

# ЛЕКЦИЯ

## ASP.Net

### Web Api

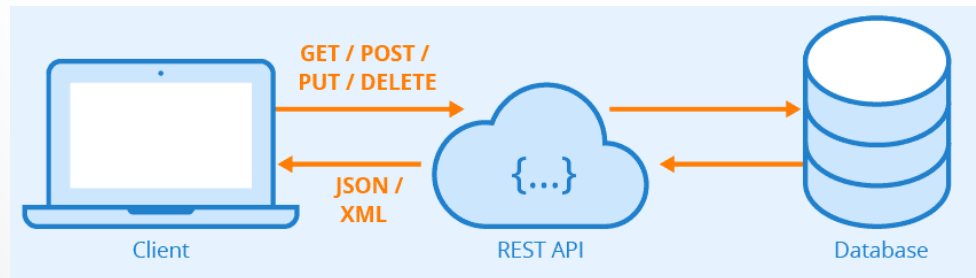
Лектор Крамар Ю.М.

# 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.



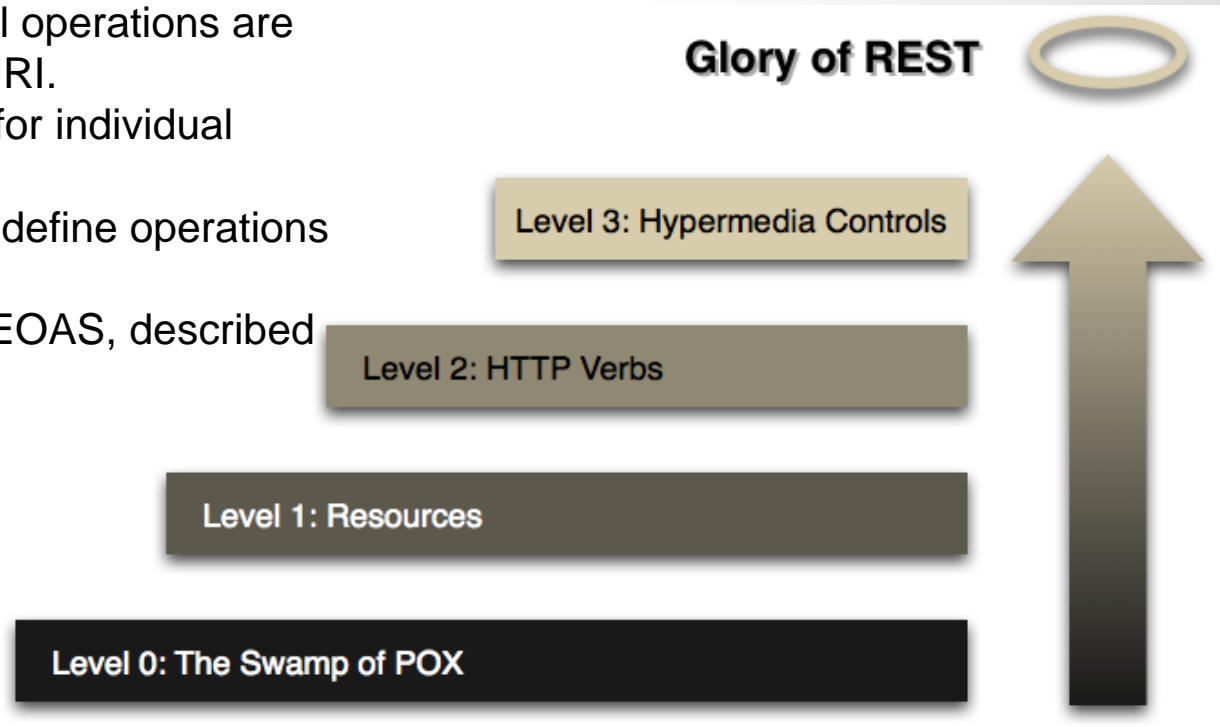
