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 заносят 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 представьте 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.

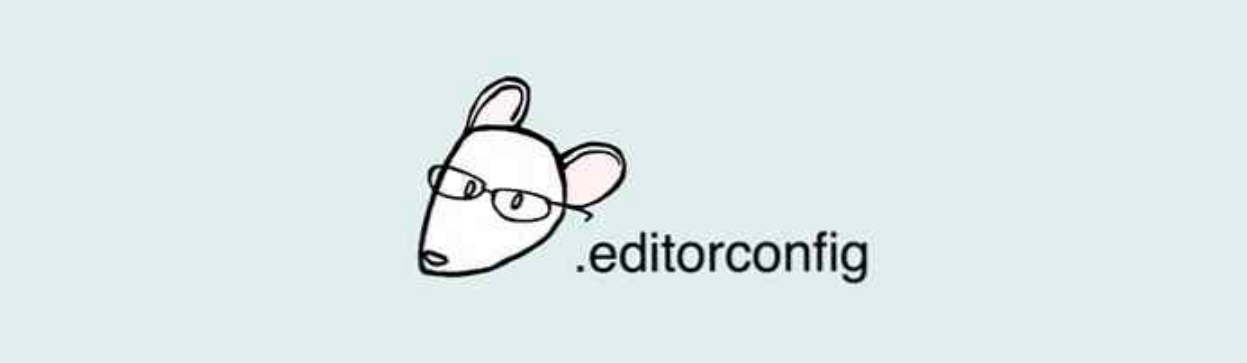
Поэтому, чтобы упростить процесс проверки кода и быть уверенным в том, что код качественный и работоспособный, в проектах используется процесс непрерывной интеграции (CI).

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 убедится 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?

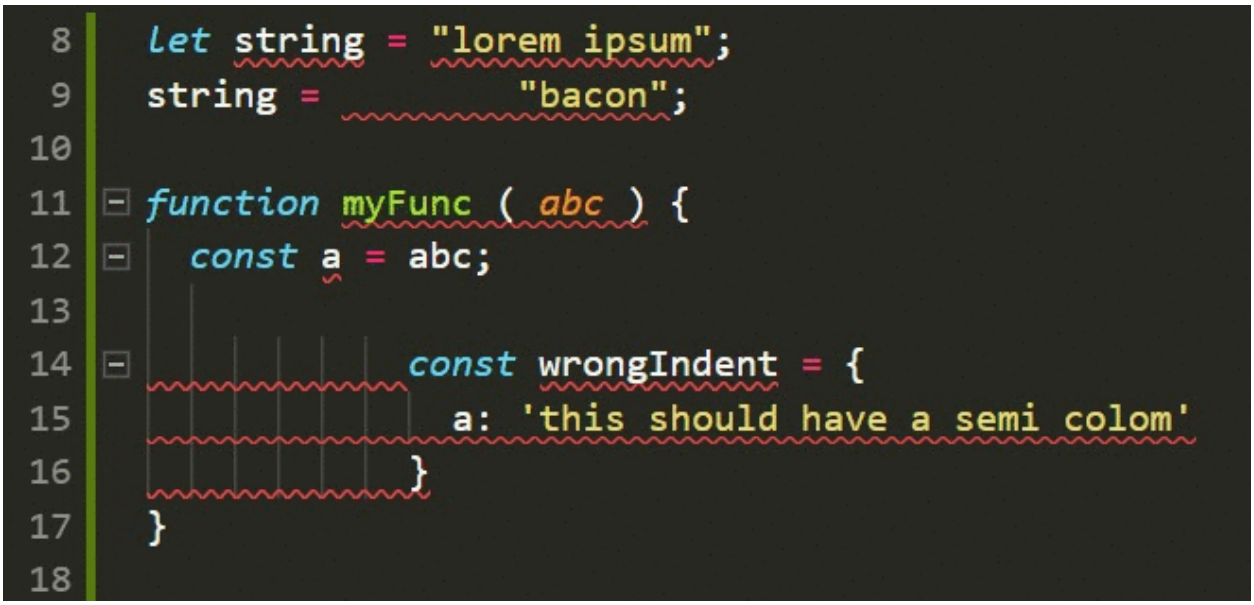
Формирование EditorConfig для существующего решения

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:



Форматирование кода, который неудобно читать из-за неправильных отступов

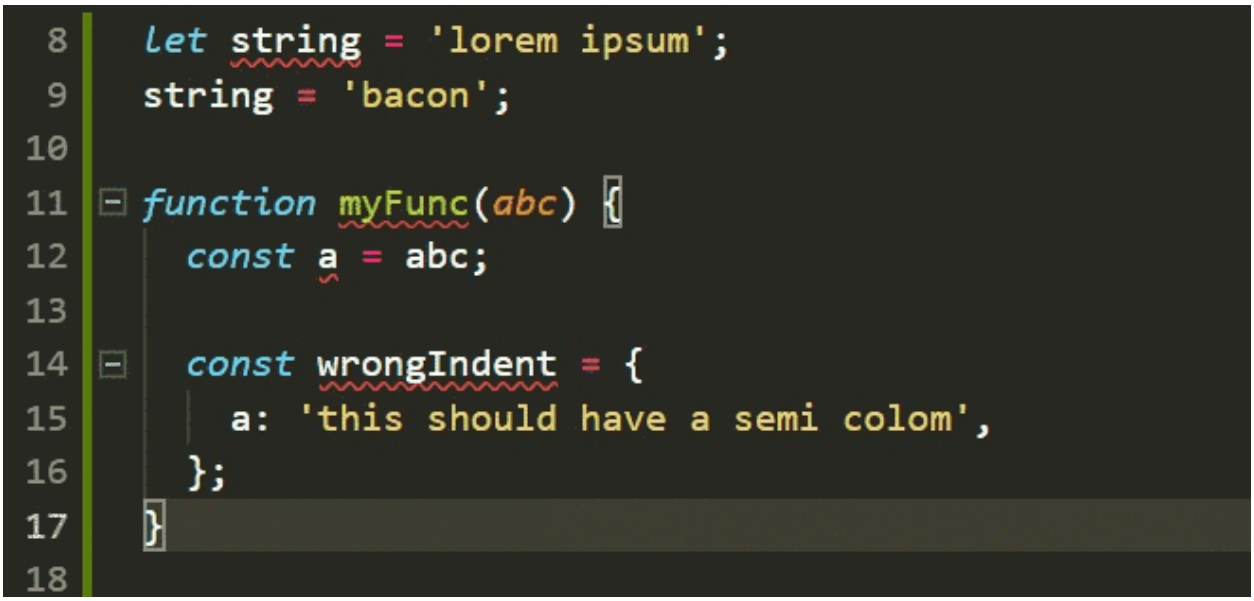
Для того, чтобы автоматически отформатировать данный код, потребуется создать файл .editorconfig и прописать в нем требуемые правила форматирования.

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



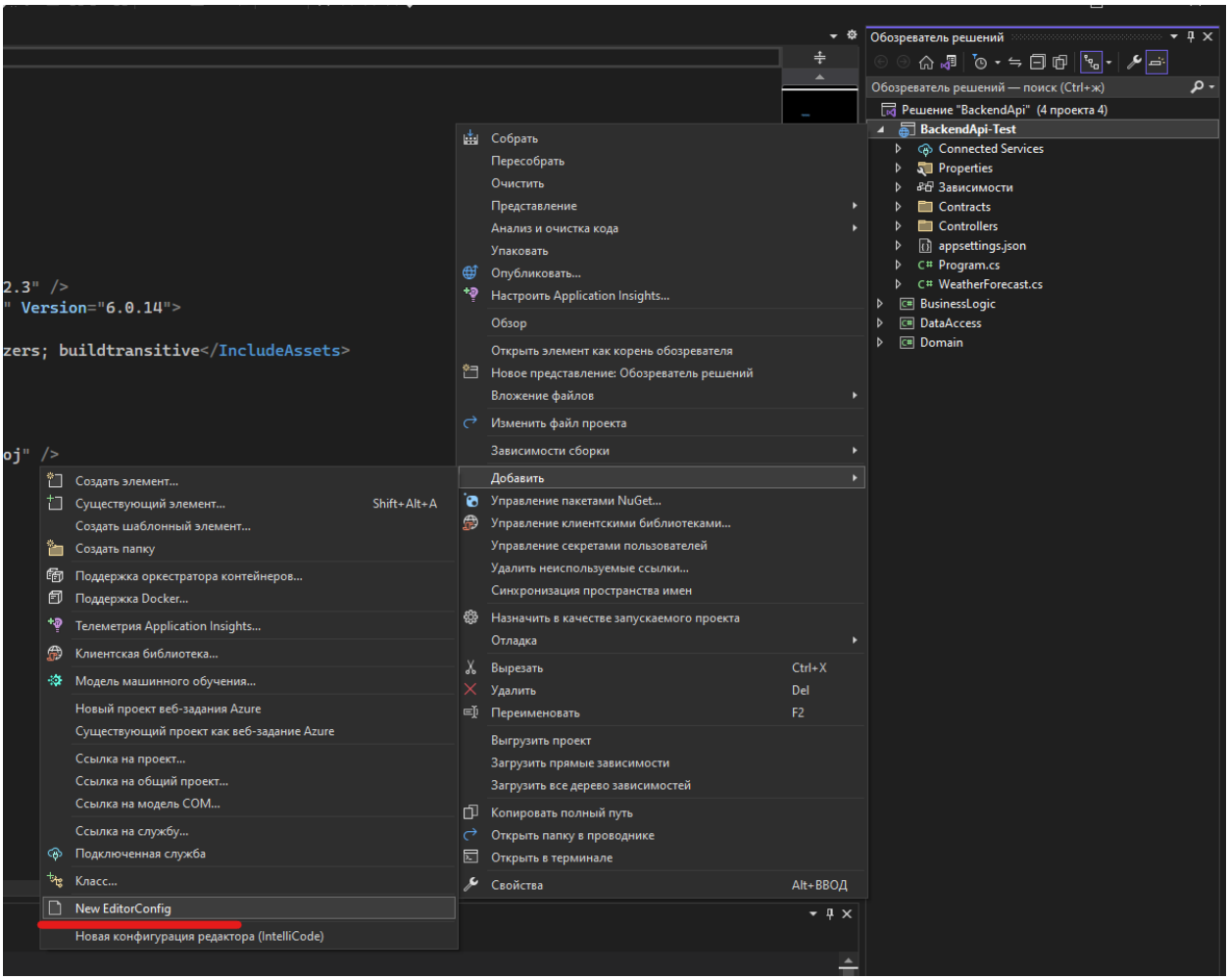
Отформатированный код

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.

