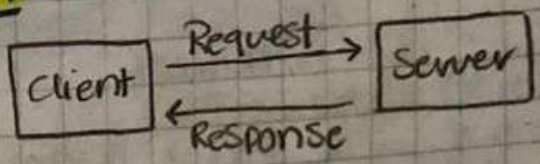
• HTTP



What is HTTP?

↳ Hyper Text Transfer Protocol بينقل البيانات بين ال Client وال Server

Network Protocol

← وبنستخدم ال browsers في عرض الصفحات الكترونية بال HTML

← ال HTML لغة قادرة تنقل الفايلات اللي من نوع XML و JSON

HTTP Characteristics:

[1] **Stateless**

المعروف بعد ما بيبعت ال request من ال Client لل Server ويرجع ال response معناه مفهوم سواء ال Client أو ال Server بيحتفظ بأي بيانات عن اليوزر

[2] **Connectionless**

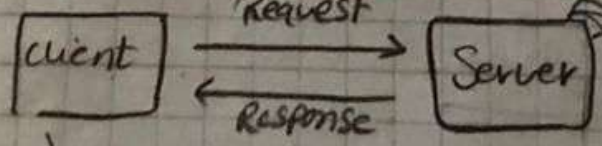
بمجرد ما ال request يبعت لل Server من ال Client وال Server يبعت ال response

ال Communication دي هتتقفل حتى هتتفتح إلا لو اليوزر أو ال Client هيبعت request مرة تانية وهكذا.

[3] **Independent**

لازم ال Client وال Server يكونوا متفقين على نوع البيانات اللي هتتنقل.

HTTP Requests and Responses:



Response have StatusCode (Number)
Payload (optional)
header

↓

بيكون initiate http request بـ

ليه URI هو العنوان اللي ال request هيروح عليه لـ Server

Headers

• URI of Request:
- have the address of resource that need to reach in Server
- have verb (Type of Request)
- Parameters

-- Http verbs --

CRUD operations.

1) Post → [Create]
2) Get → [Read / Retrive] → Select
3) put → [Update] over resource

بيعمل Update على الresource بشكل كامل يعني هيبقى واحد جديد خالص

4) Batch → [Partialy update]
5) Delete → [Remove]

-- Http Status Code --

بيرجع الresponse من ال Server لل client بتعرفه حصل ايه في الrequest

← متقسم لمجموعات :-

→ **200** [OK]

↳ the path was correctly formed, the resource was successfully found, serialized into an acceptable media type, and then returned in the response body

→ **201** [Created]

↳ The new Resource was created successfully, the response header named location contains its path, and the response body contain ~~body~~ the newly created Resource. Immediately GET-ing the resource should Return 200.

→ **202** [Accepted] لو الresponse بياخد وقت.

→ **204** [NO Content]

↳ commonly used in response to Delete request since returning the resources in the body after deleting it doesn't usually make sense! Sometimes used in response to Post, Put or Batch requests if the client doesn't need to confirm that the request was processed correctly.

→ **400** [Bad Request]

↳ the Request was invalid. For example, it used a path for a product using an integer ID where the ID value is missing.

→ 401 [Unauthorized]

↳ The Request was valid, and the resource was found but the client did not supply Credentials or is Not authorized to access that resource. Re-authenticating may enable access. For Example: by adding or changing the Authorization Request header.

→ 404 [Not Found]

غلط في الـ URL مثلاً

↳ The Request was valid, but the resources was not found. The resource may be found if the request is repeated later. To indicate that a resource will never be found, return 410 Gone.

→ 405 [Method Not Supported]

↳ Returned when the Request used a method that is not supported, For Example: a web service designed to be read-only may explicitly disallow Put, Delete and So on.

→ 500 [Internal Server error]

↳ The Request was valid, but Something went wrong on the Server Side while Processing the request. Retrying again later might work.

URI

URI: Uniform Resource Identifier

عنوان كتبته في الـ browser

Ex

https → Prefix

https : // www.devcreed.com / Courses / add ? year=2024 & month=1 # api_call

- https → Scheme: بحدد طريقة تعاملك مع Server الـ
- www.devcreed.com → authority [Server address] ↳ host + Port
- /Courses/add → Path ↳ may contain many Segments ↳ optional
- ?year=2024&month=1 → Query → Start with ?
- #api_call → Fragment

Scheme + authority → Mandatory

• and two optional parts

↳ Query → Start with ?

