнительный класс Company. Но, как можно заметить, определение класса Person совпадает со структурой json-файла.

И чтобы передать конфигурационные настройки через объект Person, мы можем использовать сервис **IOptions<TOptions>**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using Microsoft.Extensions.Options;    var builder = WebApplication.CreateBuilder();  builder.Configuration.AddJsonFile("person.json");  // устанавливаем объект Person по настройкам из конфигурации  builder.Services.Configure<Person>(builder.Configuration);    var app = builder.Build();    app.Map("/", (IOptions<Person> options) =>  {      Person person = options.Value;  // получаем переданные через Options объект Person      return person;  });  app.Run(); |

Прежде всего необходимо связать объект Person, который будет передаваться через механизм внедреия зависимостей, с конфигурацией файла json. Для этого метод builder.Services.Configure() типизирует типом Person и в качестве параметра получает конфигурацию приложения (свойство builder.Configuration реализует интерфейс IConfiguration и поэтому может передаваться в качестве параметра):

|  |  |
| --- | --- |
| 1 | builder.Services.Configure<Person>(builder.Configuration); |

Далее через механиз внедрения зависимостей мы можем получить созданный объект через сервис IOptions<Person>:

|  |  |
| --- | --- |
| 1  2  3  4  5 | app.Map("/", (IOptions<Person> options) =>  {      Person person = options.Value;  // получаем переданные через Options объект Person      return person;  }); |

Причем через механизм DI передается не просто объект Person, а объект **IOptions<Person>**, из которого мы можем получим непосредственно сам объект Person с помощью свойства **Value**.

Другой пример: определим в проекте новый класс middleware - **PersonMiddleware**, который фактически будет выводить информацию о пользователе на веб-станицу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | using Microsoft.Extensions.Options;    var builder = WebApplication.CreateBuilder();  builder.Configuration.AddJsonFile("person.json");  builder.Services.Configure<Person>(builder.Configuration);    var app = builder.Build();    app.UseMiddleware<PersonMiddleware>();  app.Run();  public class PersonMiddleware  {      private readonly RequestDelegate \_next;      public Person Person { get; }      public PersonMiddleware(RequestDelegate next, IOptions<Person> options)      {          \_next = next;          Person = options.Value;      }      public async Task InvokeAsync(HttpContext context)      {          System.Text.StringBuilder stringBuilder = new();          stringBuilder.Append($"<p>Name: {Person.Name}</p>");          stringBuilder.Append($"<p>Age: {Person.Age}</p>");          stringBuilder.Append($"<p>Company: {Person.Company?.Title}</p>");          stringBuilder.Append("<h3>Languages</h3><ul>");          foreach (string lang in Person.Languages)              stringBuilder.Append($"<li>{lang}</li>");          stringBuilder.Append("</ul>");            await context.Response.WriteAsync(stringBuilder.ToString());      }  } |

**Настройка привязки конфгурации**

При необходимости мы можем переопределить настройки с помощью перегрузки метода services.Configure():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using Microsoft.Extensions.Options;    var builder = WebApplication.CreateBuilder();  builder.Configuration.AddJsonFile("person.json");  builder.Services.Configure<Person>(builder.Configuration);  builder.Services.Configure<Person>(opt =>  {      opt.Age = 22;  });    var app = builder.Build();    app.Map("/", (IOptions<Person> options) =>  {      Person person = options.Value;  // получаем переданные через Options объект Person      return person;  });  app.Run(); |

Также можно передавать отдельные секции конфигурации. Например, передадим секцию Company:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using Microsoft.Extensions.Options;    var builder = WebApplication.CreateBuilder();  builder.Configuration.AddJsonFile("person.json");  builder.Services.Configure<Person>(builder.Configuration);  builder.Services.Configure<Company>(builder.Configuration.GetSection("company"));    var app = builder.Build();    app.Map("/", (IOptions<Company> options) => options.Value);    app.Run(); |