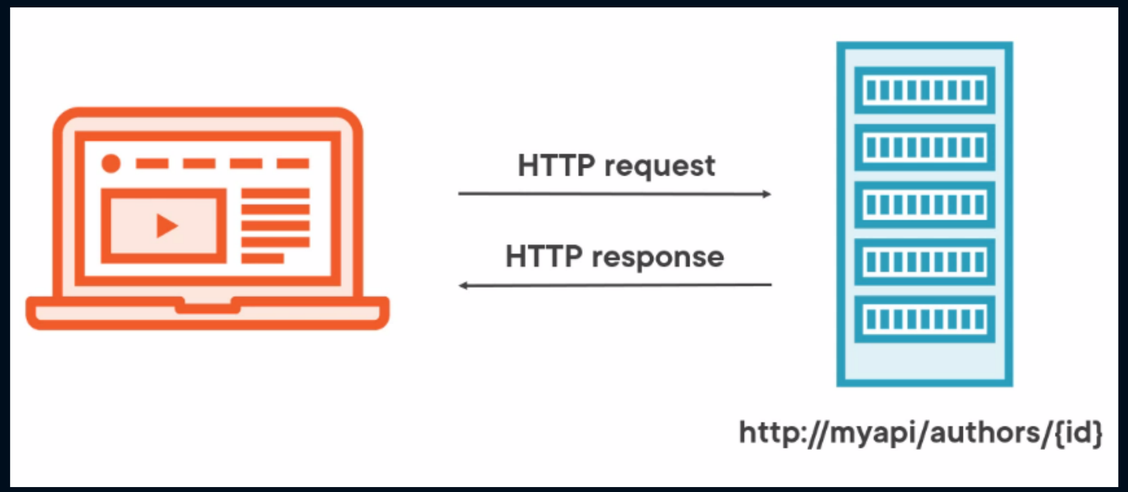
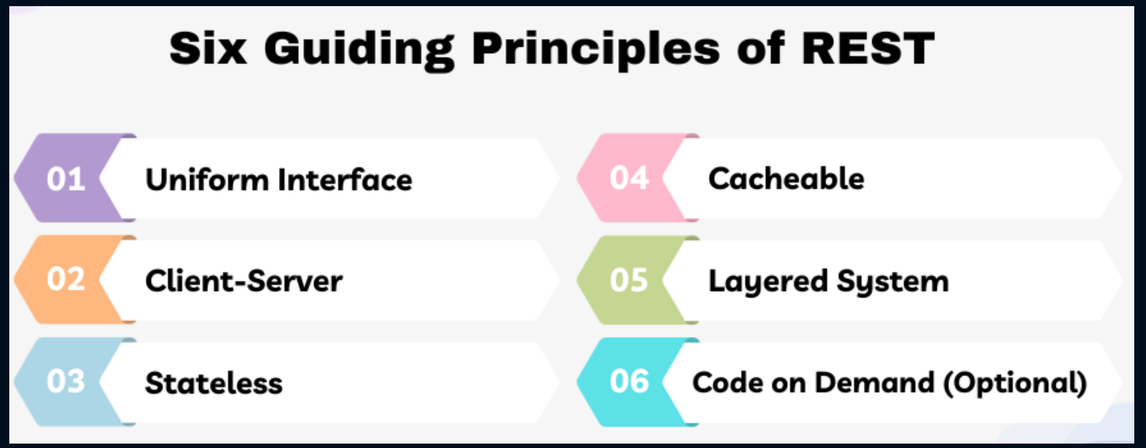
REST Overview

REST is an acronym for REpresentational State Transfer and an architectural style for distributed hypermedia systems. REST is not a protocol or a standard, it also has its guiding principles and constraints. These principles must be satisfied if a service interface has to be referred to as RESTful.



REST is, in principle, protocol agnostic



REST guiding principles

REST is based on some constraints and principles that promote simplicity, scalability, and statelessness in the design.

REST's Guiding Principles

Uniform Interface

By applying the principle of generality to the components interface, we can simplify the overall system architecture and improve the visibility of interactions.

