Rush 01. Under the weather

## Общие требования

* Убедитесь, что на вашем компьютере установлен [SDK для разработки на .NET 5](https://dotnet.microsoft.com/download) и вы используете именно его.
* Помните, ваш код будут читать! Обратите особое внимание на оформление вашего кода и именование переменных. Обязательно придерживайтесь общепринятых стандартов [C# Coding Conventions](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions).
* Самостоятельно выберите удобную для себя IDE.
* Программа должна иметь возможность запуска через командную строку dotnet.
* В задании указаны примеры ввода и вывода. Решение должно использовать их как верный формат.
* Если затрудняетесь в решении задачи, обратитесь с вопросами к другим участникам бассейна, интернету, Google, посмотрите на StackOverflow.
* С основными возможностями языка C# можно ознакомиться в [официальной спецификации](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction).
* Избегайте **хардкода** и “**магических чисел**”.
* Вы демонстрируете все решение, верный результат работы программы – лишь один из способов проверки ее корректной работы. Поэтому когда необходимо получить определенный вывод в результате работы ваших программ, запрещено показывать пред рассчитанный результат.
* Обращайте особое внимание на термины, выделенные **bold** шрифтом: их лучше сразу погуглить, их изучение пригодится вам как в выполнении текущего задания, так и в вашей дальнейшей карьере .NET разработчика.
* Have fun :)

## Требования к заданиям дня

* Используйте **var**.
* Название решения (и его отдельного каталога) должно выглядеть как d{*xx*}, где *xx* - цифры текущего дня. Названия проектов указаны в задании.
* Для форматирования выходных данных используйте **культуру** en-GB: N2 для вывода денежных сумм, d для дат.

## Intro

Если вам удалось поучаствовать в предыдущих днях бассейна, вы могли заметить частую простоту вывода в заданиях. Исключением был Rush 00, где вы могли попробовать себя в разработке desktop applications. Сегодня мы рассмотрим еще один способ обмениваться информацией с пользователем: **веб-приложения** и в частности **Web API**.

Темой сегодняшнего дня станет погода. Предмет любого разговора, повод восхищения и жалоб. И прогнозов. Не всегда точных, но всегда крайне полезных. Давайте разработаем веб-приложение, которое будет предоставлять API для получения прогноза погоды в выбранной локации.

Для разработки нашего API воспользуемся архитектурным подходом **REST**. В отличие от **SOAP-**сервисов **REST** не является стандартом, а составляет скорее набор практик и подходов, общую концепцию для имплементации веб-сервисов с интерфейсом доступа. Попробуем ее на деле.

# Exercise 00 Basics

Для [создания веб-приложения в .NET](https://docs.microsoft.com/en-us/learn/modules/build-web-api-aspnet-core/) воспользуйтесь стандартным шаблоном ASP.NET Core Web Application и выберите для него тип Web API. У вас должен получиться веб-проект, содержащий файлы конфигурации, классы настройки и **контроллер** с моделью WeatherForecast.

Ваш проект уже работает. Запустите и протестируйте написанный из коробки метод: /WeatherForecast должен возвращать 5 рандомно сгенерированных прогнозов. Пока эти данные статичны и заданы прямо в коде, далее мы изменим это. Сейчас же рассмотрим сам проект.

Рассмотрите файл Program.cs и разберитесь, что в нем происходит. Этот файл и его метод Main являются точкой входа при запуске приложения и используются во всех веб-проектах .NET.

В файле **Startup.cs**, так же важной части любого веб-проекта на .NET, происходит настройка приложения. Самостоятельно изучите, для чего нужны методы **Configure** и **ConfigureServices**, что такое **IApplicationBuilder**.

Текущей конфигурации будет достаточно для продолжения работы, но добавим документации к нашему API. В Startup есть обращения к сервису **Swagger** - функционалу для автоматической генерации документации. По адресу /swagger/index.html вы можете увидеть документацию к тем методам, которые у вас сейчас есть. Протестируйте их.

Добавьте для метода Get() в контроллере описание: саммари того, что он делает, и два типа ответа - 200, если все в порядке, и 400, если произошла ошибка. Для обозначения разных [статусов ответа](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status) метода отлично подойдет **ProducesResponseType**.

