



# Добавление процесса “Непрерывной интеграции” (CI) в репозиторий

## Overview

Разработка большого проекта обычно занимает огромное количество времени, особенно, когда данный проект разрабатывает только один человек! Для того чтобы ускорить разработку - работу над проектом, обычно, назначают сразу несколько разработчиков, а то и десяток.

Перед началом работы команда заранее определяет правила написания кода, например: “Как должны именоваться классы или переменные?”, “Какая структура папок будет в проекте?” или даже “В качестве отступов будут использоваться знак табуляции или пробел?”

В соответствии с правилами, разработчики вносят свои изменения в репозиторий проекта, код проверяется и принимается решение об отправке данного изменения на основную ветку репозитория.

**Q: Но кто проверяет каждое из изменений?**

**A: Проверяет каждое изменение другой разработчик, обычно более опытный.**

А теперь, попробуйте представить себя в роли опытного разработчика которому приходилось проверять каждое изменение на наличие проблем, а именно:

- Собрать проекты и проверить на наличие ошибок
- Протестировать проекты и убедиться, что все тесты пройдены успешно
- Проверить форматирование всех файлов проекта

Однако такие действия придётся проделать ещё десятки раз, потому что на рассмотрении лежат ещё 30 таких изменений! Подумайте, сколько времени потребуется на то, чтобы скачать и проверить каждое из таких изменений, при том, что вы сидите за ноутбуком 2010 года? Скорее всего не меньше половины дня, если не учитывать, что за такое время не придёт ещё около 20 изменений в репозиторий.

Поэтому, чтобы упростить процесс проверки кода и быть уверенным в том, что код качественный и работоспособный, в проектах используется процесс непрерывной интеграции (CI).

Данный процесс интегрируется в общую ветку разработки, который автоматически проверяет на соответствие стандартам качества, собирает и тестирует на работоспособность проект.



**CI (Continuous Integration)** — в дословном переводе обозначается как «непрерывная интеграция». Данный процесс является автоматизированным и позволяет разработчикам убедиться в качестве написанного кода, который отправляется на удалённый репозиторий.

Благодаря этому процессу, разработчики могут получить ответы на следующие вопросы:

1. Успешно ли собрался проект?
2. Проект хорошо отформатирован?
3. А код правильно работает?

## Формирование EditorConfig для существующего решения

Для начала рассмотрим хорошую практику в разработке программного обеспечения, которая может значительно улучшить качество кода в проекте, а именно использование EditorConfig в вашем проекте.



EditorConfig - это файл конфигурации, который используется для установки единого стиля кодирования в проекте. Он позволяет определить настройки форматирования кода для различных редакторов и IDE, таких как отступы, символы конца строки, кодировку и другие параметры.

Например, ваш код выглядит следующим образом:

```
8   let string = "lorem ipsum";
9   string = "bacon";
10
11  function myFunc ( abc ) {
12    const a = abc;
13
14    const wrongIndent = {
15      a: 'this should have a semi colom'
16    }
17  }
18
```

Форматирование кода, который неудобно читать из-за неправильных отступов

Для того, чтобы автоматически отформатировать данный код, потребуется создать файл `.editorconfig` и прописать в нем требуемые правила форматирования.

Как может выглядеть файл `.editorconfig`:

```
# EditorConfig is awesome: https://EditorConfig.org
```

```
# top-most EditorConfig file
root = true

# Unix-style newlines with a newline ending every file
[*]
end_of_line = lf
insert_final_newline = true

# Matches multiple files with brace expansion notation
# Set default charset
[*.{js,py}]
charset = utf-8

# 4 space indentation
[*.py]
indent_style = space
indent_size = 4

# Tab indentation (no size specified)
[Makefile]
indent_style = tab

# Indentation override for all JS under lib directory
[lib/**/*.js]
indent_style = space
indent_size = 2

# Matches the exact files either package.json or .travis.yml
[{package.json,.travis.yml}]
indent_style = space
indent_size = 2
```

На основе данных правил, код форматируется следующим образом

```

8   let string = 'lorem ipsum';
9   string = 'bacon';
10
11  [-] function myFunc(abc) {
12      const a = abc;
13
14  [-]   const wrongIndent = {
15      |   a: 'this should have a semi colom',
16      |   };
17  }
18

```

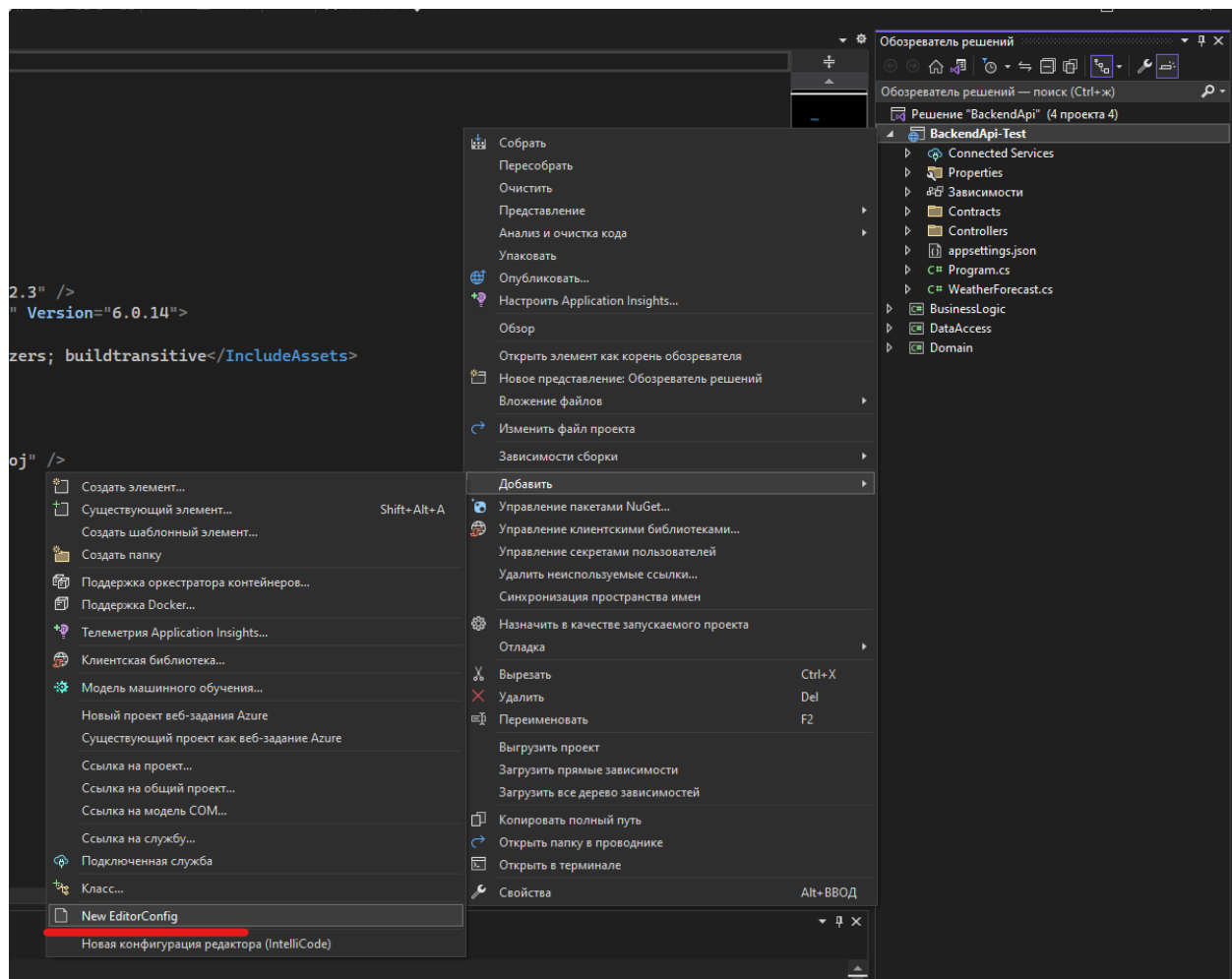
Отформатированный код

EditorConfig помогает поддерживать единый стиль кодирования в проекте, что упрощает работу команды разработчиков и повышает читаемость и понятность кода. Он также помогает избежать ошибок форматирования, которые могут возникнуть при использовании различных редакторов.

