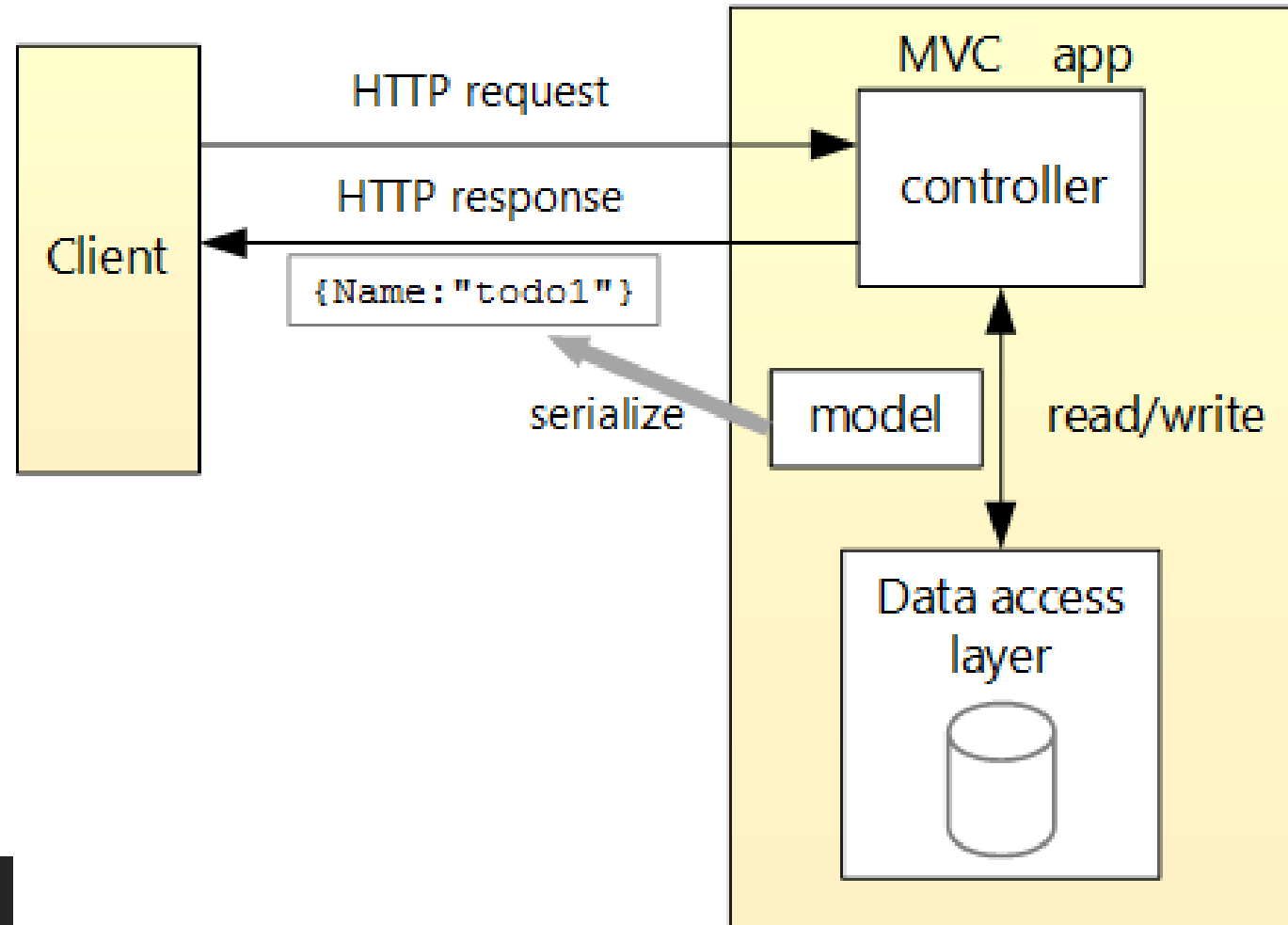# REST Fundamentals

.NET CORE

In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs). The resources are acted upon by using a set of simple, well-defined operations. The clients and servers exchange representations of resources by using a standardized interface and protocol – typically HTTP.