Откройте страницу Swagger вашего приложения и проверьте, что описание метода и возможных статусов его ответов верно отображается в сгенерированной документации.

Не забудьте проверить, что документация генерируется в XML файл.

# Exercise 01 Environment

Когда речь заходит о веб-разработке, предполагается, что **backend** приложение может быть запущено в разных **environment**: оно будет отлаживаться локально на машине разработчика, работать на dev сервере для тестирования командой и после релизиться на production.

При этом некоторые настройки приложения могут (и должны) отличаться для разных окружений. Это решается несложно. Во-первых, файлы конфигураций appsettings.json имеют постфикс с названием окружения, где применяется конкретно эта конфигурация.

Во-вторых, в файле *Startup* вы можете найти свойство типа **IConfiguration** и параметр типа **IWebHostEnvironment**. Здесь вы можете, например, обнаружить, что swagger-документация к API включена только для dev конфигурации. Это сделано для того, чтобы не давать доступ к детальному описанию вашего интерфейса в production версиях.

Приложение узнает о том, [что это за среда](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-5.0), при запуске. Для этого оно считывает значение аргумента **ASPNETCORE\_ENVIRONMENT**.

Запустите приложение в среде Production. Сделайте так, чтобы в этой конфигурации swagger-документация была недоступна.

# Exercise 02 GET weather by coordinates

Сделаем запрос прогноза погоды актуальным. Воспользуйтесь публичным [OpenWeather API](https://openweathermap.org/current): зарегистрируйтесь, получите API key и измените метод Get() своего контроллера так, чтобы он принимал географические координаты (широту и долготу) и возвращал информацию о погоде в них с помощью вызова OpenWeather.

Выведите информацию о температуре, давлении, влажности, скорость ветра, название географического объекта и описание погоды.

Добавьте описание методу и модели. Проверьте, что оно отображается в сваггер-документации.

## Пример вызова

{

* **wind**:
* {
  + **speed**: 4.47
* },
* **weather**:
* [
  + {
    - **description**: "overcast clouds"
  + }
* ],
* **main**:
* {
  + **temp**: 29.27,
  + **pressure**: 1007,
  + **humidity**: 57
* },
* **name**: "Kizicheskaya"

}

# Exercise 03 GET weather by city

Добавьте в контроллер еще один **HttpGet** метод, который будет принимать название города и возвращать прогноз погоды в этом городе. Воспользуйтесь соответствующим методом в [OpenWeather API](https://openweathermap.org/current).

Используйте формат вывода из Exercise 02. Обратите внимание на **routing**, его можно реализовать [с помощью атрибутов](https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2). Сделайте так, чтобы метод был доступен по ссылке

/WeatherForecast/{cityName}.

Добавьте описание методу и модели. Проверьте, что оно отображается в сваггер-документации.

## Пример вызова

{

* **wind**:
* {
  + **speed**: 4.49
* },
* **weather**:
* [
  + {
    - **description**: "overcast clouds"
  + }
* ],
* **main**:
* {
  + **temp**: 29.34,
  + **pressure**: 1007,
  + **humidity**: 57
* },
* **name**: "Kazan’"

}

# Exercise 04 Code refactoring

## Structuring

Давайте приберемся. Чтобы быть готовыми к расширению функционала и воспользоваться принципами **SOLID**, выделите весь код, относящийся к OpenWeather API, в отдельный класс *WeatherClient*. У вас должен получиться класс с двумя методами получения прогноза погоды: по координатам и по названию города. Оба метода должны обращаться к OpenWeather API, так что этот функционал будет **инкапсулирован**. В контроллере останется только вызвать эти методы с нужными аргументами.

Выделите модель прогноза погоды и сервис работы с OpenWeather API в отдельный проект *rush01.WeatherClient*. Чтобы проверить себя, задайтесь вопросом: что будет, если я решу использовать другой способ ввода и вывода информации, и вместо web приложения это будет desktop приложение или консольное приложение?

При таком разделении вы легко сможете переиспользовать проект, реализующий основную логику, и просто подключить к нему другое приложение с новой точкой входа.

## API KEY

Обратите внимание, что OpenWeather API имеет разные варианты тарификации и соответственно разные ограничения. Это значит, что тестовый аккаунт вполне может подойти для разработки, но в production среде возможно придется приобрести реальную лицензию (гипотетически). Значит ключ, который мы используем для подключения, может быть разным и зависеть от среды.