# REST, RESTful

## 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)

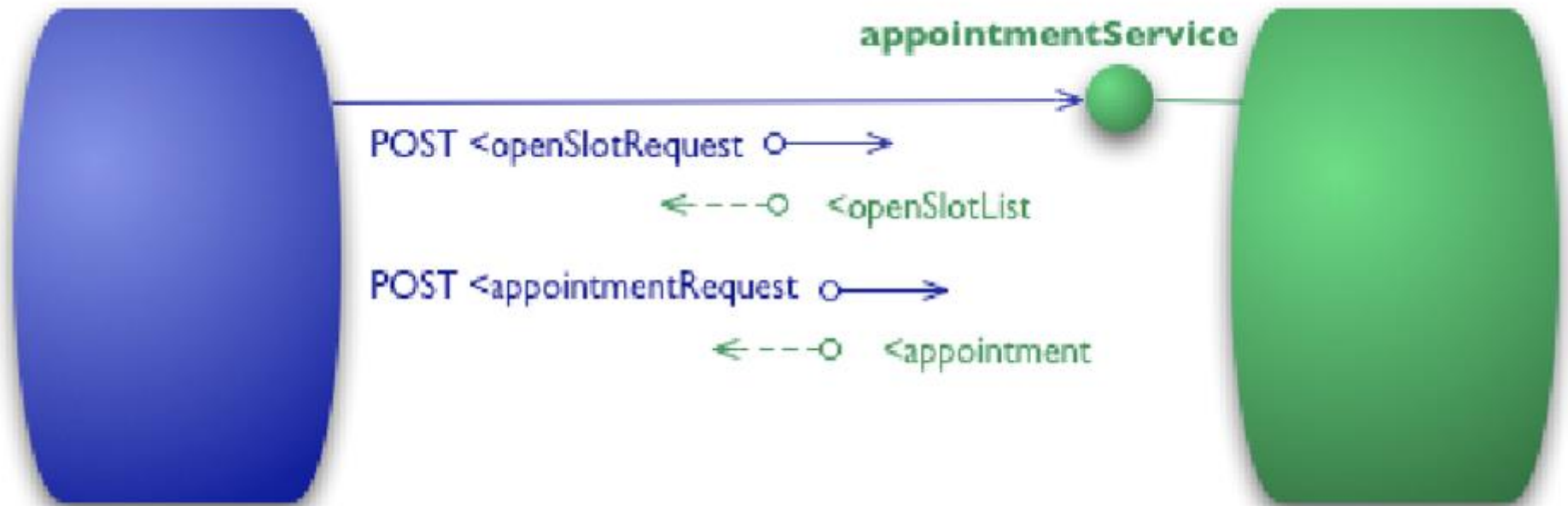
# REST, RESTful: Richardson Maturity Model

- 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).