↳ Fragment

html للتوجيه لمكان معين في الصفحة

__HttP(S) - TLS __ [Scheme]

↳ Protocol.

allow U to exchange Data between Client and Server.

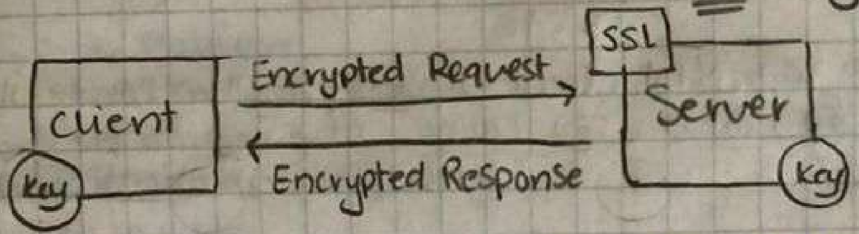
http → Port (80)
https → Port (443)

HttPS Provide:

- → Encryption
- → Data Integrity.
- → Authentication

SSL ⇒ Secure Socket layer

ال TLS تحسين ل SSL



→ Client and Server both have the Same Key used in Encryption and deeryption.

__REST__

add Some restrictions over the Communication between Server and client

REST ⇒ REpresentational State Transfer.
- ↳ Architectural Style.
- ↳ Depends on Http.

--REst Constraints-- (6)

1 Stateless

← كل request بيخرج من ال Client يروح لل Server ، لازم يكون موجود جواه كل المعلومات اللي (Server) محتاجها علشان يقدر ينفذ ال request ، لان ال Server مش لازم يحتفظ بحالة ال request

2 Client - Server

← كل واحد منهم لازم يكون مستقل بذاته علشان لو تعدل حد فيهم ميحصلش مشكلة في التاني
← ال Client مش محتاج يعرف غير ال URI بتاع ال resource اللي هيتعامل معاه وبياناته

3 Uniform Interface (.4)

3.1) Identification of Resources

← كل resource موجود عندي على ال Server لازم يكون ليه ال URI الخاص بيه

3.2) **Manipulation of Resources through Representations.**

← كل resource بيرجع من ال server ليه شكل محدد وال client لازم يكون عنده المعلومة الكاملة عن ازاي يتعامل مع ال resource دا.

3.3) **Self-Descriptive Messages.**

← كل شكل بيرجع بيه ال resource لازم يرجع بكل المعلومات اللي محتاجها.

Not used 3.4) **Hypermedia as the Engine of Application State. [HATEOAS]**

← ال client محتاج يكون initial URI لل Resource، بعد كدا ال Server هيوري محتاج اكلم اي URI بالتحديد.

[4] **Cacheability**

← ال Server هو اللي لازم يعرف ال client. اذا كان يقدر يعمل لل Cache response دا ولا لأ وهنتعامله Cache لوقت قد ايه (تتثبت في ال response header) علشان ال client ميحتاجهاش تاني.

[5] **Layered System**

تقدر تقسم ال Application لأكتر من layer، و ال client مش لازم يعرف اي حاجة عنها.

[6] **Code-on-Demand** (optional).

ال Server ممكن يرد على ال client ويرجعله Excecutable Code.

-- URL Naming --

verbs / Resource Name.

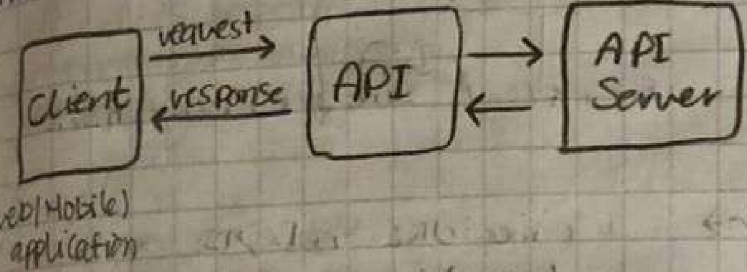
GET /Orders
POST /orders
GET /orders/{id}
PUT /Orders/{id}
Delete /Orders/{id}

GET / Orders / {orderId} / items → Select items in orders

GET / orders / {orderId} / items / {id} → يرجع حاجة معينة من ال items اللي موجودة جوا ال order

-- API --

API ⇒ Application Programming Interface.



