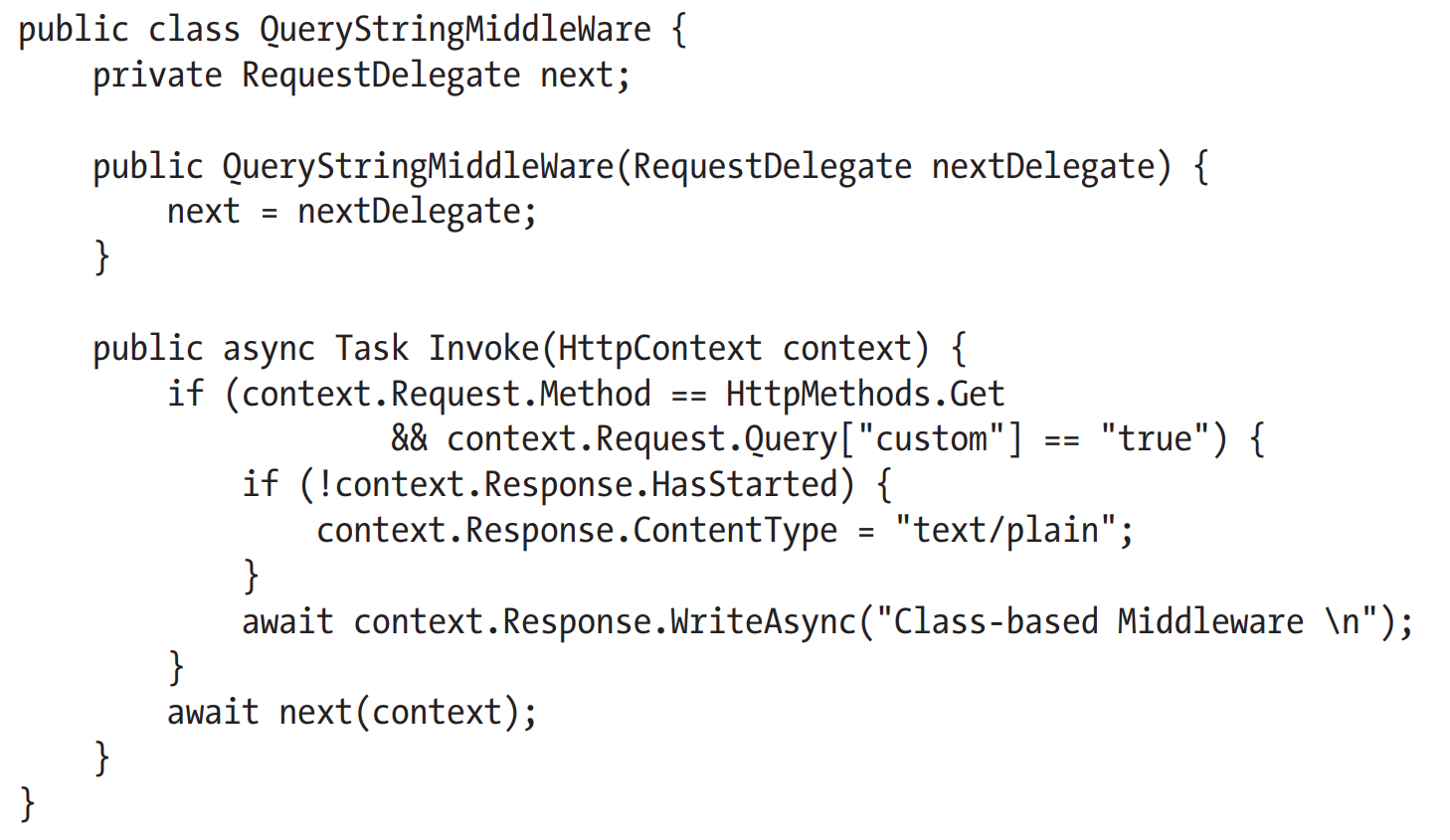
**Часть 12. Основы платформы ASP.NET Core.**

Пример создания кастомного middleware в файле Program.cs:



Пример создания кастомного middleware, используя отдельный класс:

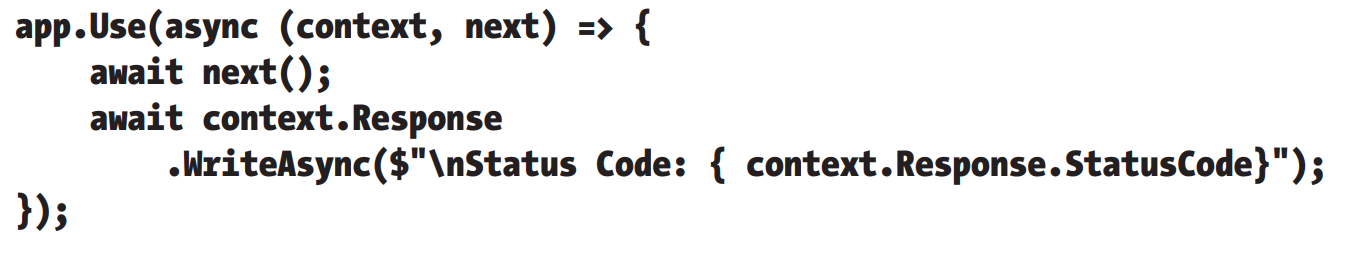


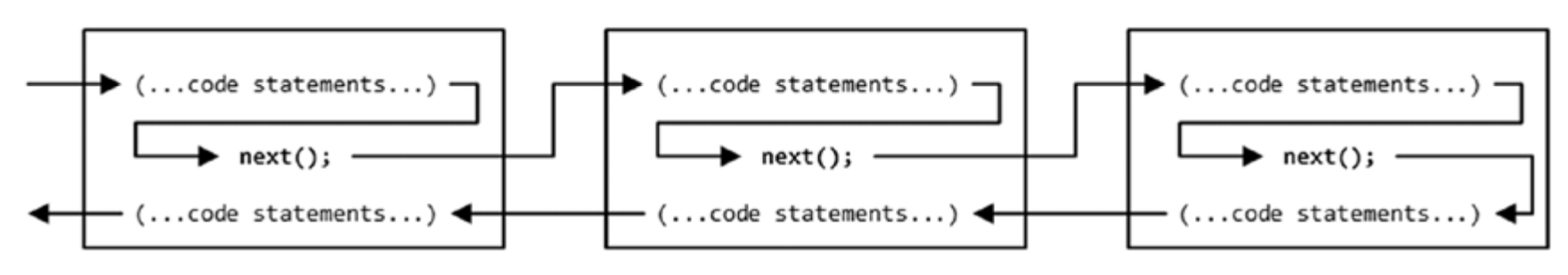
В отличие от определения в коде файла Program.cs next принимает в качестве параметра context.

В дальнейшем использование кастомного middleware в файле Program.cs:

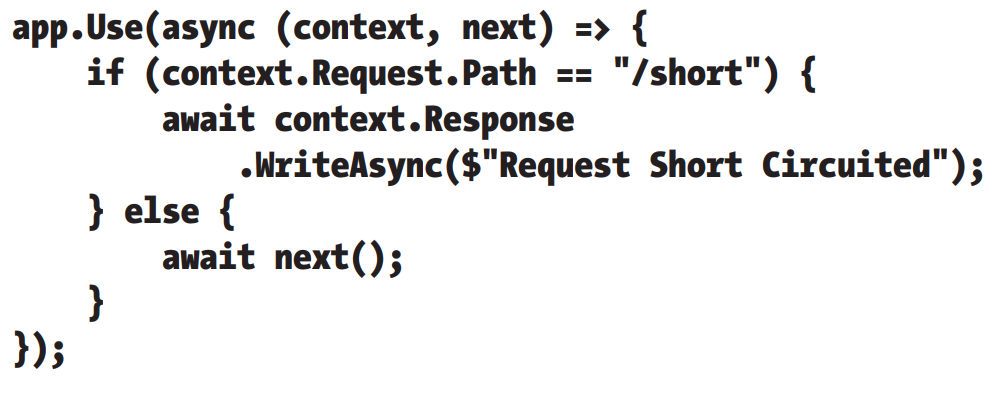


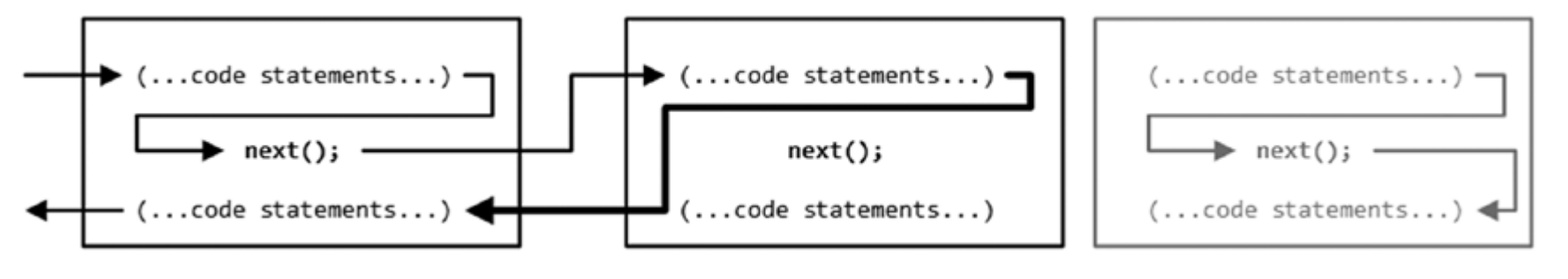
Если в middleware после метода next() имеется код, тогда этот код выполняется только после прохождения по всем middleware в цепочке pipeline:





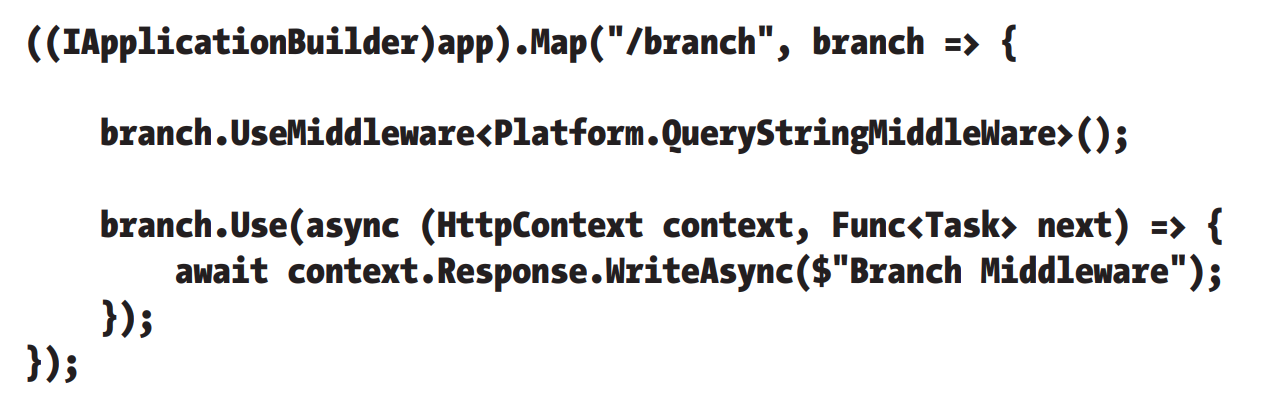
Коротко-замкнутые middleware (Short-Circuiting):



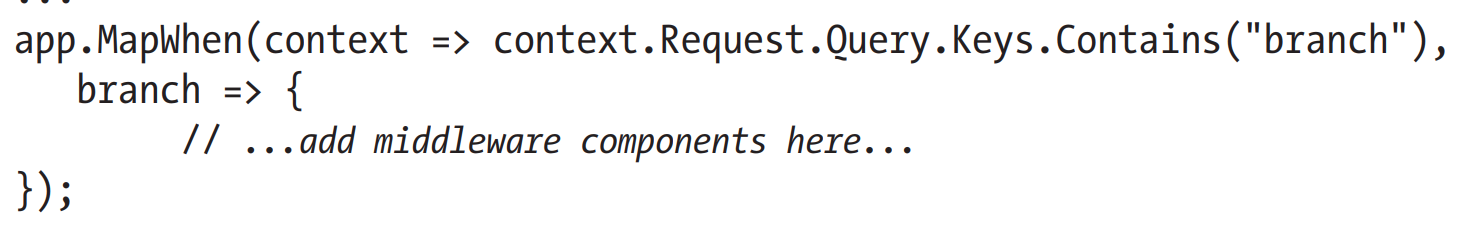


Создание отдельных ветвей pipeline:

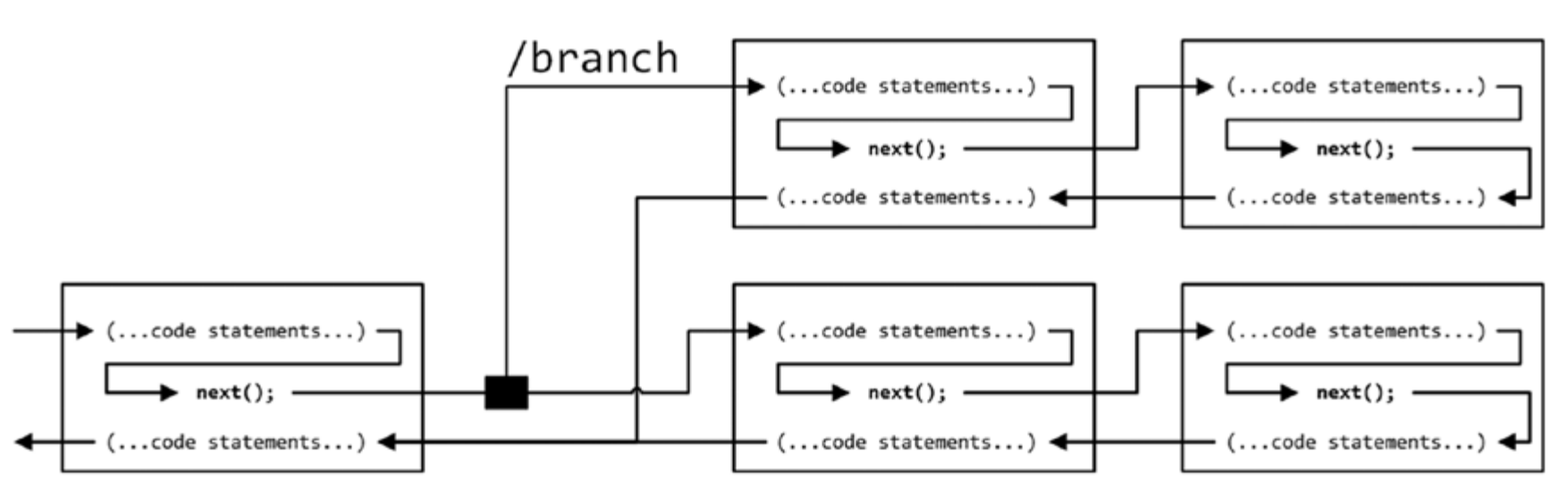
1 способ, используя метод Map() интерфейса IApplicationBuilder:



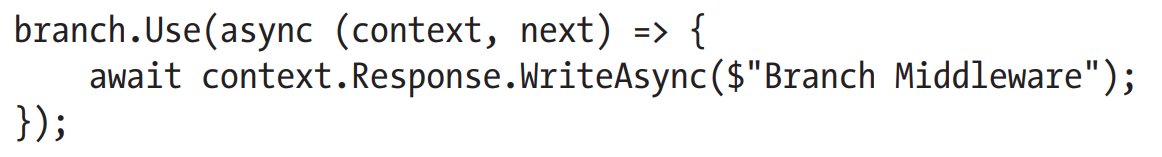
2 способ, используя метод MapWhen(), принимающий предикат:



Результат:



Терминальные middleware, это middleware, конечные в цепочке pipline, не содержащие метод next():

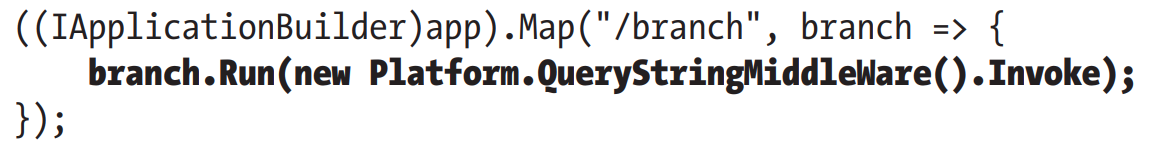


Для терминальных middleware для лучшей читаемости кода предпочтительнее использовать метод Run(), анонимный метод которого не принимает next:



Мiddleware, описанные в отдельном классе, могут также быть терминальными и регулярными, причем, одну и ту же middleware можно использовать для обоих случаев, если предусмотреть в нем конструктор без параметров, а в коде файла Program.cs использовать так:

Терминальный:

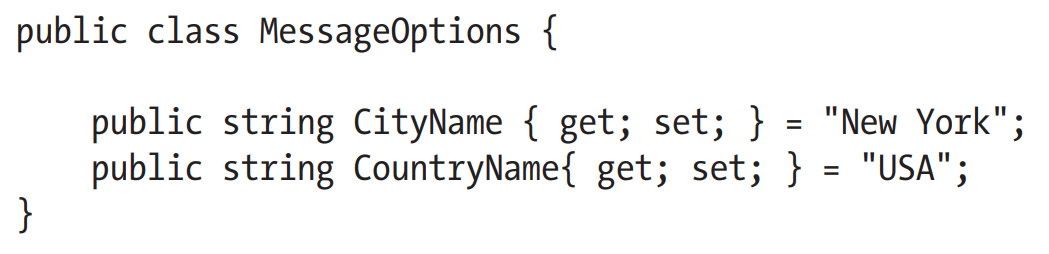


Регулярный:

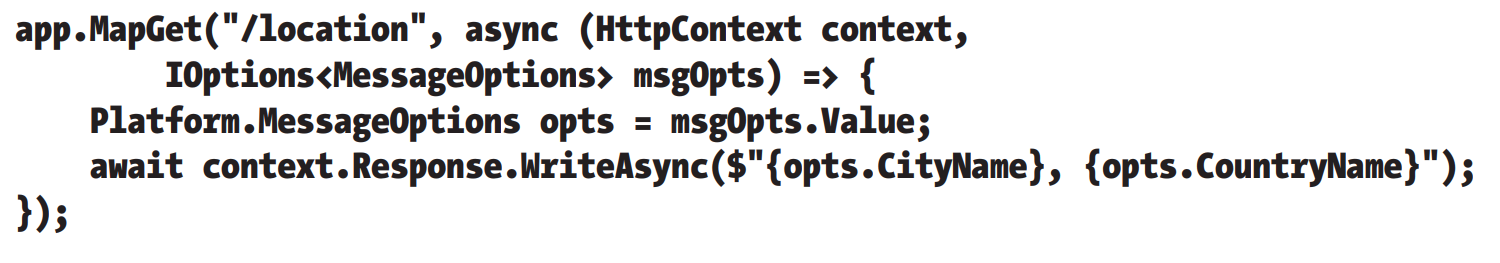


Имеется удобный паттерн для конфигурирования middleware, известный как options pattern.