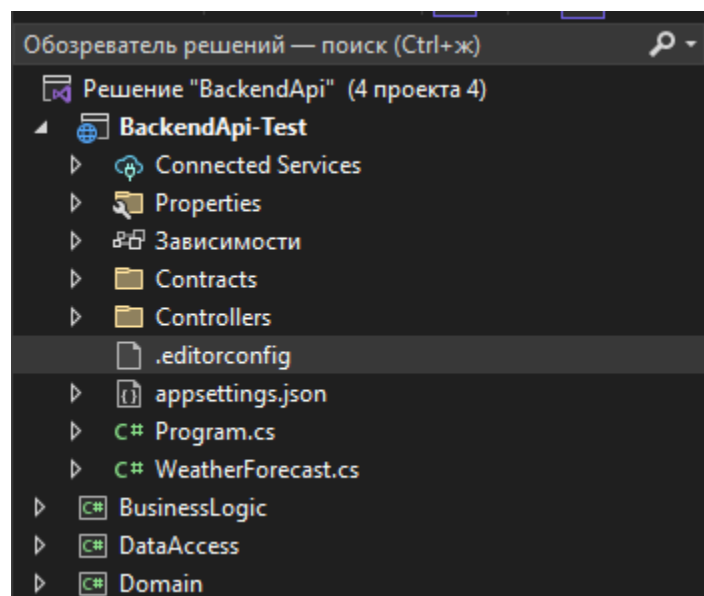
Давайте попробуем сформировать собственные правила для форматирования кода при помощи EditorConfig.

## Создание .editorconfig в Visual Studio 2022

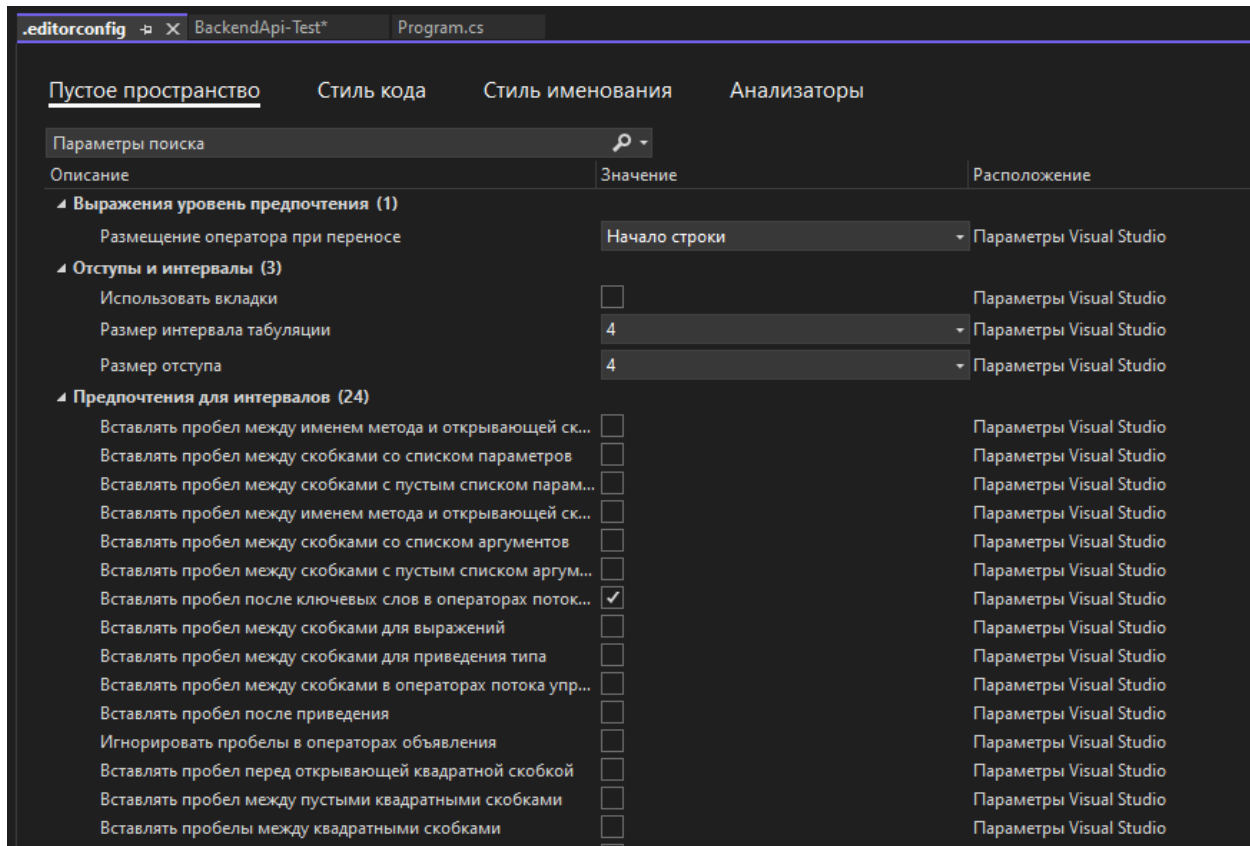
Для того, чтобы создать файл конфигурации, нажмите правую кнопку мыши по проекту и в разделе “Добавить” найдите пункт “New EditorConfig”



После чего, в проекте создается файл .editorconfig

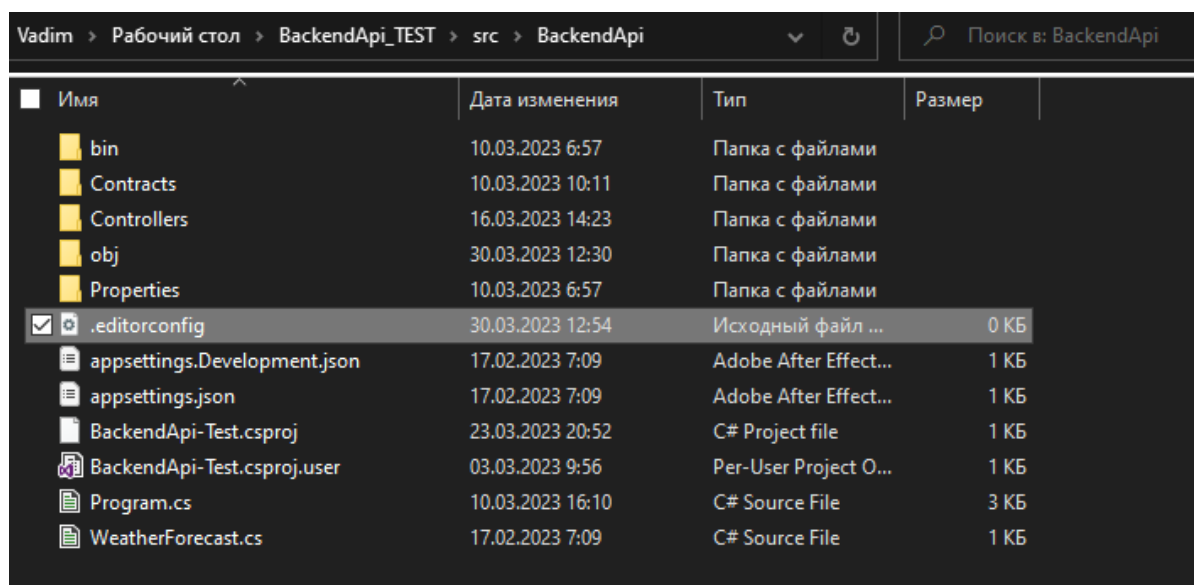
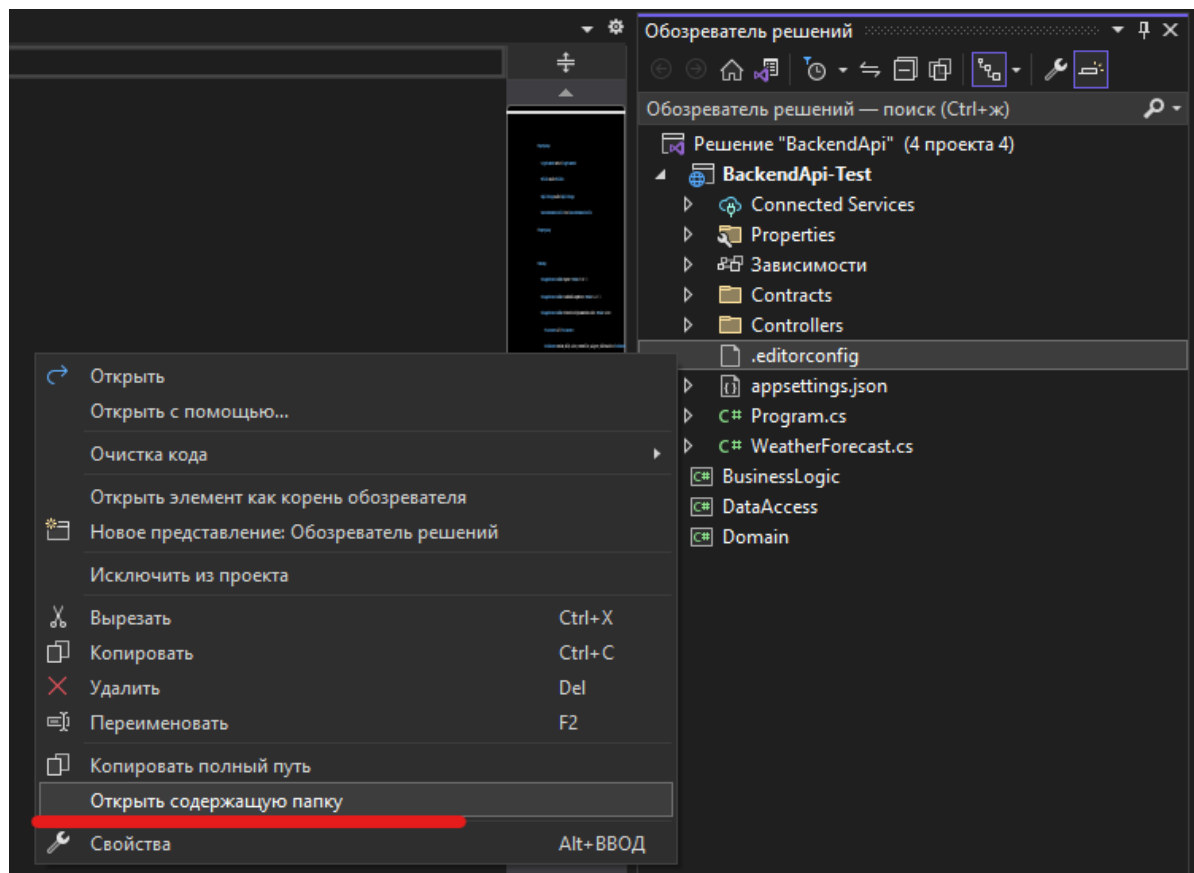


