

<epam>

Module "Web"

Submodule "Web API"

Part 1

AGENDA

- 1** REST, RESTful. RPC vs SOAP vs HTTP
- 2** HTTP request-response. HTTP processing pipeline
- 3** ASP.NET Web API overview
- 4** Controllers. Request handling
- 5** Fiddler and Postman
- 6** Action methods. Action Results in Web API

REST, RESTful. RPC vs SOAP vs HTTP

SOA & Web Services

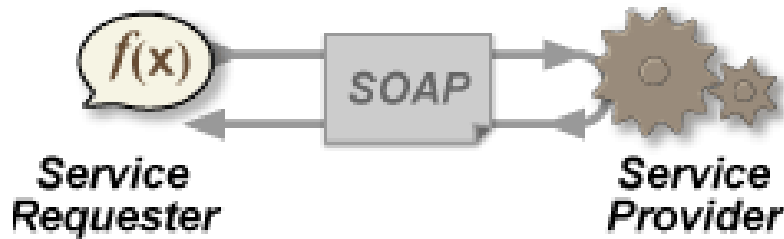


Service-Oriented Architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network.

In service oriented architecture, a number of services communicate with each other, in one of two ways: through passing data or through two or more services coordinating an activity.

SOAP vs HTTP

SOAP (Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. Its purpose is to provide extensibility, neutrality, verbosity and independence.[vague] It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP), although some legacy systems communicate over Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.



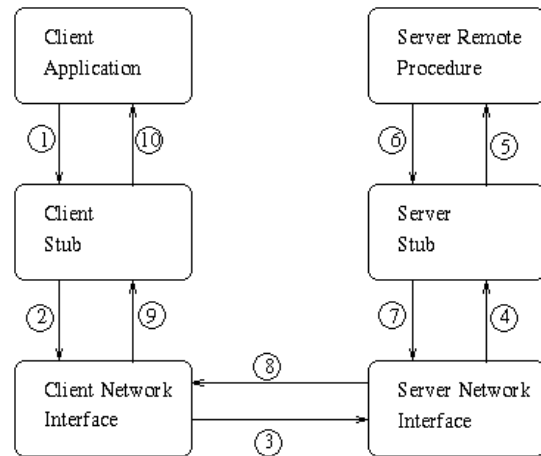
SOAP allows developers to invoke processes running on disparate operating systems (such as Windows, macOS, and Linux) to authenticate, authorize, and communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.

Remote procedure call (RPC)

In distributed computing, a **remote procedure call (RPC)** is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.

In the object-oriented programming paradigm, RPCs are represented by **remote method invocation (RMI)**.

The RPC model implies a level of location transparency, namely that calling procedures are largely the same whether they are local or remote, but usually they are not identical, so local calls can be distinguished from remote calls. Remote calls are usually orders of magnitude slower and less reliable than local calls, so distinguishing them is important.



RPC vs HTTP

Message passing

RPC is a request–response protocol. An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XMLHttpRequest.

There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

An important difference between remote procedure calls and local calls is that remote calls can fail because of unpredictable network problems.

Idempotent procedures - those that have no additional effects if called more than once

REST, RESTful

➤ XML-RPC (XML Remote Procedure Call)

Host: betty.userland.com

Content-Type: text/xml

Content-length: 181

<?xml version="1.0"?>

<methodCall>

<methodName>examples.getStateName</methodName>

<params>

<param>

<value><i4>41</i4></value>

</param>

</params>

</methodCall>

HTTP/1.1 200 OK

Connection: close

Content-Length: 158

Content-Type: text/xml

Date: Fri, 17 Jul 2020 19:55:08 GMT

Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>

<methodResponse>

<params>

<param>

<value><string>South Dakota</string></value>

</param>

</params>

</methodResponse>

➤ JSON-RPC

--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}

-- {"jsonrpc": "2.0", "result": 19, "id": 1}

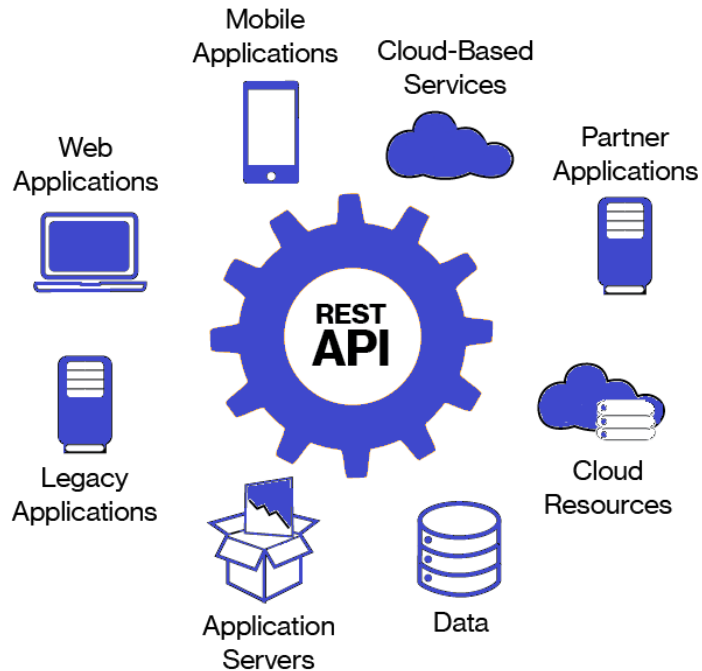
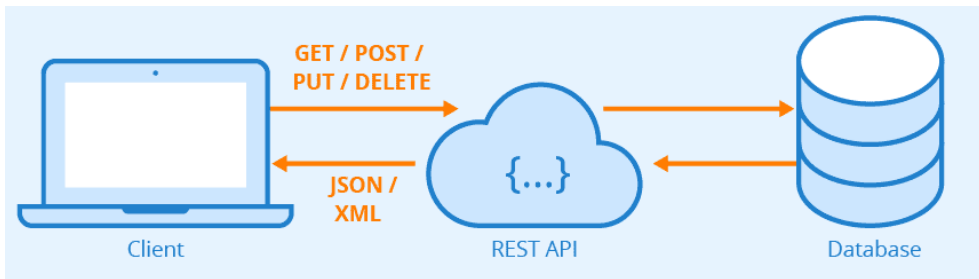
--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}

-- {"jsonrpc": "2.0", "result": 19, "id": 3}

REST, RESTful

REST, or **RE**presentational **S**tate **T**ransfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other.

REST-compliant systems, often called **RESTful** systems, are characterized by how they are stateless and separate the concerns of client and server.



Roy Fielding introduced the REST architectural pattern in a dissertation he wrote in 2000.

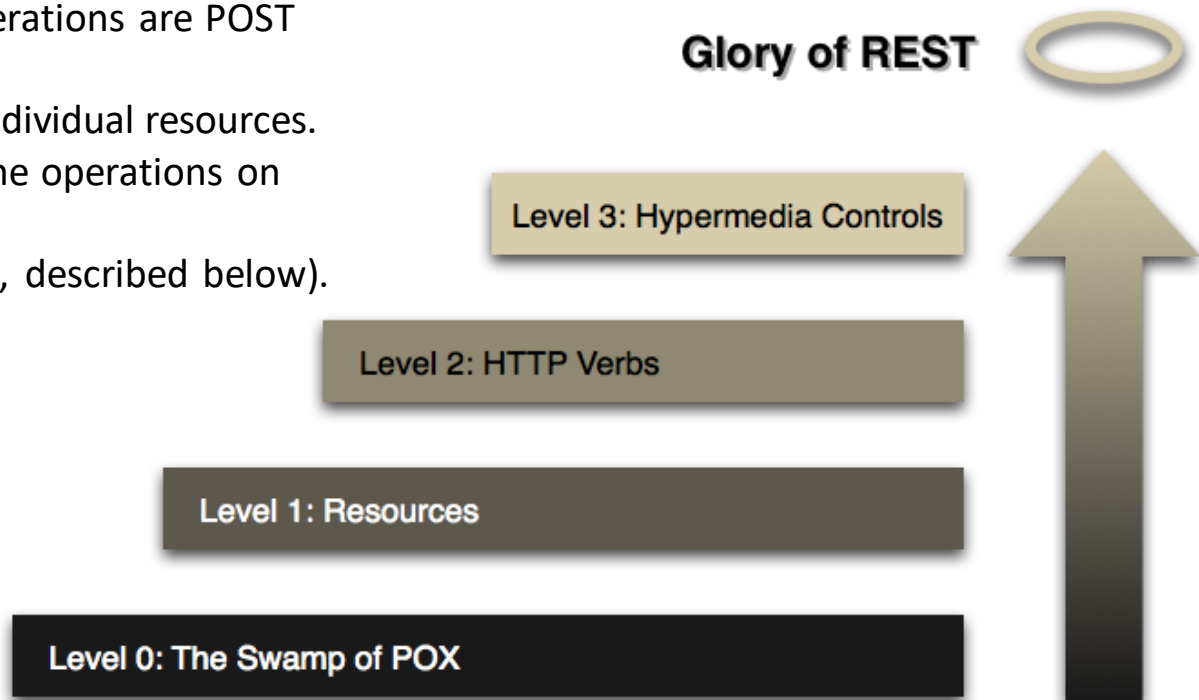
REST, RESTful

ARCHITECTURAL CONSTRAINTS:

- Client-server architecture
- Statelessness
- Cacheability
- Uniform Interface
 - Identification of resources
 - Manipulation of resources through representations
 - Self-descriptive messages
 - HATEOAS (hypermedia as the engine of application state)
- Layered System
- Code-On-Demand (optional)

REST, RESTful: Richardson Maturity Model

- Level 0:** Define one URI, and all operations are POST requests to this URI.
- Level 1:** Create separate URIs for individual resources.
- Level 2:** Use HTTP methods to define operations on resources.
- Level 3:** Use hypermedia (HATEOAS, described below).



<https://martinfowler.com/articles/richardsonMaturityModel.html>

REST, RESTful: HATEOAS

```
GET /accounts/12345 HTTP/1.1
Host: bank.example.com
Accept: application/xml
...
```

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">100.00</balance>
  <link rel="deposit" href="https://bank.example.com/accounts/12345/deposit" />
  <link rel="withdraw" href="https://bank.example.com/accounts/12345/withdraw" />
  <link rel="transfer" href="https://bank.example.com/accounts/12345/transfer" />
  <link rel="close" href="https://bank.example.com/accounts/12345/close" />
</account>
```

REST, RESTful: HATEOAS

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
  <account_number>12345</account_number>
  <balance currency="usd">-25.00</balance>
  <link rel="deposit" href="https://bank.example.com/account/12345/deposit" />
</account>
```

Comparisons between REST vs RESTful

Attributes	REST	RESTful
Definitions	It is used to develop APIs which enable interaction between the client and the server. It should be used to get a piece of data when the user connects any link to the particular URL.	It is a web application that follows the REST infrastructure which provides interoperability between different systems on the entire network.
Web services	The working of the URL is based on request and response.	The working of RESTful is completely based on REST applications.
Data format	The data format of REST is based on HTTP.	The data format of RESTful is based on JSON, HTTP, and Text.
Adaptability	It is highly adaptable and user friendly to all the business enterprises and IT.	It is too flexible when compared to RESTLESS web services.
Protocol	The protocol is strong and it inherits many security measures which are built-in architecture layers.	It is multi-layer and has a transport protocol which makes the system less secure when compared with REST.
Bandwidth	Consumes only minimum bandwidth.	Consumes less bandwidth.

REST, RESTful: Organize the API around resources

The purpose of REST is to model entities and the operations that an application can perform on those entities. A client should not be exposed to the internal implementation.

<https://adventure-works.com/orders>

// Good

<https://adventure-works.com/create-order>

// Avoid

REST, RESTful: Organize the API around resources

Entities are often grouped together into collections (orders, customers). A collection is a separate resource from the item within the collection, and should have its own URI. For example, the following URI might represent the collection of orders:

<https://adventure-works.com/orders>

It's a good practice to organize URIs for collections and items into a hierarchy. For example, [/customers](#) is the path to the customers collection, and [/customers/5](#) is the path to the customer with ID equal to 5.

