Analysis, Design and Software Architecture (BDSA)

Session 4

Sven Peldszus

Sven

- B.Sc./M.Sc. from TU Darmstadt (Electrical Engineering and Information Technology)
- PhD in Software Engineering from University of Koblenz-Landau
- PostDoc at Ruhr University Bochum
- Starting 01.09, Assist. Professor in Software Engineering at ITU

- Favorite topics: **Software Engineering** for Secure Systems

- Looks foward to **give the 2nd half of the BDSA lectures**;

  Will say hello next week! **-> Actually, only today...**

Plan for today

- First hour: Feedback/Advices, Integration Patterns, and HTTP Intro
- Second hour: HTTP recap and Intro to Minimal Web API applications
- Third hour: Properties of ASP.Net Minimal Web applications
- Fourth hour: Deployment of Web applications

Feedback/Advices

(Mainly general feedback, partially based on last year)

How is it going?

- You are working on the projects, perhaps use a bit more of time.
- Merge branches to main -> At least at the end of the day.
- Read precisely! Interface spec is fixed.

Feedback

- Why do we need to refactor?
  - In this course's project work, we are simulating that you incrementally gain knowledge in a domain.
  - The more you know about possible solutions, the more they look different.
  - Constant refactoring is the core of *agile* development processes.
- No matter what you do, do it KISS!

Feedback: Content of Repositories

- Remember to add a `.gitignore` file to your projects
  - Your repositories should not be littered with artifacts that are not part of your *Chirp!* systems! `.DS_Store` files, etc.
  - **OBS**: `.gitignore` files have to be placed in root of your repositories (`.idea` dir has to be there too, i.e., not in a subfolder)!
- Add an `.editorconfig` file, so that you apply all the same code conventions.
- Remember to clean up
  - Delete unused branches after they were merged to main.

Feedback: Use semantic versioning for your releases

- Tags and versions that look like 1.0.1 or v2.0.2 are good version numbers.
- Version numbers 0.5, v1. 2 are not

A good filename for a release contains the name of the project, the release version, and the target platform. That is,

- `Chirp.CLI-v.0.0.4-linux-x64.zip`, `Chirp-0.9.2-win-x64.zip`, or `chirp-v1.0.1-osx-arm64.zip` are all good release names.
- `linux-x64.tar.gz`, `Chirp-test_release-osx-x64.tar.gz`, `App-v0.1.8-macos.zip`, `v1.1.13`, etc. are all less good release names.

See: Semantic Versioning Specification , Image source: I. J. *emVer*

Feedback: Release what and how?

- Read precisely: publish only ZIP files for all executables, no other compressed archive format (this is only for this course)
- Release only compressed executables (single-file, platform dependent), **no other artifacts**
  - That is, no CSV files, no `*.sln` files , etc.
  - **single-file**: Release contains exactly *one* file, if you have more than that, you publish wrongly.
  - **platform dependent**: Not the entire .NET runtime bundled!
  - That is, likely your releases will be 0.5MB in size (ca. 500KB) not ca. 63MB
- Most important: **Only working executables are released**!
  - “ [Software Quality…] degree to which a software product satisfies stated and implied needs when used under specified conditions
    Source: International Organization for Standardization *Systems and software engineering — ftware Quality Requirements and Evaluation (SQuaRE) — System and software quality* ”

