ASP.NET Core - True Ultimate Guide

Section 3: HTTP - Notes

HTTP Protocol

Overview:

- HTTP (Hypertext Transfer Protocol) is a protocol used for transmitting hypertext (e.g., HTML) over the internet.
- It operates on a client-server model, where the client (usually a web browser) makes requests to a server, which then responds with the requested resources or error messages.
- **Stateless Protocol**: Each HTTP request is independent of others; the server does not retain information from previous requests.

Request/Response Model:

- **Client Request**: The client sends an HTTP request to the server.
- **Server Response**: The server processes the request and sends back an HTTP response.

HTTP Server

Definition:

- An HTTP server is software that handles HTTP requests from clients and serves back responses. It processes incoming requests, executes the necessary logic (e.g., accessing a database, generating HTML), and returns the appropriate response.

Examples:

- Apache HTTP Server, Nginx, Microsoft IIS, Kestrel (used with ASP.NET Core).

Kestrel:

- Kestrel is a cross-platform web server included with ASP.NET Core.
- It is lightweight, high-performance, and suitable for running both internal and public-facing web applications.

Request and Response Flow with Kestrel

1. **Client Sends Request:**
    - The client (e.g., web browser) sends an HTTP request to the server.
2. **Kestrel Receives Request:**

    - Kestrel receives the request and passes it through the ASP.NET Core middleware pipeline.

3. **Request Processing:**
    - Middleware components process the request and eventually pass it to the application's request handling logic.
4. **Generate Response:**
    - The application generates an HTTP response and sends it back through the middleware pipeline.
5. **Kestrel Sends Response:**
    - Kestrel sends the HTTP response back to the client.

How Browsers Use HTTP

- Browsers use HTTP to request resources such as HTML documents, images, CSS files, and JavaScript files from servers.
- When a user enters a URL or clicks a link, the browser sends an HTTP request to the server, which then responds with the requested resource.

Observing HTTP Requests and Responses in Chrome Dev Tools

1. **Open Chrome Dev Tools:**
    - Press F12 or Ctrl+Shift+I (or Cmd+Option+I on Mac) to open Chrome Dev Tools.
2. **Navigate to the Network Tab:**
    - Click on the Network tab to view HTTP requests and responses.
3. **Inspect a Request:**
    - Click on any request in the list to see detailed information:
        - **Headers**: View request and response headers.
        - **Preview/Response**: View the response body.
        - **Timing**: See the timing details of the request.

HTTP Response Message Format

Response Message Format:

- **Start Line**: Contains the HTTP version, status code, and status message.
- **Headers**: Key-value pairs providing information about the response.
- **Body**: Optional, contains the actual data (e.g., HTML, JSON).

Example:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

Commonly Used Response Headers:

- Content-Type: Specifies the media type of the resource.
- Content-Length: The size of the response body in bytes.
- Server: Provides information about the server handling the request.
- Set-Cookie: Sets cookies to be stored by the client.
- Cache-Control: Directives for caching mechanisms in both requests and responses.

Default Response Headers in Kestrel

- Content-Type: Typically defaults to text/html or application/json depending on the content being served.
- Server: Indicates the server software (e.g., Kestrel).
- Date: The date and time when the response was generated.

HTTP Status Codes

Overview:

- Status codes are issued by the server in response to the client's request to indicate the result of the request.
- Categories include:
  - **1xx Informational**: Request received, continuing process.
  - **2xx Success**: The request was successfully received, understood, and accepted.
  - **3xx Redirection**: Further action needs to be taken in order to complete the request.
  - **4xx Client Error**: The request contains bad syntax or cannot be fulfilled.
  - **5xx Server Error**: The server failed to fulfill an apparently valid request.

Common Status Codes:

- 200 OK: The request succeeded.
- 201 Created: The request succeeded and a new resource was created.
- 204 No Content: The server successfully processed the request, but is not returning any content.
- 400 Bad Request: The server could not understand the request due to invalid syntax.
- 401 Unauthorized: Authentication is required.
- 403 Forbidden: The client does not have access rights to the content.
- 404 Not Found: The server cannot find the requested resource.
- 500 Internal Server Error: The server encountered an unexpected condition.
- 502 Bad Gateway: The server was acting as a gateway or proxy and received an invalid response from the upstream server.
- 503 Service Unavailable: The server is not ready to handle the request.