Also consider the relationships between different types of resources and how you might expose these associations.

For example, the [/customers/5/orders](#) might represent all of the orders for customer 5. You could also go in the other direction, and represent the association from an order back to a customer with a URI such as [/orders/99/customer](#).

REST, RESTful: Define operations in terms of HTTP methods

- **GET** retrieves a representation of the resource at the specified URI. The body of the response message contains the details of the requested resource.
- **POST** creates a new resource at the specified URI. The body of the request message provides the details of the new resource. Note that POST can also be used to trigger operations that don't actually create resources.
- **PUT** either creates or replaces the resource at the specified URI. The body of the request message specifies the resource to be created or updated.
- **PATCH** performs a partial update of a resource. The request body specifies the set of changes to apply to the resource.
- **DELETE** removes the resource at the specified URI.

REST, RESTful: Define operations in terms of HTTP methods

GET /books/ — get list of all books

GET /books/3/ — get book number 3

POST /books/ — add book (data in query body)

PUT /books/3 – updated books (data in query body)

DELETE /books/3 – delete the book

REST, RESTful: Define operations in terms of HTTP methods

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

REST, RESTful: Uniform program Interface

The differences between POST, PUT, and PATCH can be confusing.

- A **POST** request creates a resource. The server assigns a URI for the new resource, and returns that URI to the client. In the REST model, you frequently apply POST requests to collections. The new resource is added to the collection. A POST request can also be used to submit data for processing to an existing resource, without any new resource being created.
- A **PUT** request creates a resource or updates an existing resource. The client specifies the URI for the resource. The request body contains a complete representation of the resource. If a resource with this URI already exists, it is replaced. Otherwise a new resource is created, if the server supports doing so. PUT requests are most frequently applied to resources that are individual items, such as a specific customer, rather than collections. A server might support updates but not creation via PUT. Whether to support creation via PUT depends on whether the client can meaningfully assign a URI to a resource before it exists. If not, then use POST to create resources and PUT or PATCH to update.
- A **PATCH** request performs a partial update to an existing resource. The client specifies the URI for the resource. The request body specifies a set of changes to apply to the resource. This can be more efficient than using PUT, because the client only sends the changes, not the entire representation of the resource. Technically PATCH can also create a new resource (by specifying a set of updates to a "null" resource), if the server supports this.

PUT requests must be **idempotent**. If a client submits the same PUT request multiple times, the results should always be the same (the same resource will be modified with the same values). POST and PATCH requests are not guaranteed to be idempotent.

REST, RESTful: Uniform program Interface

HTTP-Method	Safe method	Idempotent Method
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No

REST, RESTful: Media types

Clients and servers exchange representations of resources. For example, in a POST request, the request body contains a representation of the resource to create. In a GET request, the response body contains a representation of the fetched resource.

In the HTTP protocol, formats are specified through the use of media types, also called **MIME types**. For non-binary data, most web APIs support JSON (media type = application/json) and possibly XML (media type = application/xml).

```
POST https://adventure-works.com/orders HTTP/1.1
```

```
Content-Type: application/json; charset=utf-8
```

```
Content-Length: 57
```

```
{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

If the server doesn't support the media type, it should return HTTP status code 415 (Unsupported Media Type).

REST, RESTful: Conform to HTTP semantics

➤ GET methods

A successful GET method typically returns HTTP status code **200 (OK)**. If the resource cannot be found, the method should return **404 (Not Found)**.

➤ POST methods

If a POST method creates a new resource, it returns HTTP status code **201 (Created)**. The URI of the new resource is included in the Location header of the response. The response body contains a representation of the resource.

If the method does some processing but does not create a new resource, the method can return HTTP status code **200** and include the result of the operation in the response body. Alternatively, if there is no result to return, the method can return HTTP status code **204 (No Content)** with no response body.

If the client puts invalid data into the request, the server should return HTTP status code **400 (Bad Request)**.

REST, RESTful: Conform to HTTP semantics

➤ PUT methods

If a PUT method creates a new resource, it returns HTTP status **code 201 (Created)**, as with a POST method. If the method updates an existing resource, it returns either **200 (OK)** or **204 (No Content)**. In some cases, it might not be possible to update an existing resource. In that case, consider returning HTTP status code **409 (Conflict)**.

➤ DELETE methods

If the delete operation is successful, the web server should respond with HTTP status code **204**, indicating that the process has been successfully handled, but that the response body contains no further information. If the resource doesn't exist, the web server can return HTTP **404 (Not Found)**.

REST, RESTful: Filter and paginate data

`/orders?minCost=n`

`/orders?limit=25&offset=50`

`/orders?sort=ProductID`

`/orders?fields=ProductID,Quantity`

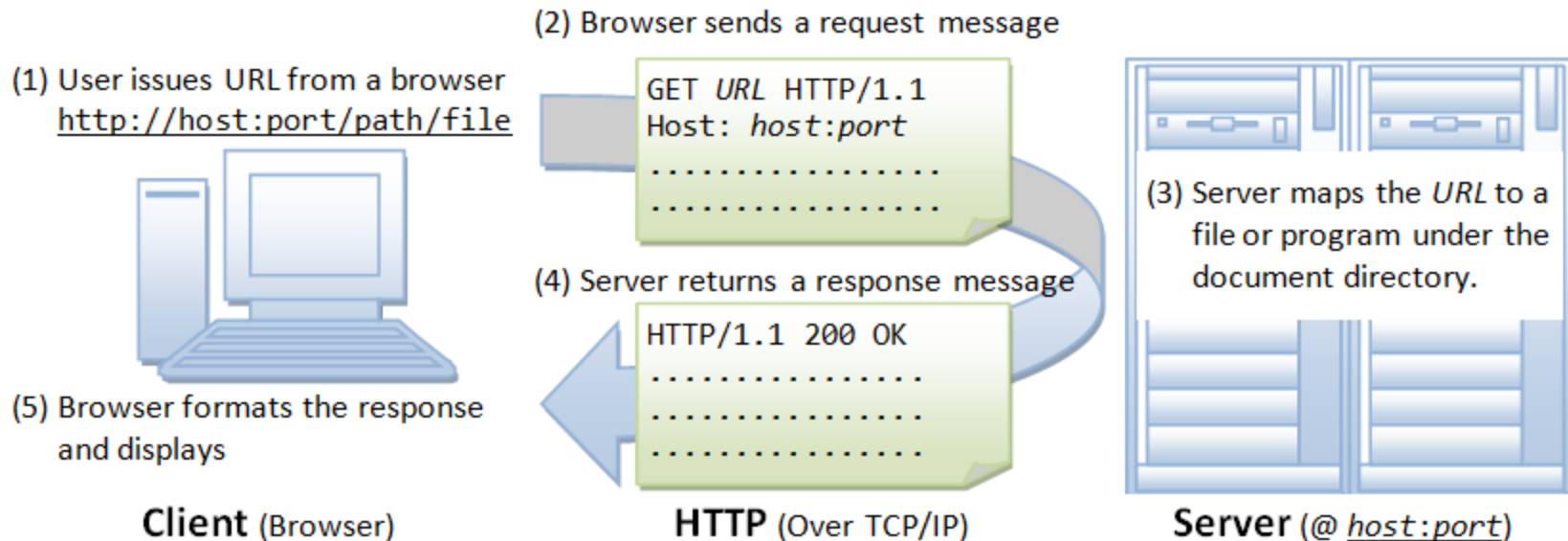
REST, RESTful: Links

<https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>

<https://github.com/microsoft/api-guidelines>

HTTP request-response. HTTP processing pipeline

HTTP request-response



WHAT IS HTTP PIPELINING?

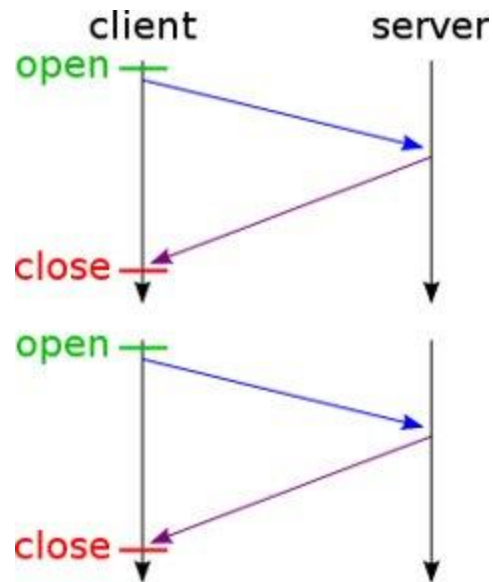
HTTP processing pipeline

- A typical HTTP request looks something like this:

```
GET / HTTP/1.1
Host: www.brianbondy.com
User-Agent: Mozilla/5.0
Connection: keep-alive
```

- A typical HTTP response looks something like this:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Server: Google Frontend
Content-Length: 12100
...content...
```



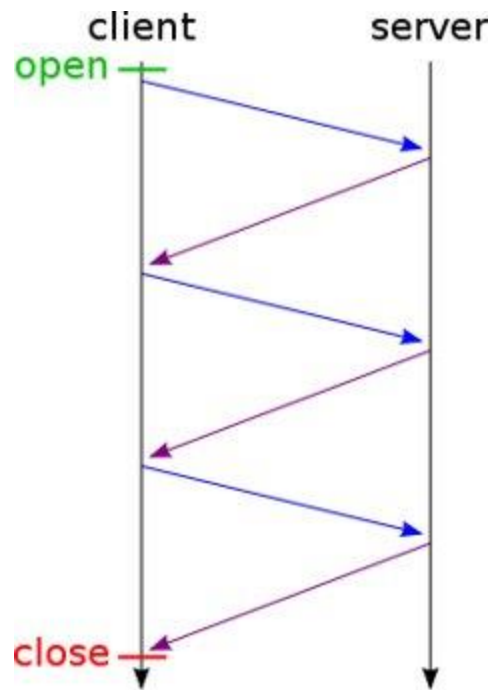
HTTP processing pipeline

- **Several requests to a single server** are very typical. For example an HTML file can have several referenced images.

To avoid creating several connections, HTTP 1.1 introduced persistent connections.

The picture on the right shows 3 requests and responses on a single persistent connection.

Having several connections can give better speed, but if you need to create a new connection for each and every request, it will use much more resources, require more TCP handshakes, and will be susceptible to TCP slow-start.

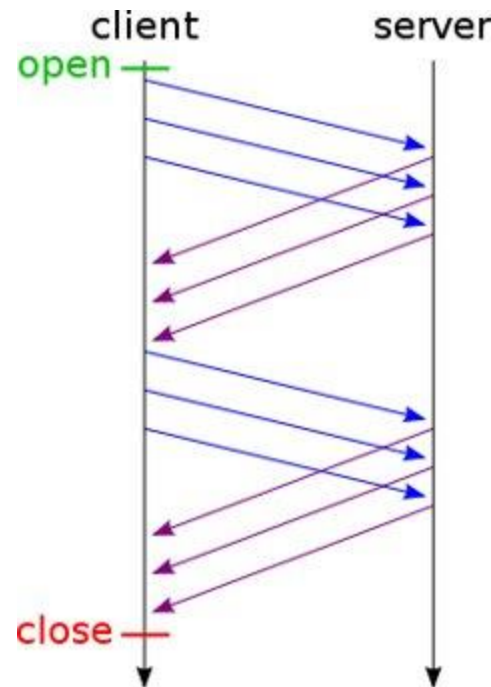


HTTP processing pipeline

- **HTTP pipelining** is a feature of HTTP 1.1 persistent connections. It means that you can send multiple requests on the same socket without waiting for each response.

The picture on the right shows 6 requests and responses using at most 3 requests at a time.