Который вы можете настраивать, дважды нажав по данному файлу.



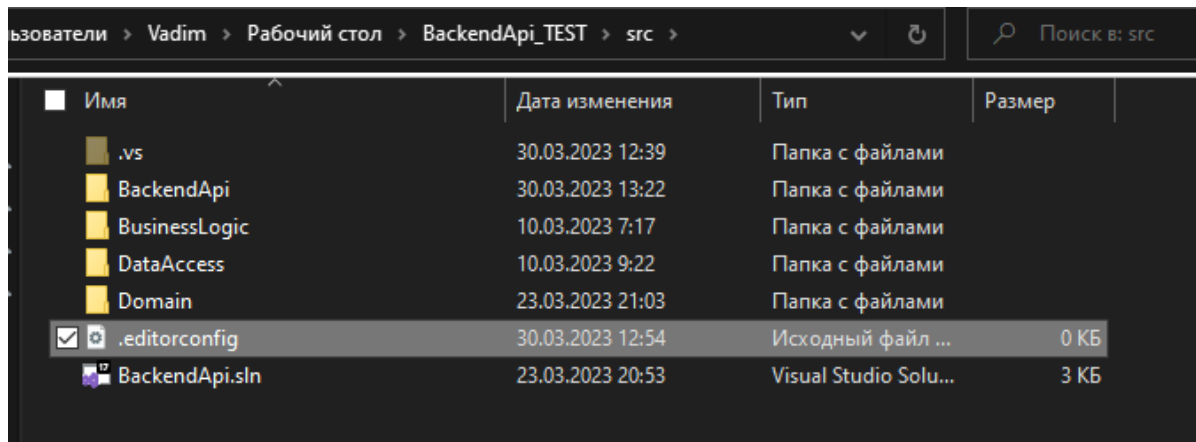
Так как файл .editorconfig расположен только в одном проекте, проверка форматирования не будет проходить за его пределами. Для того, чтобы сделать проверку форматирования кода глобальной - потребуется сделать следующие действия:

1. Отобразите папку в проводнике, в котором расположен .editorconfig

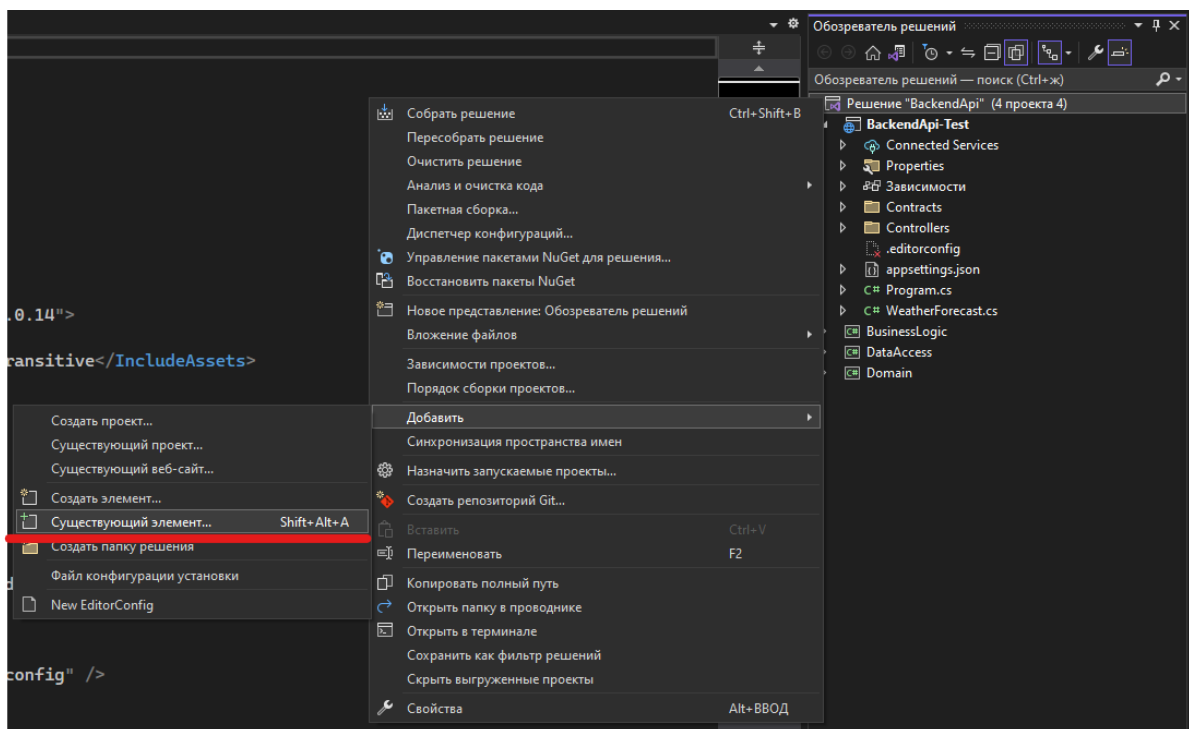


## 2. Перенесите файл к файлу решения .sln

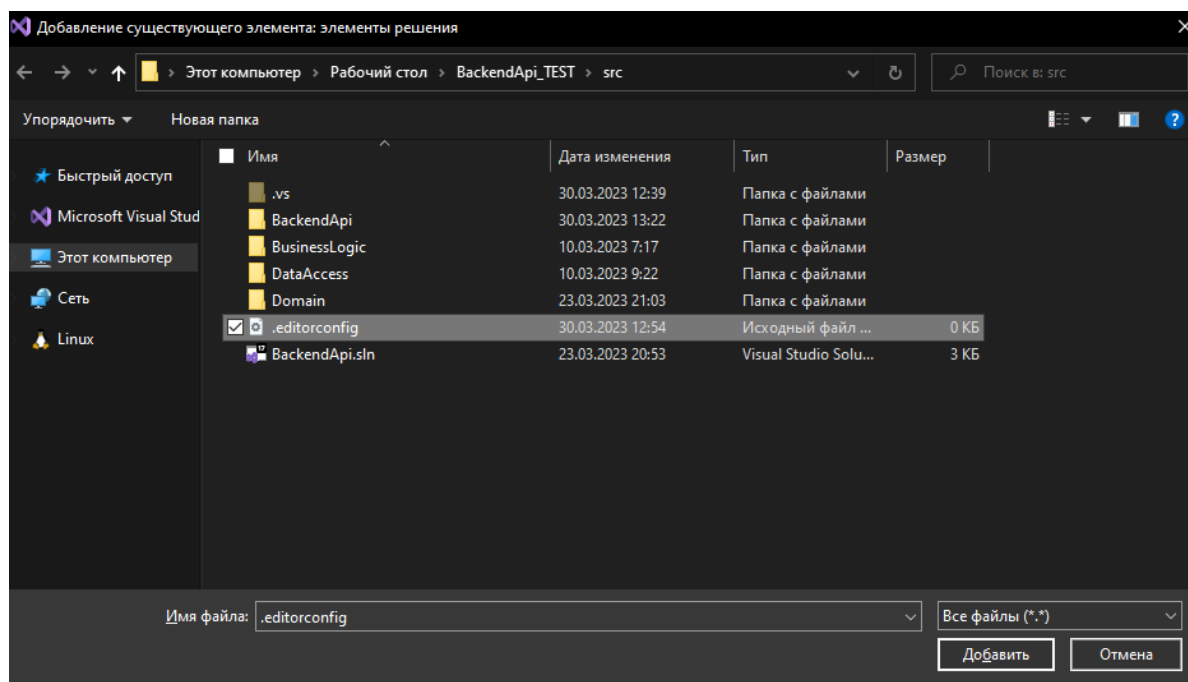




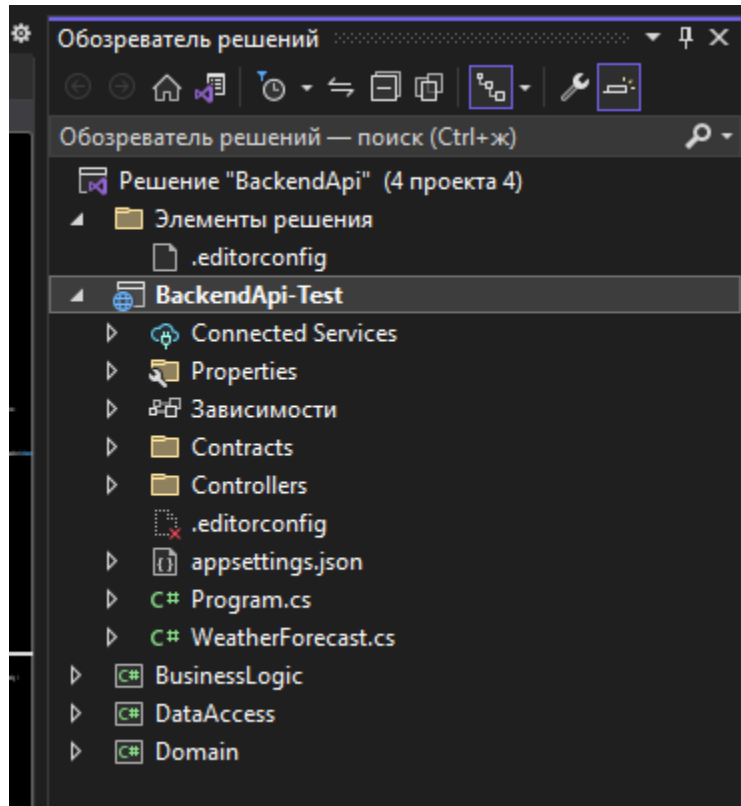
3. Нажмите ПКМ по решению. Выберите пункт “Существующий элемент...”



4. Выберите файл .editorconfig и добавьте его в решение



5. Появилась папка “Элементы решения” и файл .editorconfig в решении
6. Для удаления “отсутствующего” файла в проекте перейдите в его конфигурацию (дважды нажать по проекту ЛКМ)



7. Удалить из конфигурации проекта строку, которая отвечает за файл конфигурации

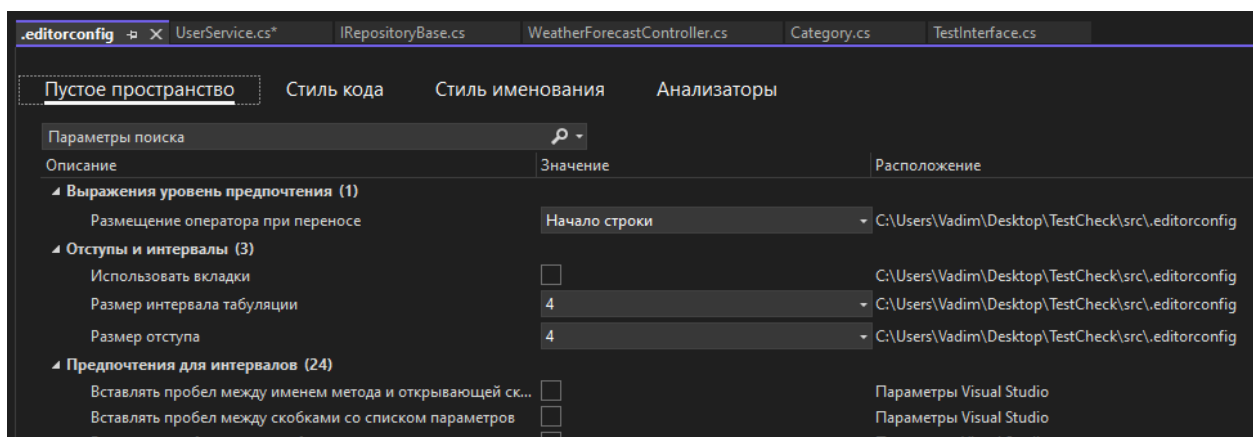
```
.editorconfig" BackendApi-Test Program.cs
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
    <GenerateDocumentationFile>True</GenerateDocumentationFile>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Mapster" Version="7.3.0" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="6.2.3" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="6.0.14">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\BusinessLogic\BusinessLogic.csproj" />
    <ProjectReference Include="..\DataAccess\DataAccess.csproj" />
  </ItemGroup>
  <ItemGroup>
    <None Include="C:\Users\Vadim\Desktop\BackendApi_TEST\src\BackendApi\.editorconfig" />
  </ItemGroup>
</Project>
```

# Установка правил форматирования для EditorConfig

Разберем каждый из разделов в EditorConfig

## Пустое пространство

Пустое пространство - раздел, в котором определяются правила для размещения операторов в коде. В данном разделе вы можете установить правила по своему вкусу.



Раздел “Пустое пространство” в EditorConfig



Для того, чтобы отформатировать код по указанным параметрам из данного раздела, используйте сочетание клавиш **Ctrl + K + D**

Попробуйте исправить форматирование для некоторых фрагментов кода, например.

Код, который требуется отформатировать:

```
Domain
using System.Linq.Expressions;

namespace Domain.Interfaces
{
    public interface IRepositoryBase<T> {
        Task<List<T>> FindAll();
        Task<List<T>> FindByCondition(Expression<Func<T, bool>> expression);
        Task Create(T entity);
        Task Update(T entity);
        Task Delete(T entity);
    }
}
```

Код у которого есть проблемы с отступами

После сочетания клавиш Ctrl + K + D преобразуется в следующий вид:

```
Domain
using System.Linq.Expressions;

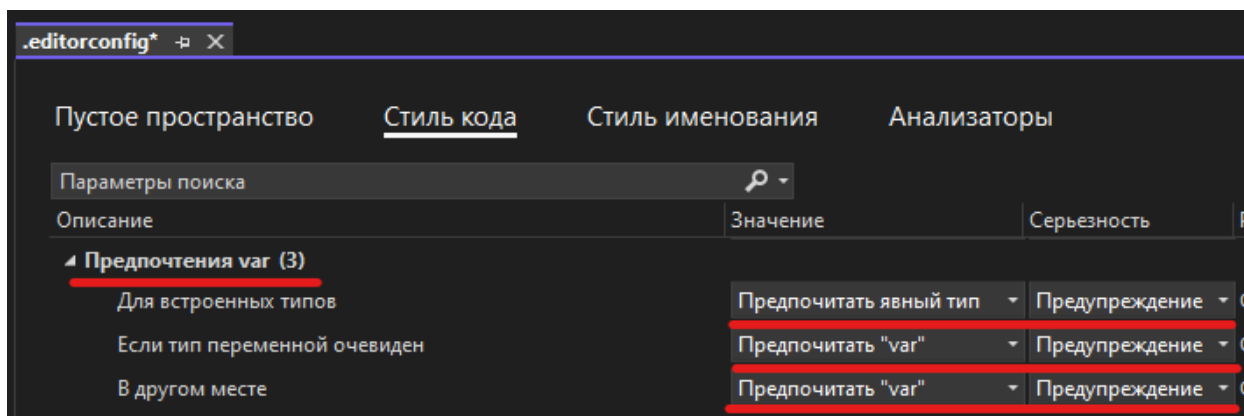
namespace Domain.Interfaces
{
    public interface IRepositoryBase<T>
    {
        Task<List<T>> FindAll();
        Task<List<T>> FindByCondition(Expression<Func<T, bool>> expression);
        Task Create(T entity);
        Task Update(T entity);
        Task Delete(T entity);
    }
}
```

Исправленный код после форматирования

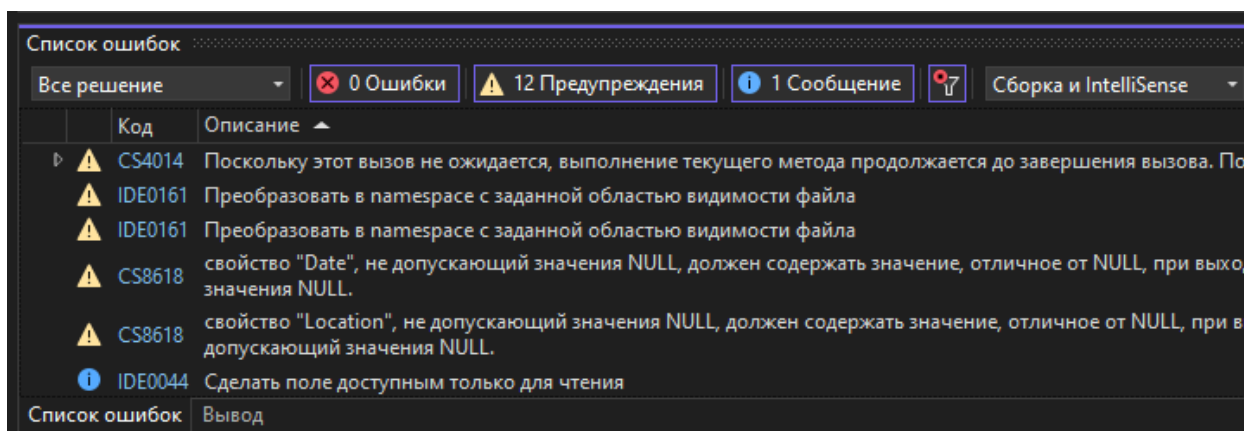
## Стиль кода

Стиль кода - раздел, в котором определяются требования к написанию кода, например.