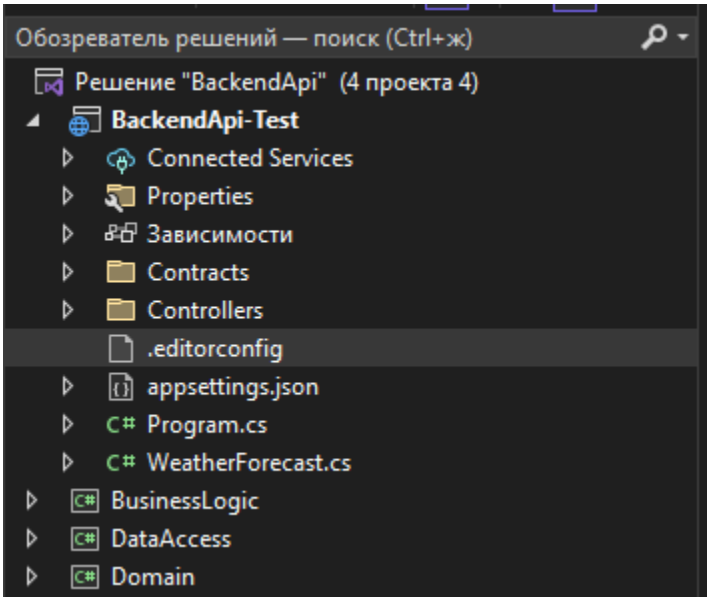
Давайте попробуем сформировать собственные правила для форматирования кода при помощи EditorConfig.

Создание .editorconfig в Visual Studio 2022

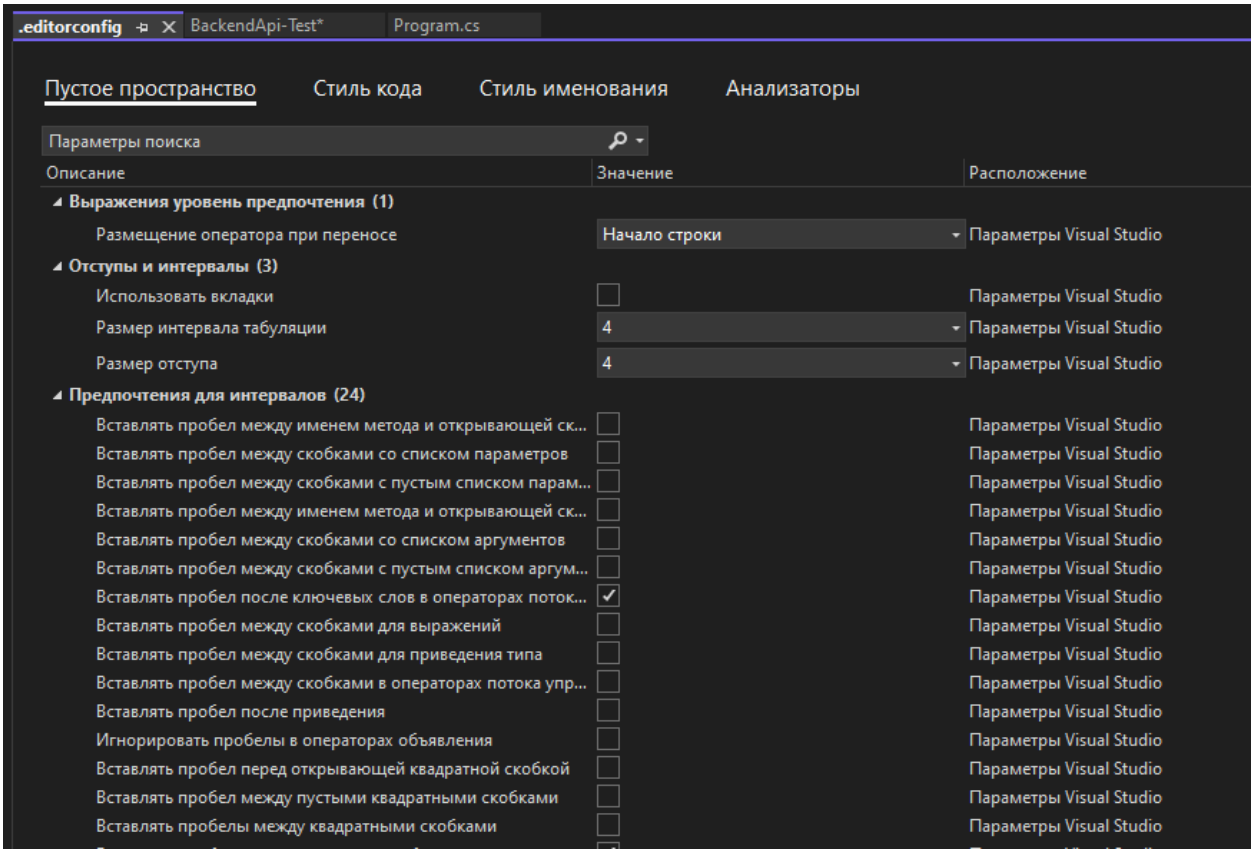
Для того, чтобы создать файл конфигурации, нажмите правую кнопку мыши по проекту и в разделе “Добавить” найдите пункт “New EditorConfig”