The following four constraints can achieve a uniform REST interface:

- Identification of resources
- Manipulation of resources through representations
- Self-descriptive messages
- Hypermedia as the engine of application state

In simpler words, REST defines a consistent and uniform interface for interactions between clients and servers. For example, the HTTP-based REST APIs make use of the standard HTTP methods (GET, POST, PUT, DELETE, etc.) and the URIs (Uniform Resource Identifiers) to identify resources.

Client-Server

The client-server design pattern enforces the separation of concerns, which helps the client and the server components evolve independently.

By separating the user interface concerns (client) from the data storage concerns (server), we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

While the client and the server evolve, we have to make sure that the interface/contract between the client and the server does not break.

REST's Guiding Principles

Stateless

Statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request.

The server cannot take advantage of any previously stored context information on the server.

Cacheable

The cacheable constraint requires that a response should implicitly or explicitly label itself as cacheable or non-cacheable.

If the response is cacheable, the client application gets the right to reuse the response data later for equivalent requests and a specified period.

Layered System

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior. In a layered system, each component cannot see beyond the immediate layer they are interacting with.

Code on Demand (Optional)

REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts.

The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

What is a Resource?

The key abstraction of information in REST is a resource. Any information that we can name can be a resource. For example, a REST resource can be a document or image, a temporal service, a collection of other resources, or a non-virtual object (e.g., a person).

The state of the resource, at any particular time, is known as the resource representation.

The resource representations consist of:

- the data
- the metadata describing the data
- the hypermedia links that can help the clients transition to the next desired state.

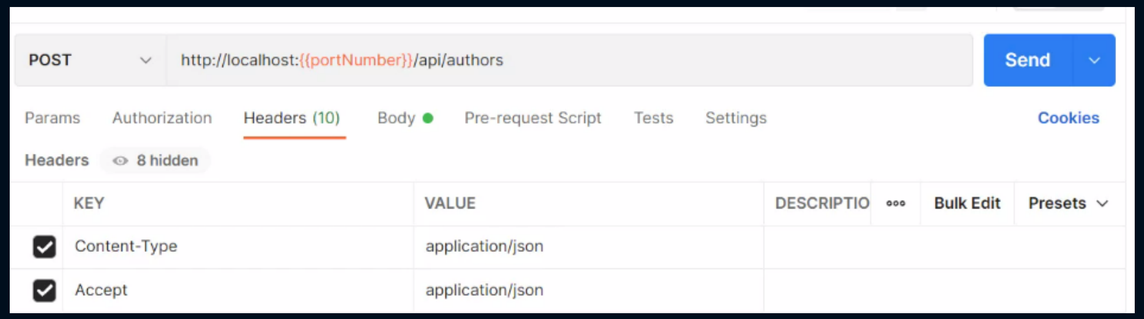
Consider the following REST resource that represents a blog post with links to related resources in an HTTP-based REST API. This has the necessary information about the blog post, as well as the hypermedia links to the related resources such as author and comments. Clients can follow these links to discover additional information or perform actions.

```json
{
  "id": 123,
  "title": "What is REST",
  "content": "REST is an architectural style for building web services...",
  "published_at": "2023-11-04T14:30:00Z",
  "author": {
    "id": 456,
    "name": "John Doe",
    "profile_url": "https://example.com/authors/johndoe"
  },
  "comments": {
    "count": 5,
    "comments_url": "https://example.com/posts/123/comments"
  },
  "self": {
    "link": "https://example.com/posts/123"
  }
}
```

