REST Fundamentals

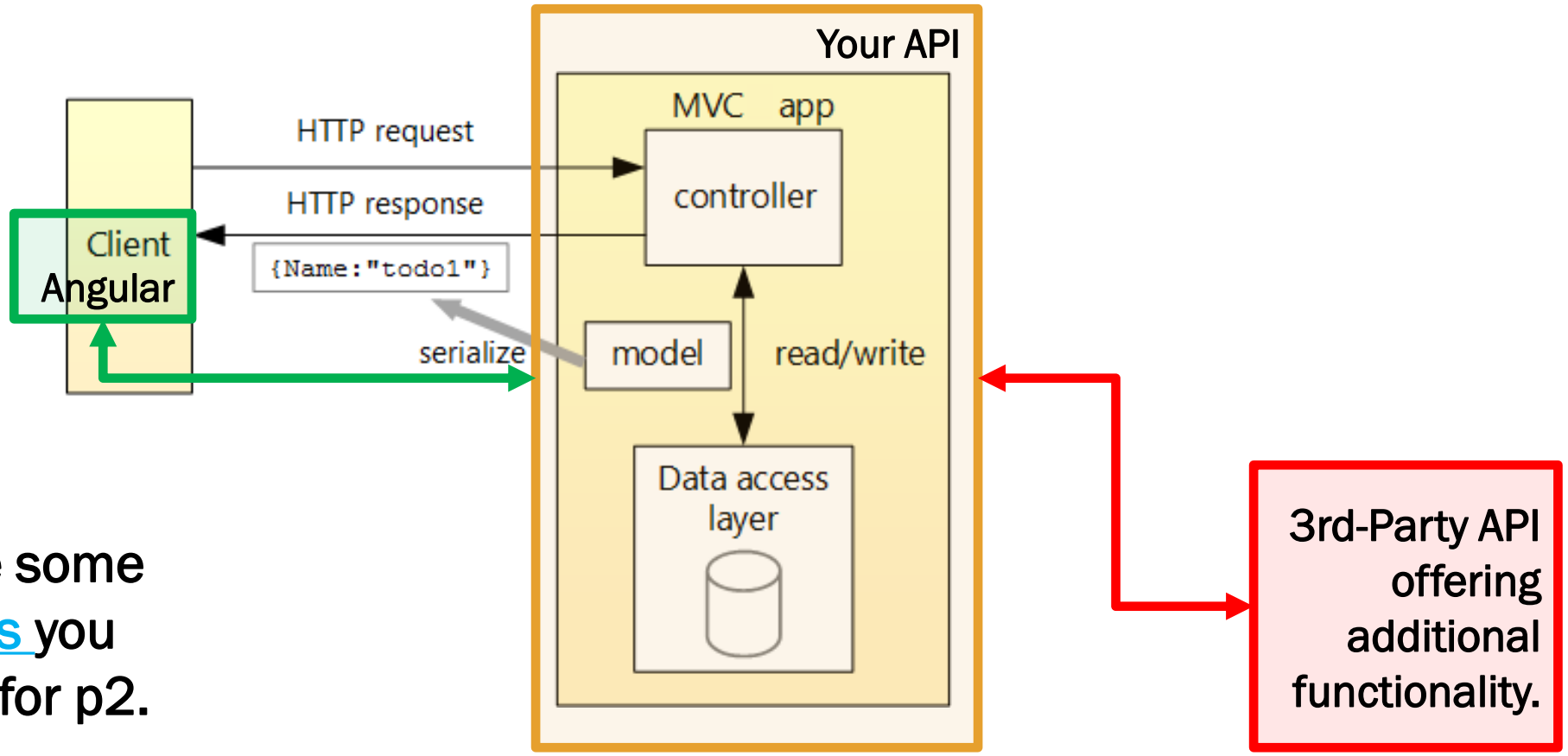
.NET

REST is an architectural style in which data and functionality are considered resources. Resources are accessed using Uniform Resource Identifiers (URIs) and are manipulated using methods. Clients and servers exchange representations of resources using a standardized interface and protocol: typically, HTTP and serialized JSON.