После чего, в проекте создается файл .editorconfig

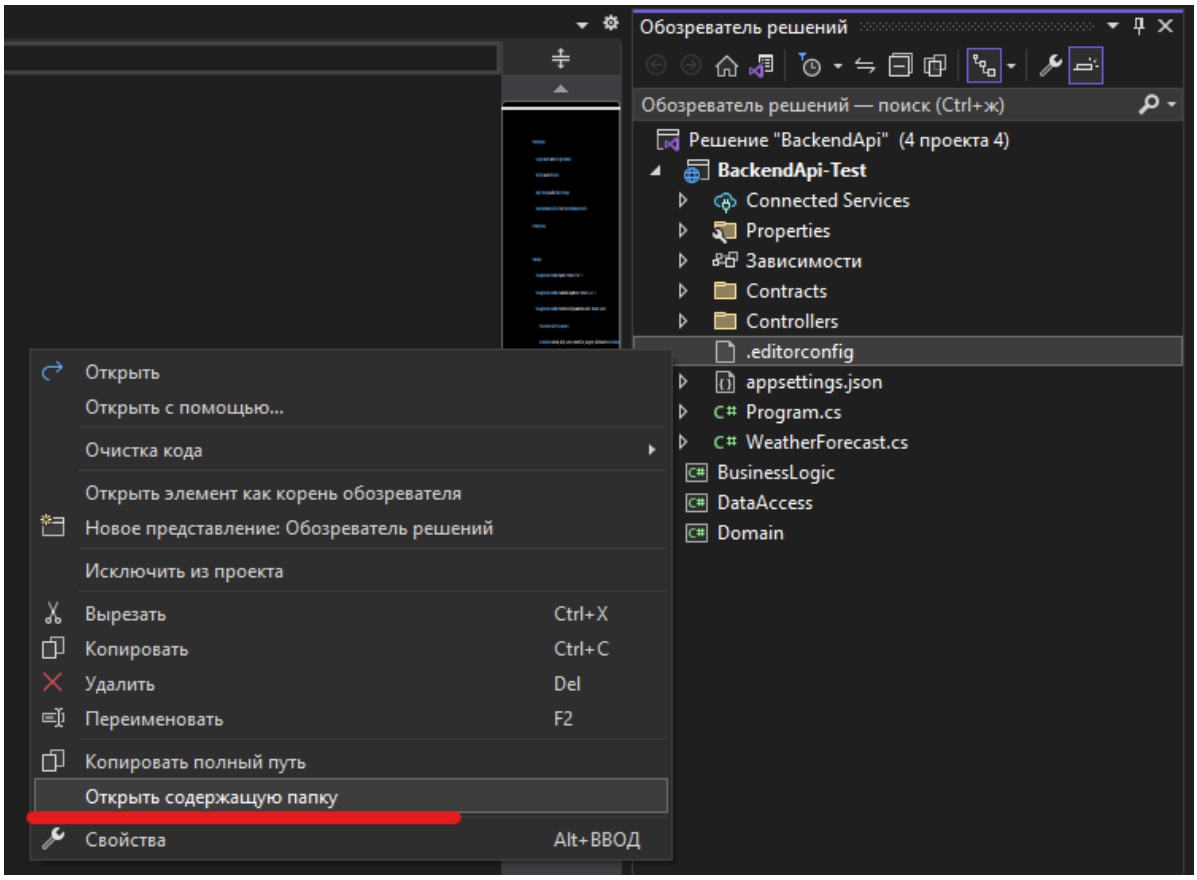


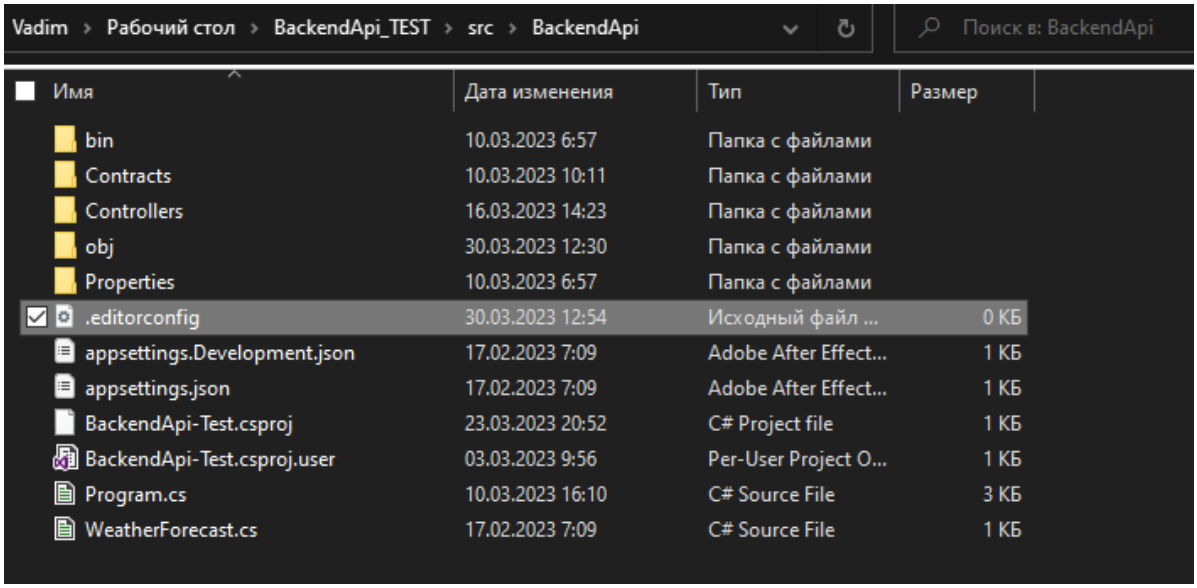
Который вы можете настраивать, дважды нажав по данному файлу.



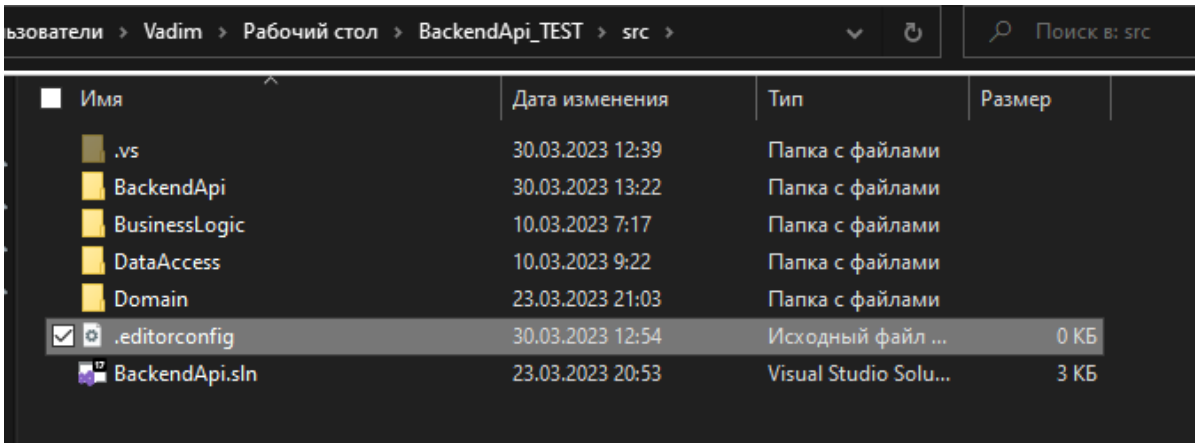
Так как файл .editorconfig расположен только в одном проекте, проверка форматирования не будет проходить за его пределами. Для того, чтобы сделать проверку форматирования кода глобальным - потребуется сделать следующие действия:

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

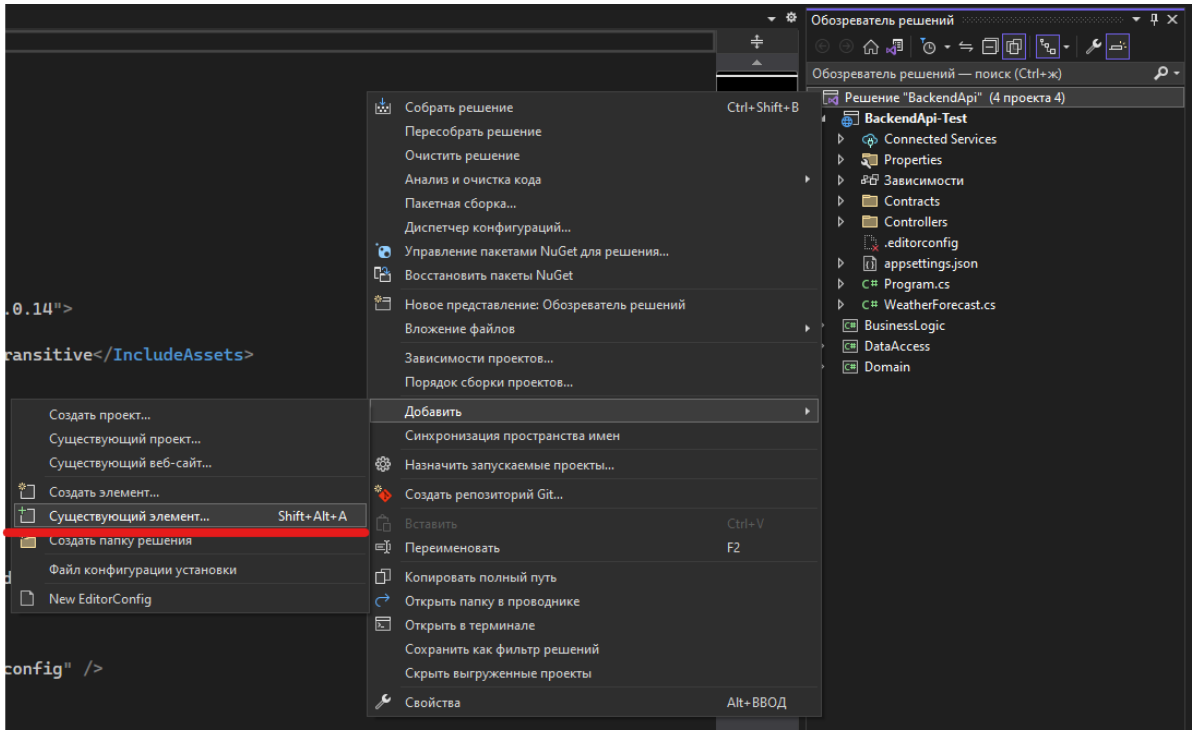




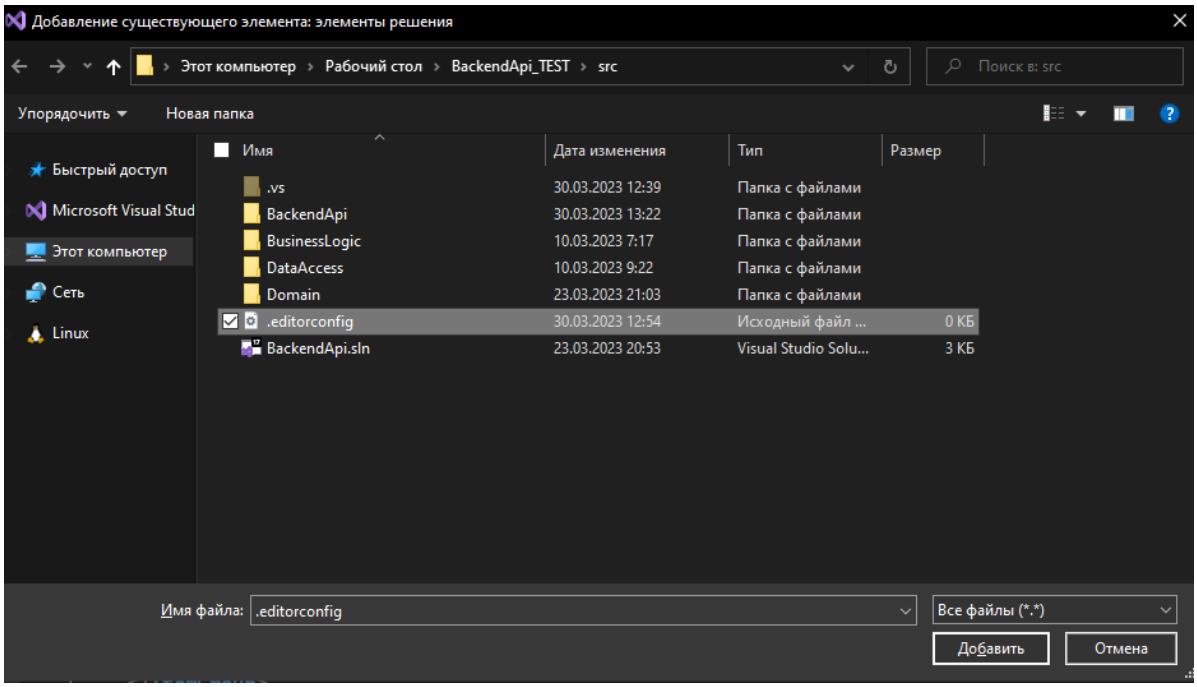
2. Перенесите файл к файлу решения .sln



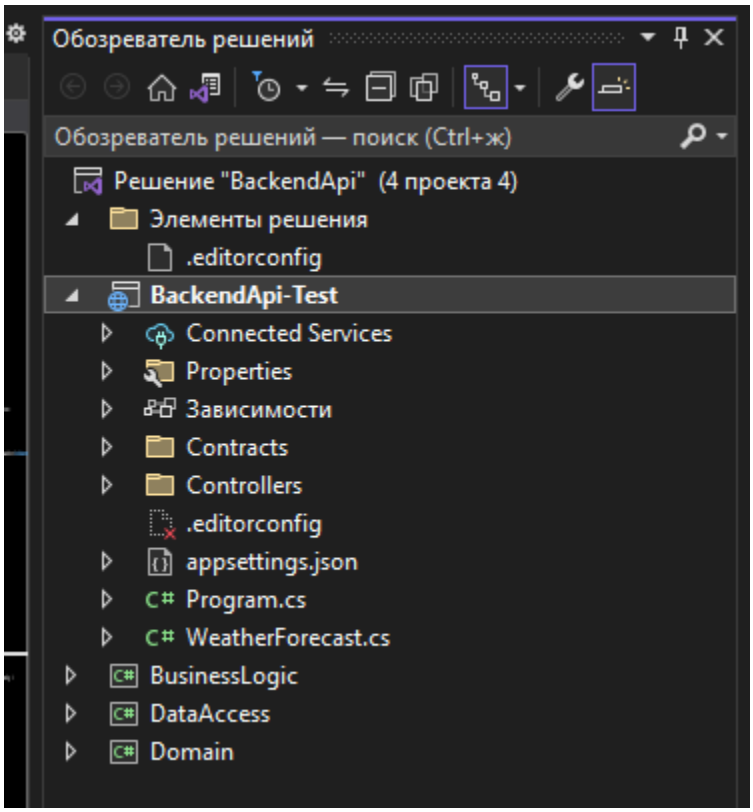
3. Нажмите ПКМ по решению. Выберите пункт “Существующий элемент…”



