Rest Api: REpresentational State Transfer

# Creating a simple REST API in PHP - ShareurCodes

**Http: Hyper Text Transfer Protocol:**

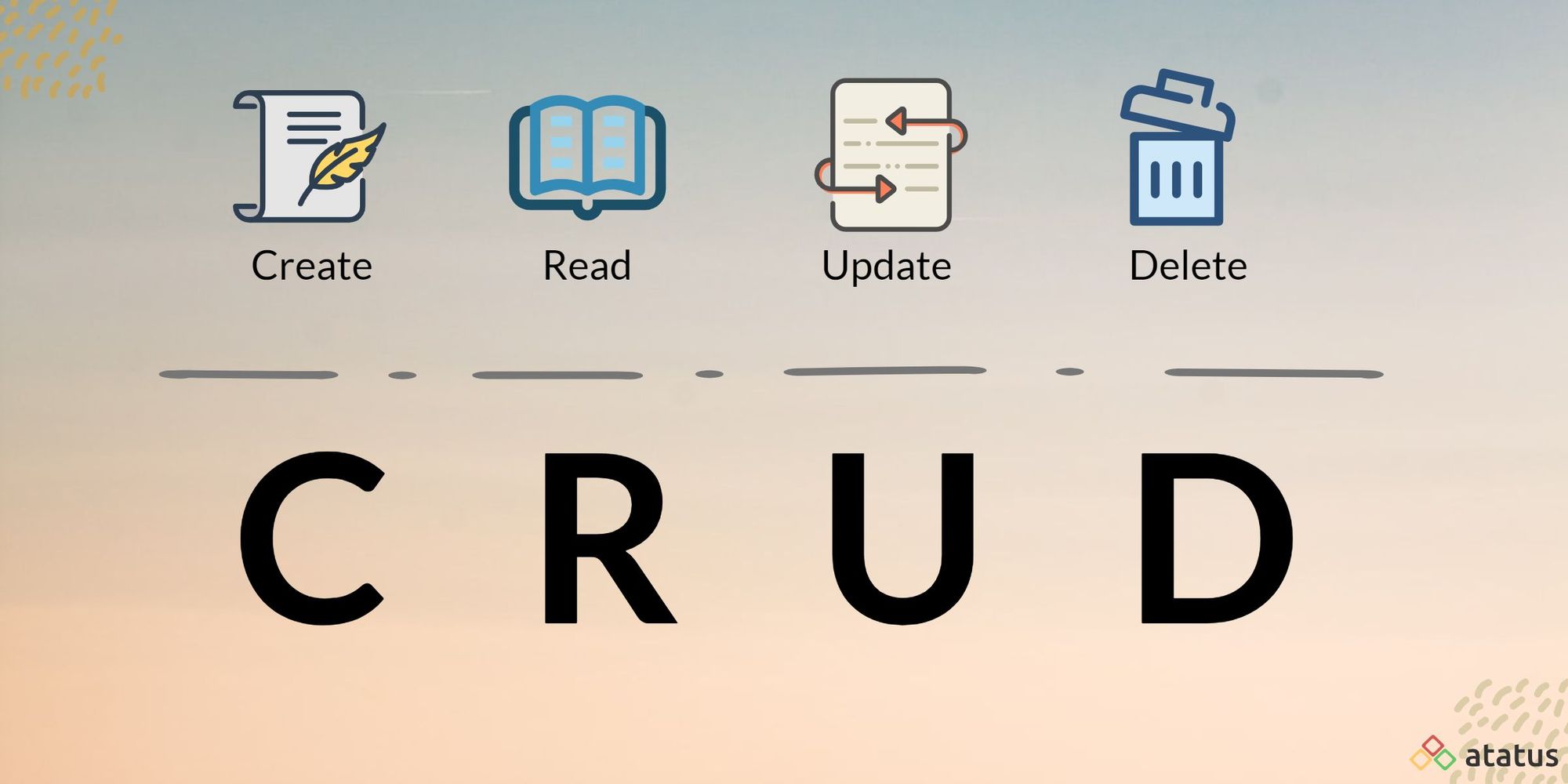
Established in 1991, HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. A typical flow over HTTP involves a client machine making a request to a server, which then sends a response message.

**What is in an HTTP request?**

An HTTP request is the way Internet communications platforms such as web browsers ask for the information they need to load a website.