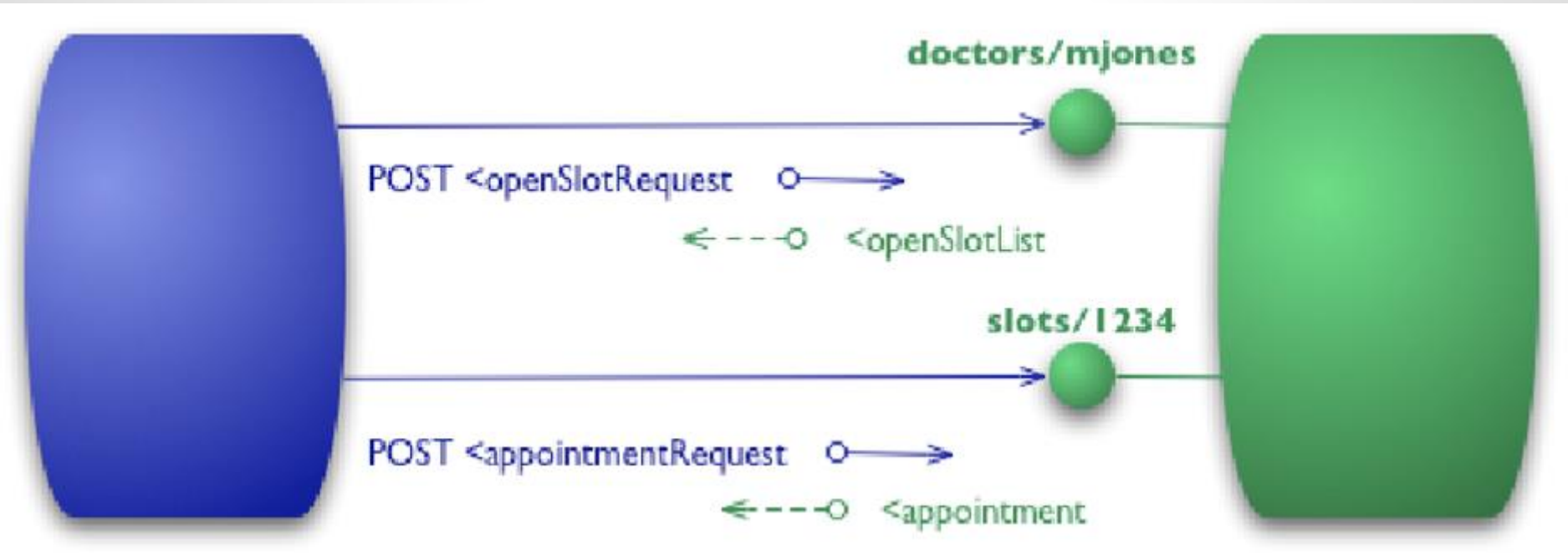


<https://martinfowler.com/articles/richardsonMaturityModel.html>

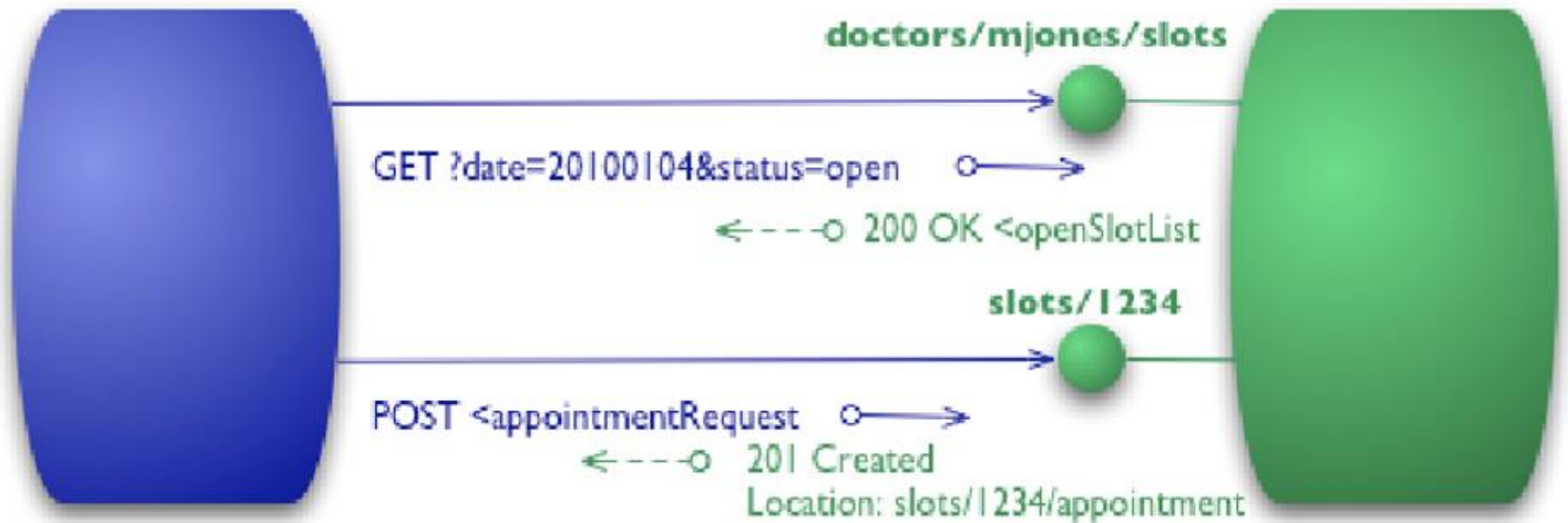
## REST, RESTful: Richardson Maturity Model – level 0