-- .NET Core --

→ Cross Platform
(windows, IOS, linux ...).
→ Open Source.
→ Performance.
→ Easy to write
→ Build different Types of applications (web, Mobile, disktop ..)

-- Files --

(1) LaunchSettings.json

- مش بينقل معايا لما بعمل deploy للـ project
- بيعرفني الـ Application هيشتغل ازاي وانا بعمل development

→ Profiles: [http , https , IIS Express]

(2) Program.cs

↳ Entery Point of Our Project.

```
var builder = WebApplication.CreateBuilder(args);
```

يعني الـ builder ده هو اللي هيضيف جواه كل الـ Services اللي الـ application بتاعي محتاجها علشان تشتغل بشكل سليم

يعني الـ builder دا يعتبر الهيكل او الـ body هبدأ اضيف الـ Services.

```
builder.Services.AddControllers();
```

هنا هضيف Services بقا وهعرفه إني هستخدم الـ controller

```
builder.Services.AddEndPointsApiExplorer();
builder.Services.AddSwaggerGen();
```

دي الـ Configuration علشان الـ Swagger يشتغل.

```
var app = builder.build();
```

بعد ما بنضيف كل الـ Services بنعمل build للـ builder بتاعي وبنعمل cache للحاجات دي جوا الـ app.

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

الجزء داخل الـ Pipeline جواه مجموعة من الـ Middlewares

الجزء داخل مرحلة الـ development علشان نعرض الـ Swagger على الـ EndPoint

```
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

دي Middlewares الـ Functionality اللي هتشتغل (هتشتغل إزاي)

دي اللي بتبدأ تشتغل الـ app بتاعي

Note

- Services: اللي انا محتاجه علشان أشتغل صح.
- Middlewares: بيشتغل إزاي.

-- Dependency injection --

- **Dependency inversion Principle [DIP]:**
  - High level Modules Should not depend on low level Modules, Both Should depend on abstractions.

يعني كلاس (A) وكلاس (B)، الكلاس B بيعتمد في شغله على الكلاس (A)
كما كلاس (A) ← High level، كلاس (B) ← low level
فهنا بيقول مينفعش يكون فيه كلاس بيعتمد على التاني، الاتنين المفروض يعتمدوا على الـ abstraction.

  - Abstractions Should not depend on details. Details Should depend on abstractions. → OS

في الأول خالص الـ .NET Development كانت معتمدة على الـ windows بس يعني معتمد بشكل كلي على الـ windows وفقط ← كدا هو معتمد على implementation أو Details

- يكون الـ principle دا بيقول انك متش عاوزك تعتمد على OS معين. فال .Net الجديد خلاك تقدر تعتمد في الـ development على الـ OS بشكل عام، يعني تقدر تعمل development من windows أو iOS أو linux

.NET Core ← متش معتمدة على OS معين

- **Dependency Injection:**
  - is a technique in Software development that manages the dependencies between different components in a System.

→

- ال dependency Injection مش مرتبط بالتحديد بال .NET أو ال C# (عادي نقدر نستخدمه مع أي Framework وأي language تانين.

- وظيفة ال dependency injection إنها مثلاً لو عندي كلاس A معتمد في شغله على ميثود موجودة جوا كلاس B (فمتكونش وظيفة A إنه يكريت object من B علشان يكون قادر يستخدم الميثودز اللي محتاجها من جواه) فال dependency injection من نفسها هتعرف ان A محتاج B فهتعمل لـ B عملية inject جوا A علشان يقدر يستخدم الميثودز اللي هو محتاجها.

→ How to apply dependency injection?
  ↳ Constructor injection.

NOTE
we use dependency injection to apply dependency inversion.

1. هيكون عندي interface للـ Services فيه ال method signature.

```
Public interface IOperatingSys
{
    String RunApp();
}
```

2. هيكون عندي Services تعمل implement للـ interface دا اللي هم ال Services عندي
   - windowsOSService.
   - MacOSService.
   - LinuxOSService

```
Public Class windowsOSService : IOperatingSys
{
  Public String RunApp()
  {
     return "Running from Windows";
  }
```

نفس الطريقة باقي ال classes [Linux, Mac] هيعملوا implement للـ interface

3. هروح في ال Controller اللي هعمل فيه inject وأكريت Private field من interface و أعمله inject بال constructor

```
[APIController]
Public class DevelopmentController : ControllerBase
{  Private readonly IOperatingSys _ioperatingSys; // Private Field.
   Public DevelopmentController (IOperatingSys ios)
    {
       _ioperatingSys = ios;
    }
```

⇒ Cont.