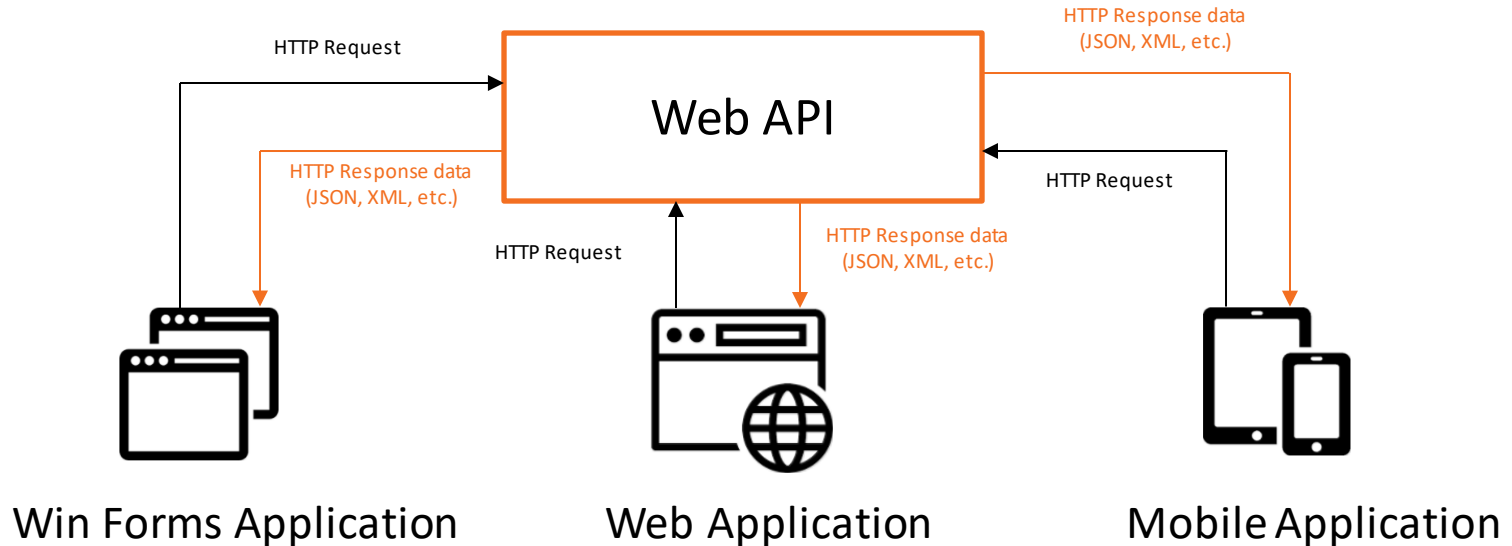
HTTP is based on TCP, and one of TCP's guarantees is ordered delivery. This means that all of the requests sent out on the same socket, will be received in that order on the server. An HTTP server that supports HTTP pipelining will send its responses in the same order.



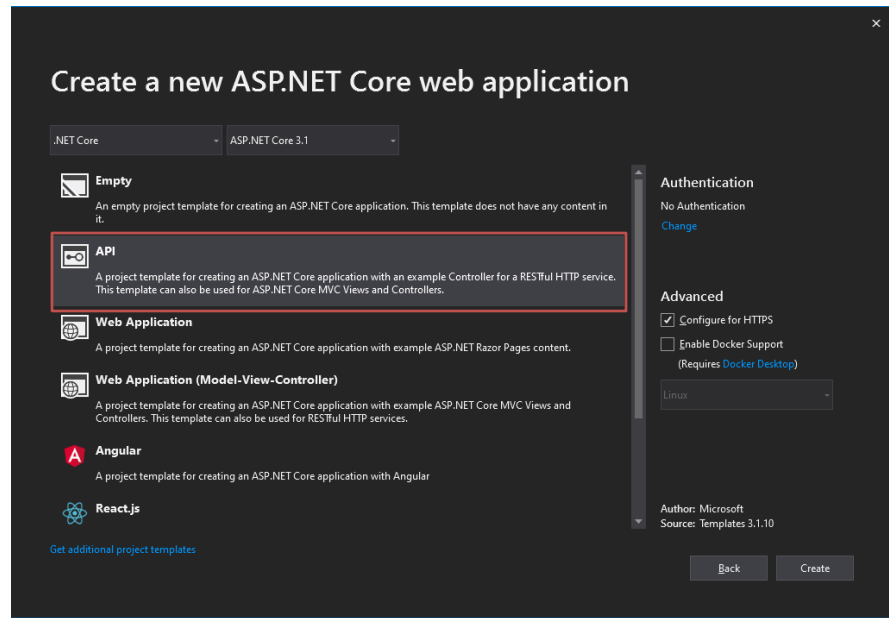
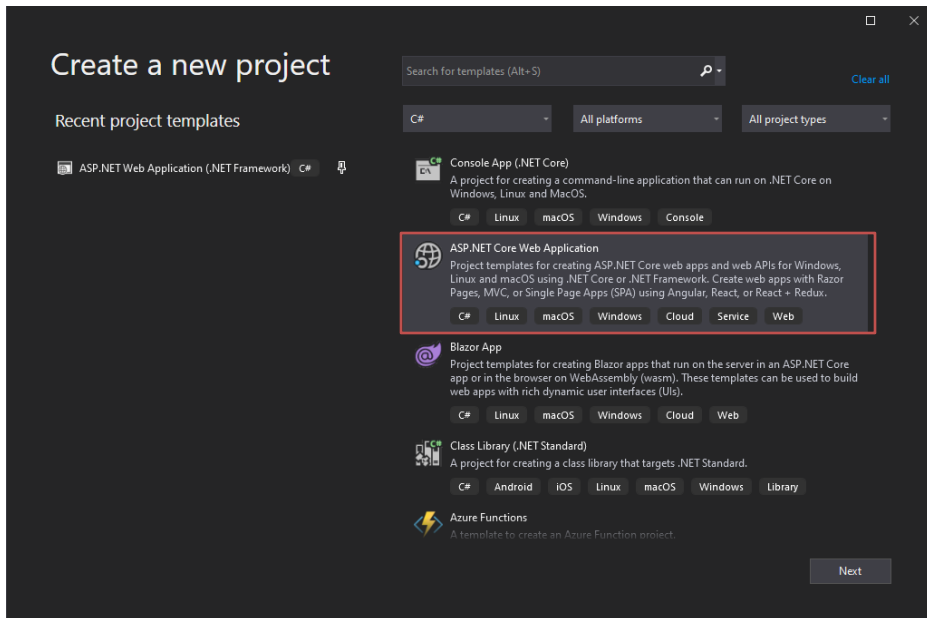
ASP.NET Web API overview

ASP.NET Web API overview: What is Web API?

The ASP.NET Web API is an extensible framework for building **HTTP** based services that can be accessed in different applications on different platforms such as web, windows, mobile etc. It works more or less the same way as ASP.NET MVC web application except that it sends data as a response instead of html view. It is like a webservice or WCF service but the exception is that it only supports **HTTP** protocol.



ASP.NET Web API: DEMO



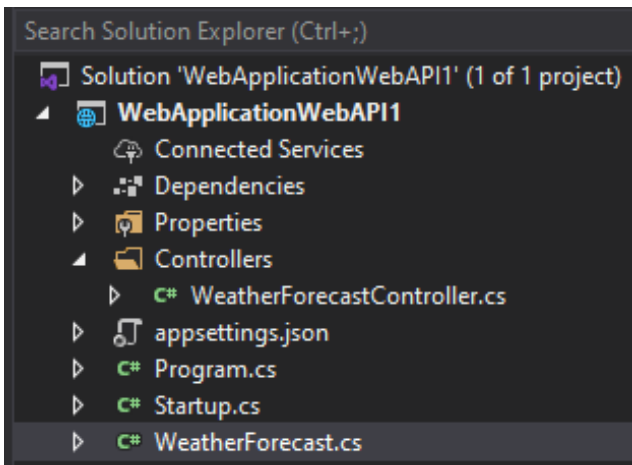
You can create a Web API project in two ways:

- Web API with MVC Project
- Stand-alone Web API Project

.NET Core CLI:

`dotnet new webapi -o TodoApi cd TodoApi dotnet add package`

ASP.NET Web API overview



```
// ~/WeatherForecast.ch
```

```
public class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
    public string Summary { get; set; }
}
```

```
[ApiController]
```

```
[Route("[controller]")]
```

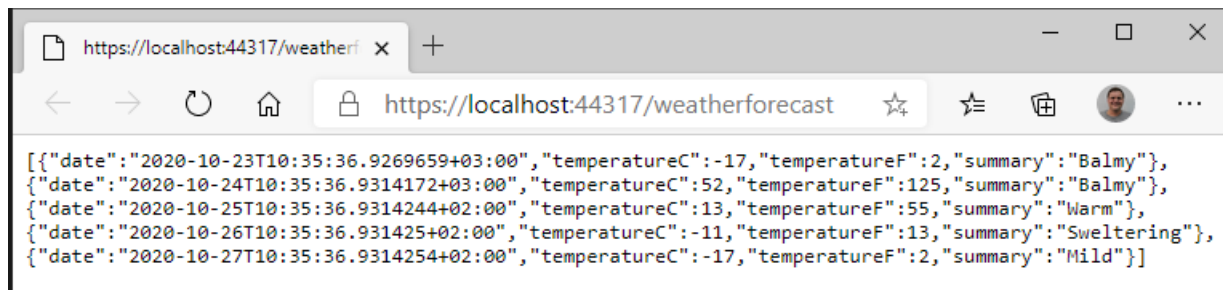
```
// ~/Controllers/WeatherForecastController.ch
```

```
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };
    private readonly ILogger<WeatherForecastController> _logger;
    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }
}
```

```
[HttpGet]
```

```
public IEnumerable<WeatherForecast> Get()
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

ASP.NET Web API overview



ASP.NET Web API overview

```
public class Startup
```

```
{  
    public Startup(IConfiguration configuration)  
    {  
        Configuration = configuration;  
    }  
  
    public IConfiguration Configuration { get; }  
  
    // This method gets called by the runtime.  
    // Use this method to add services to the container.  
    public void ConfigureServices(IServiceCollection services)  
    {  
        services.AddControllers();  
    }  
}
```

```
// This method gets called by the runtime.  
// Use this method to configure the HTTP request pipeline.  
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }  
  
    app.UseHttpsRedirection();  
  
    app.UseRouting();  
  
    app.UseAuthorization();  
  
    app.UseEndpoints(endpoints =>  
    {  
        endpoints.MapControllers();  
    });  
}
```

Configure Web API (.Net Core)

Web API supports code based configuration. It **cannot** be configured in web.config file. We can configure Web API to customize the behavior of Web API hosting infrastructure and components such as routes, formatters, filters, DependencyResolver, MessageHandlers, ParamterBindingRules, properties, services etc.

Program.cs

```
public class Program
{
    [0 references | 0 exceptions]
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }
    [2]
    [1 reference | 0 exceptions]
    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Startup.cs

```
public class Startup
{
    [0 references | 0 exceptions]
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    [1 reference | 0 exceptions]
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    [0 references | 0 exceptions]
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    [0 references | 0 exceptions]
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseMvc();
    }
}
```

Configure Web API (.Net Standard)

Web API supports code based configuration. It **cannot** be configured in web.config file. We can configure Web API to customize the behavior of Web API hosting infrastructure and components such as routes, formatters, filters, DependencyResolver, MessageHandlers, ParamterBindingRules, properties, services etc.

Global.asax

```
0 references
public class WebApiApplication : System.Web.HttpApplication
{
    0 references
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

WebApiConfig.cs

```
1 reference
public static class WebApiConfig
{
    1 reference
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Web API routes
        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```


Web API Routing

Web API routing is **similar** to ASP.NET MVC Routing. It routes an incoming HTTP request to a particular action method on a Web API controller.