Architecture: Integration Patterns

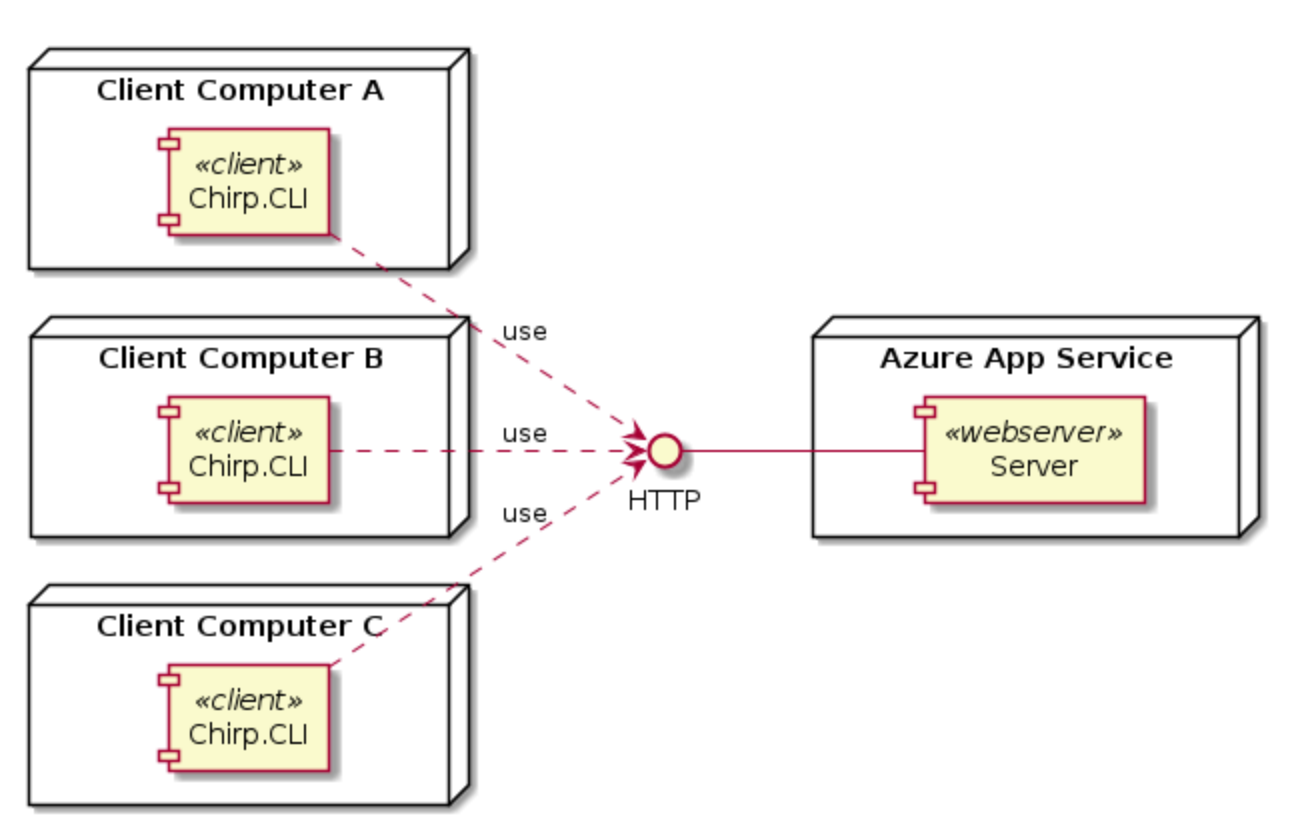
- Why is our *Chirp!* application not cool yet?
  - It is not a social network, all cheeps are stored per user on the computer that runs the program.
- How can we modify our *Chirp!* application so that it becomes a social network, i.e., so that we can read each other's cheeps?

Task: Experiment with a distributed *Chirp!* CLI system

- Download a new *Chirp!* CLI client for your operating system: Linux (and WSL), Windows ARM, Windows x64, MacOS ARM, MacOS x64
  - On MacOS/WSL/Linux make the file executable: `chmod u+x ~/Downloads/chirp-<os>`
- Write a cheep from the terminal with the new client
  - **OBS**: the usual netiquette applies, i.e., write only messages that you would utter at your grandma's coffee table
  - **OBS**: For now, write short cheeps, i.e., a single sentence.
  - `cd ~/Downloads`
  - `chirp-<os>[.exe] cheep "Hello!"` (Only Windows users have to add the `.exe` suffix)
- Now, read yours and your peer's cheeps on the terminal, e.g., via:
  - `chirp-<os>[.exe] read`
- Can you see cheeps that were not written by you? This is the goal for this lecture and this week's project work.