HTTPS://RESTFULAPI.NET/

REST API Tutorial

https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio



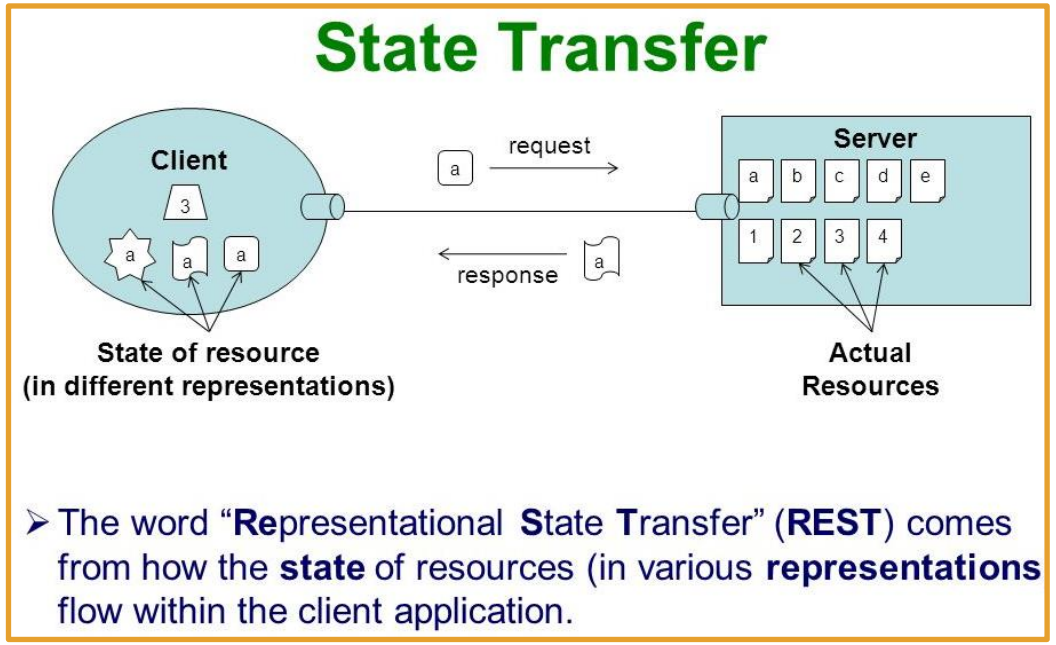
Here are some free API's you can use for p2.

Rest – Overview

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#introduction-to-rest

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 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 an example of an 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.

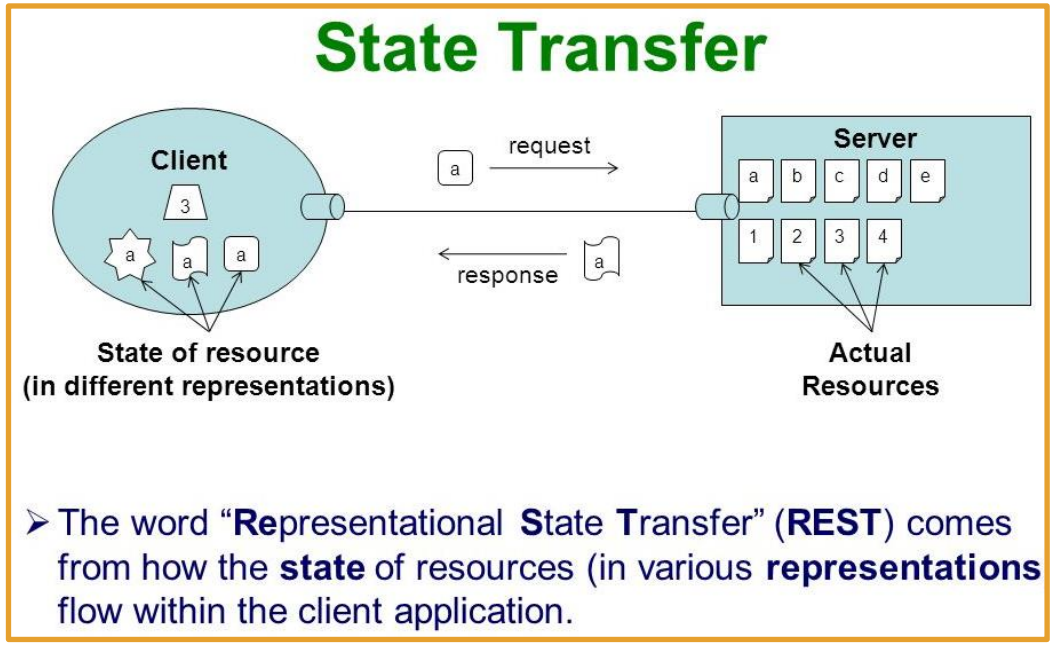


Rest – Overview

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#introduction-to-rest

REST APIs use a uniform interface that allows decoupling of client and service applications. They agree on a data transfer format (XML, JSON, JPG, etc) 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 is 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 current state of the permanent resource (DB).



