INTERNET

We talk about web application when we expose a process to communications with other processes,
hosted on the same computer or on other computers, through network sockets and drivers.

The default communication platform is the INTERNET (INTER NETwork): a network that connects local networks.

Often the resources are exchange with a client-server logic:
- **server**: the node that provides the resource
- **client**: the node that asks and gets the resource

Usually a machine is dedicated to a single purpose, and they are equipped differently.
Usually an operating system is dedicated to a single purpose and they have different capabilities.

An Exception are the **Peer-To-Peer** systems, where a computer can be both client and server, and a big network of such nodes is created to exchange data in a more distributed way.

In order to communicate, there is a stack of protocols known as INTERNET PROTOCOL SUITE:

```
|-------------------------------------------------------------------------------|
| Security Layer (TSL Protocol, SSL Protocol, ...)                              |
|-------------------------------------------------------------------------------|
| Application Layer (HTTP Protocol, FTP Protocol, MQTT Protocol, DNS Protocol, ...) |
|-------------------------------------------------------------------------------|
| Transport Layer (UDP Protocol, TPC Protocol, ...)                             |
|-------------------------------------------------------------------------------|
| Internet Layer (IP Protocol)                                                  |
|-------------------------------------------------------------------------------|
| Link Layer (Ethernet Protocol, WiFi Protocol, ...)                            |
|-------------------------------------------------------------------------------|
| Physical/Hardware Layer (Ethernet, WiFi, ...)                                 |
|-------------------------------------------------------------------------------|
```

PHYSICAL/HARDWARE LAYER
It physically connects the Devices

LINK LAYER
It's the "language" spoken by a certain hardware, connecting the nodes of a single local network.

INTERNET LAYER
It connects together every public node of every network exposed internationally, using the IP (Internet Protocol).
Every node of the network has an IP address.
IPv4: 0-255.0-255.0-255.0-255 (es: 192.173.168.3)
IPv6: X.X.X.X.X.X
    IPv6 is not used because they created the NAT technology to "multiply" the IPv4 possibilities)
command `ping <ip>`: I can see if that IP is used and is reachable.
An IP is registered through international official providers.
`127.0.0.1`: is a special IP: in any computer I use, this IP is the local IP of that computer.
    => There is no computer with that public IP.
localhost: alias for 127.0.0.1.
`192.168.x.x`: is a special IP range used for local networks.
    Usually 192.168.0-1.0-1 is used for the router, that is the device by which all communications
    enter and exit the local network.

TRANSPORT LAYER
It's the protocol for the base communication.
It needs a **PORT** (an int number up to ~65.000 (2^16)).

A computer can communicate with other nodes using a SOCKET, which is a piece of hardware that can receive data, has a little buffer, etc.
Every port is bounded to a socket.

UDP
- message-oriented
- no guarantee of order, some messages may be lost, some messages could be duplicated
- => unreliable
- => faster
- (usually used from streaming)

TCP
- connection-oriented
- messages cannot be lost and are received in order
- => reliable
- => slower
- (usually used for everything else)

Application Layer
A Protocol to organize/communicate the actual content.

HTTP (Hyper-Text-Transfer-Protocol)
Based on cycles of communication between two nodes, based on TCP:

client: makes a **REQUEST**, which is composed by:
- a URL: a string composed by the following parts:
    <protocol>://<host>:<port>/<path(segment1/segment2/...)>?<query(key1=value1&key2=value2&...)>#<anchor>
    http://www.mysite.com/css/my-styles.css
    https://api.mysite.net/products/51/?name=carta&page=4
    (Browsers use default ports if they're not explicited: HTTP: 80; HTTPS: 443)
    The chars of the URL must be encoded: there are invalid chars like " ", "?", "#".
    Reference: https://www.w3schools.com/tags/ref_urlencode.ASP
    Example: I want to search for products with code "#AB 123" => the query will be: "?code=%23AB%20123"
    The path doesn't have to map exactly the actual file path in the server's operating system;
    further more, many path are just virtual paths to resources that have nothing to do with files.
- a METHOD:
    - GET: to obtain a resource from the server (without requesting any edit)
    - POST: to add a resource to the server
    - PUT: to edit a resource in the server
    - DELETE: to remove a resource from the server
    - ...
- a BODY (i.e. a content; usually it's present for POST and PUT requests. The browsers disable GET calls with a body).
- a list of HEADERS (meta-information, for example: "give me the response with this language", or "this is the authentication token")