Setting Status Codes and Response Headers in ASP.NET Core

Example Code 1:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();


app.Run(async (HttpContext context) =>
{
   context.Response.Headers["MyKey"] = "my value";
```

```
    context.Response.Headers["Server"] = "My server";
    context.Response.Headers["Content-Type"] = "text/html";
    await context.Response.WriteAsync("<h1>Hello</h1>");
    await context.Response.WriteAsync("<h2>World</h2>");
});

app.Run();
```

Explanation:

- context.Response.Headers["MyKey"] = "my value";: Adds a custom header to the response.
- context.Response.Headers["Server"] = "My server";: Modifies the Server header.
- context.Response.Headers["Content-Type"] = "text/html";: Sets the Content-Type header to text/html.
- await context.Response.WriteAsync("<h1>Hello</h1>");: Writes the first part of the response body.
- await context.Response.WriteAsync("<h2>World</h2>");: Writes the second part of the response body.

Example Code 2:

```
csharpCopy codevar builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Run(async (HttpContext context) =>
{
    if (1 == 1)
    {
        context.Response.StatusCode = 200;
    }
    else
    {
        context.Response.StatusCode = 400;
    }
    await context.Response.WriteAsync("Hello");

    await context.Response.WriteAsync(" World");

});


app.Run();
```

Explanation:

- context.Response.StatusCode = 200;: Sets the status code to 200 OK.
- context.Response.StatusCode = 400;: Sets the status code to 400 Bad Request (this line won't be executed due to the condition).
- await context.Response.WriteAsync("Hello");: Writes the first part of the response body.
- await context.Response.WriteAsync(" World");: Writes the second part of the response body.

Summary

- **HTTP Protocol**: A fundamental protocol for web communication, following a request/response model and operating statelessly.
- **HTTP Server**: Software that processes HTTP requests and responses, such as Kestrel.
- **Request/Response Flow**: From client request to server response, involving middleware processing in Kestrel.
- **Browser Usage**: Browsers request resources via HTTP, which are then processed and rendered.
- **Dev Tools**: Chrome Dev Tools can inspect HTTP traffic in detail.
- **Message Format**: HTTP requests and responses consist of a start line, headers, and an optional body.
- **Headers**: Key-value pairs providing additional information about requests and responses.
- **Status Codes**: Indicate the result of HTTP requests, categorized into informational, success, redirection, client error, and server error codes.
- **Setting Status Codes and Headers**: ASP.NET Core allows customization of responses using code, enabling setting of status codes and headers as demonstrated.

HTTP Requests

In the world of web applications, an HTTP request is a client's way of saying, "Hey server, I need something." This "something" could be a web page, an image, data from a database, or the result of some server-side calculation. The client, typically a web browser, sends this request to the server, which processes it and returns a response.

Anatomy of an HTTP Request

An HTTP request consists of several parts:

1. **Start Line:** This is the first line of the request, and it contains three crucial pieces of information:
    - **Method:** This indicates the action the client wants the server to perform. Common methods include:
        - GET: Retrieve data from the server.
        - POST: Submit data to the server (e.g., form data).
        - PUT: Update an existing resource on the server.
        - DELETE: Remove a resource from the server.
    - **Request URI (Uniform Resource Identifier):** This is the path to the resource on the server that the client is requesting.
    - **HTTP Version:** This specifies the version of the HTTP protocol being used (e.g., HTTP/1.1 or HTTP/2).

2. **Headers:** These provide additional information about the request, such as:
    - User-Agent: The client's browser or application.
    - Accept: The types of content the client can understand (e.g., HTML, JSON).
    - Host: The domain name of the server.
    - Content-Type: The type of data being sent in the request body (if any).
    - Authorization: Credentials for authentication (if required).

3. **Empty Line:** This separates the headers from the body of the request.

4. **Body (Optional):** This part of the request contains data that the client is sending to the server. For example, a POST request might include form data or JSON data.

Query Strings: Passing Parameters in URLs

A query string is a way to pass parameters to a server within the URL itself. It starts with a question mark (?) and follows the path in the URL. Each parameter is a key-value pair, separated by an equals sign (=), and multiple parameters are separated by ampersands (&).

Example:

https://example.com/products?category=electronics&brand=apple

In this example, category=electronics and brand=apple are parameters being passed to the server.

The Request Object in ASP.NET Core

ASP.NET Core provides a HttpRequest object that gives you access to all the information within an incoming request. This object has properties like:

- Method: The HTTP method (GET, POST, etc.).
- Path: The URI path requested by the client.
- Query: A collection of query string parameters.
- Headers: A collection of request headers.
- Body: A stream representing the request body (if present).

Code 1: Displaying Request Path and Method

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.Run(async (HttpContext context) =>
{
   string path = context.Request.Path;
   string method = context.Request.Method;

   context.Response.Headers["Content-type"] = "text/html";
   await context.Response.WriteAsync($"<p>{path}</p>");
   await context.Response.WriteAsync($"<p>{method}</p>");
});
```

```
app.Run();
```