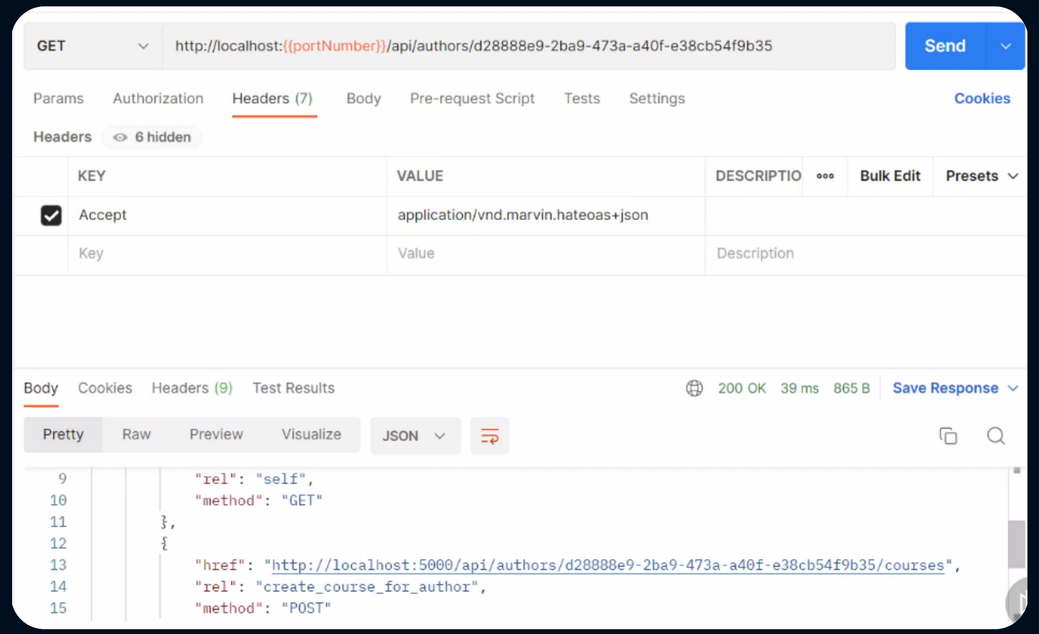
Resource Example

REST and HTTP

REST != HTTP. Though REST also intends to make the web (internet) more streamlined and standard, Roy Fielding advocates using REST principles more strictly. And that's where people try to start comparing REST with the web. Nevertheless it remains until today as the prefer way to implement a RESTFul API.



A Resource is conceptually separate from its representation. Representation media types: application/json, application/xml, custom



Representation + metadata must be sufficient to modify or delete the resource

HTTP Verbs

GET

GET /v1/products/iphone

Response

```
<HTML>
  <HEAD>iphone</HEAD>
  <BODY>
    <H1>iPhone 14</H1>
    <P>This is an iPhone 14</P>
  </BODY>
</HTML>
```

Retrieve a single item or a list of items

PUT

PUT /vi/users/123

Request Body

```
{ "name": "bob",
  "email": "bob@bytebytego.com"
}
```

Response

HTTP/1.1 200 OK

Update an item

POST

POST /v1/users

Request Body

```
{ "firstname": "bob",
  "lastname": "xxx",
  "email": "bob@bytebytego.com" }
```