server: gives a **RESPONSE**
- a STATUS CODE:
    - 200 OK (200: OK, 201: OK Created, 204: OK but there's not content to return)
    - 300 REDIRECT (301, 307, ...):
        The request is correct, but the answer is in another URL
        (in the header "Location" there is the new address)
    - 400 CLIENT ERROR:
        400: Bad Request (generic error);
        401: Unauthorized (you're not logged in);
        403: Forbidden (you're not authorized to see this content);
        404: Not Found (the resource does not exist);
        405: Method Not Allowed (for this URL the chosen HTTP method is not allowed)
        414: URI Too Long (official limit: 2048 chars; some servers may expose lower limits)
        415: Unsupported Media Type (The request has a content formatted with a format incompatible for the server)
        422: Unprocessable Entity (the passed data cannot be processed (because they are invalid? or other reasons))
    - 500 SERVER ERROR
- a BODY (the main content returned)
- a list of HEADERS

COOKIE: it's a key-value pair to store a value on the browser; it is passed via headers.
    - the server sends a response with a header with key "Set-Cookie" and value "<cookie-key>=<cookie-value>"
    - when the browser sees the key "Set-Cookie", it stores the cookie for that domain,
        and sends it back for every following request for that domain,
        until the cookie expires or the server requires a "clean cookie".

FTP (File Transfer Protocol)
A protocol to transfer files and to see the directory tree and content of another node.

DNS (Domain Name Server)
It's a protocol to handle the associations between IPs and named URLs (i.e. domain name).
For example, I can ask to the DNS: "what's the IP of www.google.com?"
    and the DNS responds with: "8.8.8.8"
There is a DNS servers network, distributed globally.
When a DNS server does not know an association, it asks to other DNS servers.
If at the end of the chain no one knows the answer, an error is returned.
(a famous hacking attack is DNS Poisoning: I change the DNS records on a server so that a domain name points to the IPs of my machines instead of the correct ones).
A domain name is divided into levels separated by points, starting from the end:
www.google.com:
- "com": 1° level domain
- "google.com": 2° level domain
- "www.google.com": 3° level domain.
1° level domains are a fixed list, decided by international foundations.
2° level domains: you must buy one.
3° level domains: you can decide to create as many as you want under your 2°l domain.

Another typical phishing attack is creating URLs similar to famous ones:
"www.google.com-xyz.hackingsite.net" => it seems like "www.google.com" if you don't look closely.

On every computer, when you define the network parameters, you define also the IP of a DNS server.
Otherwise your computer cannot resolve domain name.

Security Layer
Protocols to ensure that the connection is secure: secret and reliable.
Old: SSL (Secure Socket Layer)
New: TLS (Transport Layer Security)
They stay on top of the below protocols, making them secure.
HTTP + TLS => HTTPS.

PACKAGE TRANSMISSION
Every layer can exchange data between 2 nodes.
The data are always split in packages.
Every package has:
- a HEADER, with the necessary information to trasmit the data and coordinate the communication
- a PAYLOAD, with the actual content.
Every layer eats a bunch of bits for its header, so the actual content of the final layer (HTTP?) is just a fraction of the actual number of bits that are physically transmitted.

MODERN WEB ARCHITECTURES
Nowadays webapps are very complex.
Often they are divided into:
- a FRONT-END APP: a SPA app that runs in the browser
- a BACK-END APP: an app that acts as a server, processes requests and retrieves resources.
Developers can be divided and specialized in just one type of applications; otherwise they are FULL-STACK developers:
- front-end

- back-end
- database.