This code defines a simple middleware component (using app.Run) that:

1. Extracts the Path and Method from the Request object.
2. Sets the Content-type response header to text/html.
3. Writes the extracted path and method into the response body as HTML paragraphs.

Code 2: Handling GET Requests with Query Parameters

```
app.Run(async (HttpContext context) =>
{
  context.Response.Headers["Content-type"] = "text/html";
  if (context.Request.Method == "GET")
  {
    if (context.Request.Query.ContainsKey("id"))
    {
      string id = context.Request.Query["id"];
      await context.Response.WriteAsync($"<p>{id}</p>");
    }
  }
});
```

This code focuses on GET requests:

1. It sets the Content-type response header.
2. It checks if the request method is GET.
3. If so, it checks if a query parameter named "id" exists.
4. If found, it extracts the value of the "id" parameter and displays it.

Code 3: Extracting the User-Agent Header

```
app.Run(async (HttpContext context) =>
{
  context.Response.Headers["Content-type"] = "text/html";
  if (context.Request.Headers.ContainsKey("User-Agent"))
  {
    string userAgent = context.Request.Headers["User-Agent"];
    await context.Response.WriteAsync($"<p>{userAgent}</p>");
  }
});
```

This code:

1. Sets the Content-type response header.
2. Checks if the User-Agent header is present in the request.
3. If found, it extracts the value of the User-Agent header and displays it, indicating the client's browser or application.

Summary about HTTP Request:

HTTP requests are the messages sent from clients (like web browsers) to servers to request resources or actions. They consist of a start line (method, URI, HTTP version), headers (additional information), an empty line, and an optional body containing data. Query strings are used to pass parameters within URLs.

ASP.NET Core provides the HttpRequest object to access request details. The example codes demonstrated:

1. Displaying the requested path and HTTP method.
2. Handling GET requests and extracting query parameter values.
3. Retrieving and displaying the User-Agent header from a request.

HTTP Methods

GET: Retrieving Data

The GET method is primarily designed for fetching data from a server. Think of it as asking the server for a specific resource, like a webpage, an image, or some data from a database. Here's what characterizes GET requests:

1. **Data in the URL:** Parameters are appended to the URL as a query string. This makes the request parameters visible in the browser's address bar.
2. **Limited Data Size:** The size of data that can be sent in a GET request is restricted due to limitations in URL lengths (browsers and servers might have different limits).
3. **Idempotent:** GET requests are considered idempotent. This means you can make the same GET request multiple times, and it should have the same effect as making it once (assuming the underlying data hasn't changed).
4. **Caching:** GET requests can be cached, meaning that if a client requests the same resource again, the browser might serve the previously retrieved response from its cache, improving performance.
5. **Security:** GET requests are generally less secure than POST requests because the data is visible in the URL. Avoid using GET for sensitive information like passwords or credit card numbers.