Response

HTTP/1.1 201 Created

Create an item

DELETE

DELETE /v1/users/123

Response

HTTP/1.1 200 OK
HTTP/1.1 204 NO CONTENT

Delete an item

PATCH

PATCH /v1/users/123

Request Body

```
{
  "email": bob@bytebytego.com
}
```

Response

HTTP/1.1 200 OK

Partially modify an item

HEAD

HEAD /v1/products/iphone

Response

HTTP/1.1 200 OK

Identical to GET but no message body in the response

CONNECT

CONNECT xxx.com:80

Request

```
Host: xxx: 80
Proxy-Authorization: basic
RXhhbXBsZTphaQ==
```

Response

HTTP/1.1 200 OK

Create a two-way connection with a proxy server

OPTIONS

OPTIONS /v1/users

Response

```
HTTP/ 1.1 200 OK
Allow: GET,POST DELETE,HEAD,
OPTIONS
```

Return a list of supported HTTP methods

TRACE

TRACE /index.html

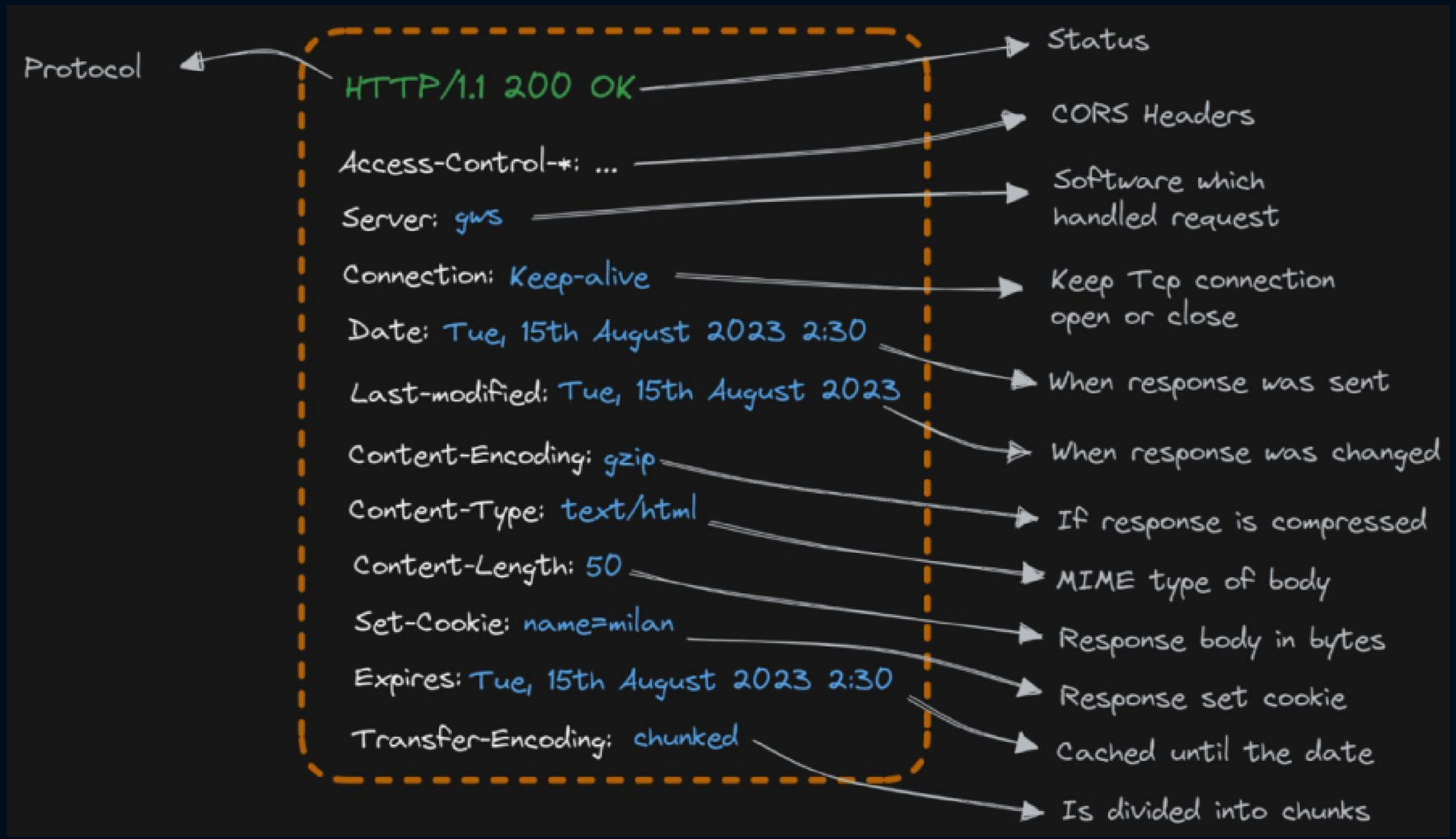
Response

```
Host: xxxxxx
Via: 1.1 xxxx: 3221
X-Forwarded-For: xx.xxx.xxx.x
```