HTTPS://RESTFULAPI.NET/

REST API Tutorial

MVC   app

HTTP request → controller

HTTP response ←

{Name:"todo1"}

Client

serialize

model

read/write

Data access layer

Rest – Overview
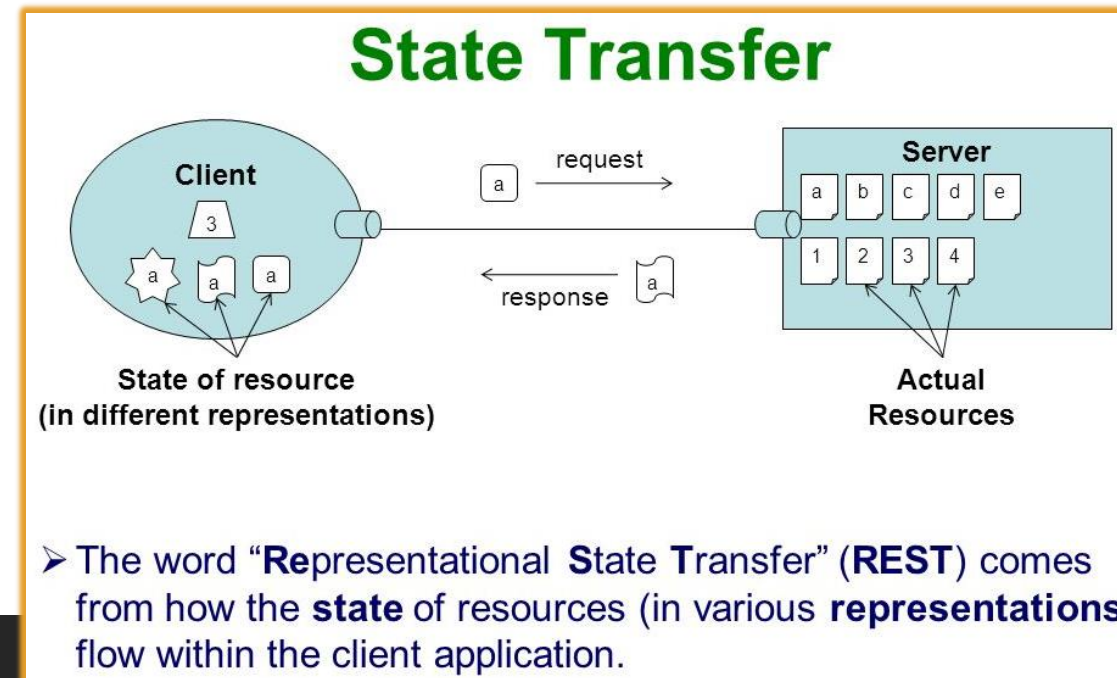
Roy Fielding proposed *Representational State Transfer (REST)* as an architectural approach to designing web services in 2000. *REST* uses open standards.
A *REST* web service could be written in ASP.NET and client applications can use any language or toolset that can generate *HTTP* requests and parse *HTTP* responses.

REST APIs are designed around *resources*.
Resources are any kind of object, data, or service that can be accessed by the client. *Resources* have identifiers. Revature.com/associates/23 is the identifier for a resource. *Routing* is used to direct the request to the correct resource.
Clients interact with a service by exchanging representations of resources (*JSON*).

State Transfer

Client

Server

request

response

State of resource
(in different representations)

Actual Resources

➢ The word "**Re**presentational **S**tate **T**ransfer" (**REST**) comes from how the **state** of resources (in various **representations** flow within the client application.

Rest – Overview
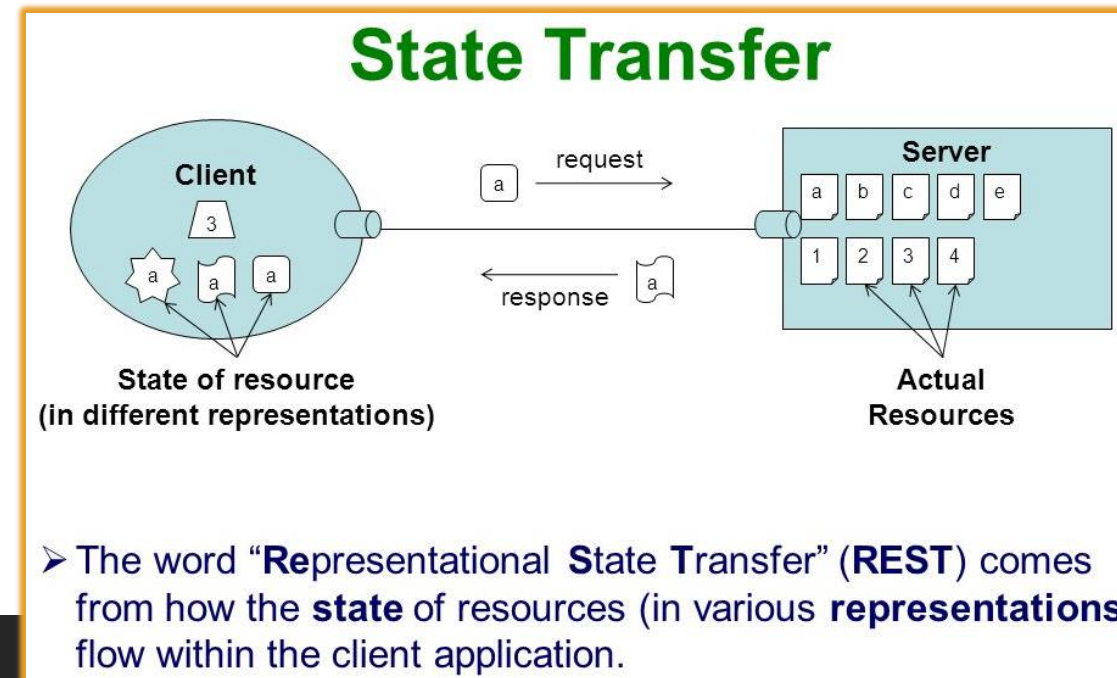
REST APIs use a uniform interface that allows decoupling of client and service applications. They agree on data transfer format (XML, JSON, JPG) and standard HTTP verbs like GET, POST, DELETE, etc.