Установите для вашего проекта требование для использования оператора var при создании переменной. Укажите в данном разделе следующие параметры:



Если в коде будет обнаружено, что стиль кода был нарушен, в разделе “Список ошибок” появится соответствующее сообщение об этом

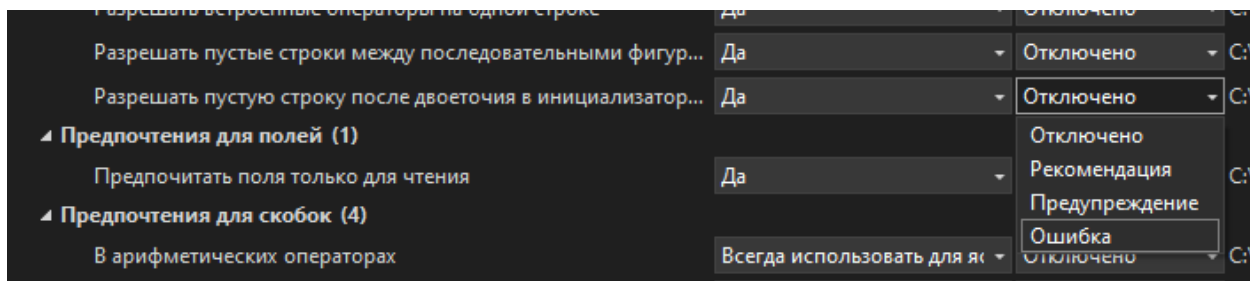


Также, сообщения помечают проблемный участок, которому требуется исправление.



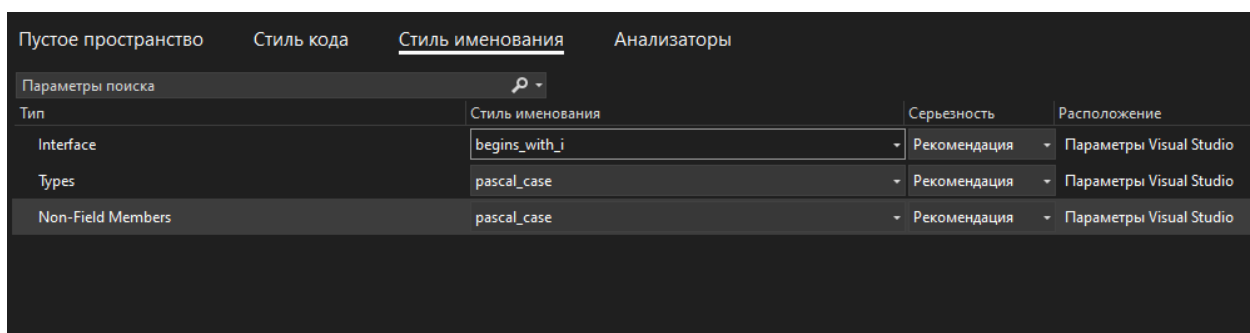
В зависимости от того, какой степенью серьезности обладает параметр в разделе “Стиль кода” - такие действия будет выполнять компилятор при обработке каждого файла.

Например, можно вообще запрещать сборку проекта, если было нарушено хотя бы одно из установленных правил.



## Стиль именования

Стиль именования - раздел, в котором определяются правила для именования различных Интерфейсов, Классов, переменных и т.д.



## Анализаторы

Анализаторы - раздел, в котором можно управлять поведением анализатора кода. Например, для того, чтобы отключить жалобы анализатора на некоторые участки кода по причине того, что в данном проекте это является нормой.

Пустое пространство    Стиль кода    Стиль именования <u>Анализаторы</u>				
ИД	Название	Категория	Серьезность	Расположение
▲ Design (49)				
CA1001	Типы, владеющие высвобождаемыми полями, должны быть...	Design	Отключено	Значения по умолчанию для анализатора
CA1032	Реализуйте стандартные конструкторы исключения	Design	Предупреждение	Значения по умолчанию для анализатора
CA1070	Не объявляйте поля событий как виртуальные	Design	Рекомендация	Значения по умолчанию для анализатора
CA1047	Не объявляйте защищенный член в запечатанном типе	Design	Рекомендация	Значения по умолчанию для анализатора
CA1000	Не объявляйте статические члены в универсальных типах	Design	Отключено	Значения по умолчанию для анализатора
CA1002	Не предоставляйте универсальные списки	Design	Предупреждение	Значения по умолчанию для анализатора
CA1003	Используйте экземпляры обработчика универсальных собы...	Design	Предупреждение	Значения по умолчанию для анализатора
CA1005	Не используйте слишком много параметров в универсальн...	Design	Предупреждение	Значения по умолчанию для анализатора
CA1008	Перечисления должны иметь нулевое значение	Design	Предупреждение	Значения по умолчанию для анализатора
CA1010	Также необходимо реализовать универсальный интерфейс	Design	Отключено	Значения по умолчанию для анализатора
CA1012	Абстрактные типы не должны иметь открытых конструкторов.	Design	Предупреждение	Значения по умолчанию для анализатора

## Знакомство с dotnet-cli. Использование команд.

Вспомните ситуацию, когда вам требовалось продемонстрировать работу вашей программы. Для этого вам приходилось открывать Visual Studio, а затем проводить операции по сборке проекта, чтобы произвести запуск программы на какой-нибудь машине.

Однако, что делать, если необходимо запустить программу на сервере у которого не установлен IDE, чтобы собрать и запустить проект? Для этого можно обратиться к инструментам командной строки, таким как dotnet cli для работы с проектами .NET.

Dotnet cli (Command Line Interface) - это инструмент командной строки, который позволяет разработчикам создавать, сборку, тестировать и развертывать



приложения на платформе .NET Core. Он представляет собой набор команд, которые можно использовать для создания проектов, управления зависимостями и сборки приложений.

Попробуем выполнить несколько команд.

Откройте командную строку и введите команду dotnet, чтобы проверить работоспособность.

```
Командная строка
Microsoft Windows [Version 10.0.19045.2728]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\Vadim>dotnet

Usage: dotnet [options]
Usage: dotnet [path-to-application]

Options:
  -h|--help           Display help.
  --info              Display .NET information.
  --list-sdks          Display the installed SDKs.
  --list-runtimes      Display the installed runtimes.

path-to-application:
  The path to an application .dll file to execute.

C:\Users\Vadim>
```

## Структура команды dotnet

```
dotnet <command> <argument> <option>
```

Все команды для платформы .NET начинаются с **dotnet**. После dotnet мы можем предоставить команду для выполнения определенного действия. За каждой командой могут следовать аргументы и опции. Ниже приведены команды .NET, благодаря которым мы можем выполнить некоторые действия

Basic Commands	Description
<u>new</u>	Создает новый проект, файл конфигурации или решение на основе указанного шаблона.
<u>restore</u>	Восстанавливает зависимости и инструменты проекта.

<u>build</u>	Выстраивает исполнимый файл с его зависимостями
<u>Run</u>	Запускает проект
<u>publish</u>	Публикует исполняемый файл с его зависимостями для определенной операционной системы
<u>test</u>	Выполняет тестирование проекта
<u>vtest</u>	Запускает тесты для указанного файла
<u>pack</u>	Упаковывает код в пакет NuGet.
<u>clean</u>	Очищает проект от файлов сборки проекта
<u>sln</u>	Модифицирует файл решения .sln
<u>help</u>	Отобразить справку по указанной команде

Project Modification Commands	Description
<u>add package</u>	Добавляет пакет (библиотеку) в проект
<u>add reference</u>	Включает в качестве зависимости проекта другой проект
<u>remove package</u>	Удаляет пакет (библиотеку) из проекта
<u>remove reference</u>	Убирает из зависимости проекта другой проект
<u>list reference</u>	Перечисляет все ссылки от проекта к проекту

## Пример работы с dotnet-cli

Давайте создадим, восстановим, соберем и запустим наш проект, использующее интерфейс командной строки без использования Visual Studio.