Perform a message loop-back test,providing a debugging mechanism

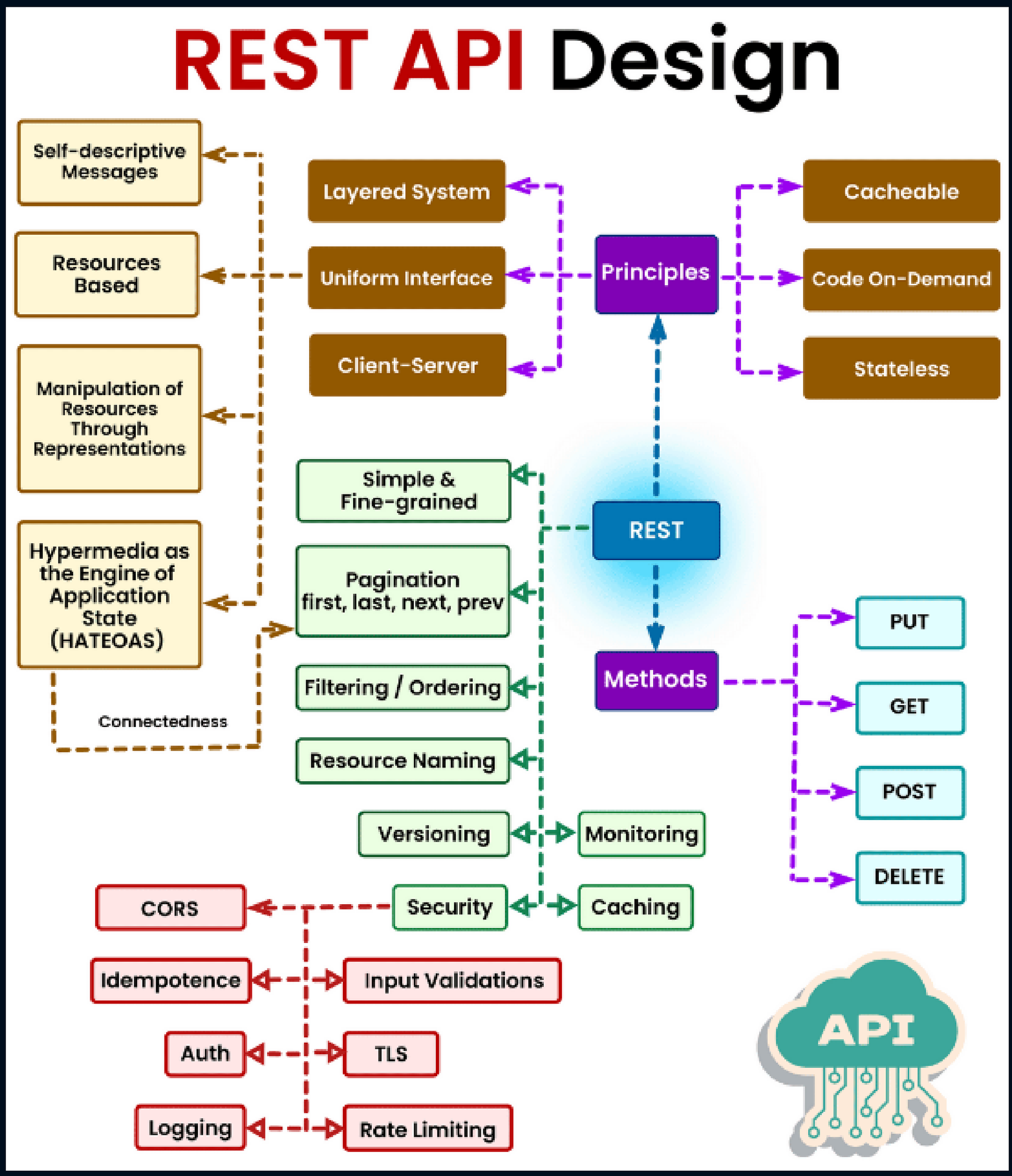
Http Verbs

HTTP Response Headers



Rest Headers

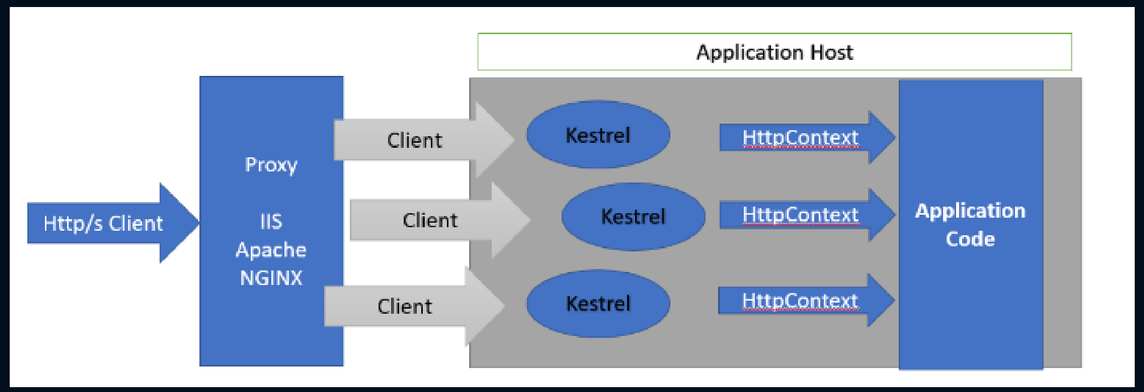
REST API Design



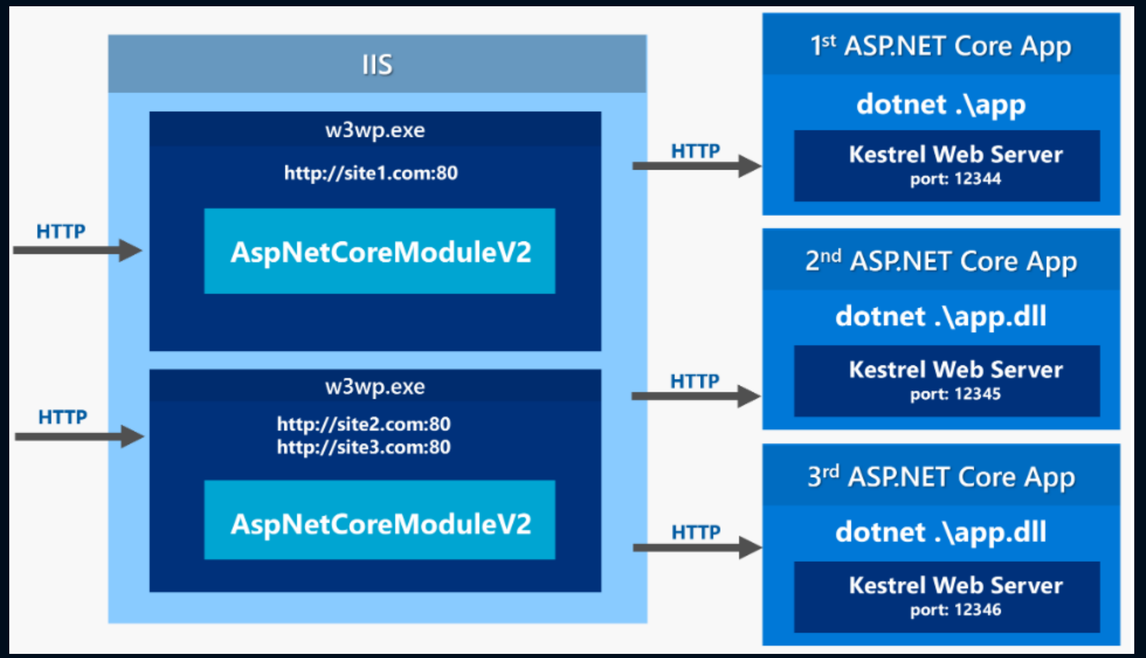
Design Rest APIs

Web Servers - Kestrel

Kestrel is a cross-platform, open-source web server developed by Microsoft and specifically designed to work with ASP.NET Core applications. It serves as the default web server for ASP.NET Core applications and provides a fast, lightweight, and efficient hosting environment.
Check this Simple manual Kestrel implementation for more information