The state of an application is not remembered between requests in *REST*.

The only place where any data is stored in the permanent resources (DB) of the application.
This means that any request can be made through any server by any client.
Each request must be atomic. It can't depend on any other request and can only read the results of other requests.

State Transfer

Client
3
a   a

State of resource
(in different representations)

request
a

response
a

Server
a b c d e
1 2 3 4

Actual
Resources

➤ The word "**Re**presentational **S**tate **T**ransfer" (**REST**) comes from how the **state** of resources (in various **representations**) flow within the client application.

REST – Guiding Principles

Client-Server	Stateless	Cacheable	Uniform Interface	Layered System	Code on Demand
Loose Coupling allows portability of the UI and scalability	Each request must be atomic.	Responses are labeled as Cacheable or non-cacheable. If, Cacheable, the client can reuse the data.	Interface Generality. Interfaces have 4 constraints – ID of resources, manipulation of resources through representations, self-descriptive messages, HATEOAS.	Hierarchical architecture. Each component can't see beyond the layer with which it is interacting.	REST allows extension of functionality by the downloading of scripts.

If you are honoring the 6 guiding principles of REST, you can call your interface RESTful.

REST – HTTP Methods

_____

HTTP Methods assign semantic meaning to a request. The effect of a specific HTTP Method used in a request should depend on whether the resource is a collection or an individual item. Here are the common conventions adopted by many RESTful implementations

Roy Fielding has never mentioned any rules around which method to use in which condition. The interface just must be uniform. If you decide HTTP POST will be used for updating a resource, you just have to be consistent within the application.