ASP.NET DEPLOY
AN ASP.NET Core app can be deployed:
- In a container (Docker)
- On IIS (Internet Information Services), a Windows web server that can host dozens of different web sites/apps.
    In this case I access a server machine online, I put there the app artifacts and I bind them to IIS.

```
1   // In this namespace there are a bunch of classes
2   // to deal with JSON serialization / deserialization;
3   // see the JsonSerializer later in the code.
4   using System.Text.Json;
5
6   namespace S05_P02_MinimalApi2;
7
8   class Program
9   {
10      static void Main(string[] args) {
11          var builder = WebApplication.CreateBuilder(args);
12          var app = builder.Build();
13          var superheroes = new List<Superhero> {
14              new Superhero {
15                  Id = 1,
16                  Nickname = "Superman",
17                  SecretName = "Klark Kent",
18                  Assets = 10000M,
19                  Birth = new DateOnly(1970, 1, 1),
20                  CanFly = true,
21                  City = "Metropolis",
22                  Gender = GenderType.Male,
23                  Strength = 100,
24              },
25              new Superhero {
26                  Id = 2,
27                  Nickname = "Batman",
28                  SecretName = "Bruce Wayne",
29                  Assets = 1000000000M,
30                  Birth = new DateOnly(1980, 1, 1),
31                  CanFly = false,
32                  City = "Gotham City",
33                  Gender = GenderType.Male,
34                  Strength = 8,
35              },
36          };
37          app.MapGet("/", () => "Welcome to the Superhero management system!");
38          app.MapGet("/superheroes", () => {
39              // These are instances internal to the process.
40              // If I return them, ASP.NET Core automatically serializes them as JSON
41              // and put them in the response's body, following by default the
                camelCase convention
42              // for the names of the properties:
43              return superheroes;
44          });
45
46          // A POST call usually has a body with the entity
47          // I want to add to the system; in this case, a JSON
48          // with the data of the new superhero:
49          app.MapPost("/add-superhero", async (HttpRequest request) => {
50              // The body is available as stream of bytes.
51              // To read it as string I can do:
52              var bodyStream = new StreamReader(request.Body);
53              var bodyJson = await bodyStream.ReadToEndAsync();
54
55              Console.WriteLine("body: " + bodyJson);
56
57              // As I have the JSON string with the body, I can deserialize it
58              // to obtain an instance of superhero (I must set he case convention):
59              var options = new JsonSerializerOptions { PropertyNamingPolicy =
                JsonNamingPolicy.CamelCase };
60              var superhero = JsonSerializer.Deserialize<Superhero>(bodyJson, options);
61
62              // I set the Id:
63              superhero.Id = superheroes.Count > 0
64                  ? superheroes.Max(s => s.Id) + 1
65                  : 1;
66
67              // I finally add it to the internal collection:
68              superheroes.Add(superhero);
69          });
70
71          // A mapped path can be dynamic in some parts.
```

```
72          // For example, the call to get the detail of a superhero
73          // is "/superhero-detail" + any number that indicates
74          // the numeric id of a superhero.
75          // I can make a path's step dynamic using {}.
76          // In that case, the step can have whatever value.
77          // I can also specify the type I expect from that step.
78          // In this case, I want the step {id} to be an integer:
79          app.MapGet("/superhero-detail/{id:int}", (HttpRequest request, HttpResponse
            response) => {
80              // I could retrieve the value of the id processing the path:
81              var id = int.Parse(request.Path.Value.Split("/")[2]);
82              // And then retrieve the superhero.
83              // Problem: if the id is not present, this code throws an Exception:
84              // var superhero = superheroes.First(s => s.Id == id);
85              // Consequently, the exception crosses backward all the middleware,
86              // there is no middleware with a catch{}, therefore ASP.NET Core