Web API supports two types of routing:

- Convention-based Routing
- Attribute Routing

```
0 references
public class StudentController : ApiController
{
    [Route("api/student/names")]
    0 references
    public IEnumerable<string> Get()
    {
        return new string[] { "student1", "student2" };
    }
}
```

```
1 reference
public static void Register(HttpConfiguration config)
{
    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "School",
        routeTemplate: "api/myschool/{id}",
        defaults: new { controller = "school", id = RouteParameter.Optional },
        constraints: new { id = "/d+" }
    );

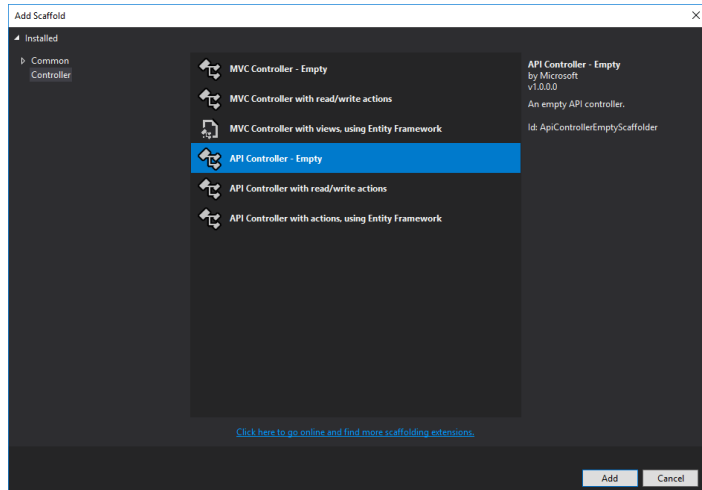
    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

Controllers. Request handling

Controllers: Web API Controller

Web API Controller is **similar** to ASP.NET MVC controller. It handles incoming **HTTP** requests and send response back to the caller.

Web API controller is a class which can be created under the Controllers folder or any other folder under your project's root folder. The name of a controller class **must** end with "Controller" and it must be derived from **System.Web.Http.ApiController** class. All the public methods of the controller are called action methods.



```
[Route("api/[controller]")]
[ApiController]
References
public class CarsController : ControllerBase
{
    // GET api/csrs
    [HttpGet]
    References | 0 requests | 0 exceptions
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "AUDI", "BMW", "Mercedes" };
    }

    // GET api/cars/5
    [HttpGet("{id}")]
    References | 0 requests | 0 exceptions
    public ActionResult<string> Get(int id)
    {
        return "AUDI";
    }

    // POST api/cars
    [HttpPost]
    References | 0 requests | 0 exceptions
    public void Post([FromBody] string value)
    {
    }

    // PUT api/cars/5
    [HttpPut("{id}")]
    References | 0 requests | 0 exceptions
    public void Put(int id, [FromBody] string value)
    {
    }

    // DELETE api/cars/5
    [HttpDelete("{id}")]
    References | 0 requests | 0 exceptions
    public void Delete(int id)
    {
    }
}
```

Controllers. Request handling

Based on the incoming request URL and HTTP verb (**GET** / **POST** / **PUT** / **PATCH** / **DELETE**), Web API decides which Web API controller and action method to execute e.g. **Get()** method will handle **HTTP GET** request, **Post()** method will handle **HTTP POST** request, **Put()** method will handle **HTTP PUT** request and **Delete()** method will handle **HTTP DELETE** request for the above Web API.

If you want to write methods that do not start with an HTTP verb then you can apply the appropriate http verb attribute on the method such as **HttpGet**, **HttpPost**, **HttpPut** etc. same as MVC controller.

```
// PUT: api/Student/5
0 references
public void PutStudent(int id, [FromBody]string value)
{
}

// DELETE: api/Student/5
0 references
public void DeleteStudent(int id)
{
}
```

```
// PUT: api/Student/5
[HttpPut]
0 references
public void Update(int id, [FromBody]string value)
{
}

// DELETE: api/Student/5
[HttpDelete]
0 references
public void Remove(int id)
{
}
```

```
0 references
public class StudentController : ApiController
{
    // GET: api/Student
    0 references
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

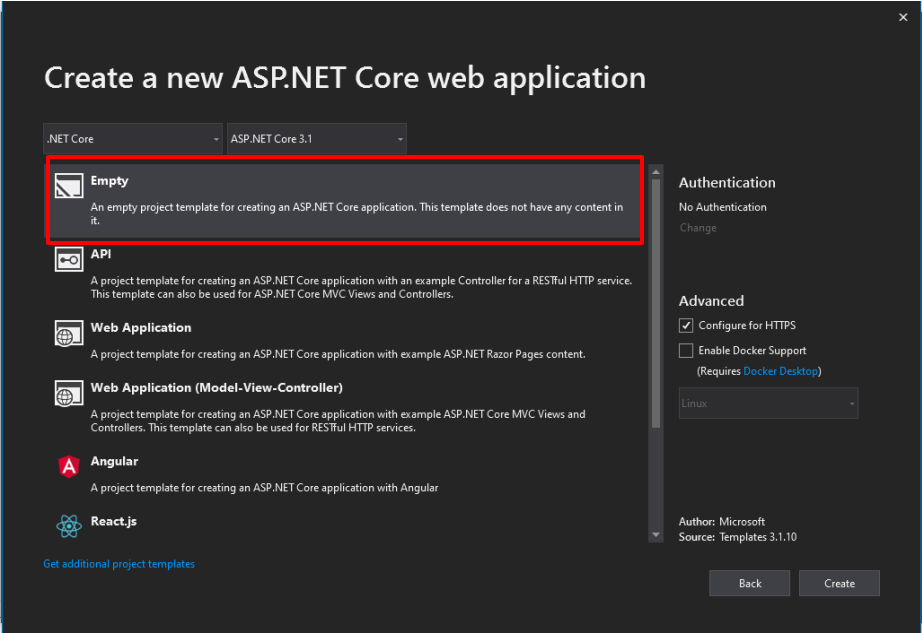
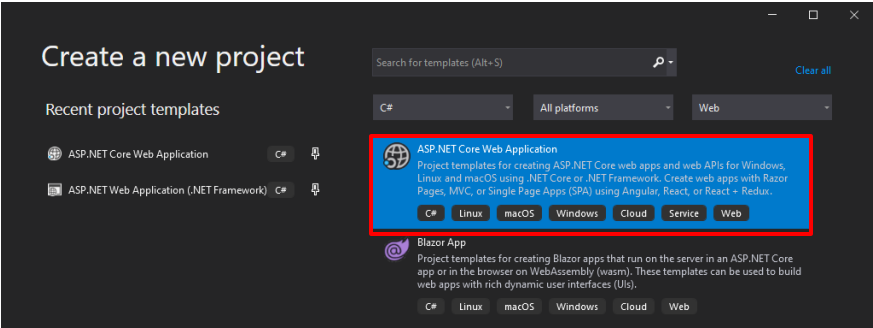
    // GET: api/Student/5
    0 references
    public string Get(int id)
    {
        return "value";
    }

    // POST: api/Student
    0 references
    public void Post([FromBody]string value)
    {
    }

    // PUT: api/Student/5
    0 references
    public void Put(int id, [FromBody]string value)
    {
    }

    // DELETE: api/Student/5
    0 references
    public void Delete(int id)
    {
    }
}
```

Controllers: DEMO



Controllers: DEMO

//~Models/User.cs

```
public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

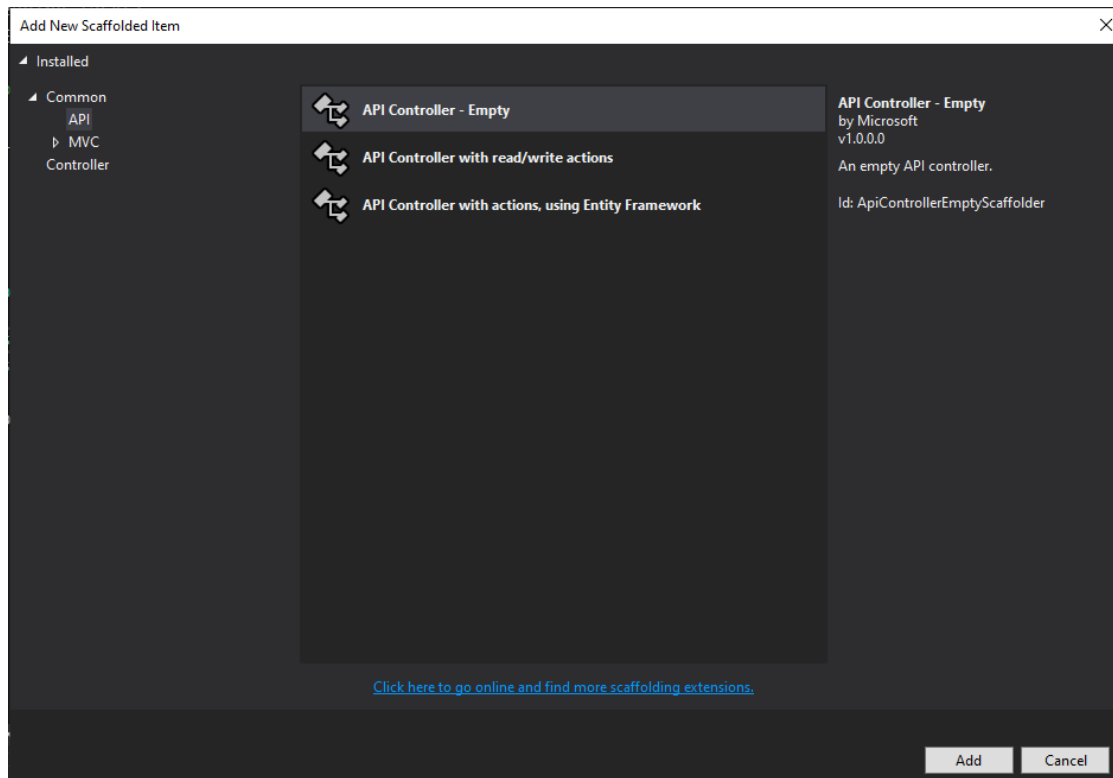
//~Models/UsersContext.cs

```
public class UsersContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public UsersContext(DbContextOptions<UsersContext> options)
        : base(options)
    {
        Database.EnsureCreated();
    }
}
```

Nuget:

Microsoft.EntityFrameworkCore.SqlServer

Controllers: DEMO



Controllers: DEMO

```
[Route("api/[controller]")]
```

```
[ApiController]
```

```
public class UsersController : ControllerBase
```

```
{
    UsersContext db;
    public UsersController(UsersContext context)
    {
        db = context;
        if (!db.Users.Any())
        {
            db.Users.Add(new User { Name = "Tom", Age = 26 });
            db.Users.Add(new User { Name = "Alice", Age = 31 });
            db.SaveChanges();
        }
    }
}
```

```
[HttpGet]
public async Task<ActionResult<IEnumerable<User>>> Get()
{
    return await db.Users.ToListAsync();
}
```

```
// GET api/users/5
[HttpGet("{id}")]
public async Task<ActionResult<User>> Get(int id)
{
    User user = await db.Users.FirstOrDefaultAsync(x => x.Id == id);
    if (user == null)
        return NotFound();
    return new ObjectResult(user);
}
```

```
// POST api/users
[HttpPost]
public async Task<ActionResult<User>> Post(User user)
{
    if (user == null)
    {
        return BadRequest();
    }

    db.Users.Add(user);
    await db.SaveChangesAsync();
    return Ok(user);
}
```

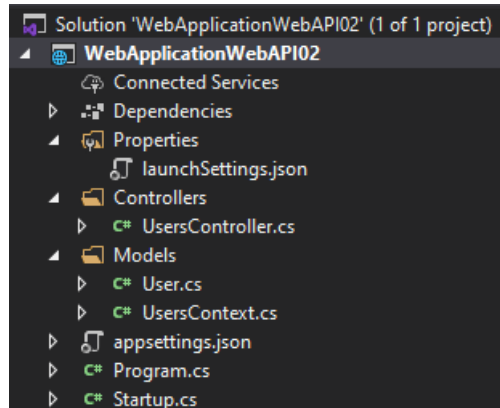
```
// PUT api/users/
[HttpPut]
public async Task<ActionResult<User>> Put(User user)
{
    if (user == null)
    {
        return BadRequest();
    }

    if (!db.Users.Any(x => x.Id == user.Id))
    {
        return NotFound();
    }

    db.Update(user);
    await db.SaveChangesAsync();
    return Ok(user);
}
```

```
// DELETE api/users/5
[HttpDelete("{id}")]
public async Task<ActionResult<User>> Delete(int id)
{
    User user = db.Users.FirstOrDefault(x => x.Id == id);
    if (user == null)
    {
        return NotFound();
    }

    db.Users.Remove(user);
    await db.SaveChangesAsync();
    return Ok(user);
}
```