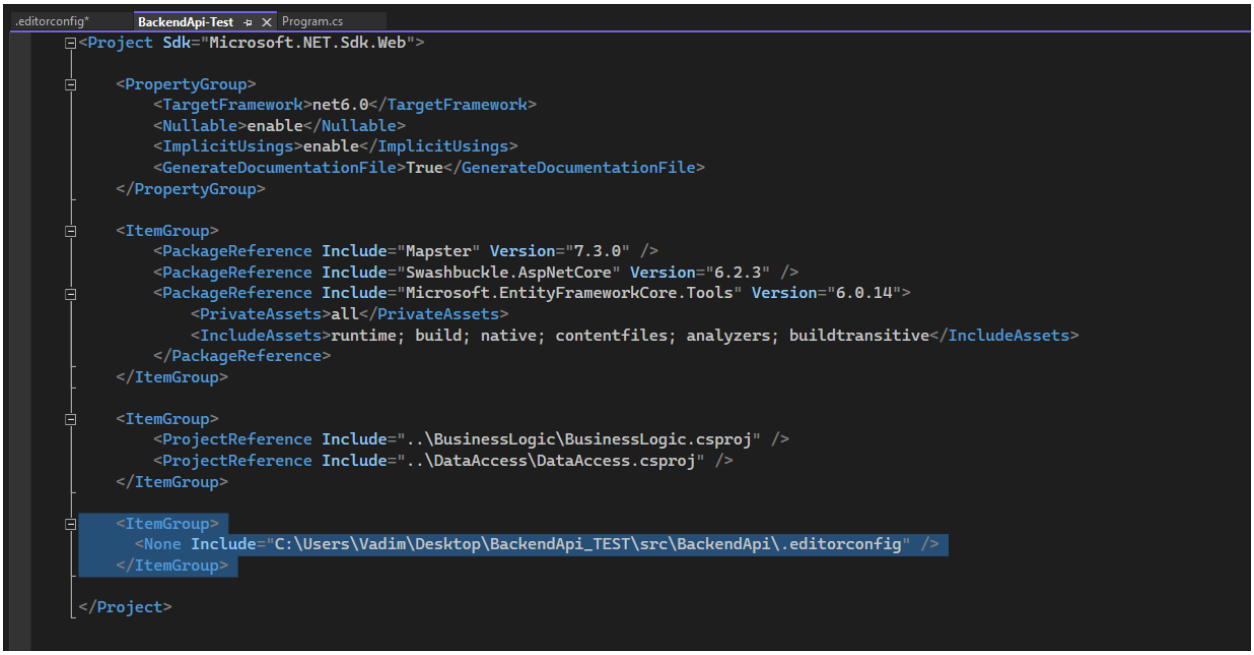
4. Выберите файл .editorconfig и добавьте его в решение



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 строчку, которая отвечает за файл конфигурации

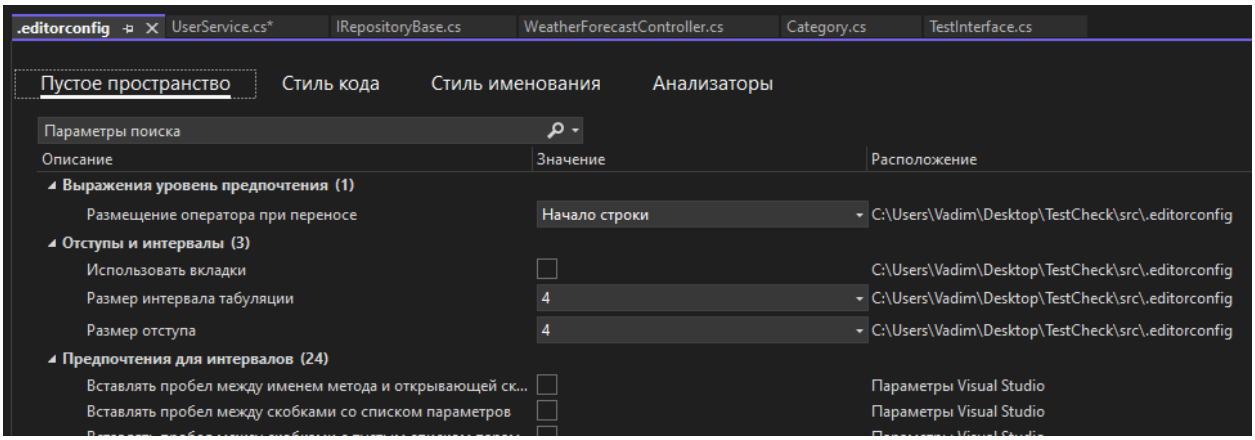


Установка правил форматирования для EditorConfig

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.

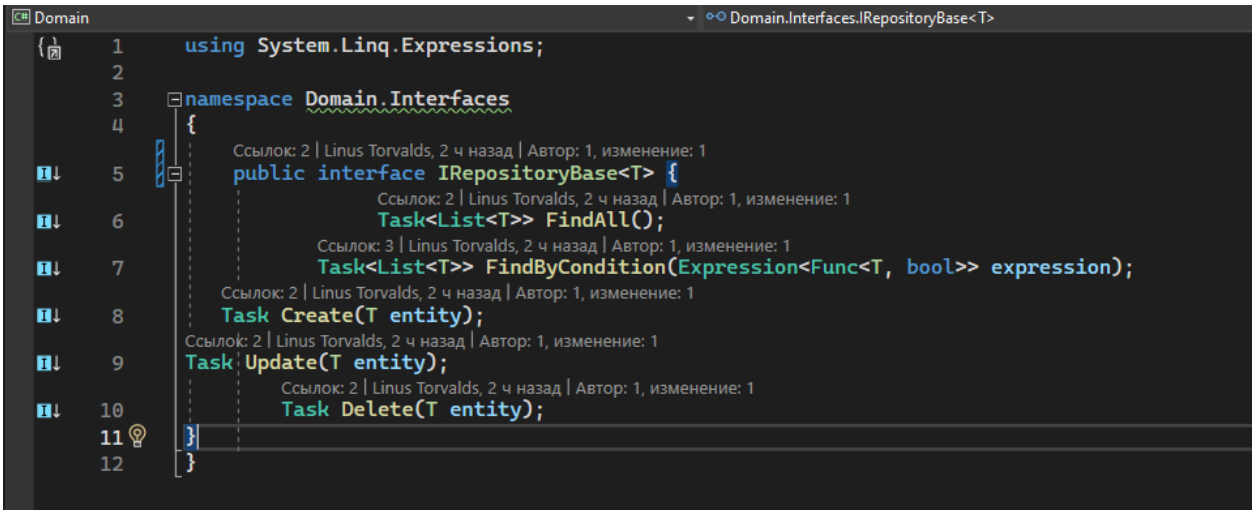


Раздел “Пустое пространство” в EditorConfig

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

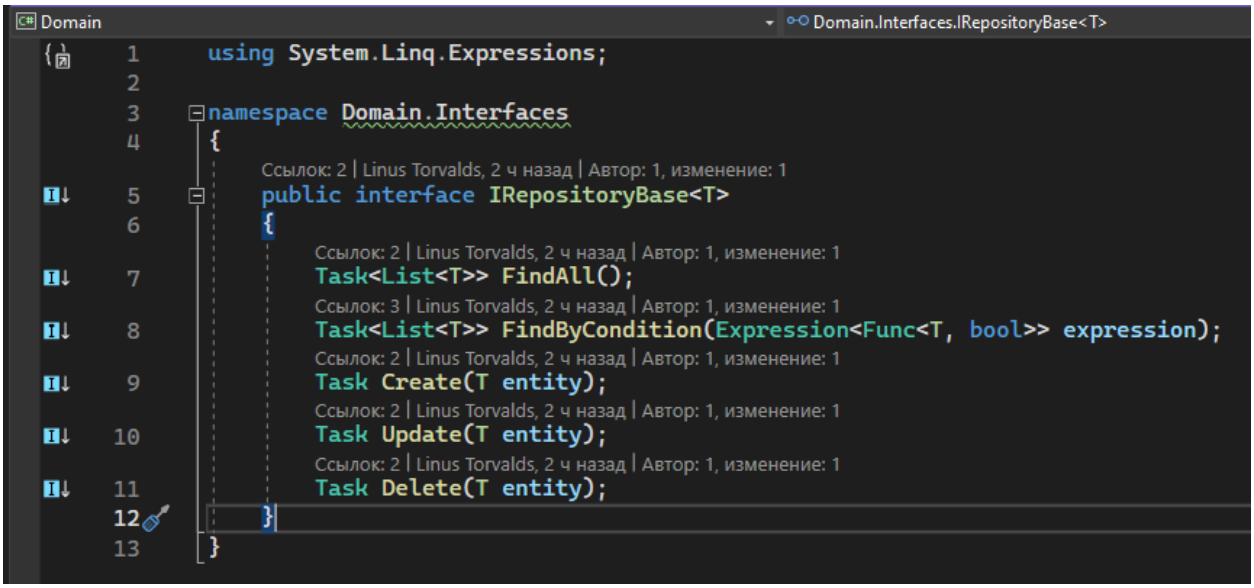
Попробуйте исправить форматирование для некоторых фрагментов кода, например.