Each HTTP request made across the Internet carries with it a series of encoded data that carries different types of information. A typical HTTP request contains:

* HTTP version type
* URL
* HTTP method:
  + Get
  + Post
  + Put/Patch
  + Delete
* HTTP request headers
* Optional HTTP body



**Constraints:**

1. **Uniform interface:**

* Standards – Uniform Resource Identifier (URI). A URI is the location of a specific web resource. A URI can be made up of several components, including the host name or IP address, port number, the path, and an optional query string.
* Resources - Text, JSON, XML
* Representations - a client and server exchange representations of a resource.

1. **Client Server**

Client and Server should be able to evolve independent of each other. As long as the

interface doesn’t change, the client shouldn’t have to care about changes on the server.

1. **Stateless**

The server should treat each request separately. No request should depend on previous requests. There should not be any client data cached on the server. The client is responsible for handling the state of the application.

1. **Layered**

REST allows you to build distributed systems. There could be load balancers,

authentication servers, databases on different machines. These layers shouldn’t affect

your client. The client should only have to send a request to one single endpoint.

**What’s an HTTP status code?**

HTTP status codes are 3-digit codes most often used to indicate whether an HTTP request has been successfully completed. Status codes are broken into the following 5 blocks:

* 1xx Informational
* 2xx Success
* 3xx Redirection
* 4xx Client Error
* 5xx Server Error

## [Good link for Codes](https://restfulapi.net/http-status-codes/)

## **ControllerBase class:**

[ApiController]

[Route("[controller]")]

public class WeatherForecastController : ControllerBase

Attributes

The Microsoft.AspNetCore.Mvc namespace provides attributes that can be used to configure the behavior of web API controllers and action methods. The following example uses attributes to specify the supported HTTP action verb and any known HTTP status codes that could be returned:

[HttpPost]

[ProducesResponseType(StatusCodes.Status201Created)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

public ActionResult<Pet> Create(Pet pet)

{

pet.Id = \_petsInMemoryStore.Any() ?

\_petsInMemoryStore.Max(p => p.Id) + 1 : 1;

\_petsInMemoryStore.Add(pet);

return CreatedAtAction(nameof(GetById), new { id = pet.Id }, pet);

}

Here are some more examples of attributes that are available:

[Route] - Specifies URL pattern for a controller or action.

[HttpGet] Identifies an action that supports the HTTP GET action verb.

[FromBody] Populates a parameter and its properties from the body of an HTTP request

[HttpGet("{id}/{name}")]

public async Task<string> GetFromId(int id, string name)

{

return "Id: " + id + " name: " + name;

}

[HttpGet("{id}")]

public async Task<string> GetFromId(int id, [FromQuery] string name)

{

return "Id: " + id + " name: " + name;

}

[HttpPost()]

[Route("GetFromBody")]

public async Task<string> Save(int id, [FromBody] string name)

{

return "Id: " + id + " name: " + name;

}

Types of Web API Action Results

Let’s see how we are going to handle returned types in Web API. Web API provides a facility to return any of the following kinds of Action Results.

1. Void
2. Any Entity/Datatype
3. IHttpActionResult

Void

Void means it doesn’t return anything. So, let’s see how it works on Web API action method.

Following action method returns void.

[HttpGet]

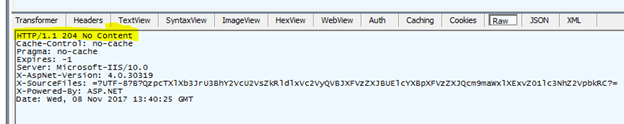
public void LogMessageinDB(string Message)

{

new UserProfileDB().LogMessageinDB(Message);

}

The response of the service.

****

We won’t get the data from the service and the status will always be 204 - no content.

Entity/Datatype

Web API action method can contain any entity type as return type. We can use any entity type as the return type. But the problem here is that we get 200 (OK) status code every time on a successful response.

[HttpGet]

[Route("UserProfile/GetUserProfile/{UserId}")]

public UserProfile GetUserProfile(int UserId)

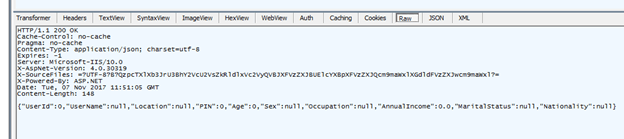
{

return new UserProfileDB().GetUserProfile(UserId);

}

Following is the result of the above service call in Fiddler.