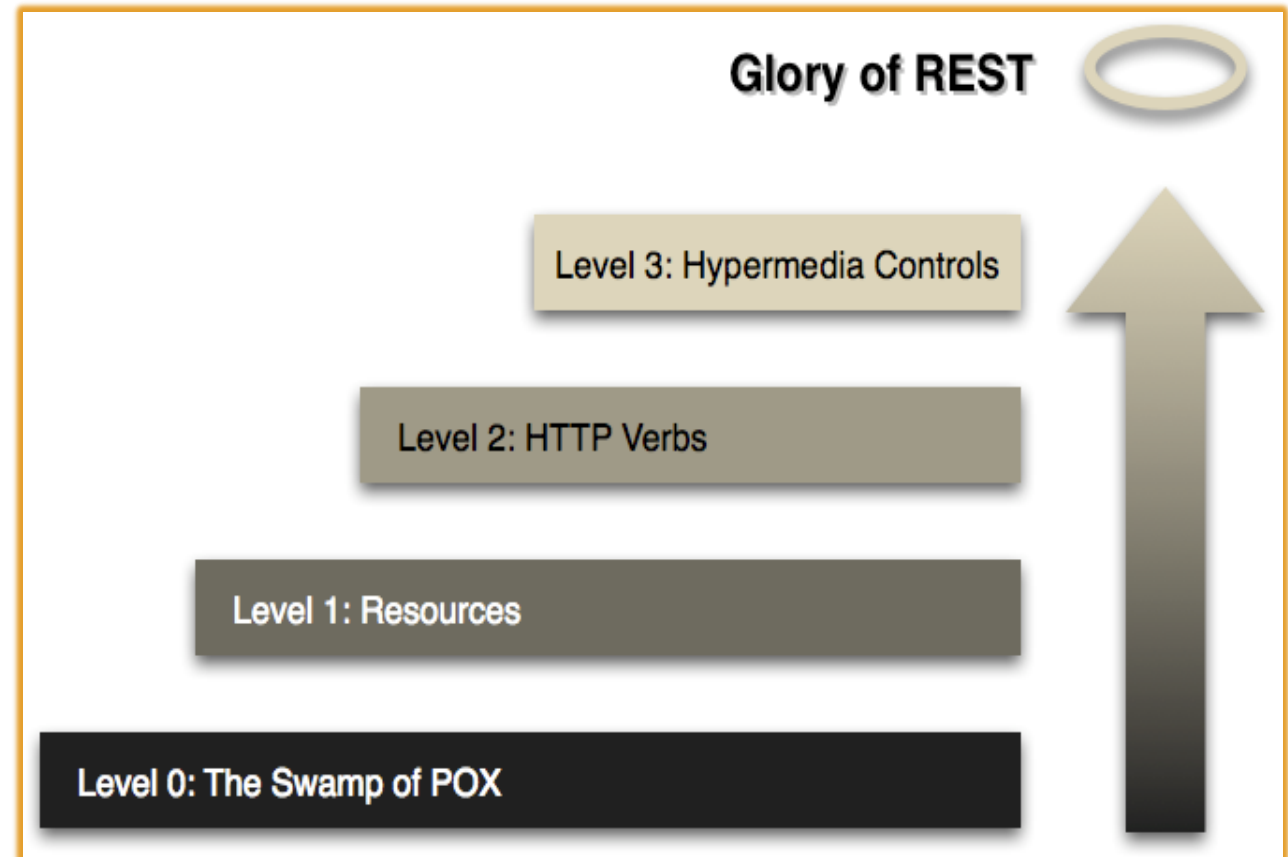| Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer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

Rest – Levels

In 2008, Leonard Richardson proposed the following maturity model for web APIs:

- 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*).

Level 3 is considered truly *RESTful* but most sites fall at Level 2.

Glory of REST

Level 3: Hypermedia Controls

Level 2: HTTP Verbs

Level 1: Resources

Level 0: The Swamp of POX

HATEOAS –
Hypertext as the Engine of Application State
https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#use-hateoas-to-enable-navigation-to-related-resources

Each HTTP GET request should return the information necessary to find the resources related directly to the requested object through hyperlinks <u>included</u> in the response. It should also be provided with information that describes the operations available on each of these resources.

HATEOAS creates a Finite State Machine. The response to a request contains the information necessary to move between states.