## REST, RESTful: Richardson Maturity Model – level 1

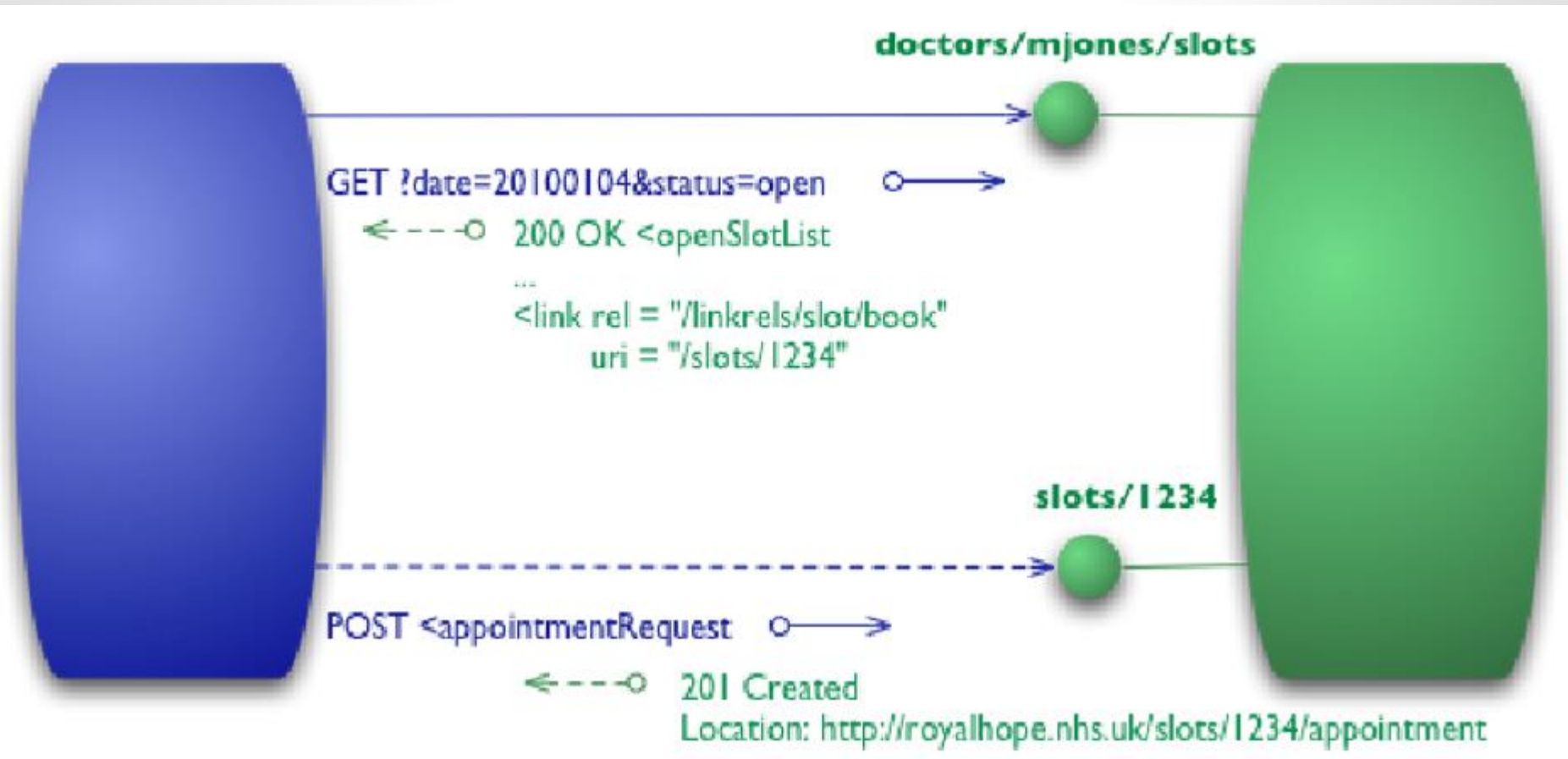


## REST, RESTful: Richardson Maturity Model – level 2





## REST, RESTful: Richardson Maturity Model – level 3





# 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>
```