Чтобы создать новый проект, мы должны использовать команду `new`, за которой следует аргумент **<template name>**. Мы можем использовать следующие имена шаблонов: `console`, `class library`, `web`, `mvc`, `webapi`, `razor`, `angular`, `react` и т.д. проекты, используя CLI. Для создания консольного приложения используем шаблон `console` для создания нового консольного приложения.

Следующий код создает новый консольный проект в текущем каталоге с тем же именем, что и текущий каталог.

```
dotnet new console
```

Следующая команда создает новый консольный проект с именем `MyConsoleApp`.

**Параметр `-n` или `--name` задает название проекта.**

```
dotnet new console -n MyConsoleApp
```

Следующая команда создает новое консольное приложение с именем MyConsoleApp в каталоге MyProjects. Параметр -o или --output используется для указания выходного каталога, в котором должен быть создан проект.

```
dotnet new console -n MyConsoleApp -o C:\MyProjects
```

После создания проекта перейдите к папке проекта в командной строке, чтобы мы могли применить специфичные команды для проекта.

## Добавить пакет (библиотеку) в проект

Для того, чтобы добавить в проект пакет NuGet для различных целей. Например, примените следующую команду, чтобы добавить пакет Telegram.Bot в наш консольный проект.

```
dotnet add package Telegram.Bot
```

После выполнения данной команды в наш проект подключится библиотека Telegram.Bot. Убедиться в этом мы сможем при просмотре файла, который имеет расширение .csproj

## Восстановление пакетов

Представим, что ваш проект был перенесен на другой компьютер. Чтобы на другом компьютере выполнялась подкачка пакетов (например библиотека Telegram.Bot), потребуется написать команду restore

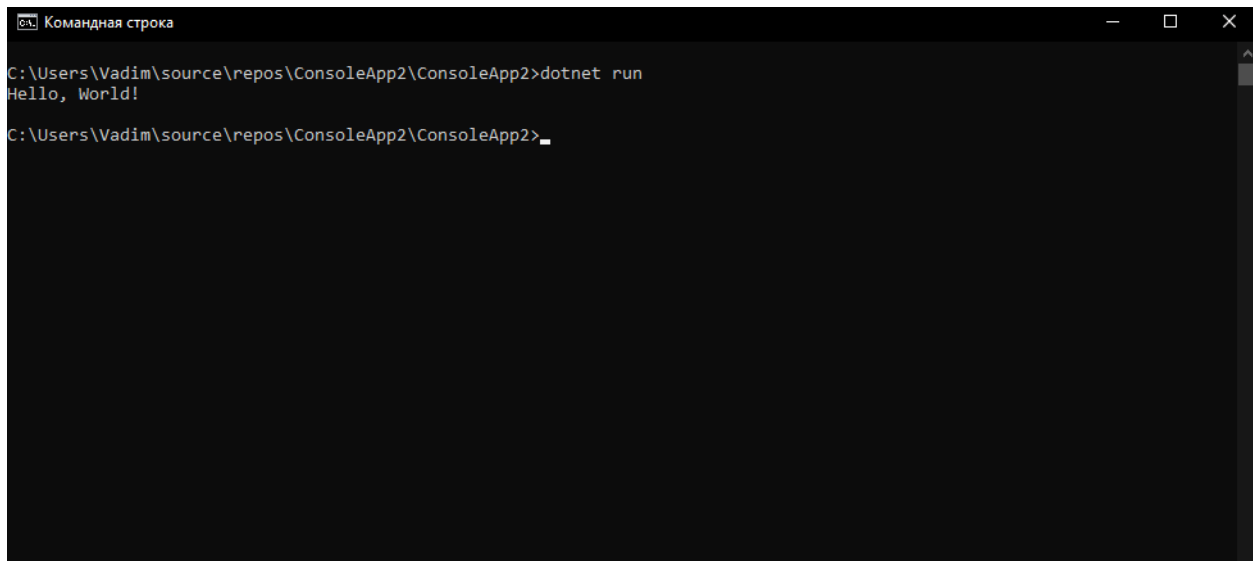
```
dotnet restore
```

## Построение проекта

Чтобы сформировать исполняемые файлы вашего проекта для запуска, напишите в консоли `dotnet build`

## Запуск проекта

Чтобы запустить проект, выполните команду в консоли `dotnet run` (Убедитесь, что вы находитесь в директории проекта)



```
Командная строка
C:\Users\Vadim\source\repos\ConsoleApp2\ConsoleApp2>dotnet run
Hello, World!
C:\Users\Vadim\source\repos\ConsoleApp2\ConsoleApp2>
```

В данном случае, результат выполнения данной команды отобразил в консоли "Hello World!".

## Подсказки для написания команд

Мы можем получить справку по любым командам .NET CLI, введя -h или -help в конце команды, по которой мы хотим получить справку.

Например, `dotnet new -h` отобразит справку по команде **new**: аргументам и параметрам, которые мы можем использовать с ней, как показано ниже.

```
Командная строка

C:\Users\Vadim\source\repos\ConsoleApp2\ConsoleApp2>dotnet new -h
Description:
  Команды по созданию экземпляров шаблонов для CLI .NET/

Usage:
  dotnet new [<template-short-name> [<template-args>...]] [options]
  dotnet new [command] [options]

Arguments:
  <template-short-name>  Краткое имя создаваемого шаблона.
  <template-args>        Используемые параметры шаблона.

Options:
  -o, --output <output>  Расположение для размещения созданных выходных данных.
  -n, --name <name>      Имя создаваемых выходных данных. Если имя не указано, используется имя выходного каталога.
  --dry-run               Отображает сводку действий, которые могли бы произойти, если бы данная командная строка была
                          запущена, если это приведет к созданию шаблона.
  --force                Заставляет создавать содержимое, даже если при этом изменяются существующие файлы.
  --no-update-check      Отключает проверку наличия обновлений пакета шаблона при создании экземпляра шаблона.
  --project <project>    Проект, который следует использовать для оценки контекста.
  -v, --verbosity <LEVEL>  Задаёт уровень детализации. Допустимые значения: q (без вывода подробных сведений), n
                          (минимальный), n (нормальный) и diag (диагностический). [default: normal]
  -d, --diagnostics      Включает диагностические выходные данные.
  -?, -h, --help         Показать справку командной строки.

Commands:
  create <template-short-name> <template-args>  Создание экземпляра шаблона с заданным коротким именем. Псевдоним
  install <package>                               'dotnet new <template name>'.
  uninstall <package>                             Устанавливает пакет шаблона.
  update                                           Удаляет пакет шаблона.
  search <template-name>                          Проверяет наличие обновлений для установленных пакетов шаблонов и
  list <template-name>                            устанавливает обновления.
                                                    Поиск шаблонов на NuGet.org.
                                                    Списки шаблонов, содержащих указанное имя шаблона. Если имя не указано,
                                                    перечисляются все шаблоны.
```

Также, dotnet cli обеспечивает возможность тестирования и сборки приложений в автоматическом режиме, что помогает ускорить процесс разработки. Рассмотрим возможности данного инструмента на изображении ниже.

```
Командная строка

C:\Users\Vadim\source\repos\ConsoleApp2\ConsoleApp2>dotnet -h
Использование: dotnet [runtime-options] [path-to-application] [arguments]

Запуск приложения .NET.

runtime-options:
--additionalprobingpath <path>    Путь к политике проверки и проверяемым сборкам.
--additional-deps <path>          Путь к дополнительному файлу deps.json.
--depsfile <path>                 Путь к файлу <приложение>.deps.json.
--fx-version <version>            Версия установленной общей платформы, которую следует использовать для запуска приложения.
--roll-forward <setting>          Накат до версии платформы (LatestPatch, Minor, LatestMinor, Major, LatestMajor, Disable).
--runtimeconfig <path>            Путь к файлу <приложение>.runtimeconfig.json.

path-to-application:
Путь к DLL-файлу приложения, который необходимо выполнить.

Использование: dotnet [sdk-options] [command] [command-options] [arguments]

Выполнение команды пакета SDK для .NET.

sdk-options:
-d|--diagnostics    Включение выходных данных диагностики.
-h|--help           Показать справку командной строки.
--info              Отображение сведений о .NET.
--list-runtimes     Отображение установленных сред выполнения.
--list-sdks         Отображение установленных пакетов SDK.
--version           Отображение используемой версии пакета SDK для .NET.

Команды SDK:
add                Добавление пакета или ссылки в проект .NET.
build              Сборка проекта .NET.
build-server       Взаимодействие с серверами, запущенными в ходе сборки.
clean              Очистка выходных данных сборки проекта .NET.
format             Применение настроек стилей к проекту или решению.
help              Показать справку командной строки.
list               Вывод списка ссылок на проекты в проекте .NET.
msbuild            Выполнение команд Microsoft Build Engine (MSBuild).
new               Создание нового файла или проекта .NET.
nuget              Предоставление дополнительных команд NuGet.
pack              Создание пакета NuGet.
publish            Публикация проекта .NET для развертывания.
remove            Удаление пакета или ссылки из проекта .NET.
restore           Восстановление зависимостей, указанных в проекте .NET.
run               Сборка и запуск проекта .NET.
sdk               Управление установкой пакета SDK .NET.
sln               Изменение файлов решения Visual Studio.
store             Сохранение указанных сборок в хранилище пакетов среды выполнения.
test              Запуск модульных тестов с помощью средства, указанного в проекте .NET.
tool              Установка и настройка инструментов, расширяющих возможности .NET.
vstest            Выполнение команд Microsoft Test Engine (VSTest).
workload          Управление необязательными рабочими нагрузками.

Дополнительные команды из входящих в пакет инструментов:
dev-certs         Создание сертификатов разработки и управление ими.
fsi               Запуск F# Interactive / выполнение скриптов F#.
user-jwts         Управлять JSON Web Token в разработке.
user-secrets      Управление развертыванием секретов пользователей.
```