Currently there are no standards or specifications that define how to model the *HATEOAS* principle.

```json
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://adventure-works.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://adventure-works.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

Filter and Paginate data

Avoid fetching large amounts of data when only a subset of the information is required.

Filtering - The API can allow passing a filter in the query string of the URI, such as /orders?minCost=n instead of retrieving all orders, then filtering. The API should create a query specifying the search parameter to retrieve just what is needed.

Pagination – You can specify a certain number of items to return with a request at a time and only retrieve the next set when the user requests the next page. /orders?limit=25&offset=50

Impose an upper limit on the number of items returned to help prevent *Denial of Service* attacks. To assist client applications, *GET* requests that return *paginated* data should also include some form of metadata that indicate the total number of resources available in the collection.
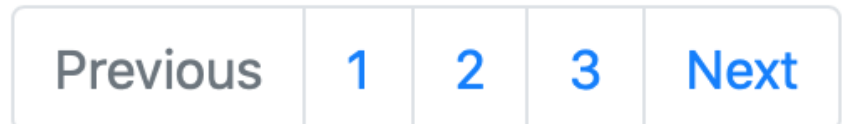
REST Best Practices

_____

- Resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource). https://adventure-works.com/orders (NOT "create-order"

- Avoid creating APIs that simply mirror the internal structure of a database.

- A collection is a separate resource from the item within the collection and should have its own URI.

- use plural nouns for URIs that reference collections.

- organize URIs for collections and items into a hierarchy.

- consider the relationships between different types of resources and how you might expose these associations: /customers/5/orders might retrieve all the orders for customer 5. (Don't go too far with this.)

- provide navigable links to associated resources in the body of the HTTP response message.(HATEOAS)

- avoid "chatty" web APIs that expose a large number of small resources, but balance this approach against the overhead of fetching data that the client doesn't need. Retrieving large objects can increase the latency.

- Avoid introducing dependencies between the web API and the underlying data sources.

- introduce a mapping layer between the database and the web API.(p2!)

Open API Initiative

The Open API Initiative was created to standardize REST API descriptions across vendors.

Swagger 2.0 specification was renamed the OpenAPI Specification (OAS) and brought under the Open API Initiative.

The OpenAPI Specification offers a set of guidelines on how a REST API should be designed. That has advantages for interoperability but requires more care when designing your API to conform to the specification.

OpenAPI promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.

Tools like Swagger can generate client libraries or documentation from API contracts.

Interesting Reads

- The N+1 Problem

- SOAP vs REST

- How to design a REST API.

- Authentication/authorization should not depend on cookies or sessions.

- The Halting Problem.

CORS

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos.[2] Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy. CORS defines a way in which a browser and server can interact to determine whether it is safe to allow the cross-origin request.

CSRF

___
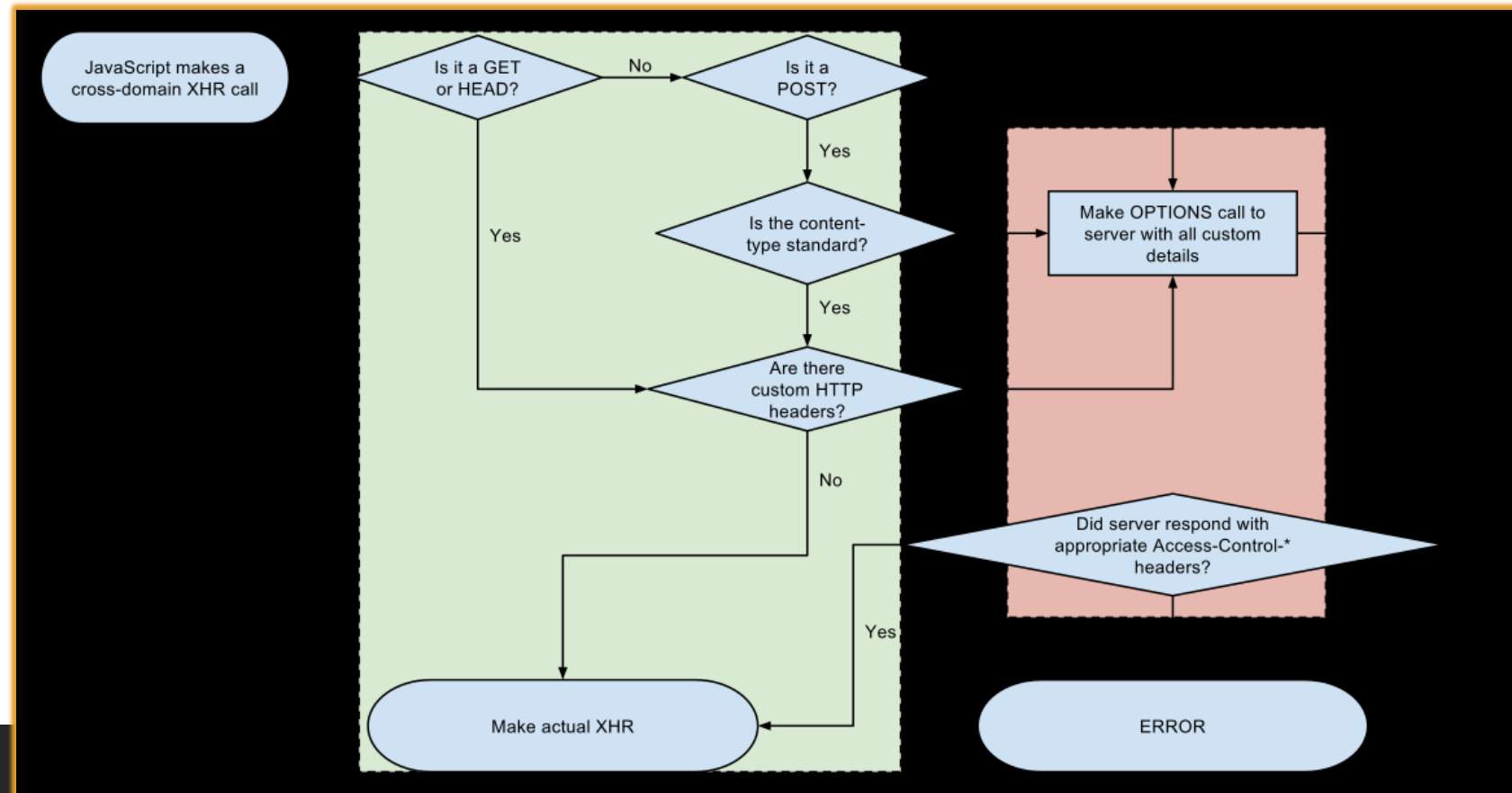
Cross-site request forgery(CSRF) is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts.

There are many ways in which a malicious website can transmit such commands; specially-crafted image tags, hidden forms, and JavaScript XMLHttpRequests, for example, can all work without the user's interaction or even knowledge. CSRF exploits the trust that a site has in a user's browser.

In a CSRF attack actions can be performed on the website that can include inadvertent client or server data leakage, change of session state, or manipulation of an end user's account.

HttpClient

Provides a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.

The HttpClient class instance acts as a session to send HTTP requests. An HttpClient instance is a collection of settings applied to all requests executed by that instance. In addition, every HttpClient instance uses its own connection pool, isolating its requests from requests executed by other HttpClient instances.

Fetch API

The Fetch API provides an interface for fetching resources (including across the network). It will seem familiar to anyone who has used XMLHttpRequest, but the new API provides a more powerful and flexible feature set.

How to implement a REST API on ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1
https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types

_____

API Tutorial

Mvc has views.. In REST, videos go away.

In VS create a asp.net core web app using c#. Of in commandLine *dotnet new webapi -o NotesApi*

Choose *api* as the app type

Create and go to *startrup.cs* in the *ConfigureServices* method use *services.AddControllers();*

The configure method you see *endpoints.MapControllers()*

We use the Class *ControllerBase* for attribute routing. This is the parent class of the MVC (with views) and doesn't support views.

With MVC you use *IActionResult* for the return type and that works also with web API's

You can write action methods specific to a different state of the source but usually you write them more generally.

Now Download *PostMan* – you don't have to sign in to use it.

In PostMan…

Use the generic app provided to you…

Send the first GET and it fails. In the details it tells you why. The SSL cert isn't trusted so you have to go to the settings(upper right. Looks like a wrench) Turn OFF SSL Cert Verification.

NOW you can send the GET and see the JSON in 'pretty print' format.

NOW add a scaffolded Item → *API Controller with read/write actions* . This will install whatever NuGet packages you need.

Looking at the template API Controller. attributes;:

◦ [Route("api/[controller]")] gives a prefix for all the action methods of this controller
◦ [ApiController] makes this action method an attribute router with automatic checking for some return status codes (like 400 Bad Request)
◦ [HttpGet("{id}", Name="Get")] the methods route will append to this. ID to the controllers overall route. Id is the route parameter or a variable for the method parameter. This is a Get method so only GET requests will be routed here.