Код, который требуется отформатировать:



Код у которого есть проблемы с отступами

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

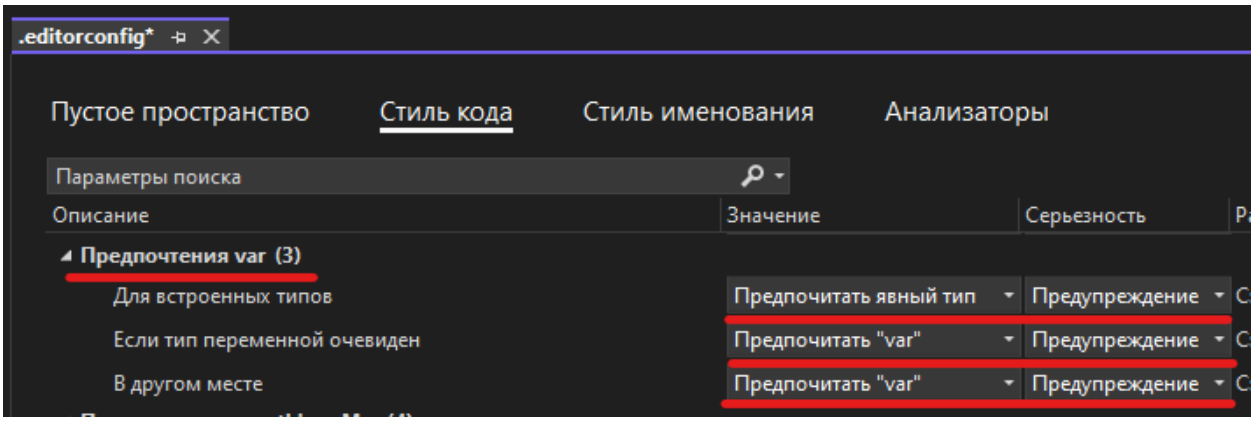


Исправленный код после форматирования

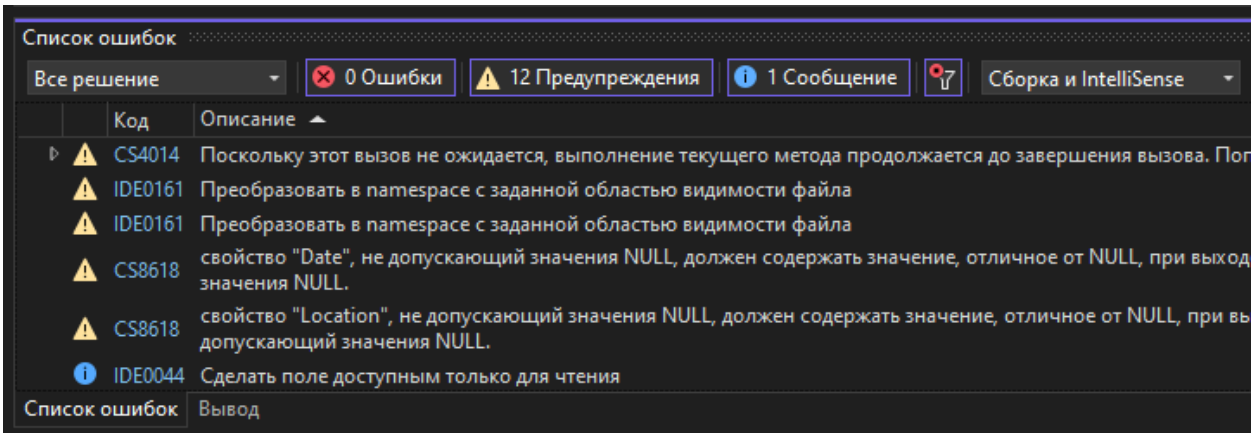
Стиль кода

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

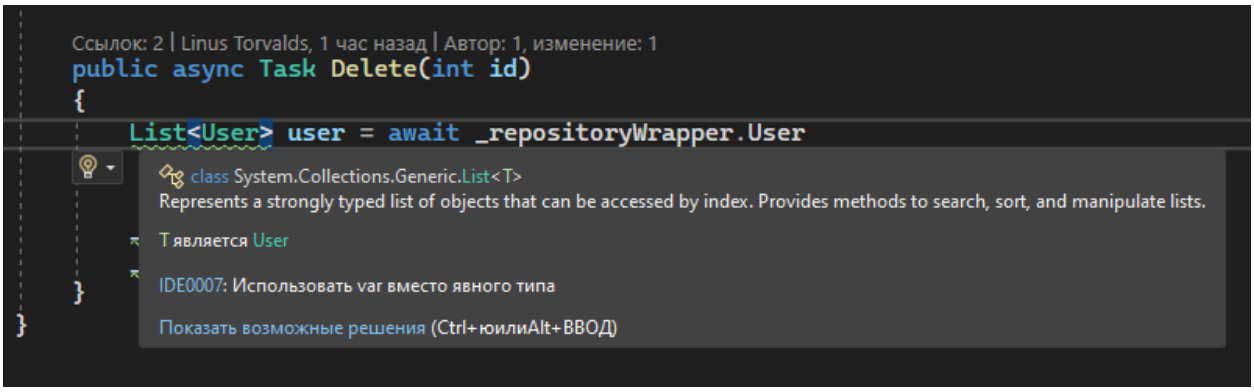
Установите для вашего проекта требование для использования оператора var при создании переменной. Укажите в данном разделе следующие параметры:



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

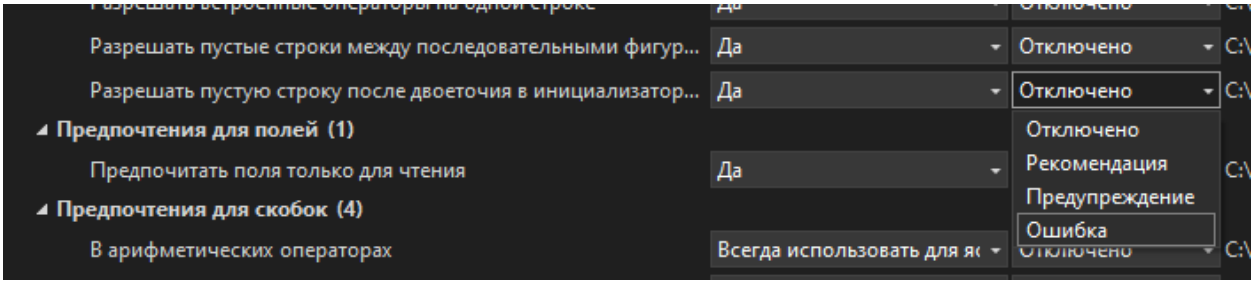


Также, сообщения помечают проблемный участок, которому требуется исправление.



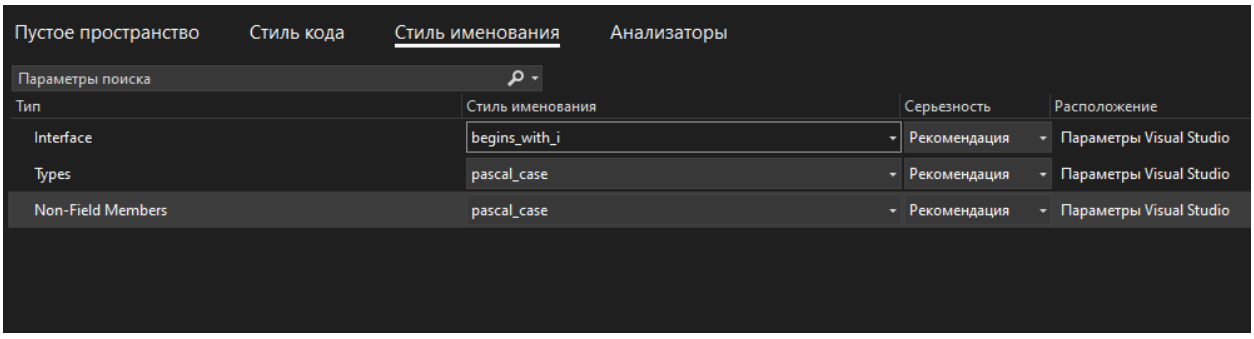
В зависимости от того, какой степенью серьезности обладает параметр в разделе “Стиль кода” - такие действия будет выполнять компилятор при обработке каждого файла.