*Result in the Fiddler*

******

When you observe the above picture, the status code is 200 and the result is in JSON format at the bottom.

When we have to use this kind of action result

It is not recommended to use because of two reasons -

1. We cannot explain/convey the client with the various status codes as we get 200 (OK) status code every time.
2. There is no guarantee that the above service will return the required result at every time. There could be an issue with the data or the DB Server or anything can happen. In these situations, the consumer won’t get the data and will get an error in response, i.e., the status code 500 always. Moreover, the client receives complete stack trace error message.

To overcome these problems, Web API provided Action results, like IHttpActionResult.

[HttpGet("{id}")]

[ProducesResponseType(StatusCodes.Status200OK, Type = typeof(Product))]

[ProducesResponseType(StatusCodes.Status404NotFound)]

public IActionResult GetById\_IActionResult(int id)

{

var product = \_productContext.Products.Find(id);

return product == null ? NotFound() : Ok(product);

}

public IActionResult SelectAllCustomers() {

var model = customerDetailsRepository.FindAll("SelectAllCustomers").ToList();

return Ok(model);

}

Common Response Types:

NotFound()

OK()

NoContent()

UnAuthorized()

ServerError()



**תרגיל:**

יש לייצר controller שמאפשר חיפוש של קבצים ותיקיות בדיסק של השרת.

**Get:**

שתי הפונקציות מקבלות שם של תיקייה - אחת מחזירה רשימה של String עם שמות התיקיות שבתוכה.

השנייה מחזירה רשימה של קבצים בתיקייה.

**עבור כל קובץ יש להחזיר:**

א. שם

ב. סיומת

ג. תאריך שינוי אחרון

ד. גודל

יש ליצור פונקציה נוספת עבור Delete שמקבלת שם מלא של קובץ ומוחקת אותו מהדיסק.

**[ProducesResponseType(200, Type = typeof(List<FileDetails>))]**

**[HttpGet]**

**[Route("GetDirs/{dirName}")]**

**public async Task<IActionResult> GetFilesByDirectory(string dirName)**

**[HttpGet]**

**[Route("GetFiles")]**

**[ProducesResponseType(StatusCodes.Status200OK,Type typeof(List<MyFile>))]**

**[ProducesResponseType(StatusCodes.Status404NotFound)]**

**public IActionResult GetFiles(string directory)**

**[HttpDelete]**

**[Route("DeleteFile")]**

**[ProducesResponseType(StatusCodes.Status200OK)]**

**[ProducesResponseType(StatusCodes.Status404NotFound)]**

**public IActionResult DeleteFile(string fileName)**

2. שנו את המימוש של חיפוש הקבצים כך שכל הלוגיקה תשב בתוך שירות נפרד ולא בController (יש לייצר Interface וClass מממש בהתאם).

3. צרו מודל של מוצר ולתמוך בכל פעולות הcrud.המידע ישב בservice נפרד.

למוצר תכונות של Id, name, price

Working with models services and contracts:

Api logics shouldn’t reside in the controller, but in repository service.

1. Add 3 Folders:

a. Contracts

b. Models

c. Services

2. In the Model Folder Add new record with the following properties:

Int Id, string Name, double Price.

3. Add Product controller with the following async Capabilities:

• List<Product> GetAllProducts()

• Product GetProductById(int id)

• Product AddNewProduct(Product p)

4. In Contracts folder add new interface IProductRepository that supports these actions asynchonously.

5. In service folder add MockForCourseProductRepository that implements IProductRepository and holds a private member of List<Products>

6. Add the Mock as a singleton service

Add Delete and Update Methods

Change the directory Assignments so that all Logics is in Class using Interface.

Add the following Capability:

In the program.cs add user and password header:

app.Use(async (context, next) =>

{

context.Request.Headers.Add("UserName", "Tally");

context.Request.Headers.Add("Password", "12345");

await next.Invoke();

});

In the controller each request must check that these header entries are present with Values. Failing to do so return UnAuthorized (401)

# Adding options to my project:

Almost every project will have some settings that need to be configured and changed depending on the environment, or secrets that you don't want to hard code into your repository. The classic example is connection strings and passwords etc.

Application settings can be stored in multiple places - environment variables, appsettings.json, user secrets.

**Strongly typed configuration (IOptions):**

In ASP.NET Core, there is now no default AppSettings["MySettingKey"] way to get settings. Instead, the recommended approach is to create a strongly typed configuration class with a structure that matches a section in your configuration file (or wherever your configuration is being loaded from):

public class UnitOptions

{

public string Temp { get; set; } = String.Empty;

public string Distance { get; set; } = String.Empty;

}

Would map to the lower section in the appsettings.json below.