REST – Guiding Principles/Constraints

https://restfulapi.net/
https://restfulapi.net/rest-architectural-constraints/

Client-Server	Stateless	Cacheable	Uniform Interface	Layered System	Code on Demand (optional)
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.	All Interfaces have 4 constraints: • ID of resources, • manipulation of resources through representations, • self-descriptive messages, • HATEOAS.	Hierarchical architecture. Each component cannot see beyond the layer with which it interacts.	REST allows extension of functionality by allowing clients to download scripts to extend functionality. Like a “widget”.

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

REST – HTTP Methods

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#define-operations-in-terms-of-http-methods
https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
https://restfulapi.net/

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 never defined any rules around which HTTP method to use in which condition. The only requirement is that the interface be uniform and consistent.

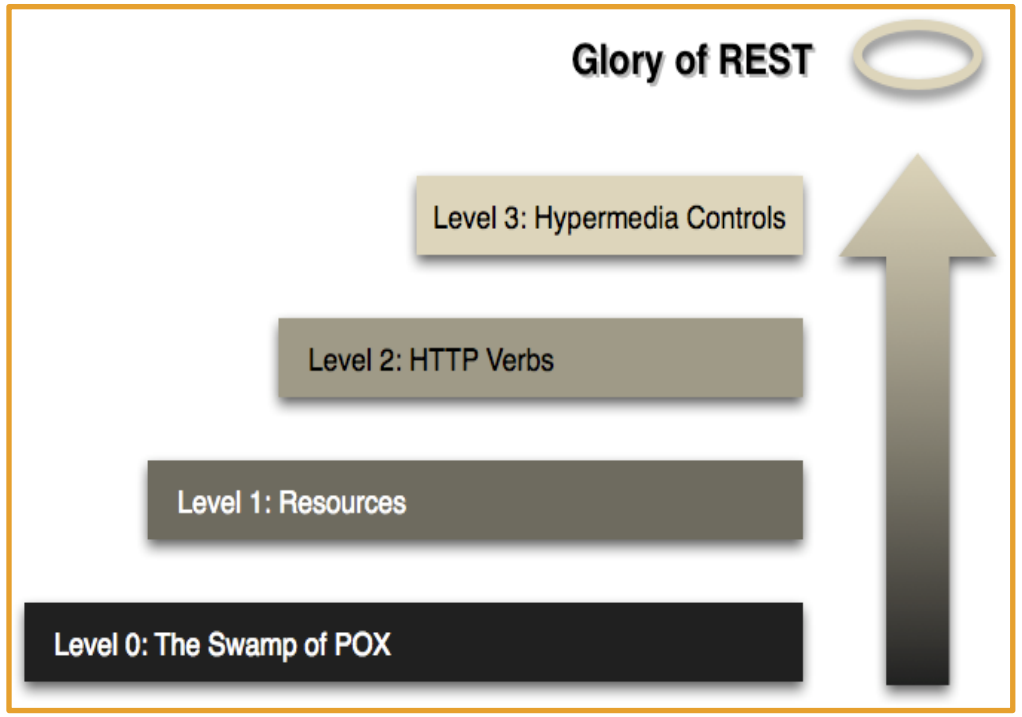
Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer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

Richardson Maturity Model

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#introduction-to-rest

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.

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 included 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.


```
{
  "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":[]
    }]
}
```

REST Best Practices (1/2)

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#organize-the-api-around-resources

- Resource URIs should be based on nouns (the resource) and not verbs (the operations on the resource). Https://adventure-works.com/Orders (NOT "GetOrders")
- Avoid creating APIs that simply mirror the internal structure of a database. This gets easier when using Microservices.
- A collection is a separate resource from the item within the collection and should have its own URI.
  - Collection => Revature.com/associates.
  - Item => Revature.com/associates/45
- Use plural nouns for URIs that reference collections.
  - Collection => Revature.com/associates/5.
- Organize URIs for collections and items into a hierarchy.

REST Best Practices (2/2)

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#organize-the-api-around-resources

- Consider the relationships between different types of resources and how you might expose these associations:
  - /customer/{id}/{resource} => /customers/5/orders might retrieve all the orders for customer 5.
- Provide navigable links to associated resources in the body of the HTTP response message.(HATEOAS)
- Avoid "chatty" web APIs that expose many small resources.
- Balance exposing many resources against the overhead of fetching data that the client doesn't need. Retrieving large objects can increase latency.
- Avoid introducing dependencies between the web API and the underlying data sources.
- Introduce a mapping layer between the database and the web API so that only requested data is returned. Use Data Transfer Objects (DTO) to minimize payloads.