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



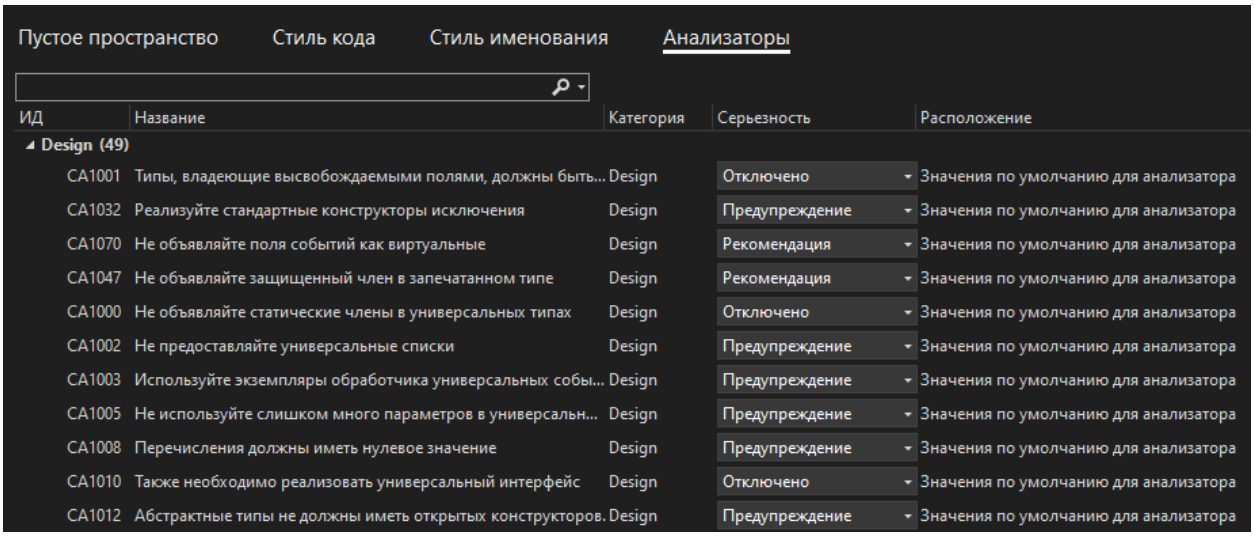
Стиль именования

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



Анализаторы

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



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.

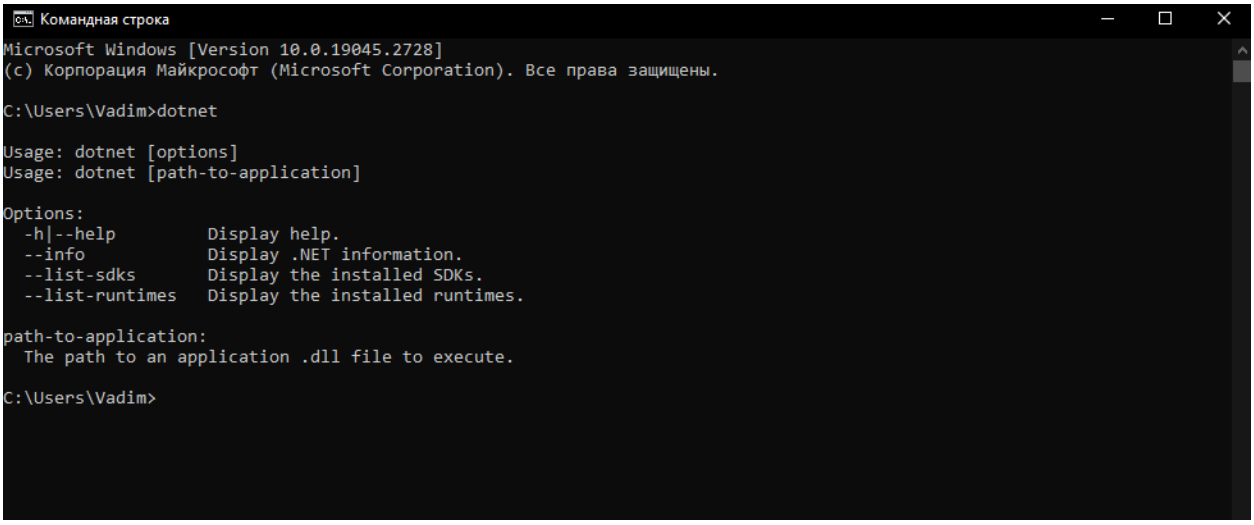
Однако, что делать, если необходимо запустить программу на сервере у которого не установлен IDE, чтобы собрать и запустить проект? Для этого можно обратиться к инструментам командной строки, таким как dotnet cli для работы с проектами .NET.

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.



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
new	Создает новый проект, файл конфигурации или решение на основе указанного шаблона.
restore	Восстанавливает зависимости и инструменты проекта.

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

Project Modification Commands	Description
add package	Добавляет пакет (библиотеку) в проект
add reference	Включает в качестве зависимости проекта другой проект
remove package	Удаляет пакет (библиотеку) из проекта
remove reference	Убирает из зависимости проекта другой проект
list reference	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 выполнилась подкачка пакетов (например библиотека Telegram.Bot), потребуется написание команды restore

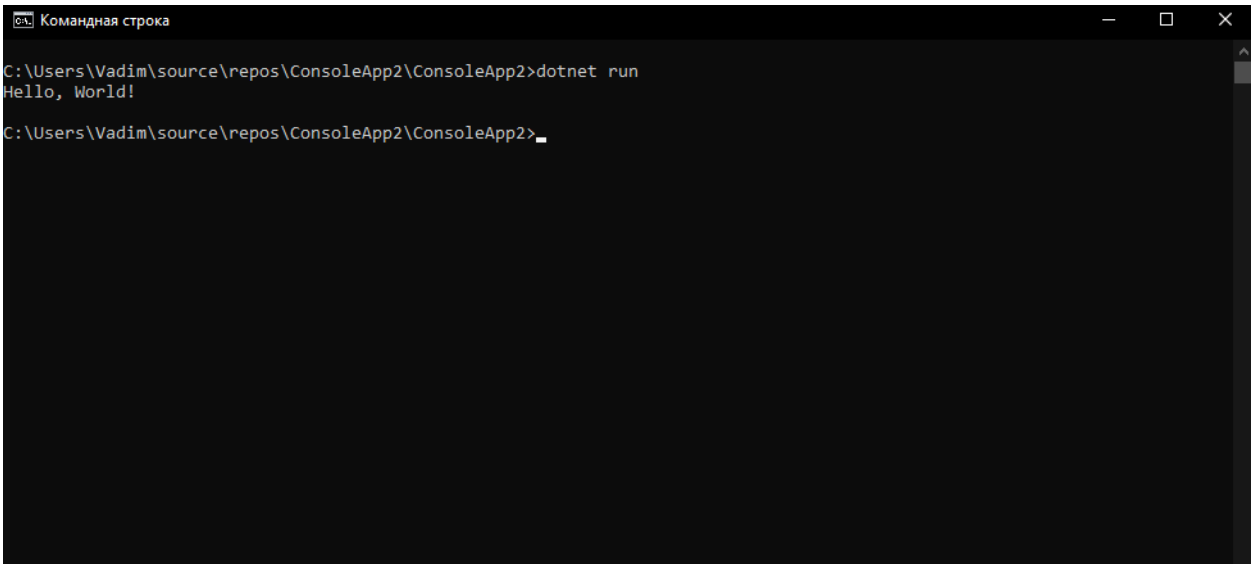
```
dotnet restore
```

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)

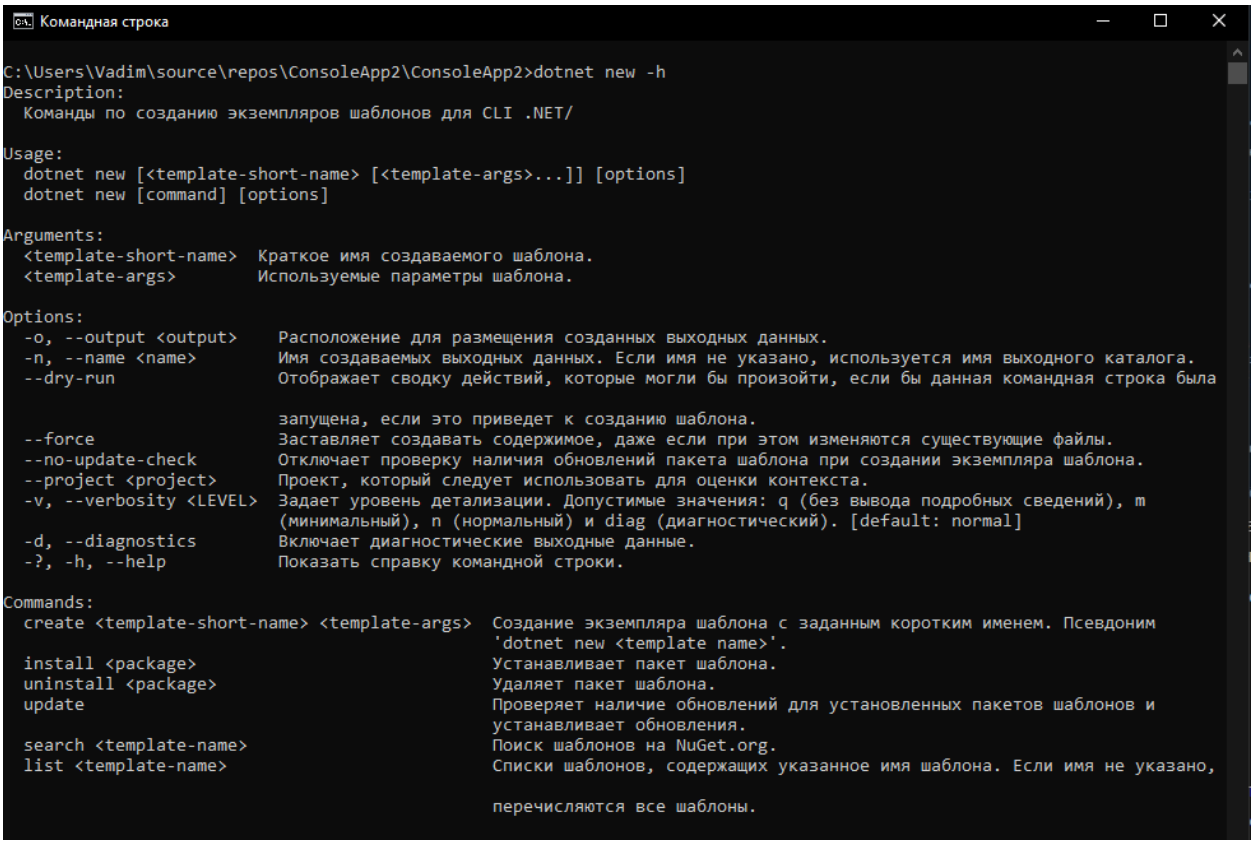


В данном случае, результат выполнения данной команды отобразил в консоли "Hello World!".