How does this work?

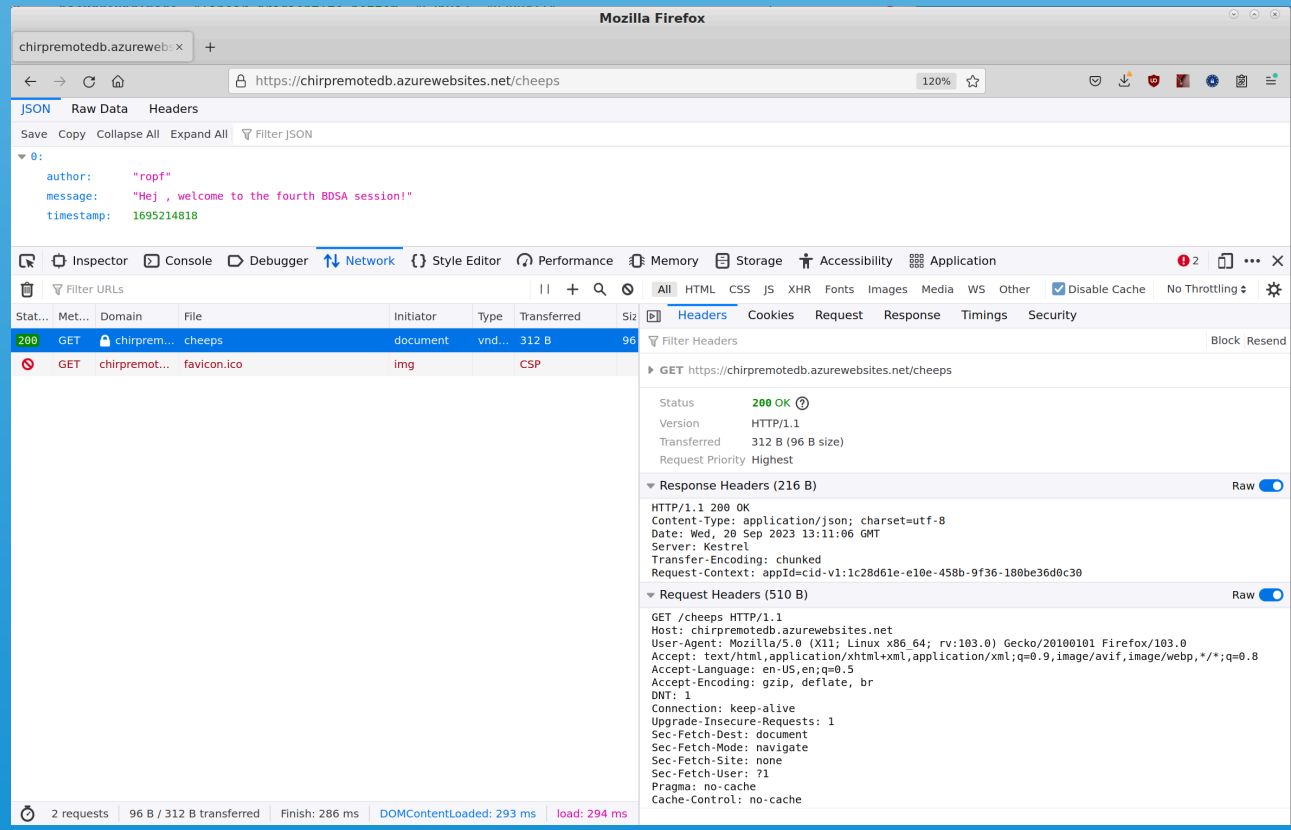
How can the client that I have locally on my computer display cheeps that were written by my peers?