Controllers: DEMO

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // set the data context
        string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted_Connection=True;";
        services.AddDbContext<UsersContext>(options => options.UseSqlServer(con));
        services.AddControllers(); // using controllers without views
    }

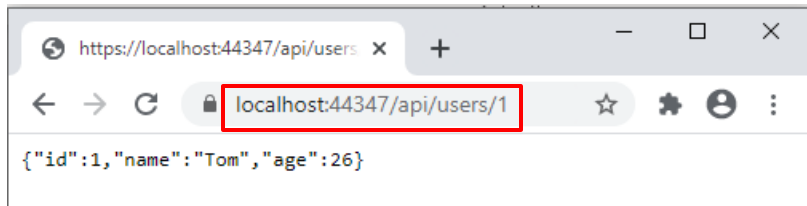
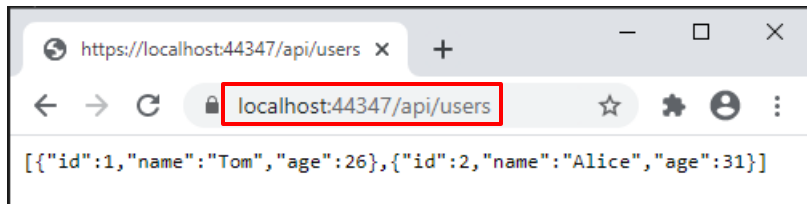
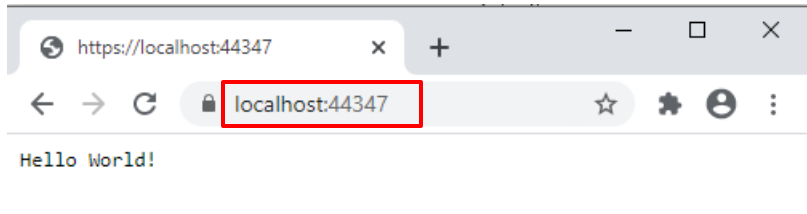
    // This method gets called by the runtime.
    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Hello World!");
            });

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers(); // connect routing to controllers
            });
        });
    }
}
```

Controllers: DEMO



```
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
```

```
[HttpGet]
public async Task<ActionResult<IEnumerable<User>>> Get()
{
    return await db.Users.ToListAsync();
}
```

```
// GET api/users/5
[HttpGet("{id}")]
public async Task<ActionResult<User>> Get(int id)
{
    User user = await db.Users.FirstOrDefaultAsync(x => x.Id == id);
    if (user == null)
        return NotFound();
    return new ObjectResult(user);
}
```

Fiddler and Postman

Fiddler and Postman

Tools:

- Custom client applications (web pages, mobile app, console app an etc.)
- Special application:
 - Fiddler
 - Postman
 - etc.
- Browser extensions
 - Fiddler
 - Postman
 - etc.
- PowerShell via windows Package Manager Console in Visual Studio
- etc.

Fiddler and Postman



<https://www.telerik.com/fiddler>



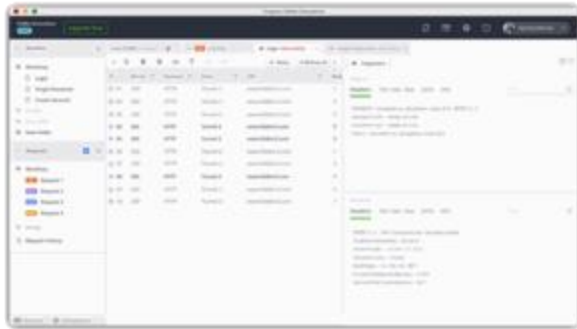
<https://www.postman.com/downloads/>

Fiddler



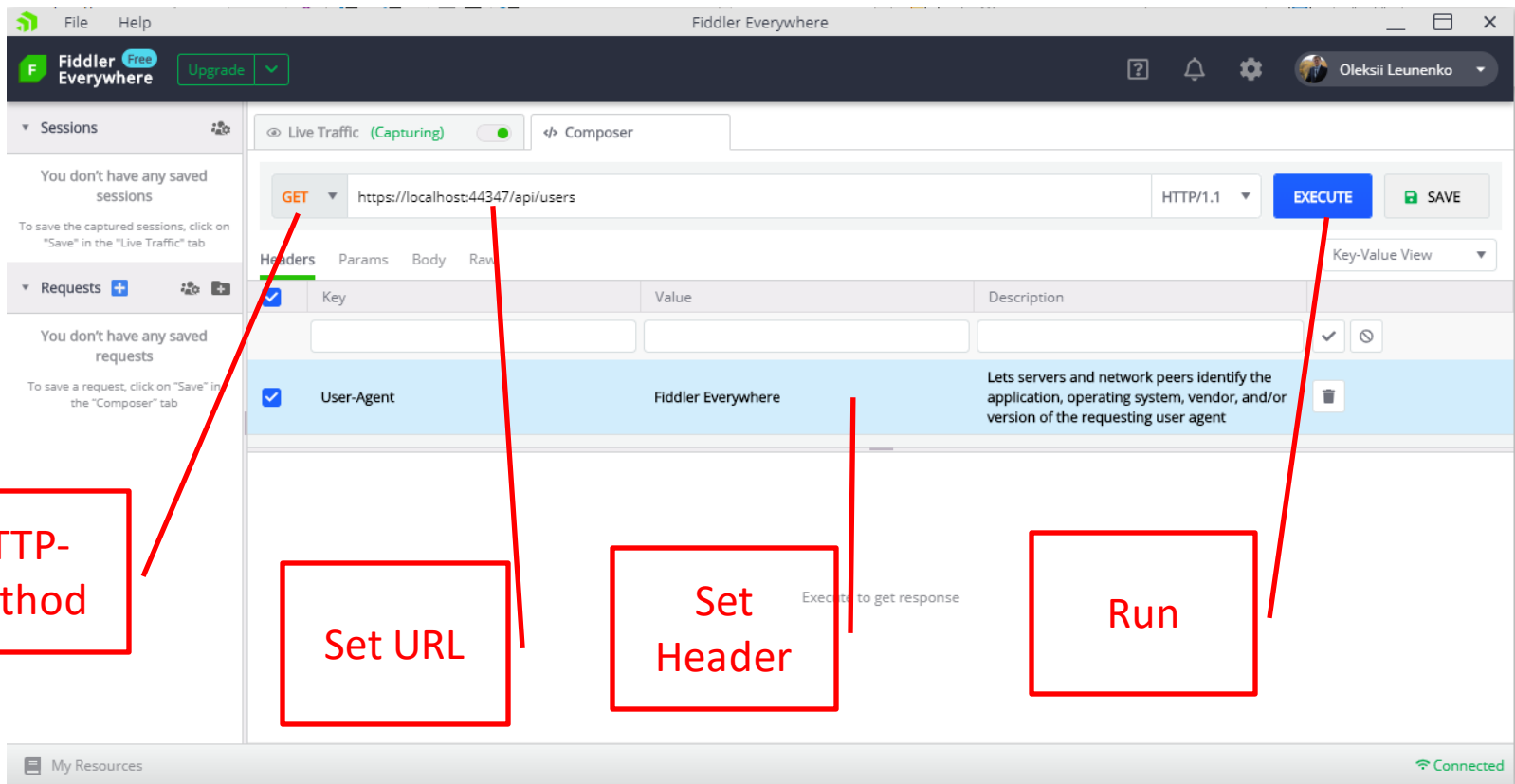
Fiddler is a debugging proxy server tool used to log, inspect, and alter HTTP and HTTPS traffic between a computer and the Internet. Fiddler was originally written by Eric Lawrence while a Program Manager on the Internet Explorer development team at Microsoft.

The usage of the name “Fiddler” has broadened to encompass additional tooling provided by Progress Telerik including Fiddler Everywhere, Fiddler Core, FiddlerCap, and Fiddler Jam.



<https://www.telerik.com/fiddler>

Fiddler



Fiddler

The screenshot displays the Fiddler Everywhere application interface. The top bar includes the 'Fiddler Everywhere' logo, an 'Upgrade' button, and a user profile 'Oleksii Leunenکو'. The main interface is divided into several sections:

- Sessions:** A sidebar on the left with instructions on how to save sessions and requests.
- Requests:** A central table listing captured requests. The table has columns for #, Result, Protocol, Host, and URL. Request #64 is highlighted with a red box.
- Inspectors:** A panel on the right showing details for the selected request. It includes tabs for Headers, Text, Web Forms, Cookies, Raw, and JSON. The 'Headers' tab is active, showing the request headers. The 'Text' tab is also visible, showing the response body.

Request List:

#	Result	Protocol	Host	URL
55	200	HTTP	Tunnel to	mobile.pipe.ai
56	200	HTTP	Tunnel to	mobile.pipe.ai
57	200	HTTP	Tunnel to	roaming.office
58	200	HTTP	Tunnel to	mobile.pipe.ai
59	200	HTTP	Tunnel to	mobile.pipe.ai
60	200	HTTP	Tunnel to	play.google.co
61	200	HTTP	Tunnel to	static.asm.sky
62	200	HTTP	Tunnel to	static-asm.sec
63	200	HTTP	Tunnel to	roaming.office
64	200	HTTPS	localhost:44347	/api/users
65	200	HTTP	Tunnel to	presence.tean
66	200	HTTP	Tunnel to	mobile.pipe.ai
67	200	HTTP	Tunnel to	roaming.office
68	200	HTTP	Tunnel to	mobile.pipe.ai
69	200	HTTP	Tunnel to	mobile.pipe.ai
70	200	HTTP	Tunnel to	self.events.da

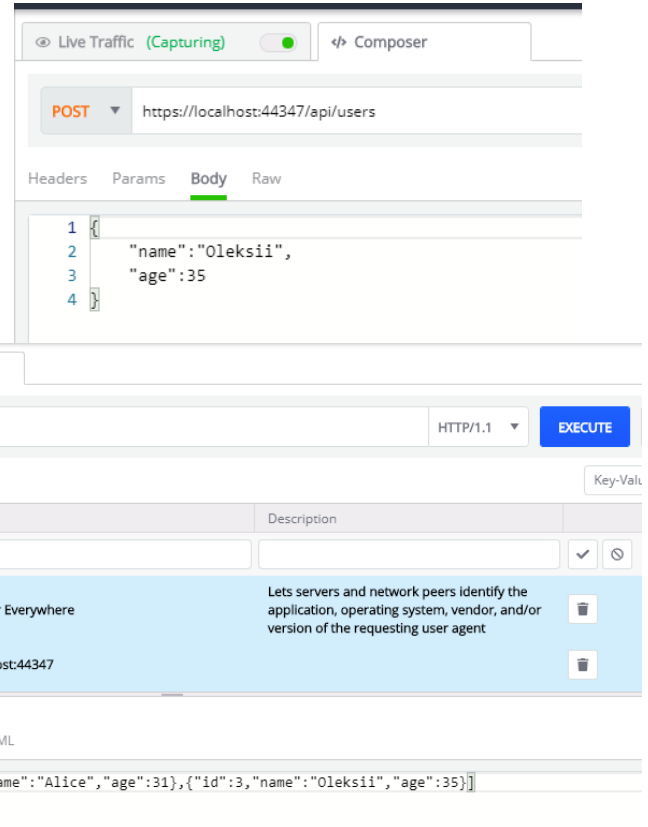
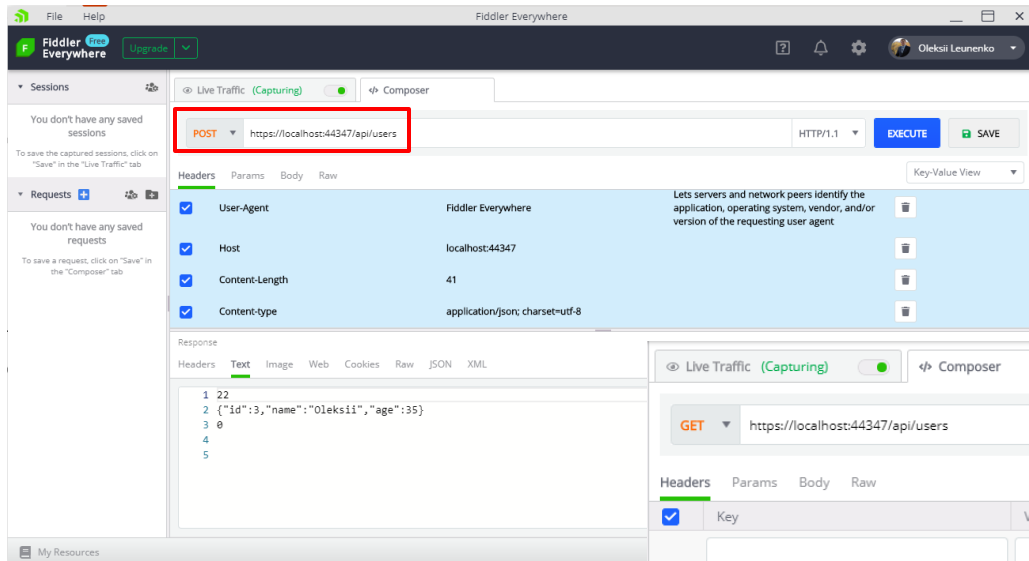
Request Details (Request #64):

Request Headers:

```
1 GET https://localhost:44347/api/users HTTP/1.1
2 User-Agent: Fiddler Everywhere
3 Host: localhost:44347
4
```

Response Data:

```
1 41
2 [{"id":1,"name":"Tom","age":26},{"id":2,"name":"Alice",
3 "age":31}]
4
5
```

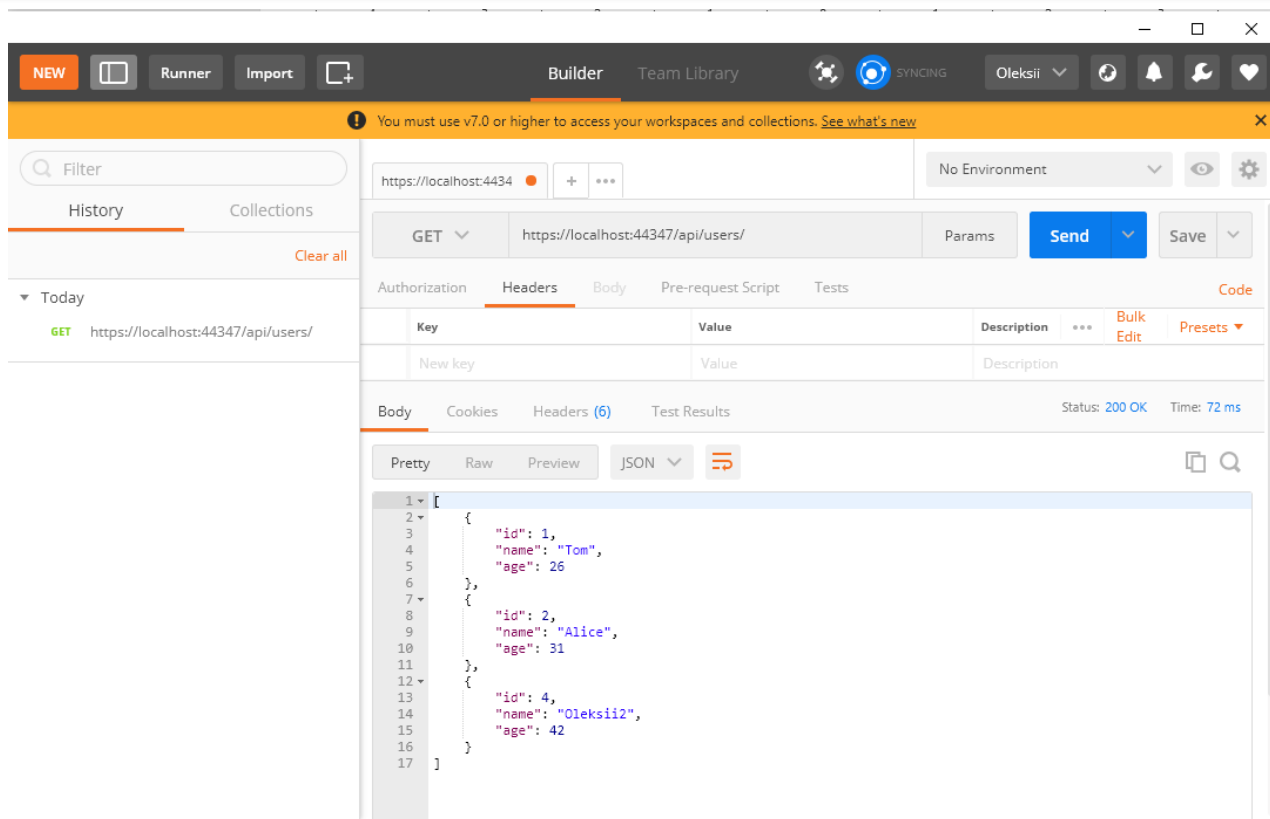
Postman



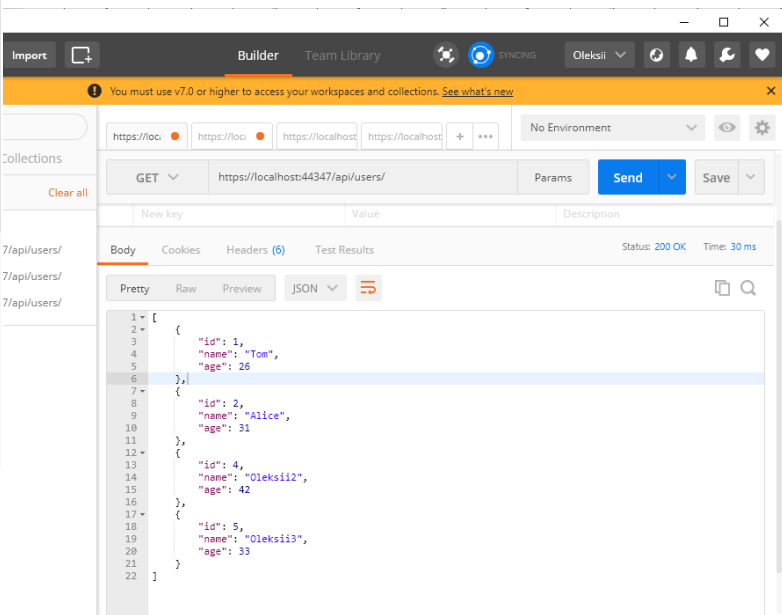
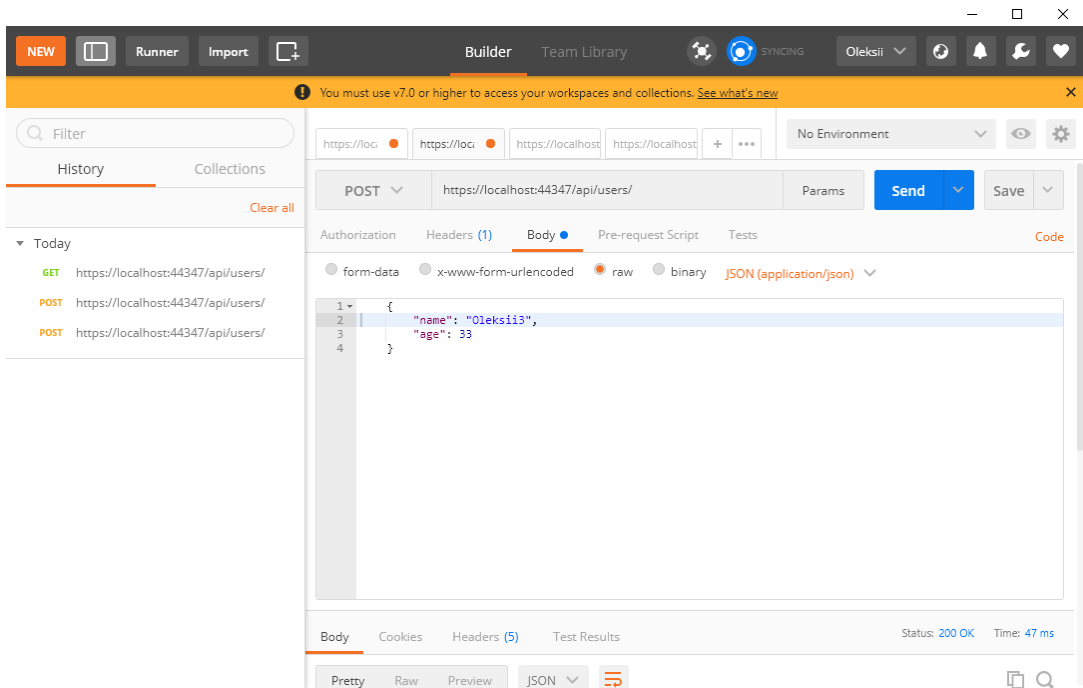
POSTMAN

<https://www.postman.com/downloads/>

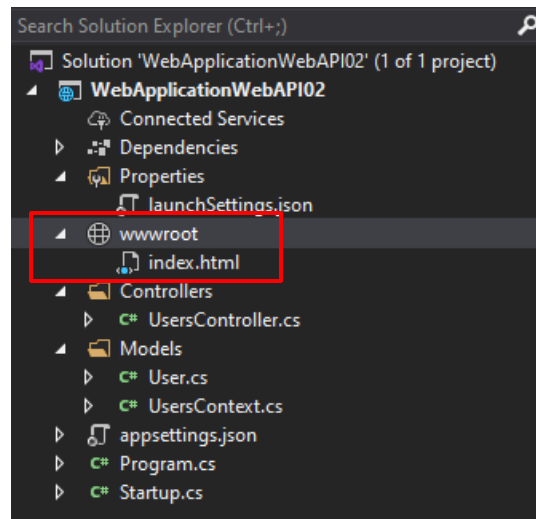
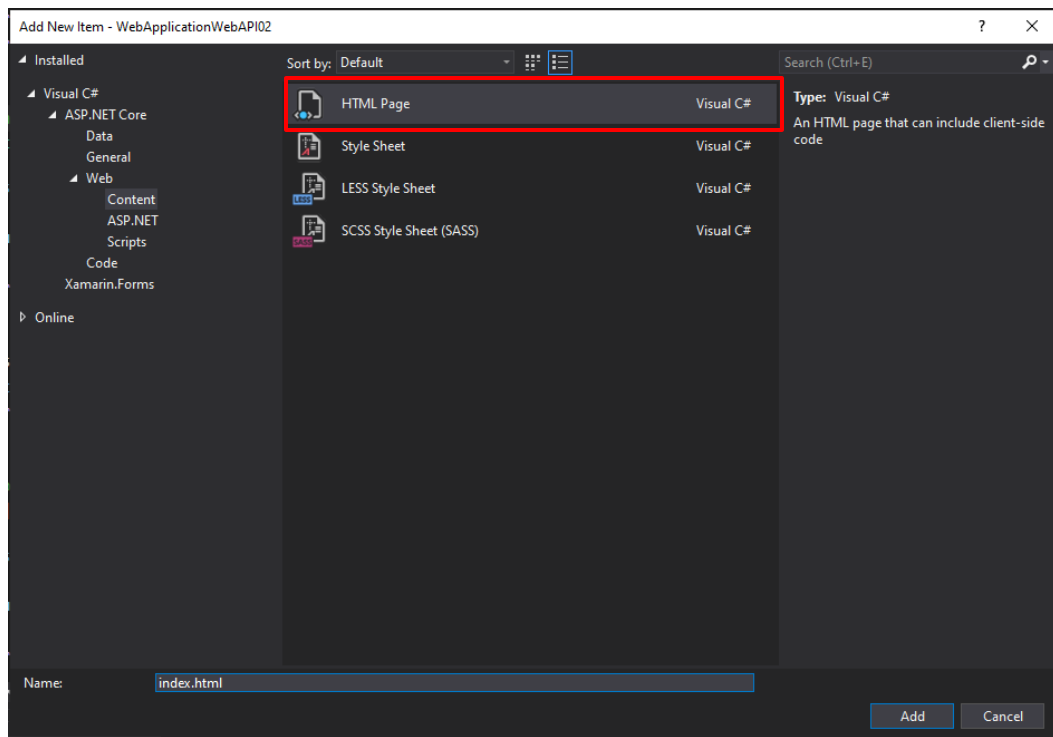
Postman



Postman



Create Web API client



Create Web API client