```
using System.Threading.Tasks;

namespace NotesService.Api.Models
{
    0 references
    public class Note
    {
        0 references
        public int Id { get; set; }

        0 references
        public string Author { get; set; }

        0 references
        public bool IsPublic { get; set; }

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

The template may be broken so you may have to replace some of the HttpPost things.

Now make a *Models* folder with a *Note* class. To do some CRUD operations. He makes a model with some relevant properties to return and show that it worked.

Nick implements a repository in the same project (for now). Will have a *private list* in the *Note Repository* to store the data (instead of using a DB connection (for now).

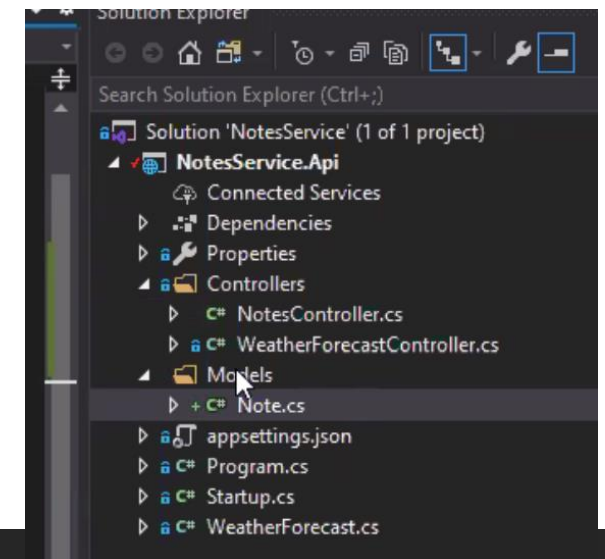Create a new controller in commandline!!!