Example GET Request:

GET /products?category=electronics&brand=apple HTTP/1.1

Host: example.com

POST: Submitting Data

The POST method is primarily used for submitting data to the server for processing. This data is typically included in the body of the request and is not visible in the URL. Here's how POST requests differ from GET:

1. **Data in the Body:** Data is sent in the request body, making it more suitable for sending large amounts of data or sensitive information.
2. **Not Idempotent:** POST requests are not idempotent. Repeated POST requests might result in different outcomes (e.g., creating multiple resources or triggering actions multiple times).
3. **Not Cachable:** POST requests are generally not cached, as they often result in changes on the server.
4. **Security:** POST requests are considered more secure than GET requests because the data is not exposed in the URL. However, they are still susceptible to attacks like cross-site request forgery (CSRF), which requires additional security measures.

Example POST Request:

POST /login HTTP/1.1

Host: example.com

```
Content-Type: application/x-www-form-urlencoded


username=john&password=secret
```

Choosing Between GET and POST

- **Use GET when:**
  - You are retrieving data from the server.
  - You want the request to be bookmarkable.
  - The data being sent is small and non-sensitive.
- **Use POST when:**
  - You are submitting data to the server for processing.
  - The request might cause changes on the server.
  - You are sending sensitive data or large amounts of data.

Postman

Postman is a versatile API development and testing tool. It allows you to easily craft HTTP requests, send them to your ASP.NET Core application (or any API), and inspect the responses. It's a fantastic way to debug, experiment, and explore your API endpoints.

Installation

1. **Download:** Head to the official Postman website (https://www.postman.com/downloads/) and download the version suitable for your operating system (Windows, macOS, Linux).
2. **Install:** Follow the on-screen instructions to install Postman. The process is usually straightforward.

Usage: Making Requests to Your ASP.NET Core App

Let's say your ASP.NET Core application is running locally at https://localhost:7070 and has an endpoint /api/products. Here's how to use Postman:

1. **Launch Postman:** Open the Postman application.

2. **Create a New Request:**
   - Click on the "New" button in the top left corner.

   - Choose "Request" from the options.

3. **Set the Request Method and URL:**
   - In the request builder, select the appropriate HTTP method (GET, POST, PUT, DELETE, etc.) from the dropdown.
   - Enter the full URL of your ASP.NET Core endpoint (e.g., https://localhost:7070/api/products) in the address bar.

4. **(Optional) Add Headers:**
   - If your endpoint requires specific headers (like Content-Type), click on the "Headers" tab and add them as key-value pairs.

5. **(Optional) Add Request Body:**
   - If you are sending data with the request (e.g., JSON data for a POST request), click on the "Body" tab.
   - Choose the format (e.g., "raw" for JSON) and enter your data.

6. **Send the Request:**
   - Click the "Send" button.

7. **Inspect the Response:**
   - The response from your ASP.NET Core application will appear in the lower part of Postman. You'll see:
     - The status code (200 OK, 404 Not Found, etc.)
     - Response headers
     - The response body (if any)

Summary

HTTP (Hypertext Transfer Protocol):

- **Foundation of the Web:** HTTP is the protocol that powers the World Wide Web. It defines how clients (browsers, apps) and servers communicate.

- **Request-Response Cycle:** Communication follows a request-response model. The client sends a request, and the server sends back a response.
- **Stateless:** HTTP is stateless, meaning each request is independent. Servers don't inherently remember past interactions.
- **Methods:** HTTP methods define actions (GET, POST, PUT, DELETE, etc.).
- **Versions:** HTTP/1.1 and HTTP/2 are the most commonly used versions.

HTTP Requests:

- **Purpose:** Initiate communication, asking for a resource or action from the server.
- **Structure:** Start line (method, URI, version), headers, empty line, optional body.
- **Methods:**
    - GET: Fetch data, idempotent, cachable.
    - POST: Submit data, not idempotent, not typically cached.
    - PUT, DELETE: Update and delete resources, respectively.
- **Headers:** Provide metadata like content type, user agent, authentication.
- **Body:** Used to send data with POST, PUT, etc.

HTTP Responses:

- **Purpose:** Server's reply to a request.
- **Structure:** Start line (version, status code, reason phrase), headers, empty line, optional body.
- **Status Codes:** Three-digit codes indicate the outcome (200 OK, 404 Not Found, 500 Internal Server Error).
- **Headers:** Provide metadata about the response (content type, length, caching).
- **Body:** Contains the requested data (HTML, JSON, etc.) or error messages.