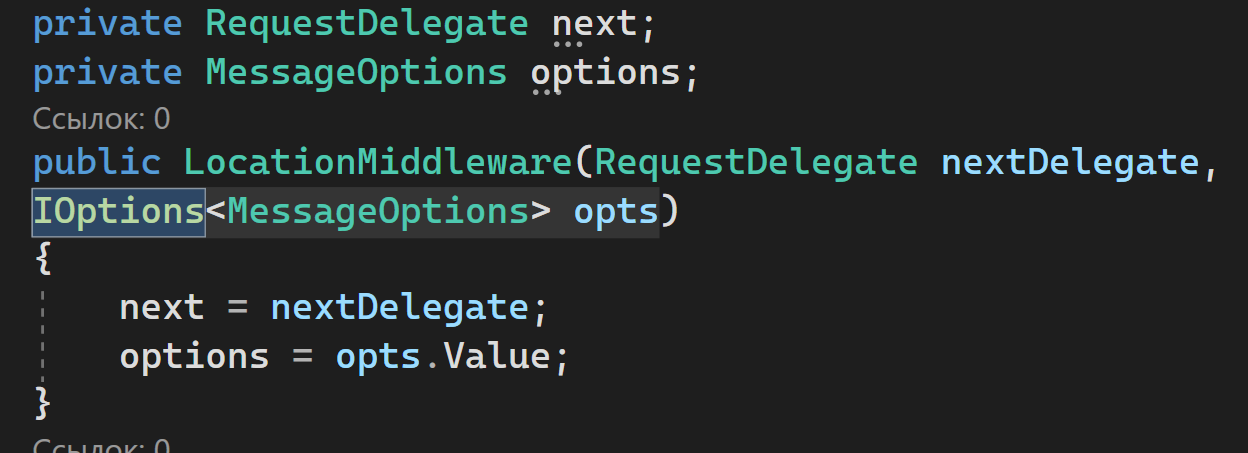
Для примера определим класс:



Далее определим middleware прямо в файле Program.cs, основывающийся на данном классе. Данный middleware во втором параметре вместо RequestDelegate next получает доступ к классу MessageOptions путем использования дженерик интерфейса IOptions:



Данный middleware также можно определить в отдельном классе, передав в конструктор IOptions<MessageOptions> opts:



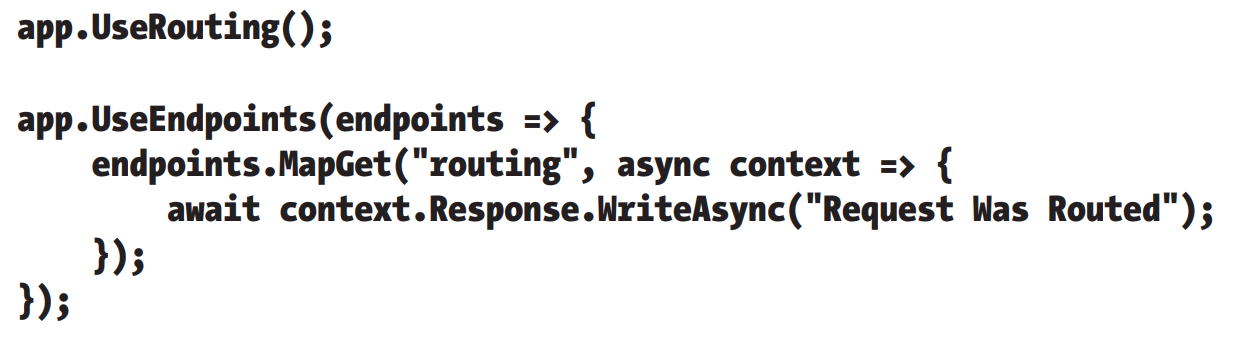
При необходимости изменить какие-то свойства класса MessageOptions налету во время выполнения, можно использовать Services.Configure дженерик метод, определенный в классе WebApplicationBuilder:



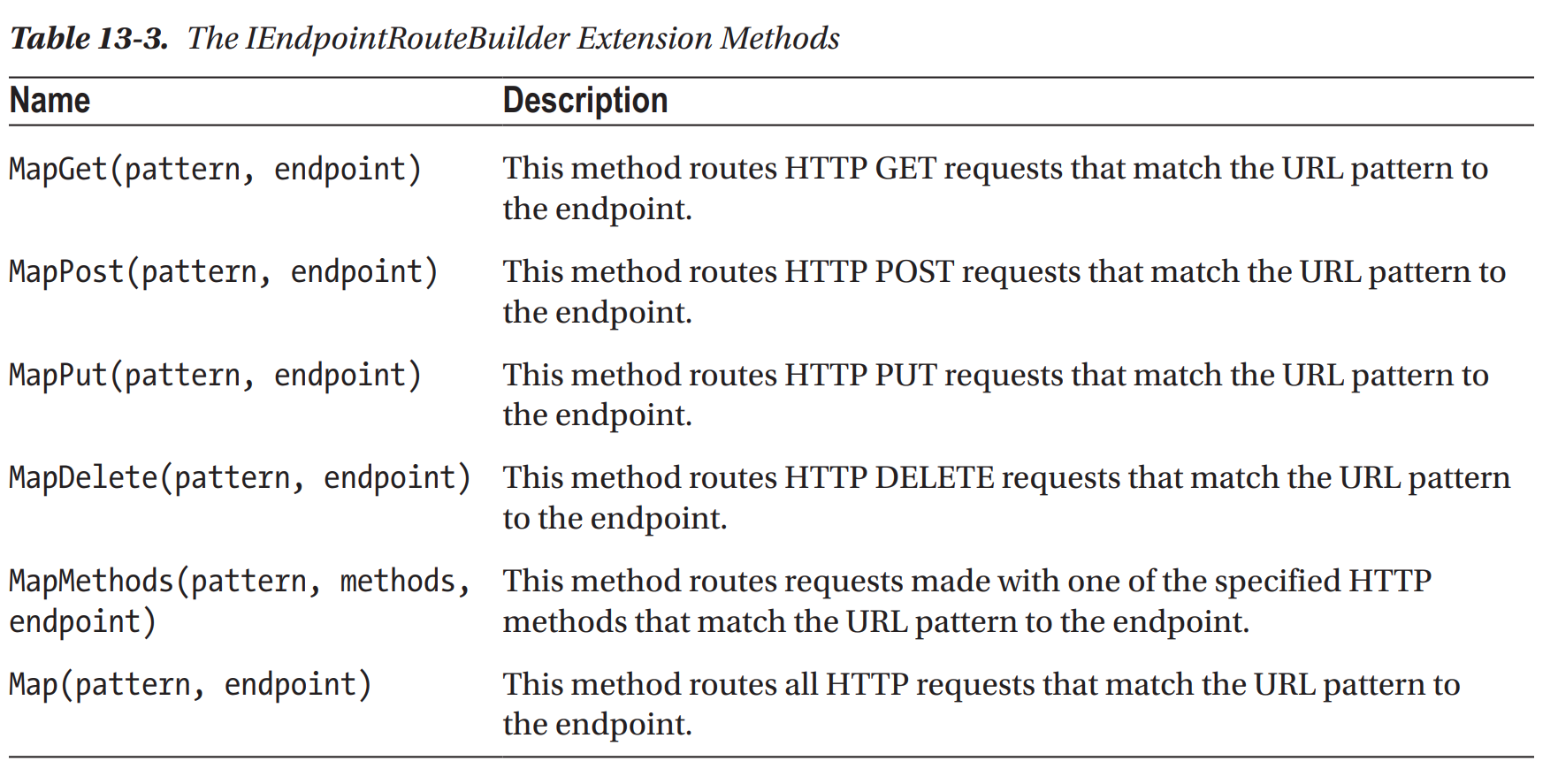
**Часть 13. Использование URL роутинга.**

Часто вместо middleware используют роутинг, который удобен в поддержке и исключает повторение кода.