Kestrel Use



IIS with Kestrel

Routing

Routing in ASP.NET Core Web API is the process of matching incoming HTTP requests to corresponding controller actions. When a client sends an HTTP request, the routing middleware in ASP.NET Core Web API examines the incoming URL against the registered route templates. If a match is found, the corresponding route is activated

```
app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints => {
        // map endpoints });
```

Learning About Routing

Middleware that runs in between selecting the endpoint and executing the selected endpoint can be injected

UseRouting and Use Endpoints

UseRouting - Marks the position in the middleware pipeline where a routing decision is made

UseEndpoints - Marks the position in the middleware pipeline where the selected endpoint is executed

```
app.UseAuthorization();

app.MapControllers();
```

Attribute-based Routing

Shortcut: call `MapControllers` on the `WebApplication` object directly

- Default in .NET 6
- Mixes request pipeline setup with route management

Attribute routing

[Route] doesn't map to an HTTP method

Used at a controller level to provide a template that will prefix all template defined at action level

API Controller

A controller-based web API consists of one or more controller classes that derive from ControllerBase. The web API project template provides a starter controller. Web API controllers should typically derive from ControllerBase rather from Controller. Controller derives from ControllerBase and adds support for views, so it's for handling web pages, not web API requests.

Attribute	Binding source
[FromBody]	Request body
[FromForm]	Form data in the request body
[FromHeader]	Request header
[FromQuery]	Request query string parameter
[FromRoute]	Route data from the current request
[FromServices]	The request service injected as an action parameter
[AsParameters]	Method parameters

The [ApiController] attribute can be applied to a controller class to enable the following opinionated, API-specific behaviors:

- Attribute routing requirement
- Automatic HTTP 400 responses
- Binding source parameter inference
- Multipart/form-data request inference
- Problem details for error status codes

Attribute parameters

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController : ControllerBase
{
    [HttpPost]
    [Consumes("application/json")]
    public IActionResult PostJson(IEnumerable<int> values) =>
        Ok(new { Consumes = "application/json", Values = values });

    [HttpPost]
    [Consumes("application/x-www-form-urlencoded")]
    public IActionResult PostForm([FromForm] IEnumerable<int> values) =>
        Ok(new { Consumes = "application/x-www-form-urlencoded", Values = values });
}
```

Example of an API

Endpoints examples

In these examples, the attribute routing syntax is used to define the routes and HTTP methods for the respective endpoints. The [HttpPost], [HttpGet], [HttpPut], and [HttpDelete] attributes are used to specify the HTTP method for each endpoint.

```
[HttpGet("users/{id}")]
public IActionResult GetUserById(int id)
{
    // Logic to fetch the user from the database based on the provided ID
    // ...

    if (user == null)
    {
        // Return a 404 Not Found if the user is not found
        return NotFound();
    }

    // Return the user with a 200 OK status code
    return Ok(user);
}
```

Get Request

```
[HttpPost("users")]
public IActionResult CreateUser([FromBody] User user)
{
    // Logic to create the user and save it to the database
    // ...

    // Return a 201 Created status code with the created user
    return CreatedAtAction(nameof(GetUserById), new { id = user.Id }, user);
}
```

Post Request

Demo

We will check the following:

Walkthrough to a new Web API project

Showing a controller implementation

Controller has GET, POST

You can check the following Repository for some examples:

C# fundamentals

Homework

Exercise/ Homework

Following a **Domain Centric Architecture,** with the Domain models designed from the Previous homework (on class 10) create a ASP.NET Core Web API:

Implement the API for a specific Domain model. You can implement for more than one but **at least the implementation for one is required**

- GET and POST is a must, the rest is optional
- Prefer using in memory collection as database, nevertheless using MSSQL, PostgresSQL, SQL Lite is also allowed

Note: You can use other ideas generated from the design process from the previous workflow if it helps

The focus of this exercise is on the implementation of the API, so it is not necessary to use a database, since we will be covering that topic later. But as mentioned it is allowed