```
[HttpGet]
Public IActionResult Run()
{
    var message = _ioperatingSys.RunAPP();
    return Ok(message);
}
}
```

[4] هنروح في الـ Program.cs اعمل registeration لـ Service بتاعتي جوا الـ Service collection

in Program.cs

```
builder.Services.AddScoped<IoperatingSys, windowsOs Service>();
```

interface بنحط الـ ← ممكن نحط أي Service أو class يعني من اللي بيـ implement الـ interface دا

-- Service Lifetime --

عندي ٣ أنواع من الـ lifetime هقدر نستخدمهم في الوقت اللي اضايقلي، Service بتاعتي جوا الـ Service collection لـ registeration

[1] Transient (Default)

↳ A new instance of the Service is created each time it is requested from the Container

← في كل موقع بحتاج فيها الـ Service دي، هيتكريت instance جديد بكرة أو تاني ولا مكان

↳ When you have a light weight and Stateless Service that does not maintain any state between requests, such as a logger or a helper class.

← هقدر استخدمها لو عندي Stateless أو lightweight Service ماليش state بتاعتها.

[2] Scoped

↳ a single instance of Service is created for each Scope. A scope typically corresponds to a web request in a web application.

← كل request بيجي عندي فيه instance واحد بس ← يعني لو الـ request بتاعي بيروح على مكانين اتنين بستخدمهم نفس النوع من الـ Service الـ dependency injection هيروح نفس الـ instance من الـ Service في على المكانين دول

الـ Scope بتاع الـ Scoped lifetime يبدأ من أول ما يوصل على الـ Server) request طول ما الـ Server بيبدأ يرجع الـ response بعد كدا الـ instance هيلوه متعرد تاني

Cont. →

Middleware:

↳ refere to Software Component that act as intermediaries between the web Server and your appication (client). these components are arranged in apipeline, and each one has the ability to:

يقدر يعدل في ال request أو ال response → Intercept incoming requests and outgoing responses: this allows Middleware to examine and modify the request data or response content before it reaches the final destination.

وكمان هو قادر يغير في جزء من ال response أو ال request

→ Perfrom various tasks: Middle ware Can handle a wide range of functionalities, including:
- Authentication
- Authorization
- logging
- Error handling

ways to add Middlewares to Your Code:

← نضيف ال Middleware في ال Program.cs في الحالة دي الترتيب مش هيفرق

1 → app.Use(); → Async arrow func. هنبعتله

```
Ex ↳ app.Use( async ( context , next) =>
    {
        Func. هنا هنكتب ال body بتاع ال
    });
```

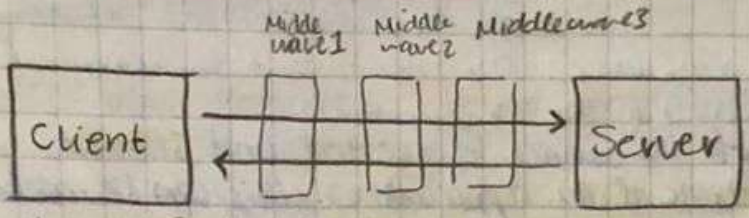
- context → HttpContext اللي جواه كل معلومات ال request
- next → ال delegation المسئول انه يباصي ال request دا لل Middle ware اللي بعده

2 → CustomMiddlewareFile — هنفصل ال Middleware عن ال Program.cs

```
Ex ↳
Public class CustomMiddleware
{
    Private read only RequestDelegate _next;

    Public CustomMiddleware( RequestDelegate next )
    {
        _next = next;
    }
    Public async Task InvokeAsync( HttpContext context )
    {
        هنا بنكتب اللوجيك
        await _next(context);
    }
}
```

--Middleware--



← ال request وهو راجع من ال Client لل Server بيعدي على حاجة اسمها Pipeline وبردو ال response من ال Server لل Client نفس الكلام

← ال Pipeline ← مجموعة من ال Middlewares اللي بيعدي عليها ال request وال response

Middleware1