Роутинг в приложение добавляется при помощи двух методов в классе Program.cs: UseRouting и UseEndpoints. UseRouting добавляет middleware, ответственный за обработку запроса в pipeline. UseEndpoints используется для определения соответствующего маршрута. URL определяются на соответствие с использованием шаблона, сравнивающегося с путем в запросе:

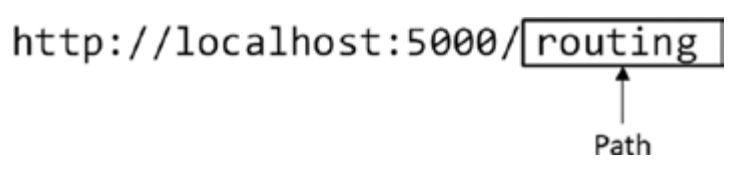


В данном примере UseRouting вызывается без аргументов, а UseEndpoints содержит анонимную функцию, принимающую объект IEndpointRouteBuilder (context) и использующую ее для создания роутинга и использованием метода расширения MapGet. Также имеются и другие методы расширения:



Ендпоинты определяются с использованием RequestDelegate, который используется и для создания обычных middleware, поэтому ендпоинты являются асинхронными методами, принимающими объект HttpContext и использующие его для генерации ответа. Это означает, что middleware могут быть использованы в ендпоинтах.

Когда сопоставляется запрос, применяется шаблон сопоставления секции пути URL, следующей за первой косой чертой:



Путь в URL адресе соответствует шаблону, указанному в маршруте:

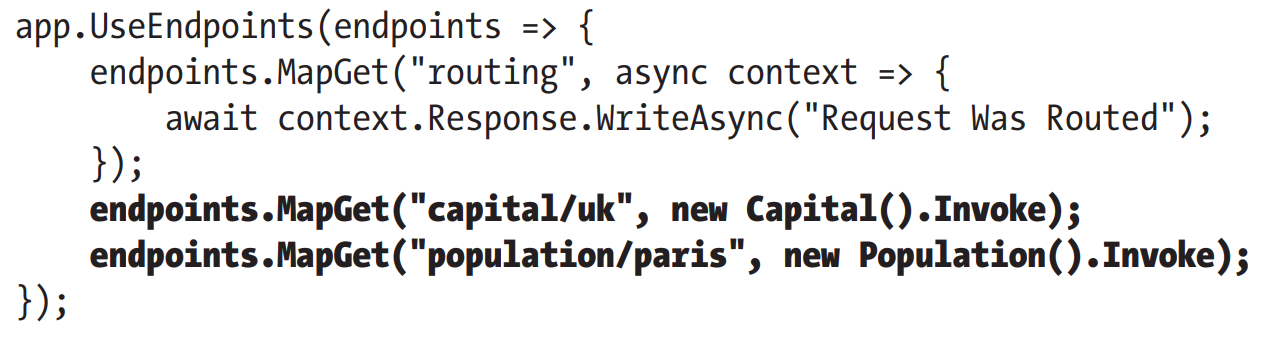


При этом шаблон указывается без косой черты перед ним.

Роутинг middleware коротко-замкнутая (short-circuits) и если запрос соответствует роутингу запрос не передается дальше в цепочке middleware.

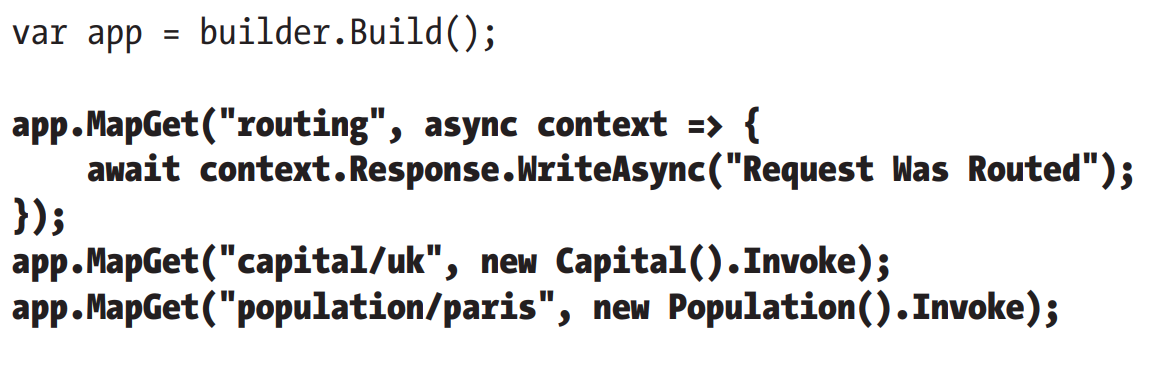
Если запрос не соответствует ни одному роутингу, тогда Роутинг middleware пропускает запрос следующей middleware в цепочке.

Ендпоинты генерируют ответ так же, как и middleware, это значит, что middleware также могут использоваться как ендпоинты.



Вышеуказанное использование middleware немного неуклюже, потому как здесь жестко заданы uk, paris, хотя middleware поддерживает и другие варианты.

Роутинги можно определять и с использованием упрощенного синтаксиса без использования методов UseRouting и UseEndpoints, прямо на объекте возвращающемся методом WebApplication.CreateBuilder:



**Понимание шаблонов URL.**

В вышеуказанных примерах middleware компоненты будут сравнивать на точное соответствие сегментов URL, а именно их количество и содержание, если точного соответствия не будет, данный middleware не сработает.

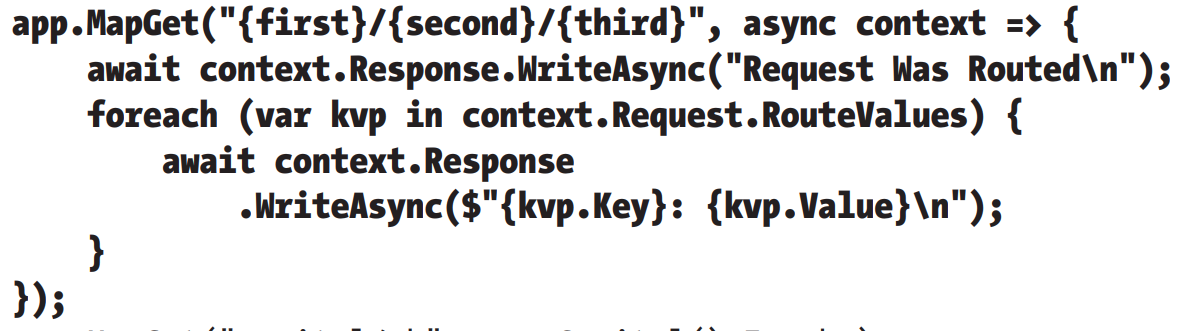
Использование сегментных переменных в URL шаблонах.

URL шаблоны, использованные в вышеуказанных примерах, используют литеральные сегменты, также известные как статические сегменты, которые сравниваются с фиксированными строками.

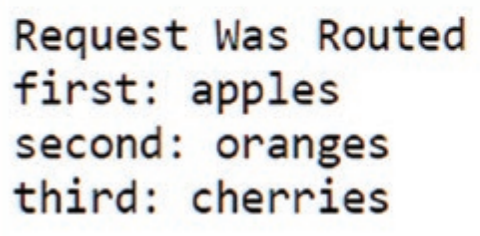
Сегментные переменные, также известные как параметры маршрута, позволяют более гибкий роутинг. Данные переменные имеют определенное имя и обрамляются фигурными скобками {first}/{second}. Данный шаблон будет соответствовать любому URL, который содержит два сегмента, без разницы в их содержимом. Когда сегментные переменные используются, роутинг middleware могут использовать содержимое сегментов URL. Это содержимое доступно через свойство HttpRequest.RouteValues, которое возвращает объект RouteValuesDictionary.

Имеются зарезервированные слова, которые не могут быть использованы в переменных сегментов: action, area, controller, handler, and page.

Пример использования переменных сегментов в ендпоинте:



Результат работы данного ендпоинта при строке запроса <http://Localhost:5000/apples/oranges/cherries>:



Правила выбора роутинга из нескольких роутингов, соответствующих строке запроса (URL).

В процессе запроса все соответствующие роутинги получают определенное место в иерархии и, впоследствии роутинг, имеющий первое место выполняется. Более специфичный роутинг получает запрос. Так шаблон с литеральными сегментами имеет преимущество перед переменными сегментов, а переменные сегментов с ограничениями имеют преимущество перед переменными сегментов без ограничений. Если шаблоны роутингов получили одинаковое место в иерархии, то выдается исключение, говорящее о двусмысленности.

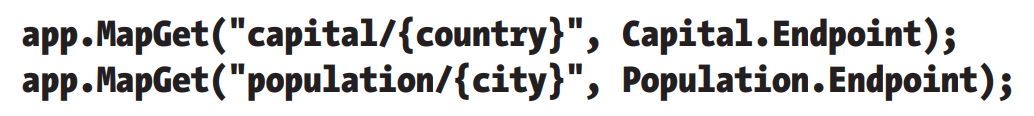
Компонент middleware может использоваться в качестве ендпоинта, обратное не возможно.