87              // responds with a 500 error.
88              // This is semantically wrong: the non-existent id was chosen by the
                client,
89              // therefore the error must be in the 400 family, in particular, 404: NOT
                FOUND.
90              // This is the correct approach:
91              var superhero = superheroes.FirstOrDefault(s => s.Id == id);
92              if (superhero != null) {
93                  return (object)superhero;
94              } else {
95                  response.StatusCode = 404;
96                  // If the superhero is not found, I must return an empty body:
97                  return (object)"";
98              }
99          });
100
101         // Since a normal app has dozens of endpoint,
102         // I don't want to map and parse them by hand one be one.
103         // I can use the automatic binding of ASP.NET Core:
104         app.MapPut("update-superhero/{id:int}", (int id, Superhero superhero,
            HttpResponse response) => {
105             var entity = superheroes.FirstOrDefault(s => s.Id == id);
106             if (entity != null) {
107                 MapProperties(superhero, entity);
108             } else {
109                 response.StatusCode = 404;
110             }
111         });
112         app.MapDelete("delete-superhero/{id:int}", (int id) => {
113             var superhero = superheroes.FirstOrDefault(s => s.Id == id);
114             if (superhero != null) {
115                 superheroes.Remove(superhero);
116             }
117         });
118         app.Run();
119     }
120
121     static void MapProperties(Superhero from, Superhero to) {
122         to.Nickname = from.Nickname;
123         to.SecretName = from.SecretName;
124         to.CanFly = from.CanFly;
125         to.Strength = from.Strength;
126         to.Assets = from.Assets;
127         to.Gender = from.Gender;
128         to.Birth = from.Birth;
129         to.City = from.City;
130     }
131 }
132
```

```
/*
To create a simple minimal web app:
    dotnet new web -n <name>
In Visual Studio can be created with the template "ASP.NET Core Empty".

There are some JSON files with a bunch of configurations:
- appsettings
- appsettings.Development.json
- Properties/launchSettings.json
Now we don't need special configurations, so we can safely delete those files.

If I run the webapp (via VS Code's task, via terminal or via Visual Studio, ...),
the terminal will show something:
    Now listening on: http://localhost:5000
This means that a web server in up and running and is listening for HTTP requests
at the host \'localhost' and port '5000' (which is the default for ASP.NET Core apps).

The project contains a Program.cs with this code:

    var builder = WebApplication.CreateBuilder(args);
    var app = builder.Build();
    app.MapGet("/", () => "Hello World!");
    app.Run();

The creation of an ASP.NET Core is made in these steps:

1) a WebApplicationBuilder is built
        var builder = WebApplication.CreateBuilder(args);

2) the builder can be configured with dependencies, accesses to other services,
        authentication and authorization, ...
    (we will see later some way to do this)

3) the builder is used to create an instance of the webapp
    var app = builder.Build();

4) the webapp's middlware pipeline is configured
    app.MapGet("/", () => "Hello World!");
For a wide explanation and images of the pipeline:
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-7.0

5) the app is run.
    app.Run();
*/
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// I can add a middlware through the Use() method:
// app.Use((httpContext, nextMiddleware) => {
//     return nextMiddleware();
// });
// Use() accepts a delegate that is invoked at every request.
// The delegate accept as parameters a HttpContext and the next Middleware to be called.
// The delegate returns a Task, which means that it is intended to be used in an asynchronous way.
// In C# I can 'await' a Task.
// In order to use 'await' inside a delegate, it must be marked as 'async':
// Therefore, this is the template of our middlewares:
// app.Use(async (httpContext, nextMiddleware) => {
//     // code
//     await nextMiddleware();
//     // code
// });

// In particular, I create a middleware to measure the execution performance:
app.Use(async (httpContext, nextMiddleware) => {
    // Code that is executed before invoking the next middleware:
    var sw = Stopwatch.StartNew();