## 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`

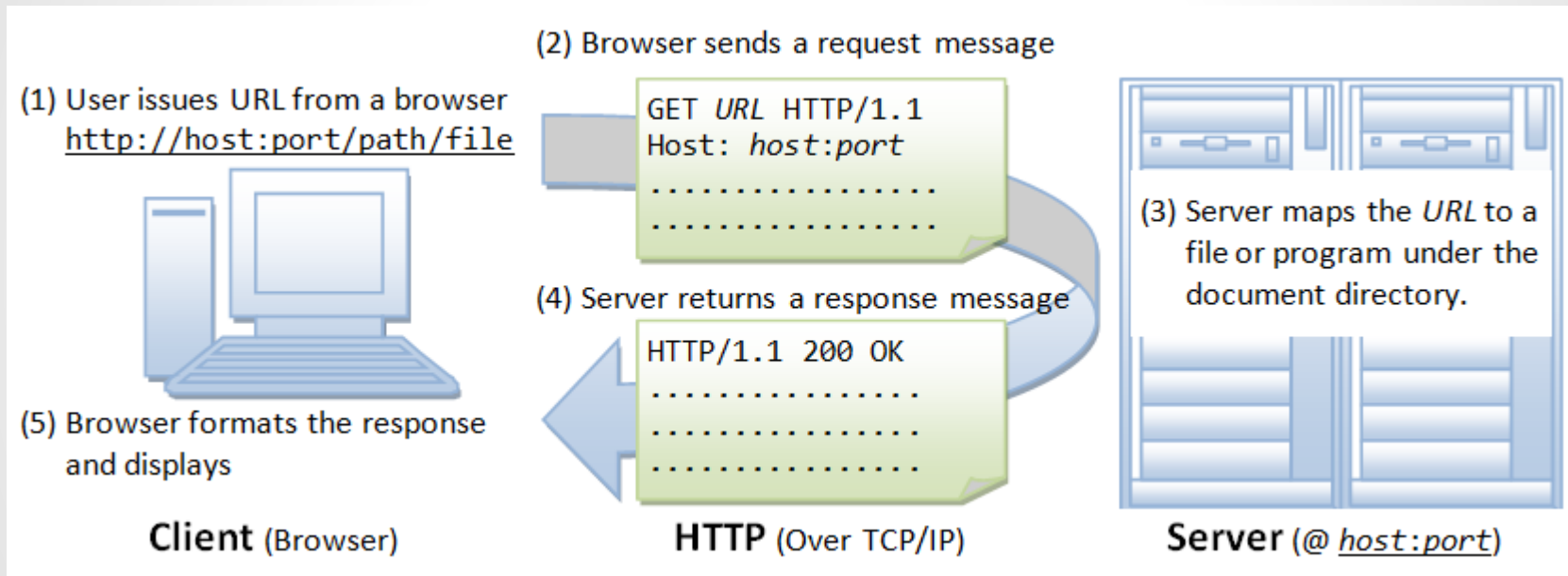
## REST, RESTful: Links

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

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



# HTTP request-response



CONFIDENTIAL

# 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)
    {
    }
}
```