Пример определения ендпоинта:



Ендпоинт отличается от middleware тем, что у него нет конструкторов, Invoke метод заменен на статический метод с наименованием Endpoint.

В файле Program.cs ендпоинты используются так:



Подход с ендпоинтами гораздо удобнее, потому как легче увидеть URL-адреса, которые поддерживает каждая middleware.

**Генерация URLs из роутингов.**

Если из одного из ендпоинтов есть ссылка на URL другого ендпоинта, то есть жестко зашитая зависимость по типу, указанной выше

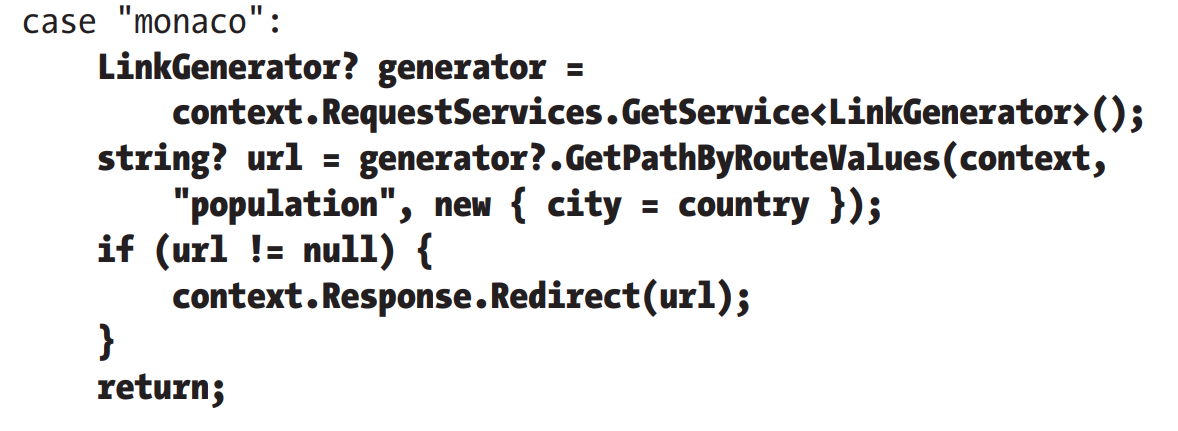
, то при смене URL приходится его менять в нескольких местах, что сопряжено с трудностями сопровождения. Данная проблема решается системой маршрутизации, которая позволяет генерировать URL адреса, предоставляя значения данных для переменных сегмента.

Первый шаг решения этой проблемы, это назначить имя маршрута, который будет целью сгенерированного URL адреса:



Единственными метаданными, необходимыми для создания URL адресов, является имя, которое назначается путем передачи нового объекта RouteNameMetadata, конструктор которого принимает имя, которое будет использоваться для ссылки на маршрут. Данное имя должно совпадать с указанным в параметре метода GetPathByRouteValues в ендпоинте.

URL адреса генерируются с использованием класса LinkGenerator, экземпляр данного класса можно получить с использованием DI:

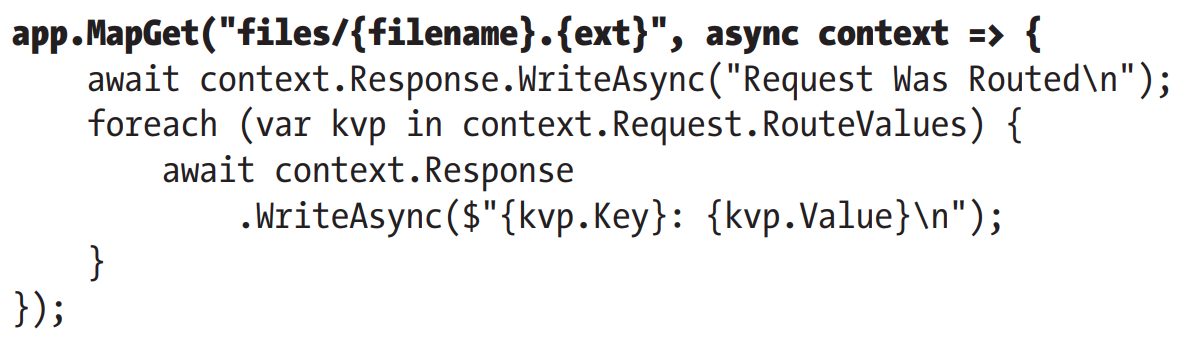


Аргументами метода GetPathByRouteValues являются объект HttpContext, имя статического сегмента роутинга, которое будет использоваться для генерации URL и объект, доставляющий значение переменной сегмента.

Польза данного подхода в том, что URL, генерируемый из URL шаблона в именованном маршруте означает, что изменение в URL шаблоне отражается в генерируемом URL, без необходимости вносить изменения в ендпоинт, так например population в шаблоне URL можно поменять на любое другое слово.

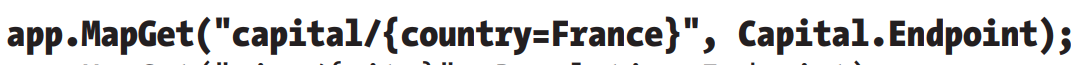
**Управление URL сопоставлением.**

Сопоставление множества значений из одного URL сегмента:

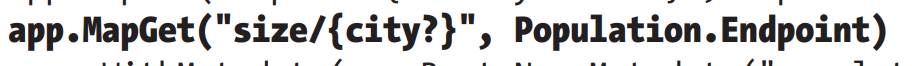


По-хорошему, URL шаблоны нужно делать как можно проще, чтобы получать ожидаемые результаты. В вышеуказанном случае чтение переменных в цикле может происходить справа налево.

Использование дефолтных значений для переменных сегментов. При этом в адресе можно опустить значение country и запрос в котором будет указан только один сегмент capital благополучно сопоставится.



Использование опциональных сегментов. Для этого ставится знак вопроса в шаблоне.

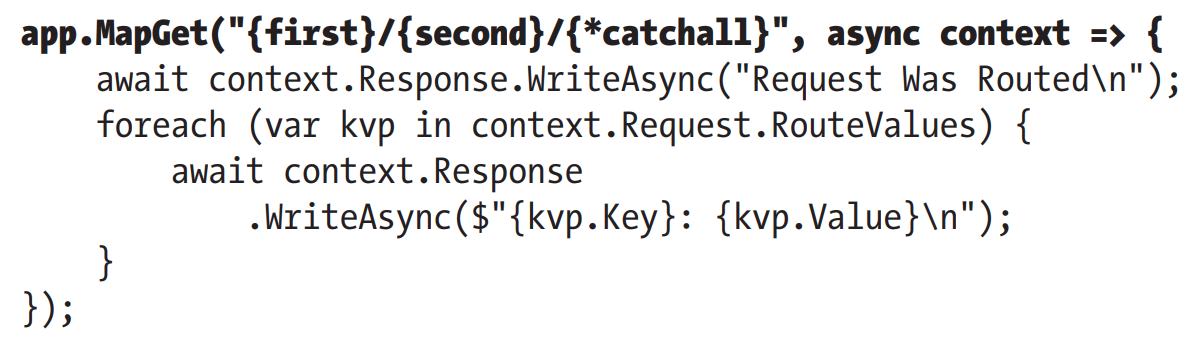


А в ендпоинте его можно обработать так:

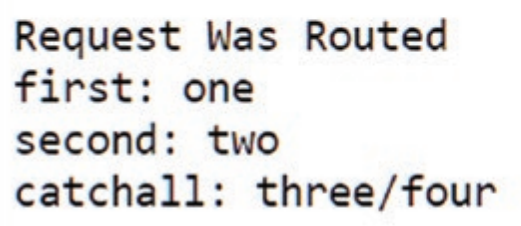


Использование универсальной (catchal) переменной сегмента.