10 minutes

Task: Send and Inspect HTTP Requests and Responses in the Browser

- Download the minmal example and run it `dotnet run`
- Open your web browser
- Open its development tools (press F12 on Firefox, Edge, ...)
- Switch to the `Network` tab
- Enter the URL http://localhost:5012 in the address bar and hit return
- In the `Network` tab, click on the `GET` (blue document icon in Safari/Edge) request to the domain `localhost:5012`
- Inspect the headers and bodies of the sent request and the received response.
- Discuss with your neighbors what you see.



5 minutes

Task: Send and Inspect HTTP Requests and Responses via `curl`

- Open your terminal
- Send an HTTP request with the `curl` command

```
curl -v --header "accept: application/json" http://localhost:5012/public
```

- Inspect the output and discuss it with your neighbors.
- If in doubt about the meaning of the provided options, read the corresponding documentation via `man curl` in the terminal.
- Discuss with your neighbors the output of the `curl` command and compare it to what you saw in the browser's network tool in the previous task.

15 Minute Break

Recap: Structure of HTTP Requests and Responses

“
- A start-line
- An optional set of HTTP headers
- A blank line indicating all meta-information for the request has been sent.
- An optional body containing data associated with the request.

Source and image source: Mozilla Developer Network ”

Recap: HTTP headers

Header Field	Description	Examples

Accept	I understand	`text/plain`
		`application/json`
		`application/xml`

Content-Type	I'm sending	`application/x-www-form-urlencoded` `application/json; charset=utf-8`
Authorization	Who I am	`Bearer ey`

Recap: Kinds of HTTP Requests

Actions work on *resources*, i.e., the "thing" specified by the URL

Action	Verb
Request/Retrieve	`GET`
Like `GET` without body	`HEAD`
Submit	`POST`
Update/Replace	`PUT`
Delete	`DELETE`
Update/Modify	`PATCH`

More details on HTTP verbs: here

Recap: Kinds of HTTP Responses, Status Codes

Codes	Meaning
100 – 199	Informational responses
200 – 299	Successful responses
300 – 399	Redirection messages
400 – 499	Client error responses
500 – 599	Server error responses

See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Recap: Kinds of HTTP Responses, Status Codes

Code	Meaning	Code	Meaning
200	OK	409	Conflict
201	Created	415	Unsupported Media Type
202	Accepted	418	I’m a Teapot
204	No Content	422	Unprocessable Entity
301	Moved Permanently	500	Internal Server Error
302	Found (Previously "Moved temporarily")	501	Not Implemented
307	Temporary Redirect	503	Service Unavailable
308	Permanent Redirect		
400	Bad Request		
401	Unauthorized		
403	Forbidden		
404	Not Found		

How to create a Minimal Web API?