## Проверка форматирования кода через командную строку

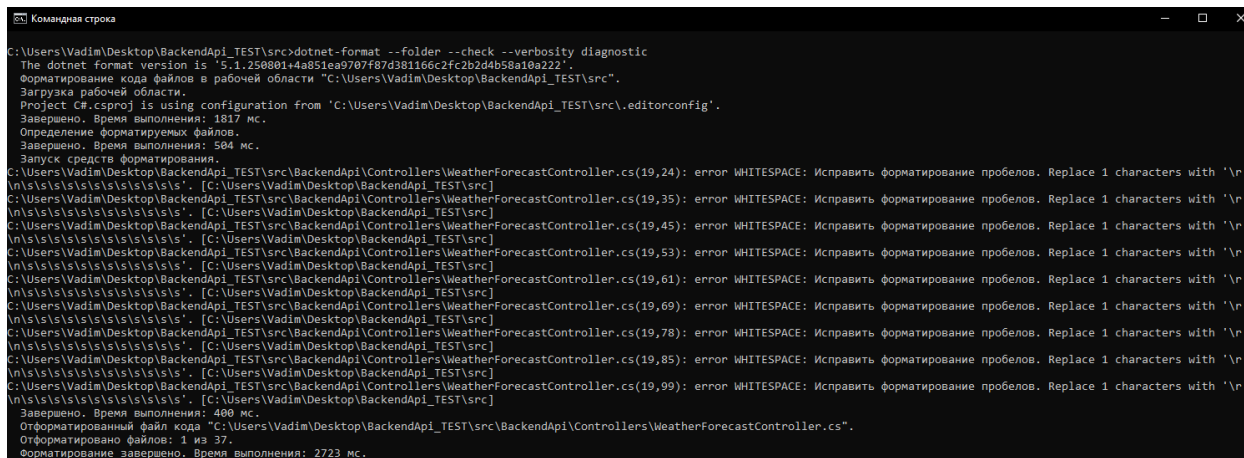
Для того, чтобы проверить качество форматирования кода вы можете использовать команду `dotnet-format`

В случае, если данная команда не установлена на вашем устройстве, выполните команду для установки:

```
dotnet tool install -g dotnet-format
```

Для того, чтобы проверить форматирование кода используйте команду:

```
dotnet-format --folder --check --verbosity diagnostic
```



Чтобы форматирование автоматически исправила данные проблемы, уберите из команды опцию `--check`

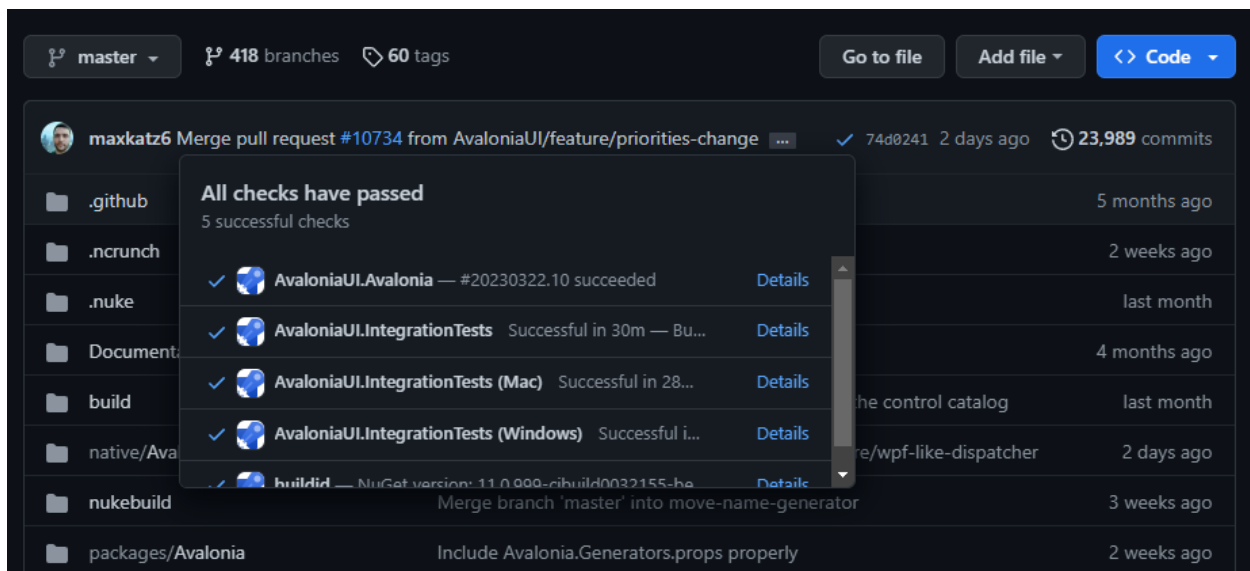
## Знакомство с GitHub Actions

Так как наш проект расположен в репозитории на GitHub, у нас есть возможность познакомиться с процессом CI/CD при помощи инструмента GitHub Actions.

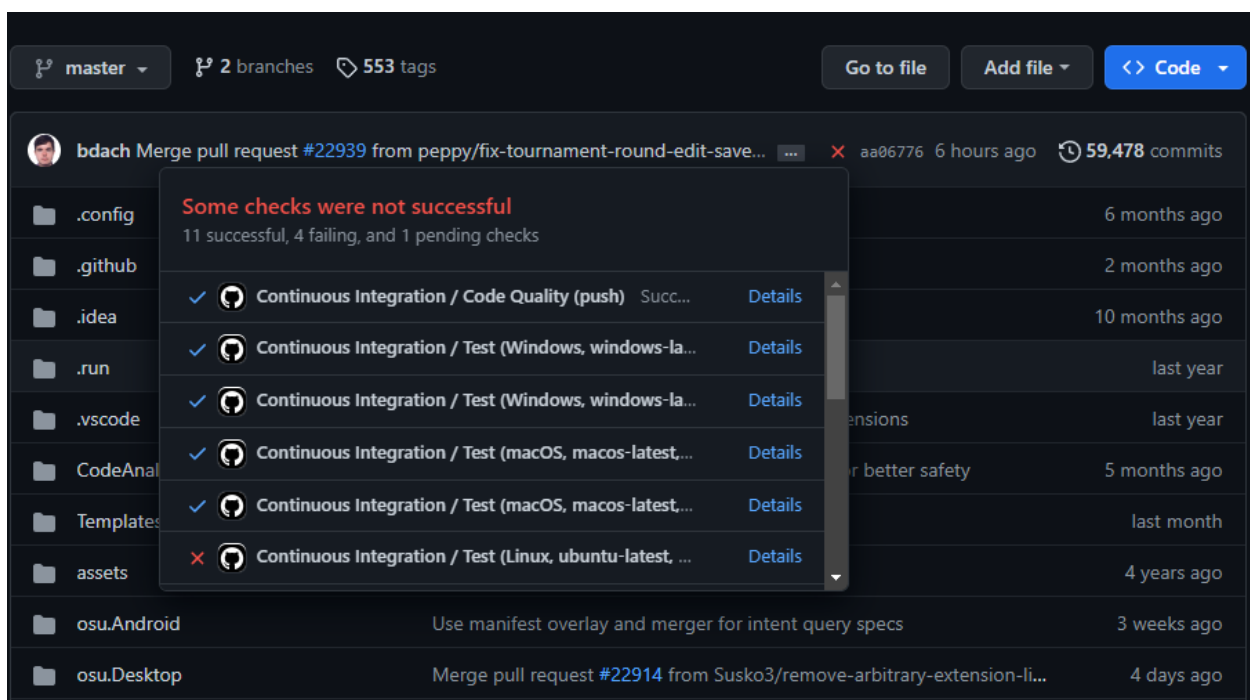


GitHub Actions - это инструмент автоматизации, который позволяет запускать различные действия при определенных условиях в репозитории на GitHub.