Опциональные сегменты, в отличие от универсальных, позволяют использовать более короткий адрес, чем в шаблоне URL, а универсальная (catchal) переменная сегмента, которая помечается звездочкой перед названием, обычно называемой catchall, позволяет указывать большее число сегментов, чем указано в шаблоне, в примере ниже от двух до бесконечности:

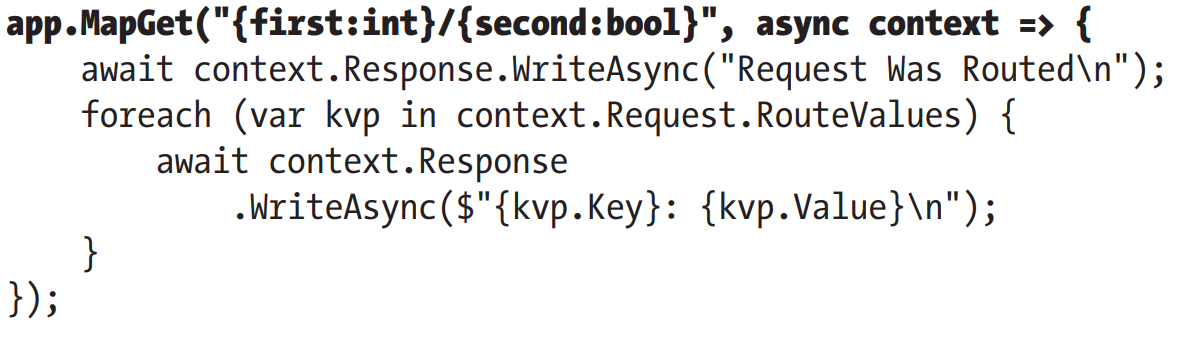


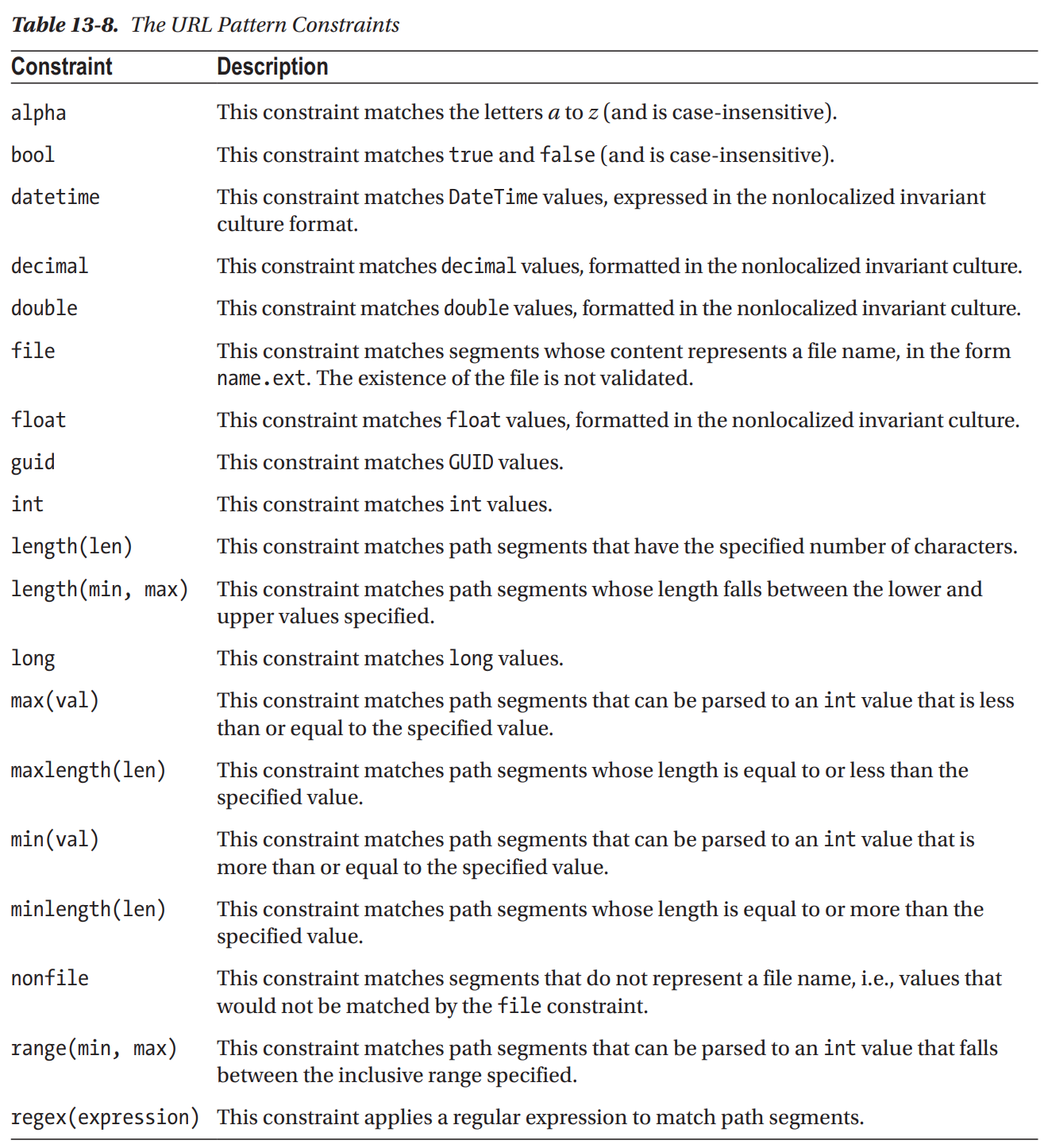
При этом вышеуказанный код при передаче URL вида localhost:5000/one/two/three/four выведет:



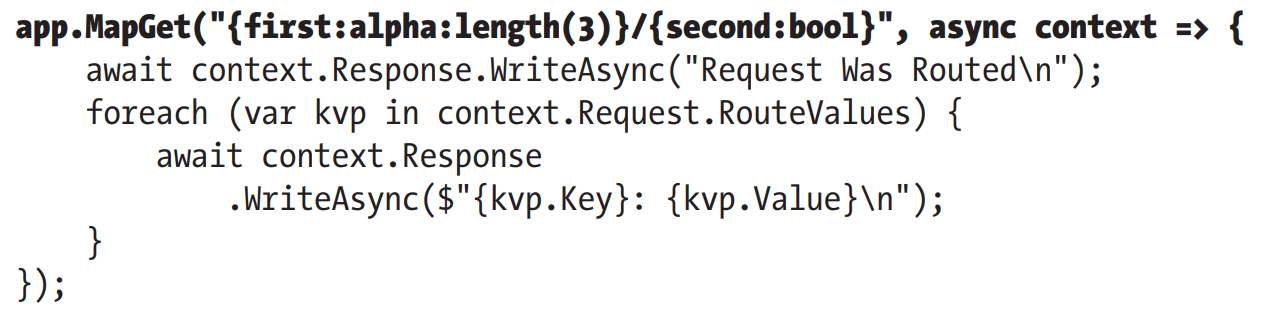
**Ограничения на соответствия сегментов.**

Дефолтные значения, опциональные сегменты и универсальные сегменты увеличивают количество вариантов URL, которые соответствуют шаблонам. Ограничения оказывают обратный эффект и ограничивают варианты URL. Это может быть полезно, если ендпоинт принимает только какое то специфичное содержание сегмента или вы хотите более точно дифференцировать URL между ендпоинтами. Ограничения реализуются путем знака двоеточия (:) после наименования переменной сегмента, после знака двоеточия устанавливается тип значения:



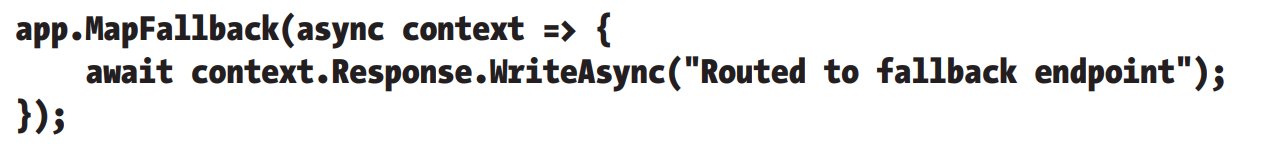


Ограничения могут комбинироваться в одном сегменте:



**Определение FallBack роутинга.**

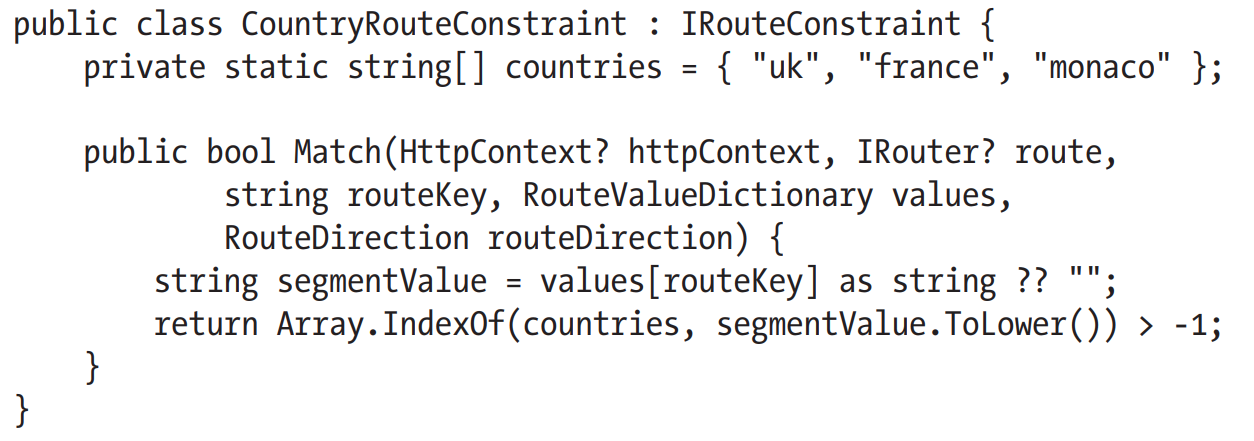
FallBack роутинг отправляет запрос на ендпоинт только когда все другие роуты не соответствуют запросу. FallBack роутинг предотвращает оставление запросов без какого либо ответа и заставляет систему в любом случае генерировать ответ.