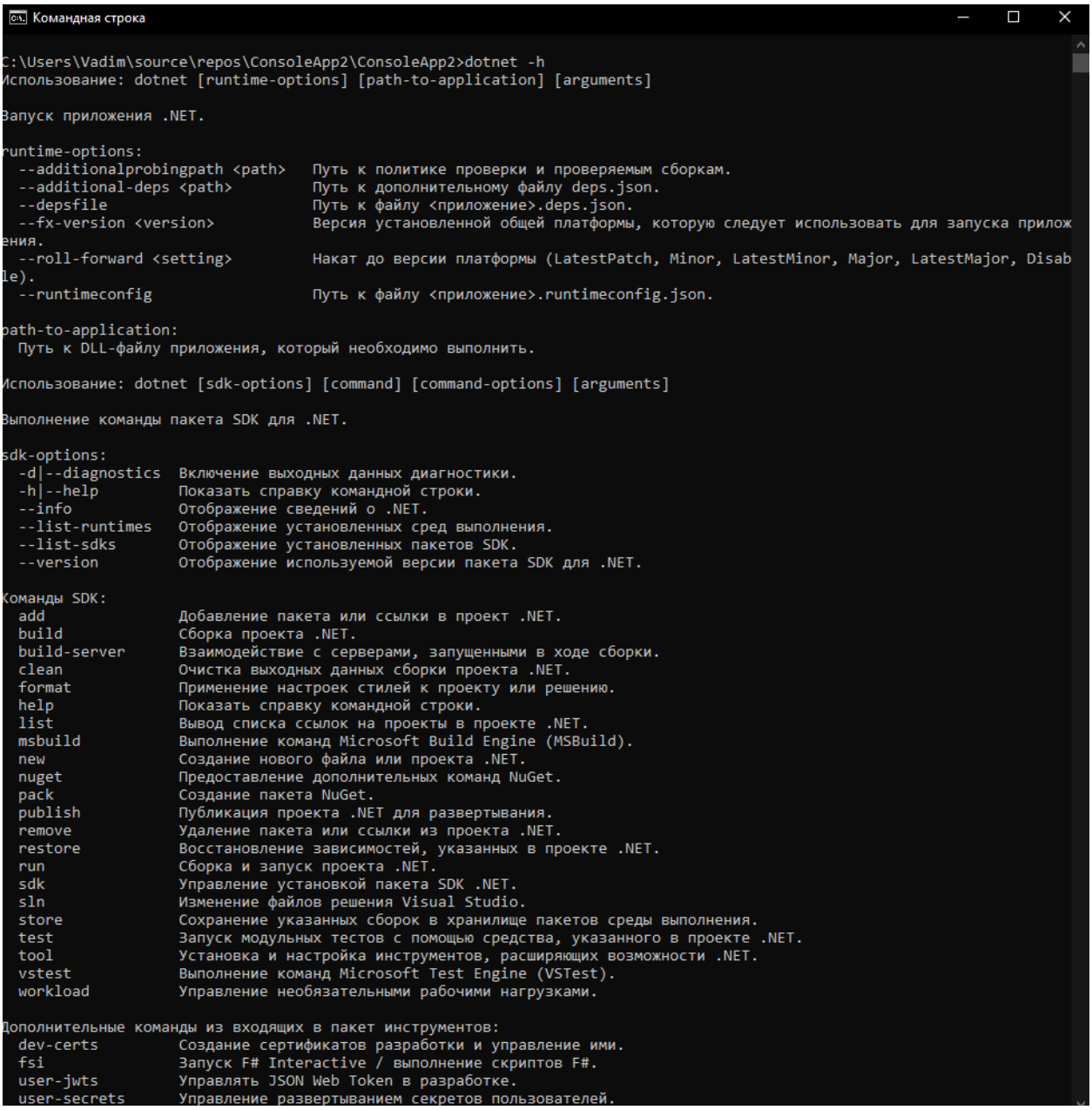
Подсказки для написания команд

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.



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



Проверка форматирования кода через командную строку

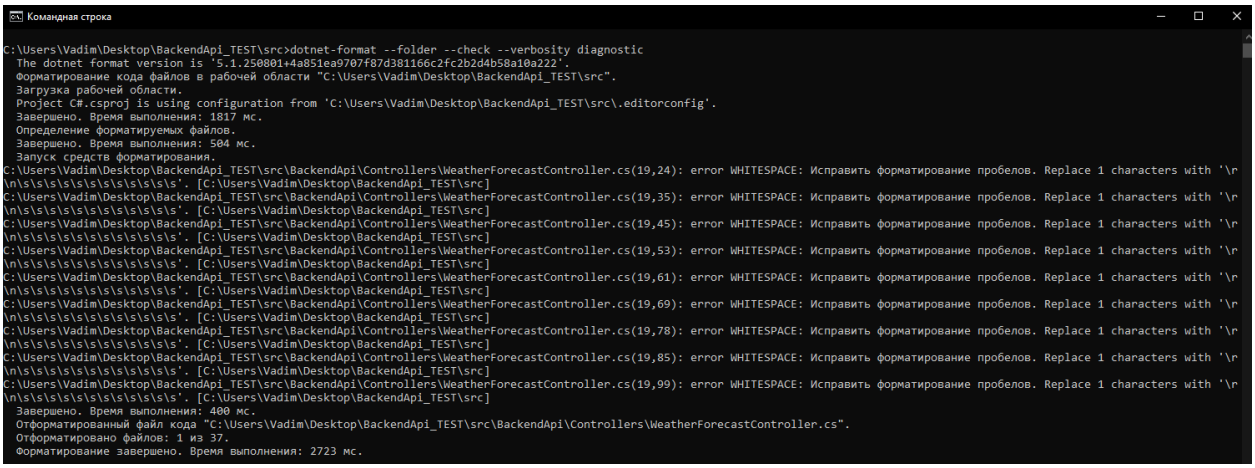
Для того, чтобы проверить качество форматирования кода вы можете использовать команду `dotnet-format`

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

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

Для того, чтобы проверить форматирование кода используйте команду:

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



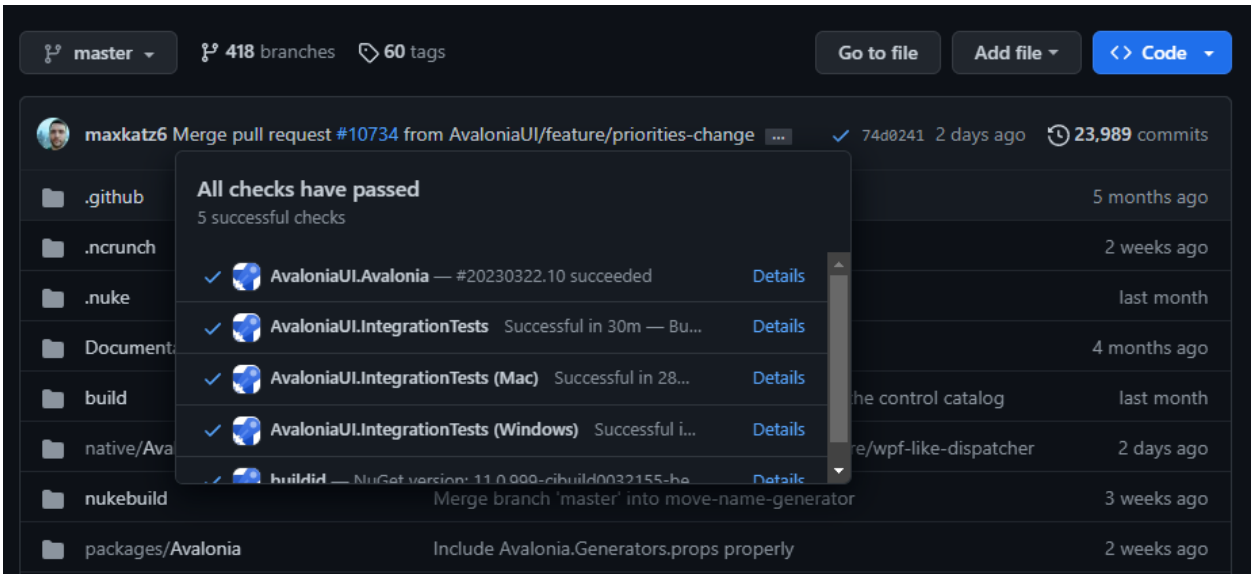
Чтобы форматирование автоматически исправила данные проблемы, уберите из команды опцию `--check`