Look at .Net application templates:

```
dotnet new list
These templates matched your input:

Template Name                                 Short Name          Language    Tags
--------------------------------------------  ------------------  ----------  --------------------------------
ASP.NET Core Empty                            web                 [C#],F#     Web/Empty
ASP.NET Core gRPC Service                     grpc                [C#]        Web/gRPC
ASP.NET Core Web API                          webapi              [C#],F#     Web/WebAPI
ASP.NET Core Web App                          webapp,razor        [C#]        Web/MVC/Razor Pages
ASP.NET Core Web App (Model-View-Controller)  mvc                 [C#],F#     Web/MVC
ASP.NET Core with Angular                     angular             [C#]        Web/MVC/SPA
ASP.NET Core with React.js                    react               [C#]        Web/MVC/SPA
Blazor Server App                             blazorserver        [C#]        Web/Blazor
Blazor Server App Empty                       blazorserver-empty  [C#]        Web/Blazor/Empty
Blazor WebAssembly App                        blazorwasm          [C#]        Web/Blazor/WebAssembly/PWA
Blazor WebAssembly App Empty                  blazorwasm-empty    [C#]        Web/Blazor/WebAssembly/PWA/Empty
Class Library                                 classlib            [C#],F#,VB  Common/Library
Console App                                   console             [C#],F#,VB  Common/Console
dotnet gitignore file                         gitignore                       Config
Dotnet local tool manifest file               tool-manifest                   Config
EditorConfig file                             editorconfig                    Config
global.json file                              globaljson                      Config
MSBuild Directory.Build.props file            buildprops                      MSBuild/props
MSBuild Directory.Build.targets file          buildtargets                    MSBuild/props
MSTest Test Project                           mstest              [C#],F#,VB  Test/MSTest
MVC ViewImports                               viewimports         [C#]        Web/ASP.NET
MVC ViewStart                                 viewstart           [C#]        Web/ASP.NET
NuGet Config                                  nugetconfig                     Config
NUnit 3 Test Item                             nunit-test          [C#],F#,VB  Test/NUnit
NUnit 3 Test Project                          nunit               [C#],F#,VB  Test/NUnit
Protocol Buffer File                          proto                           Web/gRPC
Razor Class Library                           razorclasslib       [C#]        Web/Razor/Library
Razor Component                               razorcomponent      [C#]        Web/ASP.NET
Razor Page                                    page                [C#]        Web/ASP.NET
Solution File                                 sln,solution                    Solution
                                              webconfig                       Config
                                              worker              [C#],F#     Common/Worker/Web
                                              xunit               [C#],F#,VB  Test/xUnit
```

Task: Create an ASP.NET Minimal Web API

- Create a new minimal web api project:

```
dotnet new web -o Chirp.CSVDBService
```

- Thereafter, change directory to `cd Chirp.CSVDBService`
- Inspect the source code that was generated, especially `Program.cs`
- Build and run the project via `dotnet run`
  - Note the URL that is listed after `Now listening on:`
- In another terminal, send an HTTP `GET` request with `curl` to the application. (e.g., `curl -v http://localhost:5012`, or whichever other URL is listed after `Now listening on:`)
- Study and interpret the output with your neighbors.
- Repeat the process with the development tools of your browser
- Once done, stop the web service by pressing `CTRL+C`.

Inspecting Example HTTP Request and Response

```
$ curl -v localhost:5012/public
* Host localhost:5012 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:5012...
* Connected to localhost (::1) port 5012
* using HTTP/1.x
> GET /public HTTP/1.1
> Host: localhost:5012
> User-Agent: curl/8.12.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Date: Tue, 16 Sep 2025 17:23:42 GMT
< Server: Kestrel
< Transfer-Encoding: chunked
<
Hello, BDSA students! Welcome to the course! I hope you had a good summer.
* Connection #0 to host localhost left intact
```

Building Web APIs

The application in the previous example returns a string in the body of the HTTP message, see the output of `curl`.

Web applications may return other kinds of data too.

- Web APIs often return data that is encoded as JSON.
- Web applications return most often HTML.

15 minutes

Task: Create an ASP.Net Minimal Web API

- Replace the `Program.cs` from the previous task with the following content:

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

app.MapGet("/cheeps", () => new Cheep("me", "Hej!", 1684229348));

app.Run();

public record Cheep(string Author, string Message, long Timestamp);
```

  and run the application `dotnet run`
- In another terminal, send an HTTP `GET` request with `curl` to the new `cheeps` endpoint (e.g., `curl http://localhost:5012/cheeps`)
- What is the `Content-Type` of the HTTP response?

Anonymous functions in C♯

- What was that???