```
// Some logic
Next();
// Somelogic
```

اللوجيك دا بينفذ في البداية قبل أصلاً ما ال request يوصل لل endPoint المسئولة إنها تتعامل مع ال request دا

بعد ما بينفذ الجزء دا من الكود بيروح للـ delegation اللي هو ال Next() ←

وبعد كدا بيبدأ يبحث في ال request دا لل Middleware اللي بعده.

دا معناه إن الجزء اللكتوب في أول Middleware بعد ال Next() هيتنفذ مع ال request نفسه

Middleware2

```
// Somelogic
Next()
```

ال Middleware دا هيروح ينفذ الجزء بتاع اللوجيك اللي في الأول وبعد كدا يروح على ال Next() تروح لل request اللي بعده Middleware3

Middleware3

```
// Somelogic
Next()
// Somelogic
```

نفس الكلام هينفذ اللوجيك اللي في أول Middleware وبعدين يروح لل Next ، ولأن دا آخر Middleware خلاص عندي فهو بعد ال request خلاص لآخر endPoint هتتعامل مع ال request ← فال Server هيبدأ يجهز ال response خلاص

ال response وهو راجع هيعدي على ال Middlewares بس بالعكس ، هيبدأ من ال Middleware اللي تالت

هينفذ ال Somelogic اللي تحت وبعدين يروح على ال Next() ويعدي ال Middleware التانية يروح لل Next() اللي جواها

بعدين ال Middleware الأولى تنفذ بردو ال Somelogic اللي تحت وترجع لل Next فوصل ال response لل Client

Note

Middlewares: ناخد بالنا من ترتيبهم

cont. ↳ when you want to optimize Performance by reusing a single instance of the Service across all Objects that are Created within a particular scope.

3) Singleton

↳ a Single instance of the Service is created and Shared throughout the lifetime of the application. this can be useful for Services that are expensive to create or for Services that need to maintain State.

نفس الـ instance من الـ Service هو اللي هيتكريت مع كل الـ requests إلا في حالة لو قفلت الـ application هيتكريت واحد جديد لكله بردو.

Be aware that a Single instance of Service will be created and Shared throughout the life time of the application.

__Keyed Services__

- .AddScoped
- .AddTransient
- .AddSingleton

بيضيفوا واحد implement للـ interface

```
builder.Services. AddKeyedSingleton
                . AddKeyedTransient
                . Add Keyed Scoped
```

هنقدر نكتب الأكثر من implementation لنفس الـ interface.
هيفرق بينهم باستخدام الـ **Key**

Ex

زي المثال اللي فات بتاع الـ IOperatingSys

Key ← في الـ program.cs

```
builder.Services.AddKeyedTransient<IOperatingSys , windowsOsService>("windows");
builder.Services.AddKeyedTransient<IOperatingSys, MacOSService>("MacOS");
```

↳ key.

← في الميثود بتاعت الـ EndPoint.

```
[HttpGet]
Public IActionResult Run([FromKeyedServices("windows")] IOperatingSys windowsService,
    [FromKeyedServices("MacOS")] IOperatingSys macOSService)
{
    ~
    return Ok();
}
```

عملنا الـ dependency inject في الميثود نفسها
مش في الـ Ctor بتاع الـ Controller.

← هنقدر نستخدم أكثر من class بيعمل implement لنفس الـ interface

لازم علشان الـ Middleware ما يشتغل نروح نناديه عليه في الـ Program.cs

```
app.UseMiddleware<CustomMiddleware>();
```
بنفس اسم الـ Middleware

طيب لو عندي أكتر من Middleware، هل ما كل مرة هرجع أعمل Register عليها وأعملها use لـ جوا الـ Program.cs هعمل Extension Method

الـ Program الي عندي وبعدين أناديه على الـ Middleware جوا الـ Program

```
Public static class CustomMiddlewaresExtensions
{
  Public Static IApplicationBuilder UseCustomMiddleware (this
                    ApplicationBuilder builder)
    {
      return builder.UseMiddleware<CustomMiddleware>();
    }
}
```

in Program.

```
app.UseCustomMiddleware();
```

Notes

ControllerName ⇒ مع.
DomainName ⇒ مفرد