# Postman

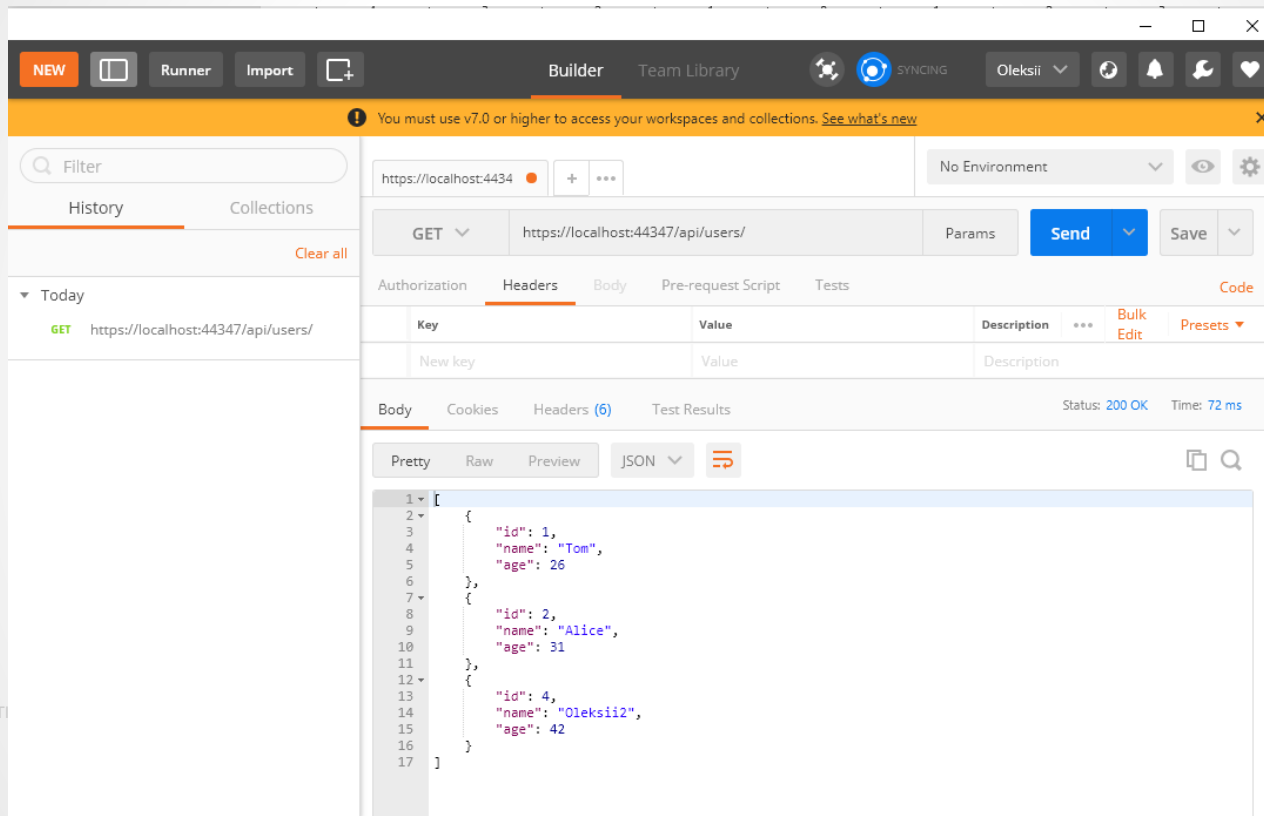


POSTMAN

CONFIDENTIAL

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

# Postman



# 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 title bar indicates the URL is `localhost:44347/swagger/index.html`. The Swagger logo and 'Supported by SMARTBEAR' are in the top left. A dropdown menu shows 'Select a definition' with 'My API V1' selected. The API title is 'WebApplicationWebAPI02' with a '1.0 OAS3' badge. Below the title is the link '/swagger/v1/swagger.json'. A section titled 'Users' contains five 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 shows a 'User' schema with a right-pointing arrow.

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 with a 'Success' description and 'No links'. A 'Try it out' button is visible. Below the response table, there is a 'Media type' dropdown set to 'text/plain', a 'Controls: accept header' section, and an 'Example Value | Schema' section with a JSON example: `{ "id": 1, "name": "string", "age": 1 }`.

Swagger UI

GET /api/Users

Parameters: No parameters

Responses

Code	Description	Links
200	Success	No links

Media type: text/plain

Controls: accept header

Example Value | Schema

```
{ "id": 1, "name": "string", "age": 1 }
```

This screenshot shows the details for the POST /api/Users endpoint. It includes a 'Parameters' section with 'No parameters' and a 'Request body' section set to 'application/json'. There is a 'Try it out' button. Below, the 'Responses' section shows a 200 status code with a 'Success' description and 'No links'. A 'Media type' dropdown is set to 'text/plain', followed by 'Controls: accept header' and an 'Example Value | Schema' section with a JSON example: `{ "id": 1, "name": "string", "age": 1 }`.