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet aspnet-codegenerator controller -name NotesController -api --readWriteActions -outDir Controllers

Repository with a list to return for PostMan/API demo

Add the interface for the repository class in startup.cs
ConfigureServices so that you can do DI in the controller.

The Get method he created work

NOW we send JSON form data (Body in PostMan, choose raw and JSON). Get the response JSON to see what format is expected.


```csharp
namespace NotesService.Api.Repositories
{
    0 references
    public class NoteRepository
    {
        private readonly List<Note> _data = new List<Note>
        {
            new Note {
                Id = 1,
                Author = "Nick",
                IsPublic = true,
                Text = "REST stands for representational state transfer"
            }
        };

        0 references
        public IEnumerable<Note> GetAll()
        {
            return _data.ToList();
        }

        0 references
        public Note GetById(int id)
        {
            return _data.FirstOrDefault(n => n.Id == id);
        }

        0 references
        public void Add(Note note)
        {
            note.Id = _data.Max(n => n.Id) + 1;
            _data.Add(note);
        }
```

```csharp
// (e.g.    automatic modelstate checking with a
1 reference
public class NotesController : ControllerBase
{
    private readonly INoteRepository _noteRepository;

    0 references
    public NotesController(INoteRepository noteRepository)
    {
        _noteRepository = noteRepository;
    }

    // GET: api/Notes
    [HttpGet]
    0 references
```

```csharp
// GET: api/Notes
[HttpGet]
0 references
public IActionResult Get()
{
    IEnumerable<Note> notes = _noteRepository.GetAll();
    return Ok(notes);
    // ( 200 OK response, with the notes serialized in the response body
}
```

homework

challenge, implement PUT or DELETE according to HTTP/REST best practices

Day 2

/api/notes
GET: get all notes
POST: create a new note /api/notes/{id}
GET: get this note
PUT: update this note
DELETE: delete this note

Starting with Postman and modeling how to GET and POST.

Gives them an assignment to... (above)

Talks about the "Halting Problem" and them tries to replicate something that I didn't catch.... Oh.. So he's talking about how an object can contain an object that contains the first object. So to display the object it continuously redisplayed each object ... like in a JSON prettyprinted format.

How to implement a REST API on ASP.NET Core

_____

API Tutorial

Mvc has views.. In REST, videos go away.

In VS create a asp.net core web app using c#. Of in commandLine *dotnet new webapi -o NotesApi*

Choose *api* as the app type

Create and go to *startrup.cs* in the *ConfigureServices* method use *services.AddControllers();*

The configure method you see *endpoints.MapControllers()*

We use the Class *ControllerBase* for attribute routing. This is the parent class of the MVC (with views) and doesn't support views.

With MVC you use *IActionResult* for the return type and that works also with web API's

You can write action methods specific to a different state of the source but usually you write them more generally.

Now Download *PostMan* – you don't have to sign in to use it.

In PostMan...

Use the generic app provided to you...

Send the first GET and it fails. In the details it tells you why. The SSL cert isn't trusted so you have to go to the settings(upper right. Looks like a wrench) Turn OFF SSL Cert Verification.

NOW you can send the GET and see the JSON in 'pretty print' format.

NOW add a scaffolded Item → *API Controller with read/write actions* . This will install whatever NuGet packages you need.

Looking at the template API Controller. attributes;:
- [Route("api/[controller]")] gives a prefix for all the action methods of this controller
- [ApiController] makes this action method an attribute router with automatic checking for some return status codes (like 400 Bad Request)
- [HttpGet("{id}", Name="Get")] the methods route will append to this. ID to the controllers overall route. Id is the route parameter or a variable for the method parameter. This is a Get method so only GET requests will be routed here.