Filtering and Pagination

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#filter-and-paginate-data

---

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

Filtering - Allow for passing a filter in the query string of the URI, such as /orders?minCost=n instead of retrieving all orders. 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

Set an upper limit on the number of items returned. This helps prevent Denial of Service (DoS) attacks.

GET requests that return paginated data should also include some form of metadata that indicate the total number of resources available in the collection.

Open API Initiative

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design#open-api-initiative
https://www.openapis.org/
https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-5.0

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

Swagger 2.0 specification was renamed to ***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. This has advantages for interoperability but requires more care when designing an API.

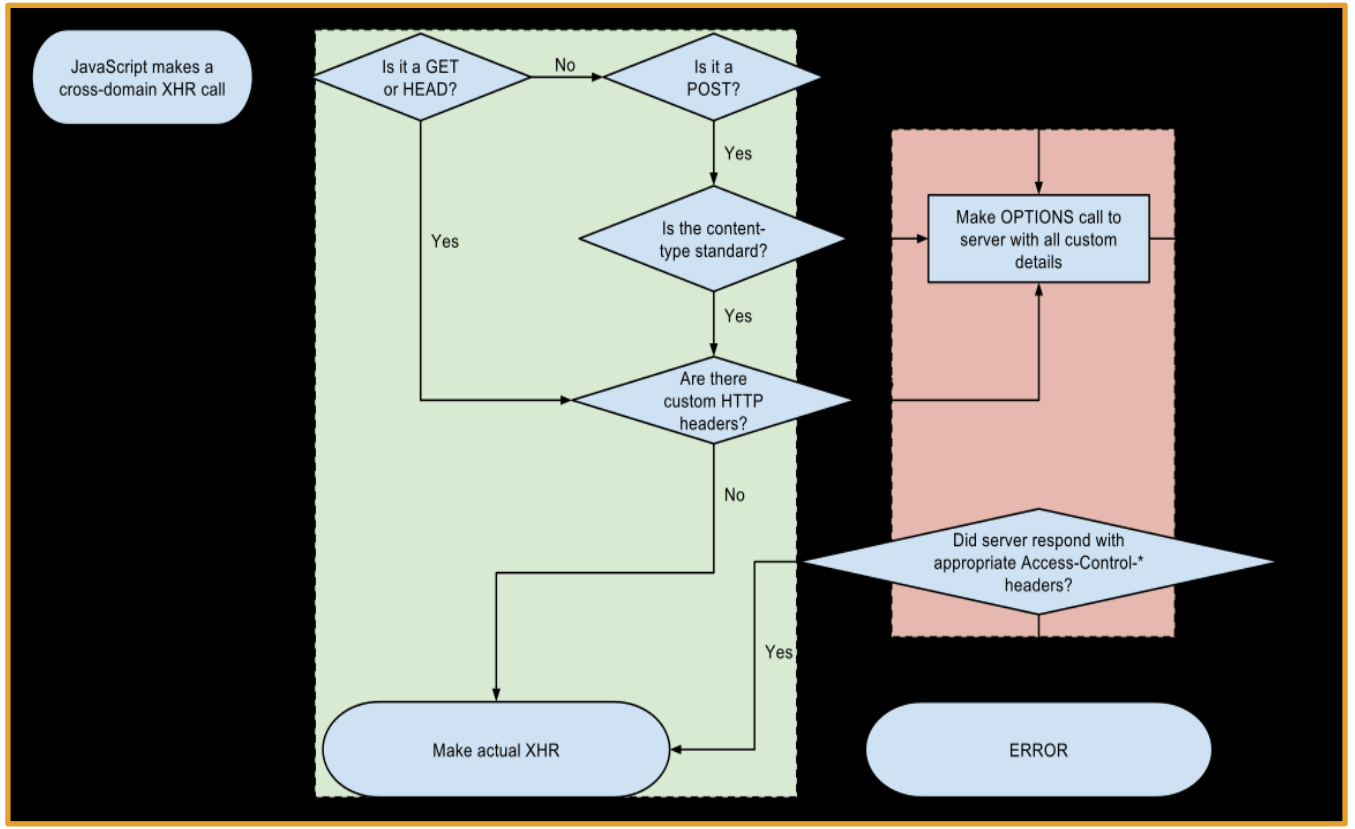
OAS promotes a contract-first approach, rather than an implementation-first approach. Contract-first means you design the API contract (the interface) first and then write code that implements the contract.(i.e. “What I want to offer. Then, “How will I get that.”)