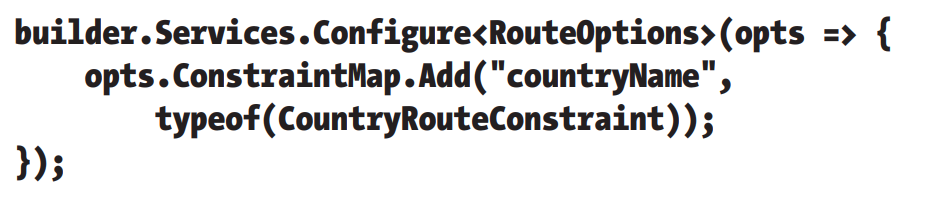
Продвинутый роутинг.

Создание кастомных ограничений.

Если ни одно из ограничений в таблице выше не подходит, можно создать свое кастомное ограничение путем создания класса, реализующего интерфейс IRouteConstraint.



Далее кастомное ограничение устанавливается с использованием option pattern, описанного ранее:



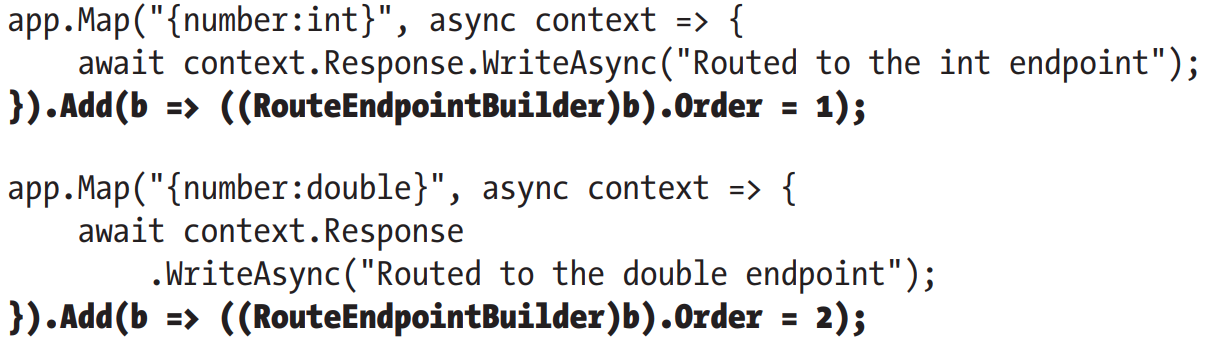
И далее используется в роутинге:



**Предотвращение исключения неоднозначности роутингов.**

Если два роутинга имеют один и тот же рейтинг, система роутинга не может выбрать между ними и выдает исключение неоднозначности. В большинстве случаев проблема решается модификацией неоднозначных роутингов путем повышения их специфичности введением литеральных сегментов или ограничений, но могут быть ситуации, когда это не возможно.

Проблема решается добавлением следующих инструкций:

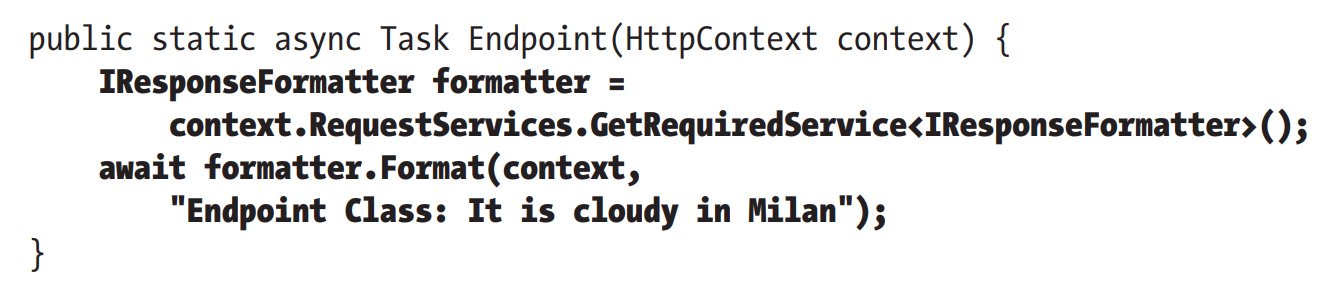


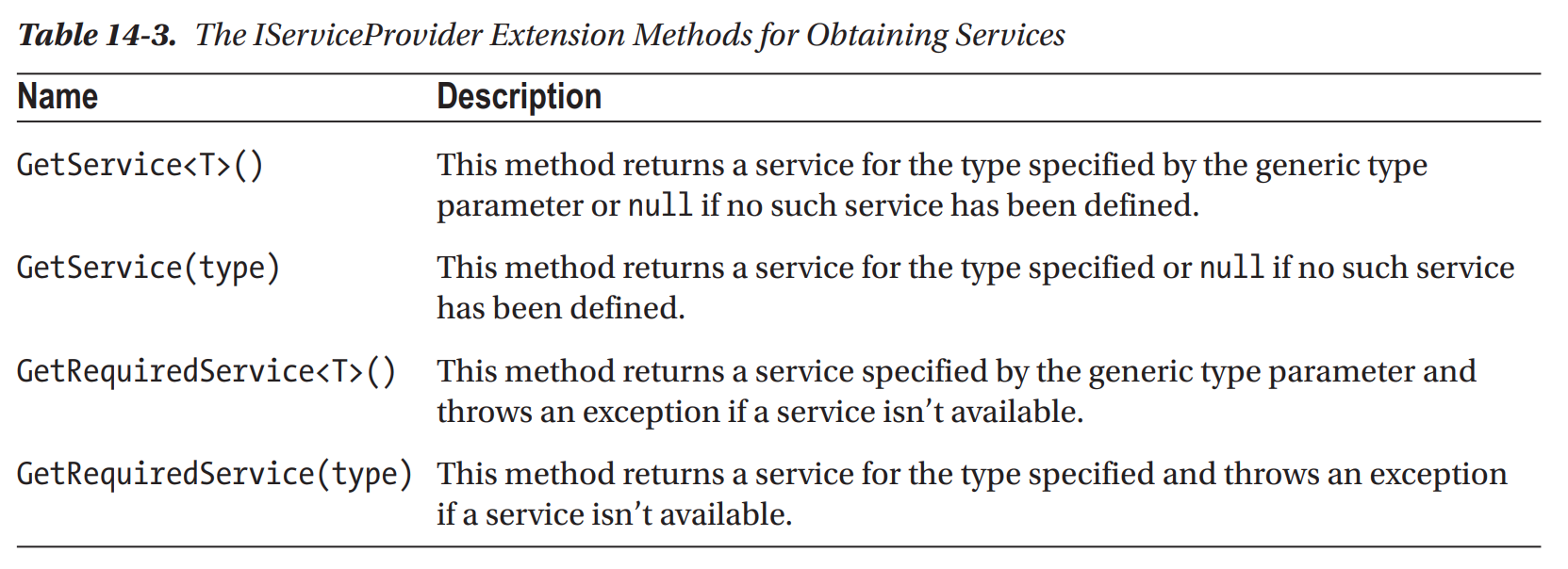
**Dependency injection.**

Зависимости могут внедряться разными способами:

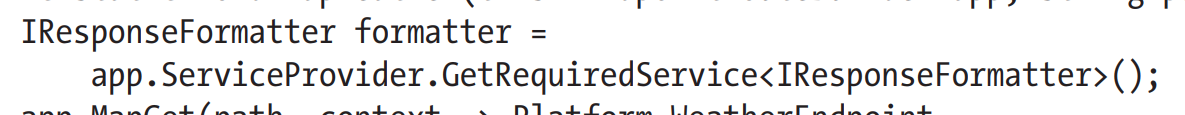
- через конструктор класса,

- с использованием объекта HttpContext (применяется в еднпоинтах, т.к. они не имеют конструктора):





- с использованием интерфейса IEndpointRouterBuilder и его метода ServiceProvider:

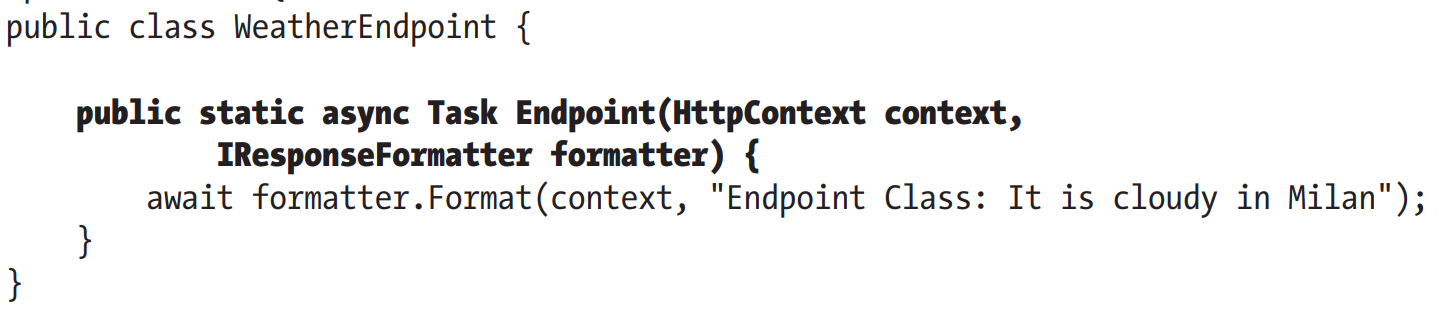


**Использование функции адаптера.**

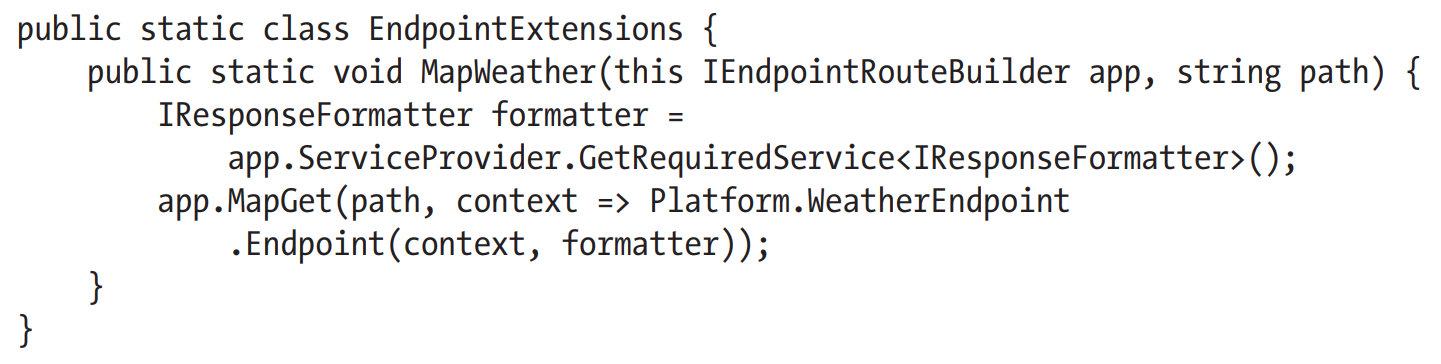
Недостаток в использовании HttpContext.RequestServices метода в том, что сервис должен быть реализован для каждого запроса. Как вы увидите далее имеются некоторые сервисы, для которых это требуется, потому что они реализуют функции, специфичные для единичного запроса или ответа. Это не относится к сервису IResponseFormatter.

Более элегантный подход, это получить услугу при создании маршрута ендпоинта, а не для каждого запроса:

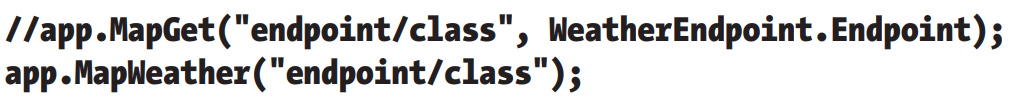
Создаем функцию адаптер:



Создаем метод расширения для интерфейса IEndpointRouterBuilder, который будет использоваться в Program.cs файле:



Используем метод расширения в классе Program.cs:



Описание жизненных циклов зависимостей я здесь не стал описывать.

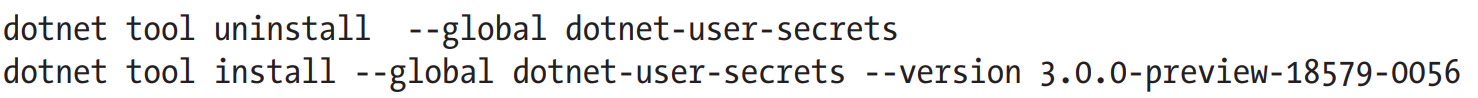
**Глава 15**

**Использование особенностей платформы. Часть 1.**

В этой части будут описаны три наиболее используемые особенности: конфигурация приложения, логгирование и использование статического контента.

**Хранение пользовательских секретов**

Первым делом нужно подготовить файл, который будет использоваться для хранения секретных данных. Для этого нужно открыть командную строку и выполнить следующие команды:

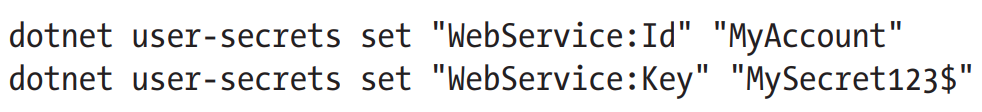


Далее нужно выполнить следующую команду, инициализирующую пользовательские секреты:



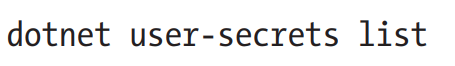
Данная команда добавляет в файл проекта Platform.csproj элемент, содержащий уникальный ID для проекта, который будет ассоциирован с секретами на каждой машине разработчиков.

Далее в папке проекта необходимо выполнить команды:

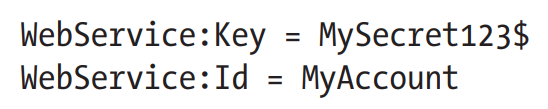


Каждый секрет имеет ключ и значение, а связанные секреты могут группироваться вместе путем использования одинакового префикса, за которым следует двоеточие и далее имя секрета. Вышеуказанные команды создают связанные Id и Key секреты, имеющие префикс WebService.

После каждой вышеуказанной команды вы увидите сообщение, подтверждающее, что секрет был добавлен в хранилище секретов. Чтобы проверить секреты для проекта нужно выполнить команду в файле проекта:



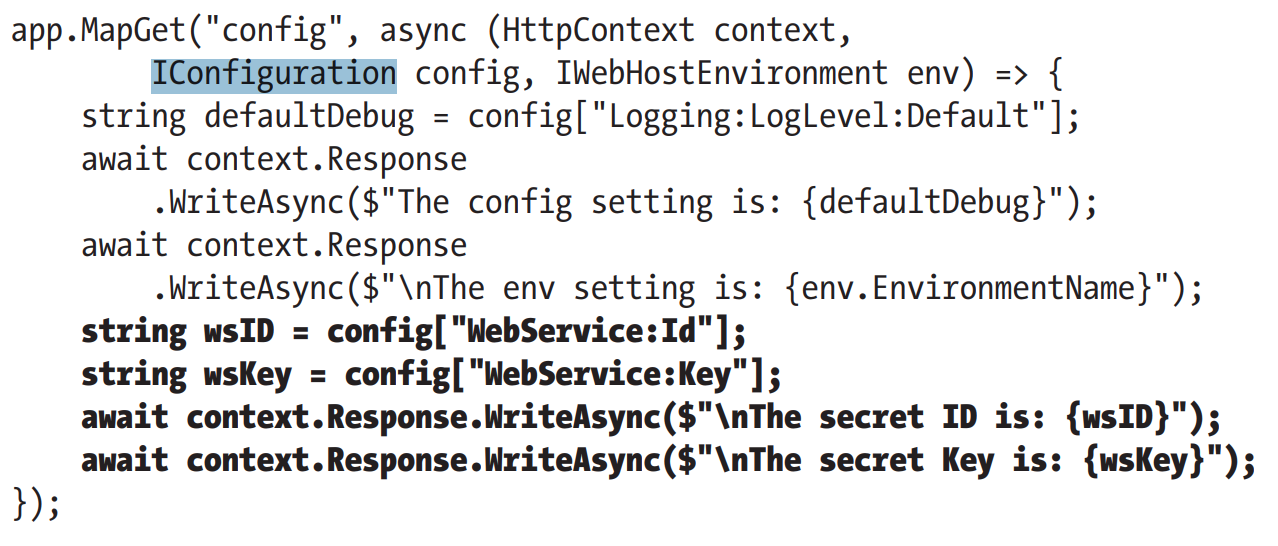
Данная команда произведет следующий вывод:



Под капотом создается JSON файл в папке, имеющей название, соответствующее секрет ID в папке %APPDATA%\Microsoft\UserSecrets на локальном компьютере. Каждый проект имеет свою собственную папку для секретов.

**Чтение пользовательских секретов.**

Получение пользовательских секретов производится с помощью объекта, реализующего интерфейс IConfiguration:



Пользовательские секреты можно получить только в режиме Development.

**Использование встроенного в платформу сервиса логирования.**

ASP.NET Core предоставляет сервис логирования, который может быть использован для записи сообщений, описывающих состояние приложения для отслеживания ошибок, исследования производительности и помощи в диагностировании проблем.

Лог сообщения отправляются лог провайдерам, ответственным за пересылку сообщений туда, где их можно увидеть, сохранить и обработать. Имеются предустановленные провайдеры для базового логирования, а также несколько сторонних провайдеров.

Три предустановленных провайдера доступны по умолчанию: консоль провайдер, дебаг провайдер и EventSource провайдер.

**Генерация сообщений логов.**