```
() => new Cheep("me", "Hej!", 1684229348)
```

- Anonymous functions [also called lambdas] in C♯:
  - *Expression lambda* that has an expression as its body:

```
(input-parameters) => expression
```

  - *Statement lambda* that has a statement block as its body:

```
(input-parameters) => { <sequence-of-statements> }
```

Source: [Microsoft Documentation]

How would this look in normal code?

- Anonymous function

```
app.MapGet("/cheeps", () => new Cheep("me", "Hej!", 1684229348));
```

- Pass function as delegate

```
static Cheep getCheep()
{
    return new Cheep("me", "Hej!", 1684229348);
}

app.MapGet("/cheeps", getCheep);
```

ASP.NET Automatically Serializes Objects into JSON for Transport over HTTP

The `Cheep` object from the following code

```
app.MapGet("/cheeps", () => new Cheep("me", "Hej!", 1684229348));
```

gets automatically serialized into JSON by the ASP.NET Core framework:

```
{"author":"me","message":"Hej!","timestamp":1684229348}
```

The same holds for the other way around. When you send a JSON object in a message body that has all corresponding fields, a C♯ object can be created automatically from that, e.g.,

```
app.MapPost("/cheep", (Cheep cheep) => { ... });
```

Web API??? What is that?

A Web *application programming interface (API)* is an API that is accessible via a network

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
...
// This is your Web API
app.MapGet("/cheeps", () => { ... });
app.MapPost("/cheep", (Cheep cheep) => { ... });

app.Run();
```

A "regular" *application programming interface (API)* is an API that is accessible in code directly

```
namespace Chirp.SimpleDB;

public interface IDatabaseRepository<T>
{
    public IEnumerable<T> Read(int? limit = null);
    public void Store(T record);
}
```

See you on Friday

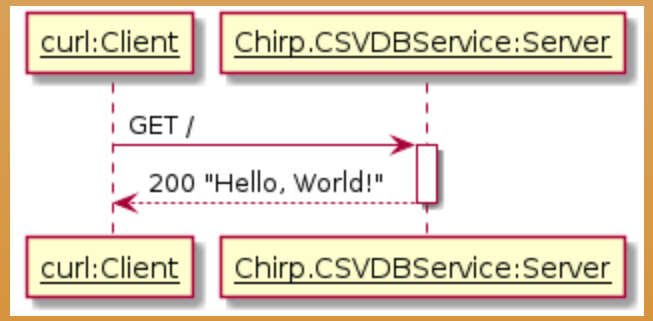
On Friday we continue with:

- Properties of ASP.Net Minimal Web applications
- Deployment of Web applications

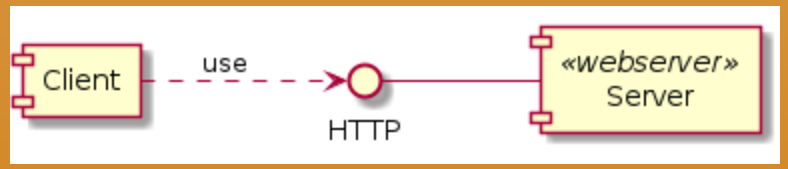
Architecture: Our First Architectural Pattern

What kind of application did we build with `Chirp.CSVDBService` and `curl` or the web browser?

- A client-server application
- Dynamic view in UML sequence diagram illustrating interactions of components



- Static view in UML component diagram illustrating dependencies of components



Sending and Receiving HTTP Requests programmatically in C#/.NET

```
using System.Net;
using System.Net.Http.Headers;
using System.Net.Http.Json;

// Create an HTTP client object
var baseURL = "http://localhost:5012";
using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
client.BaseAddress = new Uri(baseURL);

// Send an asynchronous HTTP GET request and automatically construct a Cheep object from the
// JSON object in the body of the response
var cheep = await client.GetFromJsonAsync<Cheep>("cheeps");


public record Cheep(string Author, string Message, long Timestamp);
```

10 minutes

Task: Brief Intro to Asynchronous Programming in C♯/.NET

What was this `await` keyword?