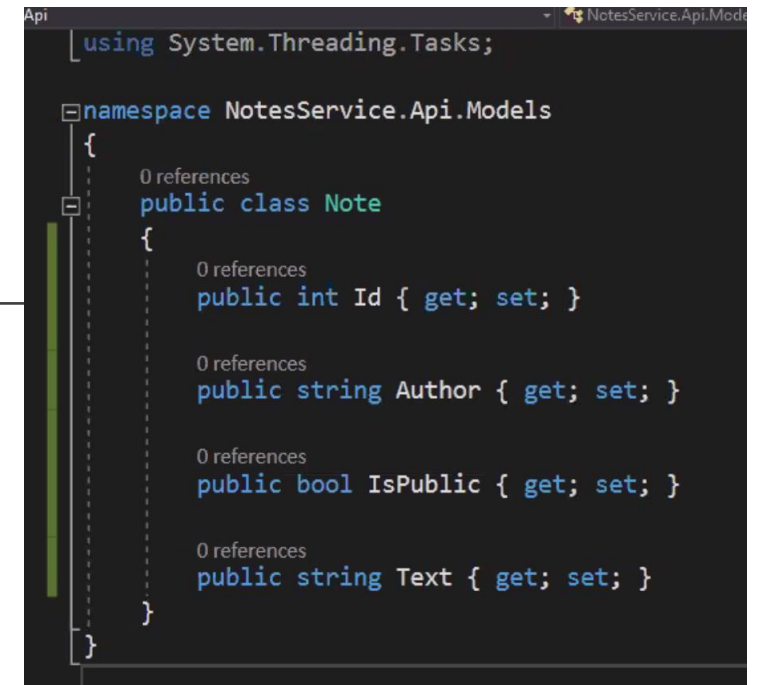The template may be broken so you may have to replace some of the HttpPost things.

Now make a *Models* folder with a *Note* class. To do some CRUD operations. He makes a model with some relevant properties to return and show that it worked.

Nick implements a repository in the same project (for now). Will have a *private list* in the *Note Repository* to store the data (instead of using a DB connection (for now).
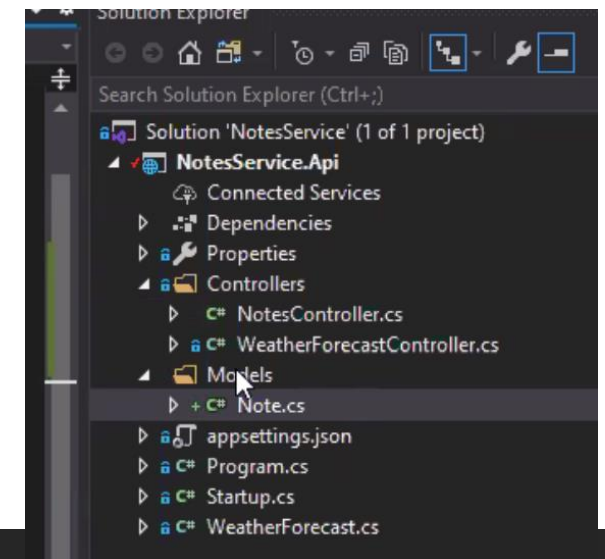
Create a new controller in commandline!!!

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet aspnet-codegenerator controller -name NotesController -api -- readWriteActions -outDir Controllers

Repository with a list to return for PostMan/API demo

Add the interface for the repository class in startup.cs *ConfigureServices* so that you can do DI in the controller.

The Get method he created work

NOW we send JSON form data (Body in PostMan, choose raw and JSON). Get the response JSON to see what format is expected.


```csharp
namespace NotesService.Api.Repositories
{
    0 references
    public class NoteRepository
    {
        private readonly List<Note> _data = new List<Note>
        {
            new Note {
                Id = 1,
                Author = "Nick",
                IsPublic = true,
                Text = "REST stands for representational state transfer"
            }
        };

        0 references
        public IEnumerable<Note> GetAll()
        {
            return _data.ToList();
        }

        0 references
        public Note GetById(int id)
        {
            return _data.FirstOrDefault(n => n.Id == id);
        }

        0 references
        public void Add(Note note)
        {
            note.Id = _data.Max(n => n.Id) + 1;
            _data.Add(note);
        }
```

```csharp
// GET: api/Notes
[HttpGet]
0 references
public IActionResult Get()
{
    IEnumerable<Note> notes = _noteRepository.GetAll();
    return Ok(notes);
    // ( 200 OK response, with the notes serialized in the response body
}
```

```csharp
// (e.g.   automatic ModelState checking with a
1 reference
public class NotesController : ControllerBase
{
    private readonly INoteRepository _noteRepository;

    0 references
    public NotesController(INoteRepository noteRepository)
    {
        _noteRepository = noteRepository;
    }

    // GET: api/Notes
    [HttpGet]
    0 references
```

homework

---

challenge, implement PUT or DELETE according to HTTP/REST best practices

Day 2

/api/notes
GET: get all notes
POST: create a new note /api/notes/{id}
GET: get this note
PUT: update this note
DELETE: delete this note

Starting with Postman and modeling how to GET and POST.

Gives them an assignment to... (above)

Talks about the "Halting Problem" and them tries to replicate something that I didn't catch.... Oh.. So he's talking about how an object can contain an object that contains the first object. So to display the object it continuously redisplayed each object ... like in a JSON prettyprinted format.