```
public class Startup
```

```
{
```

```
    public void ConfigureServices(IServiceCollection services)
```

```
    {
```

```
        // set the data context
```

```
        string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted_Connection=True;";
```

```
        services.AddDbContext<UsersContext>(options => options.UseSqlServer(con));
```

```
        services.AddControllers(); // using controllers without views
```

```
    }
```

```
    // This method gets called by the runtime.
```

```
    // Use this method to configure the HTTP request pipeline.
```

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

```
    {
```

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

```
        {
```

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

```
        }
```

```
        app.UseDefaultFiles();  
        app.UseStaticFiles();
```

```
        app.UseRouting();
```

```
        app.UseEndpoints(endpoints =>
```

```
        {
```

```
            endpoints.MapGet("/", async context =>
```

```
            {
```

```
                await context.Response.WriteAsync("Hello World!");
```

```
            });
```

```
        app.UseEndpoints(endpoints =>
```

```
        {
```

```
            endpoints.MapControllers();
```

```
        });
```

```
    });
```

```
}}
```

http://localhost:xxxx/index.html

Create Web API client

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>User list</title>
  <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <h2>User list</h2>
  <form name="userForm">
    <input type="hidden" name="id" value="0" />
    <div class="form-group col-md-5">
      <label for="name">Name:</label>
      <input class="form-control" name="name" />
    </div>
    <div class="form-group col-md-5">
      <label for="age">Age:</label>
      <input class="form-control" name="age" type="number" />
    </div>
    <div class="panel-body">
      <button type="submit" id="submit" class="btn btn-primary">Save</button>
      <a id="reset" class="btn btn-primary">Reset</a>
    </div>
  </form>
  <table class="table table-condensed table-striped col-md-6">
    <thead><tr><th>Id</th><th>Name</th><th>age</th><th></th></tr></thead>
    <tbody>
    </tbody>
  </table>
  <script> ... </script>
</body>
</html>
```

Create Web API client

// Get all users

```
async function GetUsers() {  
  // send request and get response  
  const response = await fetch("/api/users", {  
    method: "GET",  
    headers: { "Accept": "application/json" }  
  });  
  // If request is OK  
  if (response.ok === true) {  
    // get data  
    const users = await response.json();  
    let rows = document.querySelector("tbody");  
    users.forEach(user => {  
      // add elements to table  
      rows.append(row(user));  
    });  
  }  
}
```

// Get User by id

```
async function GetUser(id) {  
  const response = await fetch("/api/users/" + id, {  
    method: "GET",  
    headers: { "Accept": "application/json" }  
  });  
  if (response.ok === true) {  
    const user = await response.json();  
    const form = document.forms["userForm"];  
    form.elements["id"].value = user.id;  
    form.elements["name"].value = user.name;  
    form.elements["age"].value = user.age;  
  }  
}
```

// Create User

```
async function CreateUser(userName, userAge) {  
  const response = await fetch("/api/users", {  
    method: "POST",  
    headers: { "Accept": "application/json", "Content-Type": "application/json" },  
    body: JSON.stringify({  
      name: userName,  
      age: parseInt(userAge, 10)  
    })  
  });  
  if (response.ok === true) {  
    const user = await response.json();  
    reset();  
    document.querySelector("tbody").append(row(user));  
  }  
}
```

// Edit User

```
async function EditUser(userId, userName, userAge) {  
  const response = await fetch("/api/users", {  
    method: "PUT",  
    headers: { "Accept": "application/json", "Content-Type": "application/json" },  
    body: JSON.stringify({  
      id: parseInt(userId, 10),  
      name: userName,  
      age: parseInt(userAge, 10)  
    })  
  });  
  if (response.ok === true) {  
    const user = await response.json();  
    reset();  
    document.querySelector("tr[data-rowid=" + user.id + "]").replaceWith(row(user));  
  }  
}
```

// Delete User

```
async function DeleteUser(id) {  
  const response = await fetch("/api/users/" + id, {  
    method: "DELETE",  
    headers: { "Accept": "application/json" }  
  });  
  if (response.ok === true) {  
    const user = await response.json();  
    document.querySelector("tr[data-rowid=" + user.id + "]").remove();  
  }  
}
```


Create Web API client

```
// Reset form
function reset() {
  const form = document.forms["userForm"];
  form.reset();
  form.elements["id"].value = 0;
}

// Create table row
function row(user) {

  const tr = document.createElement("tr");
  tr.setAttribute("data-rowid", user.id);

  const idTd = document.createElement("td");
  idTd.append(user.id);
  tr.append(idTd);

  const nameTd = document.createElement("td");
  nameTd.append(user.name);
  tr.append(nameTd);

  const ageTd = document.createElement("td");
  ageTd.append(user.age);
  tr.append(ageTd);

  const linksTd = document.createElement("td");

  const editLink = document.createElement("a");
  editLink.setAttribute("data-id", user.id);
  editLink.setAttribute("style", "cursor:pointer;padding:15px;");
  editLink.append("Change");

  editLink.addEventListener("click", e => {
    e.preventDefault();
    GetUser(user.id);
  });
  linksTd.append(editLink);

  const removeLink = document.createElement("a");
  removeLink.setAttribute("data-id", user.id);
  removeLink.setAttribute("style", "cursor:pointer;padding:15px;");
  removeLink.append("Delete");
  removeLink.addEventListener("click", e => {
    e.preventDefault();
    DeleteUser(user.id);
  });
  linksTd.append(removeLink);
  tr.appendChild(linksTd);

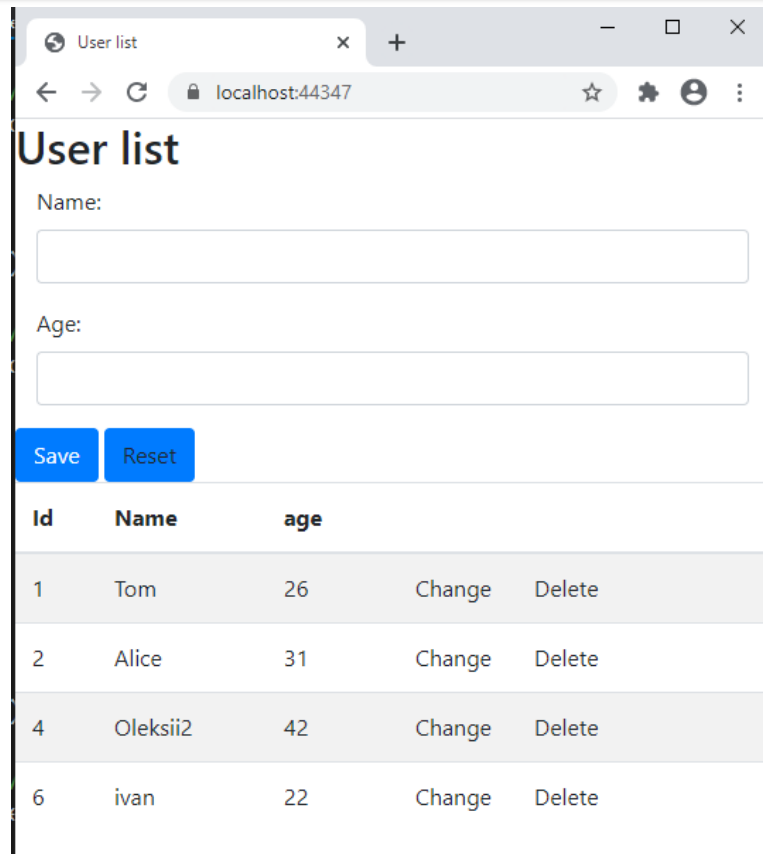
  return tr;
}

// Reset form values
document.getElementById("reset").click(function (e) {
  e.preventDefault();
  reset();
})

// Send form
document.forms["userForm"].addEventListener("submit", e => {
  e.preventDefault();
  const form = document.forms["userForm"];
  const id = form.elements["id"].value;
  const name = form.elements["name"].value;
  const age = form.elements["age"].value;
  if (id == 0)
    CreateUser(name, age);
  else
    EditUser(id, name, age);
});

// Load users
GetUsers();
```

Create Web API client



User list

Name:

Age:

Save Reset

Id	Name	age		
1	Tom	26	Change	Delete
2	Alice	31	Change	Delete
4	Oleksii2	42	Change	Delete
6	ivan	22	Change	Delete

Swagger overview

When consuming a Web API, understanding its various methods can be challenging for a developer. Swagger, also known as OpenAPI, solves the problem of generating useful documentation and help pages for Web APIs. It provides benefits such as interactive documentation, client SDK generation, and API discoverability.

Swagger UI offers a web-based UI that provides information about the service, using the generated Swagger specification. Both Swashbuckle and NSwag include an embedded version of Swagger UI, so that it can be hosted in your ASP.NET Core app using a middleware registration call.

Swagger installation

Swashbuckle can be added with the following approaches:

From the **Package Manager Console** window:

Go to **View > Other Windows > Package Manager Console**

Navigate to the directory in which the csproj file exists

Execute the following command:

Install-Package Swashbuckle.AspNetCore

From the **Manage NuGet Packages** dialog:

Right-click the project in Solution Explorer > Manage NuGet Packages

Set the Package source to "nuget.org"

Enter "**Swashbuckle.AspNetCore**" in the search box

Select the "**Swashbuckle.AspNetCore**" package from the Browse tab and click Install

Swagger configuration

Add the Swagger generator to the services collection in the Startup.ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services)
{
    string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted_Connection=True;";
    // set the data context
    services.AddDbContext<UsersContext>(options => options.UseSqlServer(con));

    services.AddControllers(); // using controllers without views

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

Swagger configuration

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

```
{  
    if (env.IsDevelopment())  
    {  
        app.UseDeveloperExceptionPage();  
    }  
}
```

```
app.UseDefaultFiles();  
app.UseStaticFiles();
```

```
// Enable middleware to serve generated Swagger as a JSON endpoint.  
app.UseSwagger();
```

```
// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),  
// specifying the Swagger JSON endpoint.  
app.UseSwaggerUI(c=>  
{  
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");  
});
```

```
app.UseRouting();
```

```
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapGet("/", async context =>  
    {  
        await context.Response.WriteAsync("Hello World!");  
    });  
    app.UseEndpoints(endpoints =>  
    {  
        endpoints.MapControllers(); // connect routing to controllers  
    });  
});  
}
```

In the Startup.Configure method, enable the middleware for serving the generated JSON document and the Swagger UI:

Swagger UI

The screenshot shows the Swagger UI interface in a web browser. The address bar displays 'localhost:44347/swagger/index.html'. The header includes the Swagger logo and a dropdown menu labeled 'Select a definition' with 'My API V1' selected. The main heading is 'WebApplicationWebAPI02' with a '1.0 OAS3' badge. Below this is the link '/swagger/v1/swagger.json'. A section titled 'Users' is expanded, showing a list of API endpoints: GET /api/Users, POST /api/Users, PUT /api/Users, GET /api/Users/{id}, and DELETE /api/Users/{id}. At the bottom, a 'Schemas' section is also expanded, showing a 'User' schema.