- In the directory of this lecture notes, you find an example called `AsyncAwaitHTTP`
- In its `src` directory, you find two .NET projects, one HTTP `Server` and one HTTP `Client`.
- Start the server with `cd ./AsyncAwaitHTTP/src/Server/` followed by `dotnet run` in one terminal window
- Start the client in another terminal window with `cd ./AsyncAwaitHTTP/src/Client/` followed by `dotnet run`.
- Inspect the output of the client program.
- Now try to understand what the client is doing, see `Program.cs`.
  - Discuss with your neighbors what is different in the two ways of calling the HTTP server.
  - Can you see, what makes the second example execute much faster than the first one?

Brief Intro to Asynchronous Programming in C♯/.NET

- What was this `await` keyword?
- Network calls take long, for sure way longer than method calls in your programs running on your CPU.

<marp-pre is="marp-pre" data-auto-scaling="downscale-only">

```
using HttpClient client = new(); client.BaseAddress = new Uri("http://localhost:5088"); // Sequential execution var watch = System.Diagnostics.Stopwatch.StartNew(); // first HTTP request var response = await client.GetAsync("/"); // second HTTP request response = await client.GetAsync("/"); watch.Stop(); Console.WriteLine($"Done after {watch.ElapsedMilliseconds}ms");
```

</marp-pre>

<marp-pre is="marp-pre" data-auto-scaling="downscale-only">

```
// Concurrent execution watch = System.Diagnostics.Stopwatch.StartNew(); // first HTTP request var fstRequestTask = client.GetAsync("/"); // second HTTP request var sndRequestTask = client.GetAsync("/"); var fstResponse = await fstRequestTask; var sndResponse = await sndRequestTask; watch.Stop(); Console.WriteLine($"Done after {watch.ElapsedMilliseconds}ms");
```