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Debug",

"System": "Information",

"Microsoft": "Information"

}

},

"Units": {

"Temp": "Celsius",

"Distance": "Miles"

}}

Bind the UnitOptions class to the corresponding section in appsettings.json by registering configuration instance in Program.cs

builder.Services.Configure<UnitOptions>(builder.Configuration.GetSection("Units"));

**IOptions** is singleton and hence can be used to read configuration data within any service lifetime. Being singleton, it cannot read changes to the configuration data after the app has started.

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

[ApiController]

public class OptionsDemoController : ControllerBase

{

private readonly ITransientService \_transientService;

public OptionsDemoController(TransientService transientService)

{

\_transientService = transientService;

}

[HttpGet]

[Route("/units/transient")]

public IActionResult GetUnitsTransient() => Ok(\_transientService.GetUnits());

}

public interface ITransientService

{

UnitOptions GetUnits();

}

public class TransientService : ITransientService

{

private readonly UnitOptions \_unitOptions;

public TransientService(IOptions<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions.Value;

}

public UnitOptions GetUnits()

{

return \_unitOptions;

}

}

IOptionsSnapshot is scoped and hence it can be used only with transient and scoped service lifetimes. Being scoped, it can recompute config data for each request.

public interface IScopedService

{

UnitOptions GetUnits();

}

public class ScopedService : IScopedService

{

private readonly UnitOptions \_unitOptions;

public ScopedService(IOptionsSnapshot<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions.Value;

}

public UnitOptions GetUnits()

{

return \_unitOptions;

}

}

While the app is still running, change the value of distance unit from ‘Miles’ to ‘Kilometres’ in the appsettings.json file and hit the same API controller action again. The response reflects the changes to the config data.

**IOptionsMonitor**

IOptionsMonitor is singleton and hence can be used to read configuration data in any service lifetime. However, as opposed to IOptions, it can retrieve current config data at any time.

Create a singleton service with an injected IOptionsMonitor instance as follows:-

public interface ISingletonService

{

UnitOptions GetUnits();

}

public class SingletonService : ISingletonService

{

private readonly IOptionsMonitor<UnitOptions> \_unitOptions;

public SingletonService(IOptionsMonitor<UnitOptions> unitOptions)

{

\_unitOptions = unitOptions;

}

public UnitOptions GetUnits()

{

return \_unitOptions.CurrentValue;

}

}

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

[ApiController]

public class OptionsDemoController : ControllerBase

{

private readonly ITransientService \_transientService;

private readonly IScopedService \_scopedService;

private readonly ISingletonService \_singletonService;

public OptionsDemoController(ITransientService transientService, IScopedService scopedService, ISingletonService singletonService)

{

\_transientService = transientService;

\_scopedService = scopedService;

\_singletonService = singletonService;

}

[HttpGet]

[Route("/units/transient")]

public IActionResult GetUnitsTransient() => Ok(\_transientService.GetUnits());

[HttpGet]

[Route("/units/scoped")]

public IActionResult GetUnitsScoped() => Ok(\_scopedService.GetUnits());

[HttpGet]

[Route("/units/singleton")]

public IActionResult GetUnitsSingleton() => Ok(\_singletonService.GetUnits());

}

**Environment variables:**

Environment variables are used to avoid storage of app secrets in code or in local configuration files. Environment variables override configuration values for all previously specified configuration sources.

**Environment.GetEnvironmentVariable("envVar");**

**Secret Manager:**

The Secret Manager tool stores sensitive data during the development of an ASP.NET Core project. In this context, a piece of sensitive data is an app secret. App secrets are stored in a separate location from the project tree. The app secrets are associated with a specific project or shared across several projects. The app secrets aren't checked into source control.

Use of nuget: Microsoft.Extensions.Configuration.UserSecrets

Generate Your User Secrets File

To generate your user secrets file, right-click on the common/config project (whichever utilizes connection strings) and select Manage User Secrets. A file named secrets.json should be opened. Any configuration values you want to store for local use should be stored here.

In file add key value pairs: "MySecret" : "12345"

builder.Configuration.AddUserSecrets(Assembly.GetExecutingAssembly(), true);

Access: builder.Configuration["MySecret"];