```

```
71      // I await the completion of the operations in the next middleware:
72      await nextMiddleware();
73
74      // Code that is executed after the next middleware completed:
75      sw.Stop();
76      // To show the Console in the integrated terminal, add this to the launch.json
        "configurations":
77      // "console": "integratedTerminal"
78      Console.WriteLine($"Time spent for the request: {sw.ElapsedTicks} ticks");
79  });
80
81  // I can create an access middleware, that blocks some calls returning an error
    status code:
82  app.Use(async(ctx, next) => {
83      Console.WriteLine("PATH: " + ctx.Request.Path.ToString());
84      if (ctx.Request.Path.ToString() == "/forbidden") {
85          ctx.Response.StatusCode = 403;
86      } else {
87          await next();
88      }
89  });
90
91  // To map the final endpoints of the middleware chain,
92  // I can use the methods that map the HTTP methods:
93  // MapGet(), MapPost(), MapPut(), MapDelete(), ...
94  app.MapGet("/", () => {
95      Console.WriteLine("MapGet");
96      return "Hello World!";
97  });
98
99  // If I want a unusual HTTP method, I can map a general endpoint
100 // and check the method inside the delegate:
101 app.Map("to-be-called-with-head-method", (HttpContext ctx) => {
102     Console.WriteLine("to-be-called-with-head-method");
103     if (ctx.Request.Method == "HEAD") {
104         return "CALL SUCCESSFUL!";
105     } else {
106         return "wrong HTTP method!";
107     }
108 });
109
110 /*
111 The several Map() methods accept as input:
112 - a string with the path
113 - a delegate that is invoked when the request matches that path.
114 The delegate is of type Delegate, i.e. it can be whatever delegate
115 with whatever input and output parameters.
116 I use this flexibility to set as input parameters instances of classes
117 that I need, for example: HttpContext, DbContext, IConfiguration, ...
118 There are a bunch of classes that are already configured by the system.
119 In the builder I can configure my custom classes (see next projects).
120 */
121
122 app.Run();
123
```

```
<!-- This is an HTML comment.
It's like a /* */ comment: multi-line and non-nestable

Usually the main page of a website is called index.html

HTML is a dialect of XML: is based on nested tags with attributes
The list of tags is decided by the international associations
and depends on the version you choose.
We will work with HTML5, which has many many tags that helps to define
the SEMANTICS of the sections of the page.
(We don't use different tags to get different graphic results:
the graphic is responsibility of the CSS).

HTML was born before XML.
Then they saw that the "<>" format was very powerful to express structured data,
so they expandend HTML into XML (eXtensible Markup Language)
"Mario": is a simple string
"<name>Mario</name>" => is a string with markup
'extensible' because I can create my own tags in XML,
there isn't a fixed list of tags.

HTML allows non-closed tag, for example <link href="style.css">, or <br>.
They created XHTML to indicate an HTML that is XML-compliant (no not-closed tags).

-->
<!-- first tag to declare the type of document -->
<!DOCTYPE html>
<!-- root tag-->
<html>
    <!-- the <head> section contains metadata for the page -->
    <head>
        <meta charset="UTF-8">
        <title>My first web site</title>
        <!-- here you import other files like CSS style files, JS script files, etc. -->
    </head>

    <!-- <body> contains the content you see on the webpage -->
    <body>

        <!-- where you put the heading content of the section -->
        <header>
            <!-- navigation tag is where I put the navigation links and info -->
            <nav>
                <!-- now I have a list of menu voice, and it's and Unordered List => <ul> -->
                <!-- If I want a list with numbers of letters, I can use <ol> (Ordered List) -->
                <ul>
```

```
47                      <!-- every item of the list if a List Item <li> -->
48                      <li>
49                          <!-- an ANCHOR <a> is a link to another page
                    (via attribute href) -->
50                          <a href="index.html">HOME</a>
51                      </li>
52                      <li>
53                          <a href="contact-us.html">CONTACT US</a>
54                      </li>
55                  </ul>
56              </nav>
57          </header>
58
59          <!-- the main content of the section -->
60          <main>
61              <!-- to write text, HTML has:
62                  - many heading tags: <h1>, <h2>, <h3>... (<h1> is the
                    most important,
63                    you increase the number when the heading is less
                    important)
64                    (again, it's not a matter of how big it is rendered
                     on the page,
65                    it's a matter of semantic importance)
66                  - a <p> tag (paragraph)-->
67              <h1>MY PUB!</h1>
68              <h2>WELCOME!</h2>
69
70              <!-- I can divide an area in separate sections, here I
                  want:
71              - "burgers of the week" section
72              - "full menu" section
73              -->
74              <section>
75                  <h3>Burgers of the week:</h3>
76                  <!-- I can use <article> for a unit of content -->
77                  <article>
78                      <h4>Veggie Top</h4>
79                      <!-- <p> for a paragraph of text -->
80                      <p>A veggie salad with mango cream, crispy onions
                    and cucumber</p>
81
82                      <!-- <img> to include an image.
83                          The 'src' attribute contains the link to the
                    image file (a local link or a link to an external
                    site)