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

CORS

https://en.wikipedia.org/wiki/Cross-origin_resource_sharing
https://www.w3.org/TR/cors/#resource-requests

Cross-origin resource sharing (CORS) 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 might embed cross-origin images, stylesheets, scripts, iframes, and videos.
Certain "cross-domain" requests (AJAX requests) are forbidden 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.



Cross-site request forgery(CSRF)

https://en.wikipedia.org/wiki/Cross-site_request_forgery

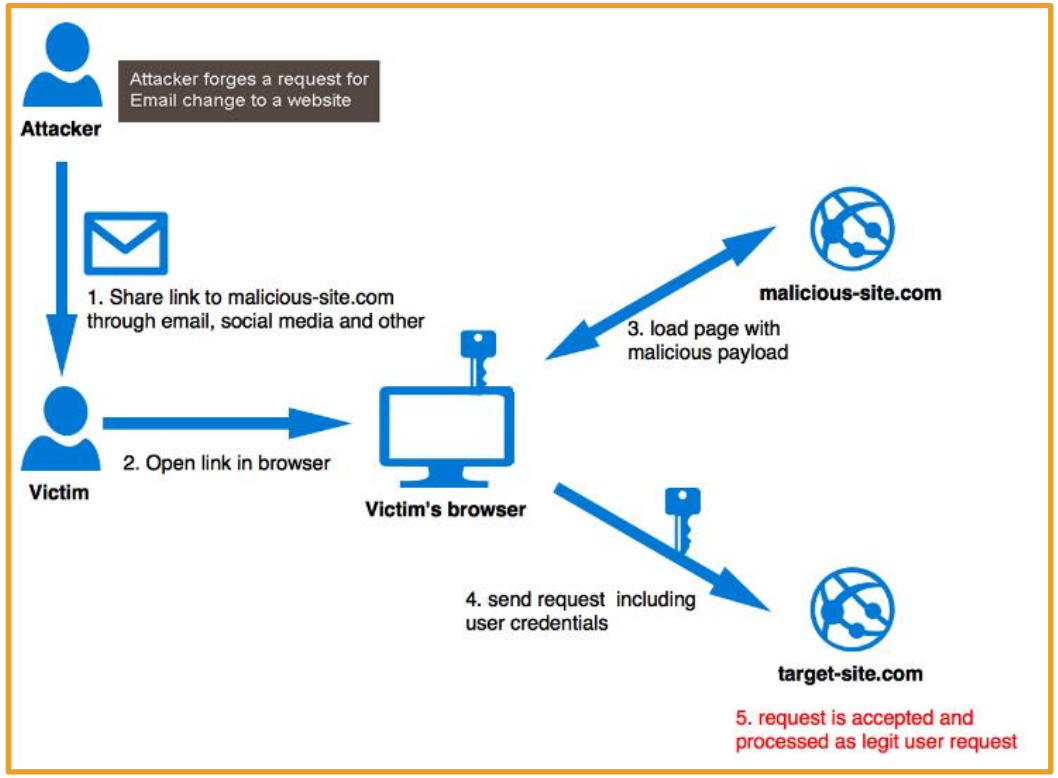
CSRF is a type of attack where unauthorized commands are transmitted from a user that the web application trusts.

A malicious website can transmit commands by using:

- specially-crafted image tags,
- hidden forms,
- *JavaScript XMLHttpRequests.*

All the above can 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

https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-5.0

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.

How to implement a REST API on ASP.NET Core

https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-5.0
https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types

---

- API Tutorial
- Mvc has views.. In REST, views go away and you are left with HTTP endpoints that you must query.
- In VS create a asp.net core web app using C#. *dotnet new webapi -o <appName>*
- Choose ***api*** as the app type
- Create and go to ***startup.cs*** in the ***ConfigureServices()*** method use ***services.AddControllers();***
- In the configure method you see ***endpoints.MapControllers()***
- The ***ControllerBase*** Class allows 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. IActionResult also works with web API's
- You can write action methods specific to a different state of the source but usually you write them more generally.
- Download ***PostMan*** – you don't have to sign in to use it.

Interesting Reads and Resources

- 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.