Темы урока

Как работает HTTP?

Hypertext Transfer Protocol - HTTP/1.1

Этот протокол описывает взаимодействие между двумя компьютерами (**клиентом** и **сервером**), построенное на базе сообщений, называемых запрос (Request) и ответ (Response). Каждое сообщение состоит из трех частей:

1. Стартовая строка
2. Заголовки
3. Тело

При этом обязательной является только стартовая строка.

Fiddler

- Устанавливаем Fiddler.
- Запускаем (F12: start / stop capturing)
- Смотрим на запросы браузера через него в контексте изучения HTTP-сообщений.

HTTP-методы

Два основных, поддерживаемых HTML-формами, а следовательно, и браузерами:

- **GET** — запрашивает представление ресурса. Запросы с использованием этого метода могут только извлекать данные.
- **POST** — используется для отправки сущностей к определённому ресурсу. Часто вызывает изменение состояния или какие-то побочные эффекты на сервере.

Ещё три часто используемых:

- **PUT** — заменяет (обновляет) все текущие представления ресурса данными запроса.
- **DELETE** — удаляет указанный ресурс.
- **PATCH** — используется для частичного изменения ресурса.

Есть ещё много экзотических используемых реже.

REST: REpresentational State Transfer

REST - это набор принципов построения веб-приложений.

Вообще REST охватывает более широкую область, нежели HTTP — его можно применять и в других сетях с другими протоколами. REST описывает принципы взаимодействия клиента и сервера, основанные на понятиях «ресурса» и «глагола» (можно понимать их как подлежащее и сказуемое). В случае HTTP ресурс определяется своим URI, а глагол — это HTTP-метод.

Проблемы?

Есть небольшая проблема с применением REST на практике. Проблема эта называется HTML. Запросы PUT и DELETE можно отправлять через XMLHttpRequest, или через небраузерное обращение к серверу (скажем, через curl или даже через telnet).

Однако, нельзя сделать HTML-форму, отправляющую полноценный PUT- или DELETE-запрос. Дело в том, спецификация HTML не позволяет создавать формы, отправляющие данные иначе, чем через GET или POST.

Postman

Устанавливаем Postman.
Запускаем.
Пробуем

ASP.NET Core MVC

Живая демонстрация

Создаём новое приложение ASP.NET Core Web Application

По идее, чтобы делать API можно выбрать заранее подготовленный для этого шаблон “API”, однако мы выберем шаблон “Empty” (пустой) для того, чтобы “с нуля” (ну, практически:) создать первый API.

Да, даже “пустой” проект ASP.NET Core Web содержит некоторый код.

Program.cs

Во-первых, у нас есть файл Program.cs, а в нём есть метод Main. Это вам ничего не напоминает? Всё верно, очень похоже на консольное приложение. В принципе, это потому, что приложения .NET Core ASP Web App и построено на базе обычного консольного приложения).

Можно указать, что в нашем примере все параметры настраиваются “по-умолчанию” через метод WebHost.CreateDefaultBuilder и бегло показать что внутри него происходит: https://github.com/aspnet/AspNetCore/blob/master/src/DefaultBuilder/src/WebHost.cs.

Startup.cs

Это точка входа для нашего веб-приложения.
Содержит метод **ConfigureServices**, который необходим для добавления сервисов в контейнер. Мы чуть-чуть позже раберём что это за контейнер и как и зачем в него добавляются сервисы. Пока просто отметим это.
Далее вызывается метод **Configure**, который описывает как наше приложение будет отвечать на HTTP-запросы. Сейчас наше приложение написано таким образом, что на любой запрос оно будет отвечать строкой “Hello World!”.

Первый запуск с точками прерывания

Расставляем точки прерывания в методах

- Program.Main
- Startup.ConfigureServices
- Startup.Configure

Запускаем и показываем последовательность вызовов (в порядке списка) и мы видим строку Hello World! на экране браузера.

Ещё раз - почему это работает так? Потому что так мы сконфигурировали наш веб-сервис. Думаю, сейчас самое время рассмотреть, как это работает?

Request Pipeline and Middleware

Когда приложению приходит HTTP-запрос, что-то должно перехватить и обработать его чтобы в итоге вернуть HTTP-ответ.

Части кода, которые обрабатывают HTTP-запросы и возвращают HTTP-ответы формируют **Request Pipeline** — конвейер запросов. Мы можем добавлять в этот конвейер **Middleware** — промежуточный или связующий код.

Далее я бы рекомендовал оперировать терминами Request Pipeline и Middleware, потому что вменяемых переводов на русский язык я не нашёл, в то же время, они являются весьма устоявшимися.

Примером таких middleware может быть система аутентификации или авторизации, система диагностики, система логирования. И сам MVC также является точно таким же middleware, который также может быть добавлен в request pipeline.

Теперь посмотрим на request pipeline:

- Request pipline ASP.NET Core состоит из последовательности делегатов, выполняющихся от одного middleware к другому.
- Каждый из них имеет возможность выполнить операции перед и после следующего делегата и дополнить или изменить Response.