Вынесите значение API KEY в файлы *appsettings.json* и *appsettings.Development.json* и сделайте так, чтобы оно передавалось в класс *WeatherClient* и использовалось как параметр.

## DI

Задумайтесь, как создается объект *WeatherClient* в контроллере, который его использует:

* Мы можем создавать новый экземпляр каждый раз, когда в нем есть необходимость.
* Мы можем сделать его свойством контроллера и создавать новый экземпляр в конструкторе контроллера один раз, после использовать заполненное свойство.
* Мы можем заполнять свойство значением, которое является параметром конструктора и передается уже готовое в него.

Все три способа легитимны, их использование зависит от требований к правилам создания, времени жизни, количеству экземпляров объекта. Чаще всего удобно использовать именно третий вариант - к нему же куда удобнее писать unit тесты.

Откуда же будет вызываться конструктор контроллера, чтобы передать ему нужный параметр? Здесь на помощь нам придет **Dependency Injection**. Это форма **Inversion of control** (IoC), которая позволяет внедрять зависимости извне. По сути, стиль настройки объекта, при котором поля объекта задаются внешней сущностью. В нашем случае - [контейнером служб, встроенным в .NET средством](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0).

Отредактируйте WeatherForecastController так, чтобы он содержал

private readonly WeatherClient \_weatherClient;

и чтобы значение этого свойство передавалось в конструктор контроллера и задавалось там. Обратите внимание, что свойство является **private** и **readonly**, зачем это нужно?

Отредактируйте *Startup.cs* так, чтобы зависимость *WeatherClient* регистрировалась в методе *ConfigureServices* при запуске приложения. Служба внедрит объект в конструктор контроллера там, где он используется. Платформа берет на себя создание экземпляра зависимости и его удаление, когда он больше не нужен.

Запустите приложение и проверьте, что все работает верно.

## Опции

Используя Dependency Injection можно [зарегистрировать объект опций](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options?view=aspnetcore-5.0), который позволит внедрять элементы конфигурации туда, где они нужны. Помните, мы вынесли значение ключа в appsettings?

Создайте класс *ServiceSettings* с полем *ApiKey* и отредактируйте конструктор *WeatherClient:*

public WeatherClient(IOptions<ServiceSettings> options)

Сделайте так, чтобы объект заполнялся значением из файла appsettings. Эта зависимость также должна быть зарегистрирована в методе *ConfigureServices Startup.cs.*

# Exercise 05 POST/PUT/DELETE, repeat

Кроме HTTP метода **GET** вам могут пригодится и [другие](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods), у каждого из них есть своя область применимости. Обратите, например, внимание на разницу между использованием методов **PUT**, **POST**, **PATCH**. Прочитайте, что такое **идемпотентность**.

Давайте сделаем так, чтобы при желании можно было установить город по умолчанию. Если он установлен, то в методе Get из Exercise 03 будет возвращаться:

* если название города передано в метод: погода для этого города;
* если название города не передано в метод и город по умолчанию задан: погода для города по умолчанию;
* если название города не передано в метод и город по умолчанию не задан: [404](https://developer.mozilla.org/ru/docs/Web/HTTP/Status).

Добавьте в контроллер метод Post(), принимающий название города. Не забудьте добавить методу описание! Если все в порядке, метод должен возвращать HTTP status OK.

Так как одной из особенностей принципа REST является **serverless**, а для хранения одной строки заводить целую базу данных расточительно, давайте воспользуемся [кэшированием](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-5.0).

Измените код приложения так, чтобы метод Post() сохранял переданный ему город в кэше, а метод Get из Exercise 03 использовал его при необходимости. Проверьте, что все работает верно:

1. Вызовите метод Get без указания города. Метод должен вернуть статус 404.
2. Вызовите метод Get с указанием города Барнаул. Метод должен вернуть погоду для города Барнаул.
3. Вызовите метод Post с указанием города Альбукерке. Метод должен вернуть статус 200.
4. Вызовите метод Get с указанием города Барнаул. Метод должен вернуть погоду для города Барнаул.
5. Вызовите метод Get без указания города. Метод должен вернуть погоду для города Альбукерке.