84                          The 'alt' attribute defines the text to be
                    shown if the image is not available
85                          <img> is always an auto-closed tag.
86                      -->
87                      <img src="/images/veggie.png" alt="veggie burger" /
                    >
88
89                      <!-- I could put measures on the images: -->
```

```
90                      <!-- <img src="/images/veggie.png" alt="veggie
                    burger" width="300" height="200" /> -->
91                      <!-- but it's a bad approach: I'm mixing semantics/
                    structure and aesthetic. -->
92                      <!-- We better decide the dimensions in the CSS. --
                    >
93
94                      <!-- I can enclose the <img> into a <picture> tag
                    if I want a complete image
95                           (with a caption, diffent sources for different
                    page sizes, ecc.) -->
96                      <!-- <picture>
97                           <img src="/images/veggie-big.png" />
98                           <source media="(max-width: 300px)"
                    srcset="veggie-small.png" />
99                           <figcaption>try it!</figcaption>
100                     </picture> -->
101                 </article>
102
103                 <article>
104                     <h4>Brutal Bacon</h4>
105                     <p>A lake of delicious bacon on top of you burger,
                    with cheddar, BBQ sauce and salad.</p>
106                     <img src="/images/brutal-bacon.png" alt="brutal
                    bacon" />
107                 </article>
108             </section>
109
110             <section>
111                 <h3>Look at the full menu!</h3>
112
113                 <article>
114                     <h4>Merlin</h4>
115                     <!-- to create a general division, I use the tag
                    <div>: -->
116                     <div>
117                         <img src="images/merlin.png" />
118                     </div>
119                     <div>
120                         <!-- blank characters (even multiple ones) are
                    always considered as a single white space.
121                          If I want to go to a new line, this does not
                    work: -->
122                          <!-- <p>Taste the power of a sorcerer!
123                              Bread, chicken burger, eggplant, tomato.</
                    p>
124                          <p>Price: 12.00 €</p> -->
125                          <!-- I can use the <br /> tag: -->
126                          <p>Taste the power of a sorcerer! <br />
127                              Bread, chicken burger, eggplant, tomato,
                    cheddar.</p>
128                          <p>Price: 12.00 €</p>
129                     </div>
```

```
130                 </article>
131
132                 <article>
133                     <h4>Semola</h4>
134                     <div>
135                         <img src="images/semola.png" />
136                     </div>
137                     <div>
138                         <p>The King of the Burger! <br />
139                             Bread, double burger (200 g), salad,
                      tomato, ketchup, cucumber, eggs.</p>
140                         <p>Price: 16.00 €</p>
141                     </div>
142                 </article>
143
144                 <article>
145                     <h4>Veggie Top</h4>
146                     <div>
147                         <img src="images/veggie.png" />
148                     </div>
149                     <div>
150                         <p>A veggie salad with mango cream, crispy
                      onions and cucumber</p>
151                         <p>Price: 10.00 €</p>
152                     </div>
153                 </article>
154
155                 <article>
156                     <h4>Brutal Bacon</h4>
157                     <div>
158                         <img src="images/brutal-bacon.png" />
159                     </div>
160                     <div>
161                         <p>A lake of delicious bacon on top of you
                      burger, with cheddar, BBQ sauce and salad.</p>
162                         <p>Price: 15.00 €</p>
163                     </div>
164                 </article>
165
166             </section>
167         </main>
168
169         <!-- for information that you usually put at the end (contacts,
               vat number, ...) -->
170         <footer>
171             <div>
172                 <!-- <span> is a in-line subsection of the container --
                  >
173                 <span>Copyright © 2022 MyPub</span>
174                 <span>vat: 123123123123</span>
175                 <span>country: Italy</span>
176             </div>
177             <div>
```

```
178                 <!--
179                     If a part of a text is important, I enclose it in a ↵
                       tag <strong> (old tag <b> (bold))
180                     If a part of a text has emphasis, I enclose it in a ↵
                       <em> tag (old <i> (italic))
181                 -->
182                 <span>This site uses cookies. We adhere to the          ↵
                      <strong>GDPR</strong>: we value <em>very much</em>    ↵
                      your privacy!</span>
183             </div>
184         </footer>
185     </body>
186 </html>
187
```

```
Compilazione avviata...
1>------ Inizio compilazione: Progetto: S05_P06_HTML1, Configurazione: Debug
   Any CPU ------
1>S05_P06_HTML1 -> C:\Users\marco\Documents\GitHub\ires-2022\kraus\05-web-
  dynamic\S05_P06_HTML1\bin\Debug\net7.0\S05_P06_HTML1.dll
========== Compilazione: 1 completato/i, 0 non riuscito/i, 0 aggiornato/i, 0
   ignorato/i ==========
========== Trascorso 00:08,147 ==========
```