Последовательность middleware важна!

Важно понимать, что каждый компонент middleware решает, передать ли Request дальше по цепочке middleware или нет. Таким образом, порядок, в котором мы добавляем middleware имеет важное значение.

Хорошим примером здесь будет компонент middleware, отвечающий за авторизацию. Если пользователь не авторизован для доступа к запрашиваемому ресурсу, middleware сам ответит кодом ошибки доступа 403 Forbidden и не будет передавать запрос дальше по цепочке.

В данном примере авторизационный middleware должен стоять раньше по цепочке конвейера, чтобы отфильтровывать неавторизованные запросы.

Живая демонстрация

Дальше работаем в **L21_C01_empty_asp_net_core_app**.
Приводим его постепенно к **L21_C02_empty_asp_net_core_app_final** :)

Middleware

Давайте посмотрим, Как мы можем добавить middleware в request pipeline.

В этом примере мы сконфигурируем ASP.NET Core Request Pipeline.

Мы добавим вывод диагностики в удобном для разработчика виде для случая, если в коде сгенерировалось исключение. И мы хотим, чтобы этот вывод происходил только в окружении разработчика (Development Environment).

Когда я сказал: “Мы добавим…”, на самом деле, я немного преувеличил, так как разработчики Visual Studio уже добавили необходимый код в пустой шаблон ASP.NET Core Web Application. Так что, давайте посмотрим, как он выглядит (файл Startup.cs метод Configure).

Строчка

```
app.UseDeveloperExceptionPage();
```

добавляет middleware обработки исключений для вывода информативной отладочной информации разработчику. Этот middleware - часть сборки `Microsoft.AspNetCore.Diagnostics` (можно показать по F12). Зачастую middleware лежат в разных сборках, отвечающих за определённую часть функционала. Так достигается модульность ASP.NET Core.

- Заменяем код согласно слайду (генерируем исключение)
- Нажимаем F5
- Ловим исключение в коде.
- Переходим в браузер и видим понятное разработчику сообщение с расширенной информацией об исключении.

На самом деле, если вы будете внимательны, вы заметите, что ловите исключение дважды, так как браузер кроме самой страницы всегда запрашивает иконку favicon.ico. Нажмите F12 в браузере, перейдите на вкладку Console, вы увидите дважды красную строку:

```
Failed to load resource: the server responded with a status of 500
```

Environments

Теперь давайте посмотрим на условие

```
if (env.IsDevelopment())
```

Это условие обеспечивает нам показ расширенной информации об исключении **только** для окружения разработчика (Development Environment). И, думаю, это удобное место, чтобы поговорить о различных окружениях. Как видно, мы можем запускать различный код в зависимости от окружения, в котором работает наше приложение.

Это как раз отличный пример, так как отладочная информация

- Очень полезна разработчику и её хотелось бы видеть в окружении разработчика (development environment)
- Недопустима для вывода в производственном окружении (production environment), так как она даёт потенциальному злоумышленнику увидеть детали реализации вашего приложения.

Давайте посмотрим на вкладку Debug свойств проекта.

В пункте Environment Variables установлена переменная окружения `ASPNETCORE_ENVIRONMENT`, которой выставлено значение `Development`.

ASP.NET Core использует эту переменную окружения для того, чтобы определить, в какой среде (или в каком окружении) он запущен. По соглашению используется 3 возможные среды выполнения:

- `Development`
- `Staging`
- `Production`

Переменные окружения задаются в ОС напрямую, они внешние по отношению к приложению.

В ОС Windows для работы с переменными окружения используется команды set и setx:

- Посмотреть:
  `set ASPNETCORE_ENVIRONMENT`
  или
  `echo %ASPNETCORE_ENVIRONMENT%`
- Задать:
  `set ASPNETCORE_ENVIRONMENT=Staging`

Вернёмся к коду

Чтобы программно получить доступ к значению текущего окружения мы используем сервис `IHostingEnvironment`., который предоставляет основу для работу с окружениями

Самостоятельная работа (Environments)

Нажмите кнопку `Windows` ❖, наберите `cmd` и нажимте `Enter`, чтобы открыть консоль.

В консоли перейдите в папку `bin\Debug\netcoreapp2.2` относительно папки вашего проекта. Посмотрите, чему равна переменная окружения `ASPNETCORE_ENVIRONMENT`:

```
set ASPNETCORE_ENVIRONMENT
```

Скорее всего вы увидите `Environment variable ASPNETCORE_ENVIRONMENT not defined`. Установите эту переменную окружения в значение `Development`:

```
set ASPNETCORE_ENVIRONMENT=Development
```

Запустите ваше приложение:

```
dotnet имя_проекта.dll
```

Откройте в браузере URL http://localhost:5000, и убедитесь, что вы видите информацию об исключении.

Теперь остановите приложение `[Ctrl+C]`, поменяйте переменную окружения на `Production`, снова запустите приложение, в браузере (F5) убедитесь в отсутствии информации об исключении.

Домашнее задание

- Домашнее задание к уроку №21 не предполагается.