Знакомство с GitHub Actions

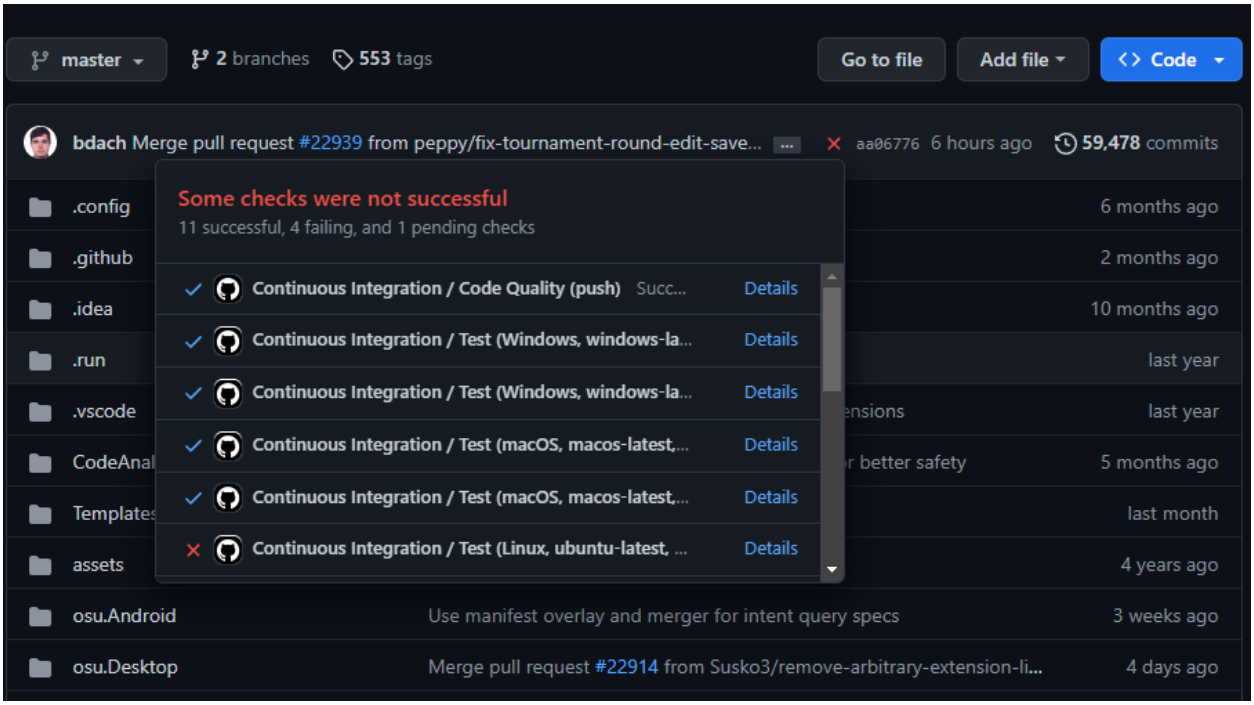
Так как наш проект расположен в репозитории на GitHub, у нас есть возможность познакомиться с процессом CI/CD при помощи инструмента GitHub Actions.

> 💡 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:



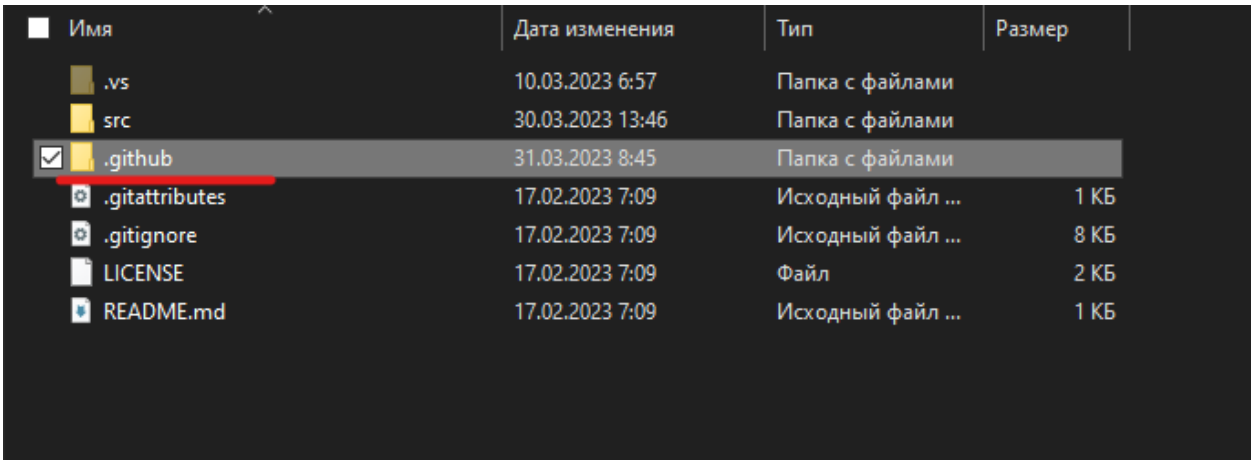
В главной ветке репозитория AvaloniaUI/Avalonia все проверки были пройдены успешно



В главной ветке репозитория ppy/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.



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
```

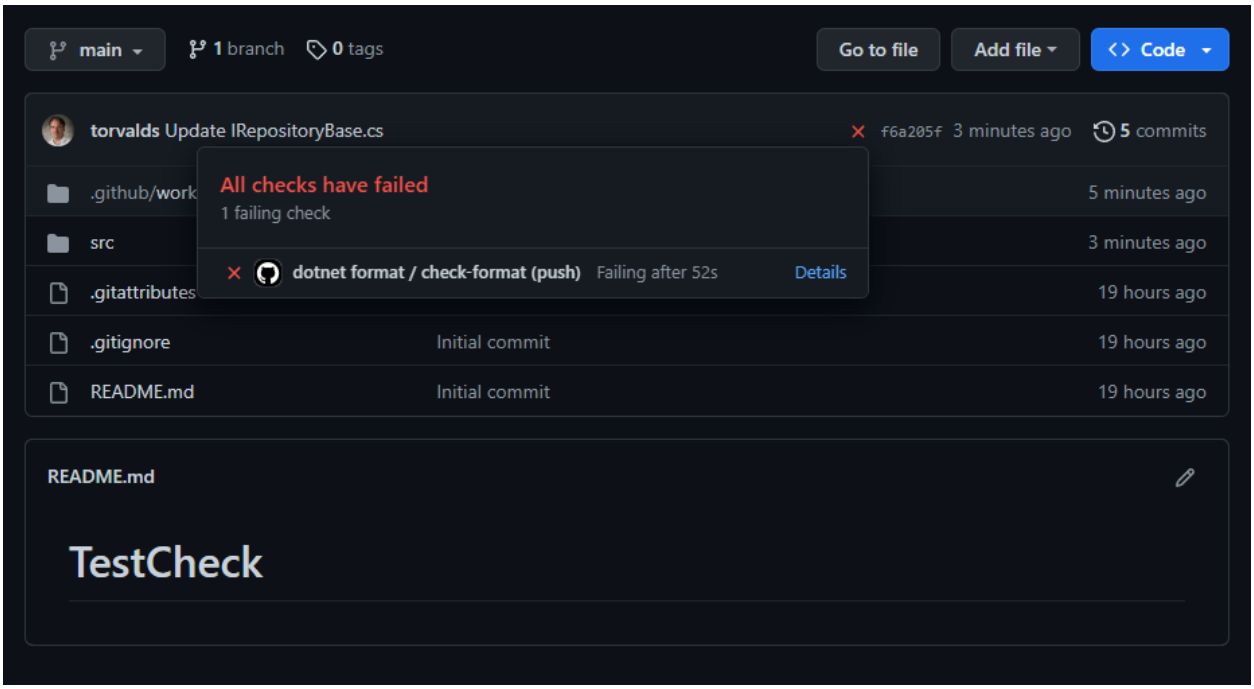
В этом примере мы настроили GitHub Actions для выполнения следующих действий:

- 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".

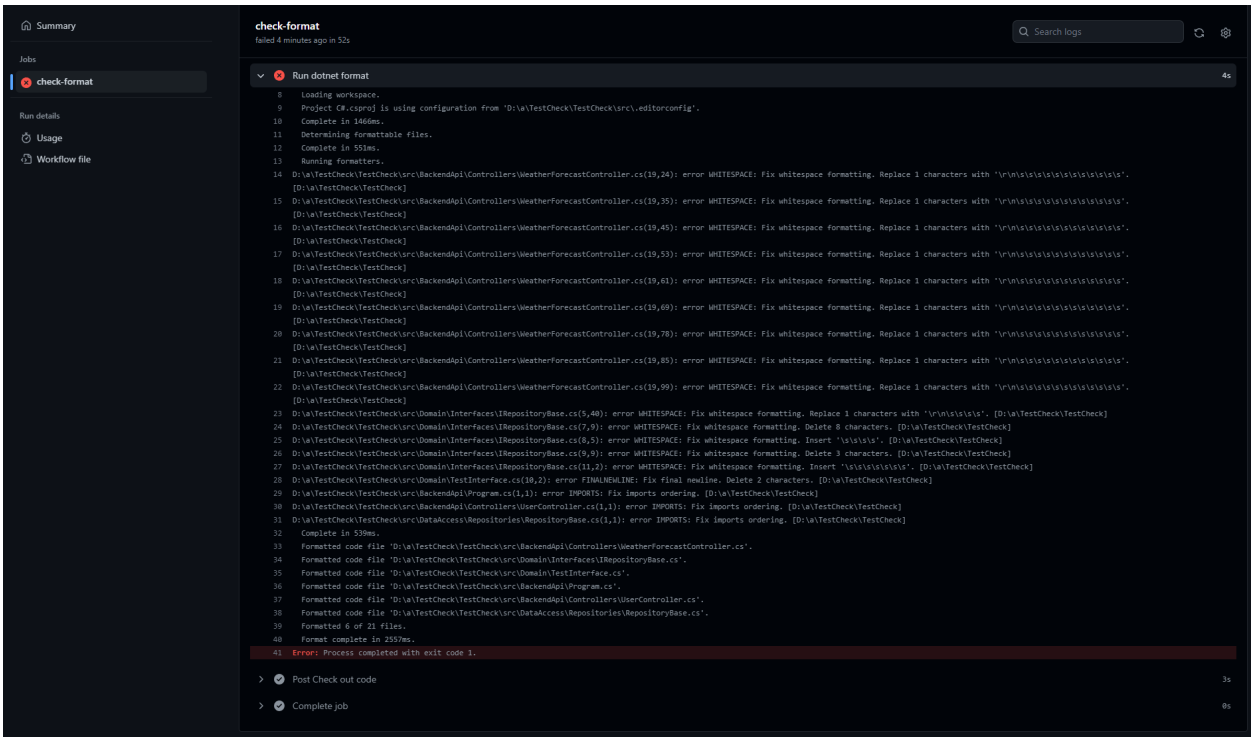
Запуск GitHub Actions

После того, как вы создали файл конфигурации для GitHub Actions, его нужно закоммитить и запушить 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 .yml-файл, который будет выполнять сборку проекта при каждом commit-е.