При открытии репозитория GitHub у которого имеется галочка или крестик рядом с описанием коммита - вы можете сразу понять в каком состоянии находится проект:



В главной ветке репозитория AvaloniaUI/Avalonia все проверки были пройдены успешно



В главной ветке репозитория rpr/osu упали некоторые тесты

Попробуем добавить свой вариант CI в нашем проекте и таким образом разберём работу GitHub Actions.



Для того, чтобы настроить проект для использования GitHub Actions, необходимо разместить .yml файлы по пути ".github/workflows/" в корне вашего репозитория.

Имя	Дата изменения	Тип	Размер
.vs	10.03.2023 6:57	Папка с файлами	
src	30.03.2023 13:46	Папка с файлами	
<input checked="" type="checkbox"/> .github	31.03.2023 8:45	Папка с файлами	
.gitattributes	17.02.2023 7:09	Исходный файл ...	1 КБ
.gitignore	17.02.2023 7:09	Исходный файл ...	8 КБ
LICENSE	17.02.2023 7:09	Файл	2 КБ
README.md	17.02.2023 7:09	Исходный файл ...	1 КБ

Назовем файл для проверки форматирования кода как dotnet-format.yml

```
name: dotnet format

on:
  push:
    paths:
      - "**.cs"
      - ".editorconfig"

jobs:
  check-format:
    runs-on: windows-latest

    steps:
      - name: Setup .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '6.0.x'

      - name: Install dotnet-format tool
        run: dotnet tool install -g dotnet-format

      - name: Check out code
        uses: actions/checkout@v2

      - name: Run dotnet format
        run: dotnet-format --folder --check --verbosity diagnostic
```

В этом файле мы будем описывать действия, которые должны быть выполнены при определенных условиях.

Давайте разберем пример другого файла конфигурации.

Ниже приведен пример файла конфигурации для GitHub Actions:

```
name: .NET Core CI

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Setup .NET Core
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '3.1'
      - name: Build
        run: dotnet build --configuration Release
      - name: Test
        run: dotnet test --configuration Release --no-build
```

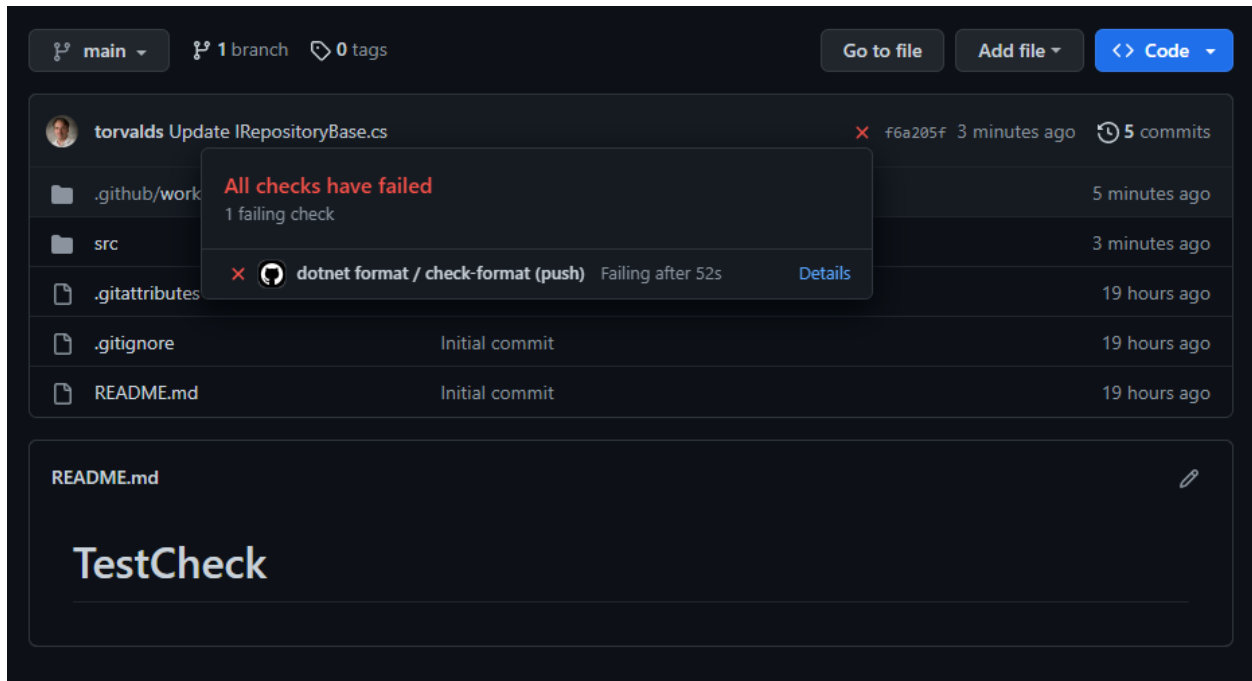
В этом примере мы настроили GitHub Actions для выполнения следующих действий:

- Запуск сборки проекта при пуше в ветку master или при создании pull request'a в master.
- Использование последней версии Ubuntu.
- Установка .NET Core версии 3.1.
- Сборка проекта с помощью команды "dotnet build".
- Запуск тестов с помощью команды "dotnet test".

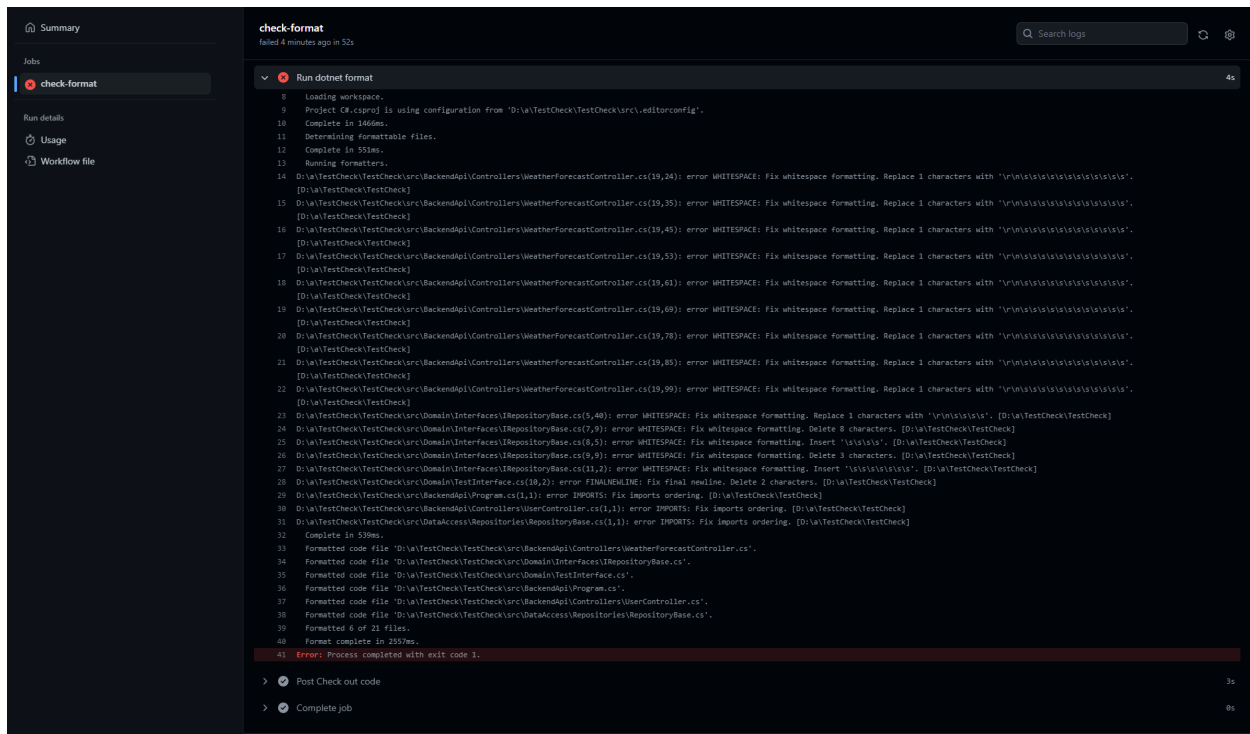
## Запуск GitHub Actions

После того, как вы создали файл конфигурации для GitHub Actions, его нужно закоммитить и запустить в ваш репозиторий на GitHub.

GitHub автоматически запустит GitHub Actions при выполнении условий, указанных в файле конфигурации.



Вы можете просмотреть результаты выполнения GitHub Actions, перейдя на страницу вашего репозитория на GitHub и выбрав вкладку "Actions".



# Задание

Создайте .yaml-файл, который будет выполнять сборку проекта при каждом commit-е.