Swagger UI

POST /api/Users

Parameters: No parameters

Request body: application/json

Try it out

Example Value | Schema

```
{ "id": 1, "name": "string", "age": 1 }
```

Responses

Code	Description	Links
200	Success	No links

Media type: text/plain

Controls: accept header

Example Value | Schema

```
{ "id": 1, "name": "string", "age": 1 }
```

This screenshot shows the 'Schemas' section of the Swagger UI. It contains a single schema named 'User' with a dropdown arrow. The schema definition is as follows:

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

- ▶ **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

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
<input checked="" type="checkbox"/> name	Vlad	
<input checked="" type="checkbox"/> age	27	

... Bulk Edit

```
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; }
}
```

# [FromBody] and [FromUri]

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

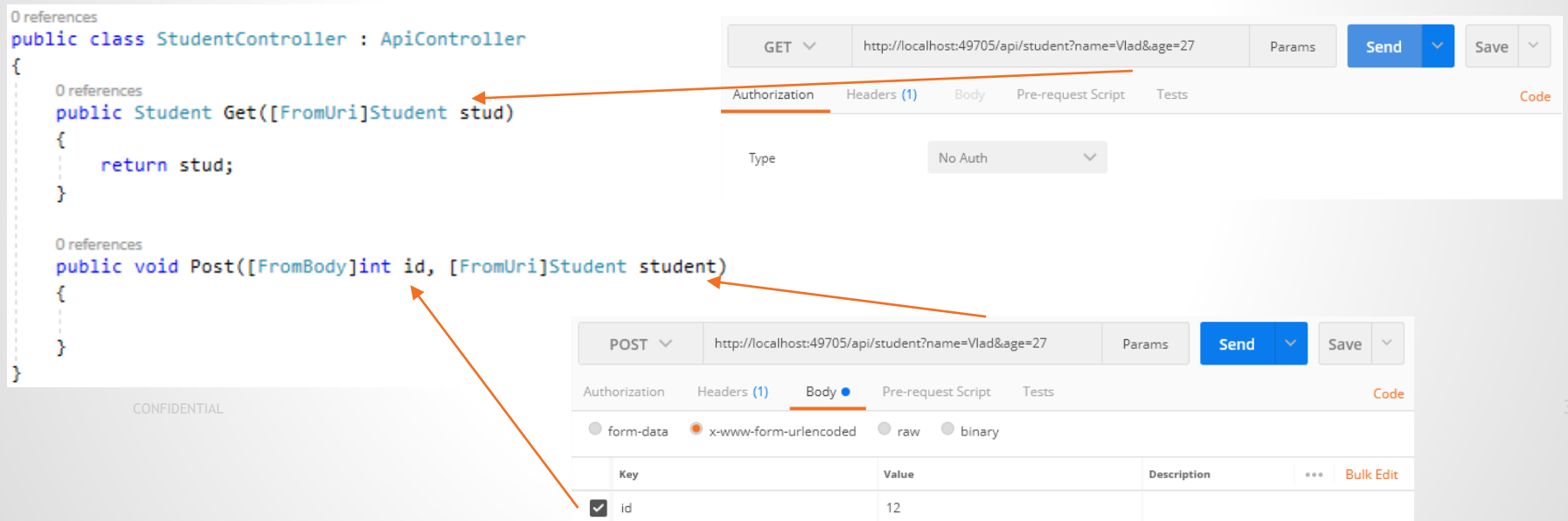
Type No Auth ▼

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			



CONFIDENTIAL

# Action Method Return Type

The Web API action method can have following return types:

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

CONFIDENTIAL

```
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);
}
```

# IHttpActionResult

The **IHttpActionResult** was introduced in Web API 2 (.NET 4.5). An action method in Web API 2 can return an implementation of **IHttpActionResult** 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'
}
```

# 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

## ASP.Net Web Api

### **Ресурсы:**

<https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0>

<https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-6.0>

[Создание приложения Web API в ASP.NET](#)

[Создание десктопного клиента на C# для ASP.NET Web API](#)

## ASP.Net Web Api

### **Ресурсы:**

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

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

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