Swagger UI
Supported by SMARTBEAR

Select a definition **My API V1**

WebApplicationWebAPI02 ^{1.0} OAS3

/swagger/v1/swagger.json

Users

- GET** /api/Users
- POST** /api/Users
- PUT** /api/Users
- GET** /api/Users/{id}
- DELETE** /api/Users/{id}

Schemas

- User >

This screenshot shows the details for the 'GET /api/Users' endpoint. It includes a 'Parameters' section with 'No parameters' and a 'Responses' section. The response table shows a 200 status code for 'Success'. A 'Test' button is visible, and a dropdown menu is set to 'testplan'. An 'Example Value' is displayed as a JSON object: { "id": 1, "name": "John", "age": 30 }.

Swagger UI
localhost:44347/swagger/index.html

Users

GET /api/Users

Parameters: No parameters

Responses:

Code	Description	Links
200	Success	No links

Media type: testplan

Example Value: { "id": 1, "name": "John", "age": 30 }

This screenshot shows the details for the 'POST /api/Users' endpoint. It includes a 'Parameters' section with 'No parameters', a 'Request body' section set to 'Application/json', and a 'Responses' section. The response table shows a 200 status code for 'Success'. A 'Test' button is visible, and a dropdown menu is set to 'testplan'. An 'Example Value' is displayed as a JSON object: { "id": 1, "name": "John", "age": 30 }.

Swagger UI
localhost:44347/swagger/index.html

POST /api/Users

Parameters: No parameters

Request body: Application/json

Example Value: { "id": 1, "name": "John", "age": 30 }

Responses:

Code	Description	Links
200	Success	No links

Media type: testplan

Example Value: { "id": 1, "name": "John", "age": 30 }

This screenshot shows the 'Schemas' section of the Swagger UI. It displays the 'User' schema as a JSON object with the following properties: id (integer(\$int32)), name (string, nullable: true), and age (integer(\$int32)).

Schemas

```
User {
  id: integer($int32)
  name: string
  age: integer($int32)
}
```

Swagger annotations

XML comments

Decorate the model with attributes (Required, DefaultValue, Name, etc...)

Produces attribute to the API controller

```
/// <summary>
/// Students endpoint
/// </summary>
[Route("api/[controller]")]
[Produces("application/json")]
[ApiController]
References
public class StudentsController : ControllerBase
{
    #region [Get]

    /// <summary>
    /// Get list of students
    /// </summary>
    /// <remarks>
    /// Sample request:
    ///
    /// GET /Students
    /// [
    ///   "Student 1",
    ///   "Student 2"
    /// ]
    /// </remarks>
    /// <returns>List of students</returns>
    /// <response code="200">OK</response>
    [ProducesResponseType(200)]
    [HttpGet]
    References | 0 requests | 0 exceptions
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "value1", "value2" };
    }
}
```

Students

GET

/api/Students

Get list of students

Sample request:

```
GET /Students
[
  "Student 1",
  "Student 2"
]
```

Parameters

Try it out

No parameters

Responses

Response content type: application/json

Code	Description
200	<div>OK</div> <div>Example Value Model</div> <div><pre>["string"]</pre></div>

Swagger annotation configuration

Right-click the project in **Solution Explorer > Properties**
Build > Output > checked **Xml documentation file**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    /// Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });

        // Set the comments path for the Swagger JSON and UI.
        var xmlFile = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        var xmlPath = Path.Combine(AppContext.BaseDirectory, xmlFile);
        c.IncludeXmlComments(xmlPath);
    });
}
```

Action methods. Action Results in Web API

Action method Naming Conventions

Verb	Action Method Name	Usage
GET	Get() / get() / GET() / GetAllStudent() *any name starting with Get *	Retrieves data.
POST	Post() / post() / POST() / PostNewStudent() *any name starting with Post*	Inserts new record.
PUT	Put() / put() / PUT() / PutStudent() *any name starting with Put*	Updates existing record.
PATCH	Patch() / patch() / PATCH() / PatchStudent() *any name starting with Patch*	Updates record partially.

Name of the action methods in the Web API controller plays an important role. Action method name can be the same as HTTP verbs like Get, Post, Put, Patch or Delete as shown in the Web API Controller example above. However, you can append any **suffix** with HTTP verbs for more readability. For example, Get method can be **GetAllNames()**, **GetStudents()** or any other name which starts with **Get**.

Difference between Web API and MVC controller:

Web API Controller	MVC Controller
Derives from System.Web.Http.ApiController class	Derives from System.Web.Mvc.Controller class.
Method name must start with Http verbs otherwise apply http verbs attribute.	Must apply appropriate Http verbs attribute.
Specialized in returning data.	Specialized in rendering view.
Return data automatically formatted based on Accept-Type header attribute. Default to json or xml.	Returns ActionResult or any derived type.

Parameter Binding

Action methods in Web API controller can have one or more parameters of different types. It can be either **primitive** type or **complex** type. Web API binds action method parameters either with URL's query string or with request body depending on the parameter type. By default, if parameter type is of .NET **primitive** type such as int, bool, double, string, GUID, DateTime, decimal or any other type that can be converted from string type then it sets the value of a parameter from the query string. And if the parameter type is **complex** type then Web API tries to get the value from request body by default.

HTTP Method	Query String	Request Body
GET	Primitive Type, Complex Type	NA
POST	Primitive Type	Complex Type
PUT	Primitive Type	Complex Type
PATCH	Primitive Type	Complex Type
DELETE	Primitive Type, Complex Type	NA

Parameter Binding

POST http://localhost:49705/api/student/12

Authorization Headers (1) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary

Key	Value	Description
name	Vlad	
age	27	

0 references

```
public class StudentController : ApiController
{
    0 references
    public void Post(int id, Student student)
    {
        ≤ 50,864ms elapsed
    }
}
```

student {WebApiProject.Models.Student}

- Age "27"
- Name "Vlad"

1 reference

```
public class Student
{
    0 references
    public string Name { get; set; }

    0 references
    public string Age { get; set; }
}
```

[FromUri] and [FromBody]

Use **[FromUri]** attribute to force Web API to get the value of complex type from the query string and **[FromBody]** attribute to get the value of primitive type from the request body, opposite to the default rules.

```
0 references
public class StudentController : ApiController
{
    0 references
    public Student Get([FromUri]Student stud)
    {
        return stud;
    }

    0 references
    public void Post([FromBody]int id, [FromUri]Student student)
    {
    }
}
```

GET ▼ http://localhost:49705/api/student?name=Vlad&age=27 Params Send Save ▼

Authorization Headers (1) Body Pre-request Script Tests Code

Type No Auth ▼

POST ▼ http://localhost:49705/api/student?name=Vlad&age=27 Params Send Save ▼

Authorization Headers (1) **Body** Pre-request Script Tests Code

☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> id	12			

Action Method Return Type

The Web API action method can have following return types:

- Void
- Primitive type or Complex type
- HttpResponseMessage
- IHttpActionResult

0 references

```
public void Delete(int id)
{
    DeleteStudentFromDB(id);
}
```

0 references

```
public int GetId(string name)
{
    int id = GetStudentId(name);

    return id;
}
```

0 references

```
public Student GetStudent(int id)
{
    var student = GetStudentFromDB(id);

    return student;
}
```

HttpResponseMessage

Web API controller always returns an object of **HttpResponseMessage** to the hosting infrastructure.

The advantage of sending **HttpResponseMessage** from an action method is that you can configure a response your way. You can set the status code, content or error message (if any) as per your requirement.

```
0 references
public HttpResponseMessage Get(int id)
{
    Student stud = GetStudentFromDB(id);

    if (stud == null)
    {
        return Request.CreateResponse(HttpStatusCode.NotFound, id);
    }

    return Request.CreateResponse(HttpStatusCode.OK, stud);
}
```


IHttpRequestResult

The **IHttpRequestResult** was introduced in Web API 2 (.NET 4.5). An action method in Web API 2 can return an implementation of **IHttpRequestResult** class which is more or less similar to **ActionResult** class in ASP.NET MVC.

0 references

```
public IHttpActionResult Get(int id)
{
    Student stud = GetStudentFromDB(id);

    if (stud == null)
    {
        return NotFound();
    }

    return Ok(stud);
}
```

ApiController Method	Description
BadRequest()	Creates a BadRequestResult object with status code 400.
Conflict()	Creates a ConflictResult object with status code 409.
Content()	Creates a NegotiatedContentResult with the specified status code and data.
Created()	Creates a CreatedNegotiatedContentResult with status code 201 Created.
CreatedAtRoute()	Creates a CreatedAtRouteNegotiatedContentResult with status code 201 created.
InternalServerError()	Creates an InternalServerErrorResult with status code 500 Internal server error.
NotFound()	Creates a NotFoundResult with status code 404.
Ok()	Creates an OkResult with status code 200.
Redirect()	Creates a RedirectResult with status code 302.
RedirectToRoute()	Creates a RedirectToRouteResult with status code 302.
ResponseMessage()	Creates a ResponseMessageResult with the specified HttpResponseMessage.
StatusCode()	Creates a StatusCodeResult with the specified http status code.
Unauthorized()	Creates an UnauthorizedResult with status code 401.

Web API Request/Response Data Formats

Media type (aka MIME type) specifies the format of the data as type/subtype e.g. text/html, text/xml, application/json, image/jpeg etc.

In HTTP request, MIME type is specified in the request header using **Accept** and **Content-Type** attribute. The **Accept** header attribute specifies the format of response data which the client expects and the **Content-Type** header attribute specifies the format of the data in the request body so that receiver can parse it into appropriate format.

HTTP GET Request Example:

```
GET http://localhost:60464/api/student HTTP/1.1
User-Agent: Fiddler
Host: localhost:1234
Accept: application/json
```

HTTP POST Request Example:

```
POST http://localhost:60464/api/student?age=15 HTTP/1.1
User-Agent: Fiddler
Host: localhost:60464
Content-Type: application/json
Content-Length: 13
```

```
{
  id:1,
  name: 'Vlad'
}
```

Web API Request/Response Data Formats

Web API converts request data into CLR object and also serialize CLR object into response data based on **Accept** and **Content-Type** headers. Web API includes built-in support for **JSON, XML, BSON, and form-urlencoded data**. It means it automatically converts request/response data into these formats OOB out-of-the box.

GET Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Accept	application/json				
New key	Value	Description			

Body Cookies Headers (10) Test Results Status: 200 OK Time: 24 ms

Pretty Raw Preview JSON

```
1 {
2   "Name": "Vlad",
3   "Age": "27"
4 }
```

GET Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Accept	application/xml				
New key	Value	Description			

Body Cookies Headers (10) Test Results Status: 200 OK Time: 20 ms

Pretty Raw Preview XML

```
1 <Student xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org
2   /2004/07/WebApiProject.Models">
3   <Age>27</Age>
4   <Name>Vlad</Name>
5 </Student>
```

Media-Type Formatters

Media type formatters are classes responsible for serializing **request/response** data so that Web API can understand the request data format and send data in the format which client expects.

Web API includes following built-in media type formatters:

Media Type Formatter Class	MIME Type	Description
JsonMediaTypeFormatter	application/json, text/json	Handles JSON format
XmlMediaTypeFormatter	application/xml, text/json	Handles XML format
FormUrlEncodedMediaTypeFormatter	application/x-www-form-urlencoded	Handles HTML form URL-encoded data
JQueryMvcFormUrlEncodedFormatter	application/x-www-form-urlencoded	Handles model-bound HTML form URL-encoded data

.NET Online UA Training Course Feedback

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report on to the electronic address:

Oleksii_Leunenko@epam.com

With the note [.NET Online UA Training Course Feedback]

Thank you.

Q&A



DRIVEN



CANDID



CREATIVE



ORIGINAL



INTELLIGENT



EXPERT

UA .NET Online LAB