</marp-pre>

Note, Concurrency vs. Parallelism

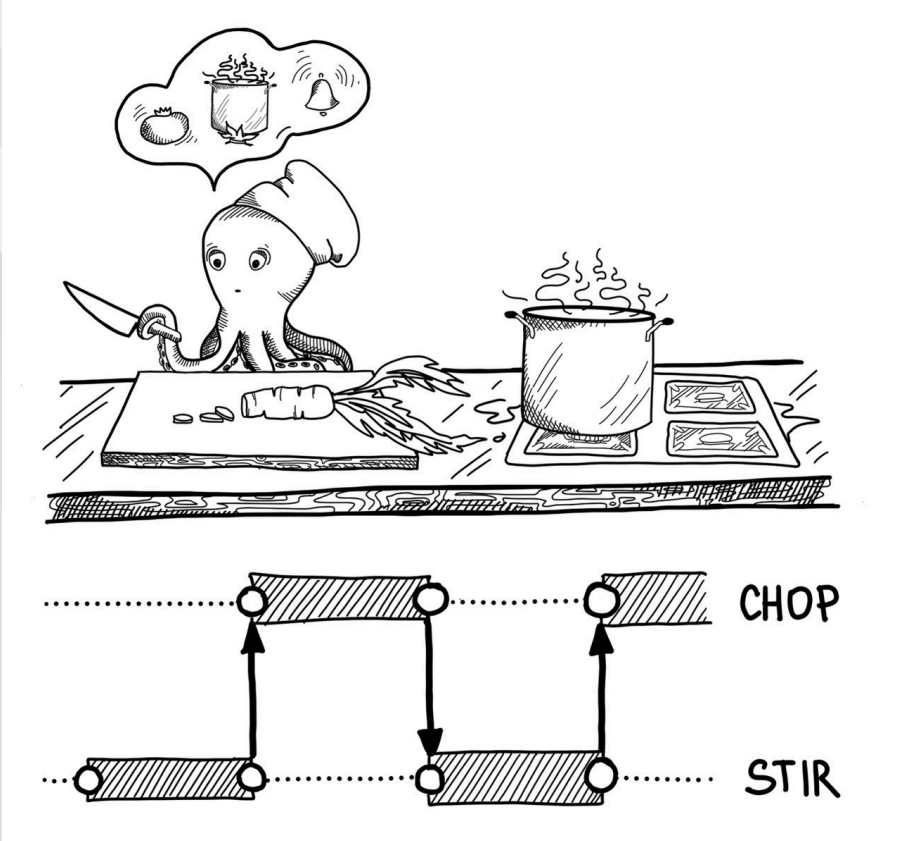
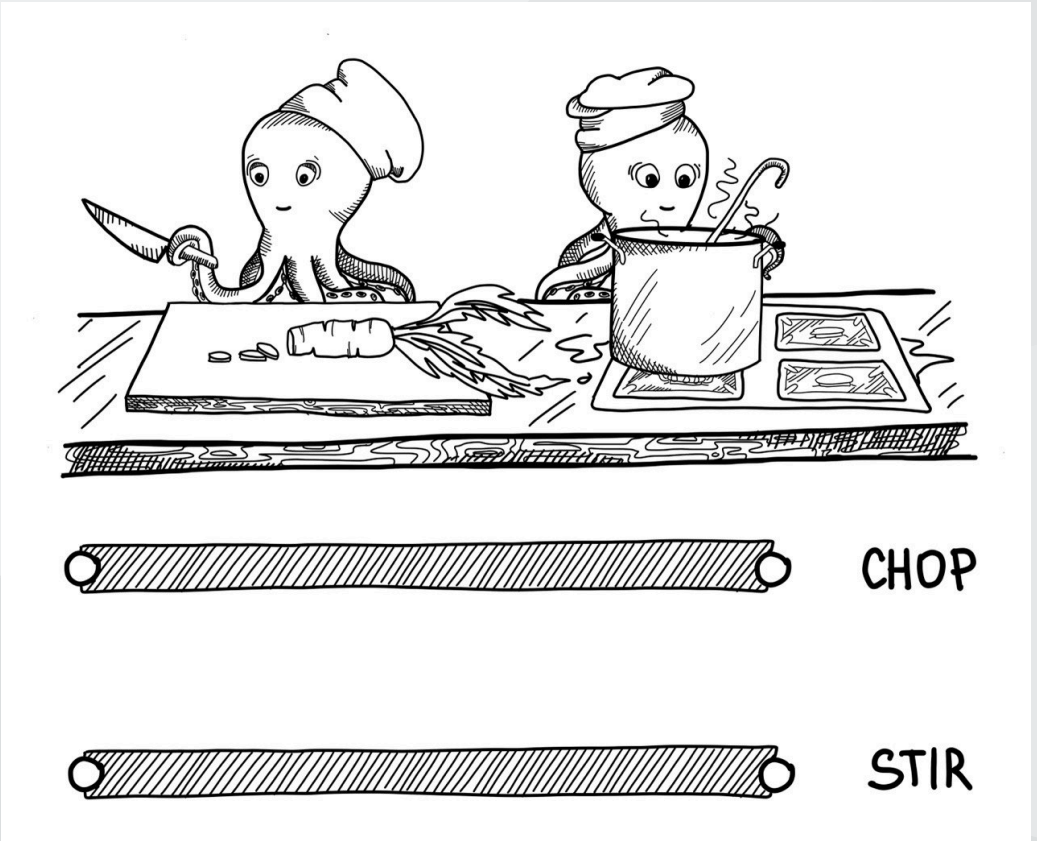
Concurrency	Parallelism
	

Image source: Kirill Bobrov *Grokking Concurrency*

Why does it matter?

Methods that are declared as `async` and that return a `Task` object in C♯ and whose results you have to `await` are comparable to concurrent functions that you start with `go` in Go in your distributed systems class.

Usually, long running IO operations are encapsulated in `async` methods, i.e., those returning a `Task` object. Network requests are typically very long running IO operations.

15 minute break

Making the CSV DB Server run on the internet

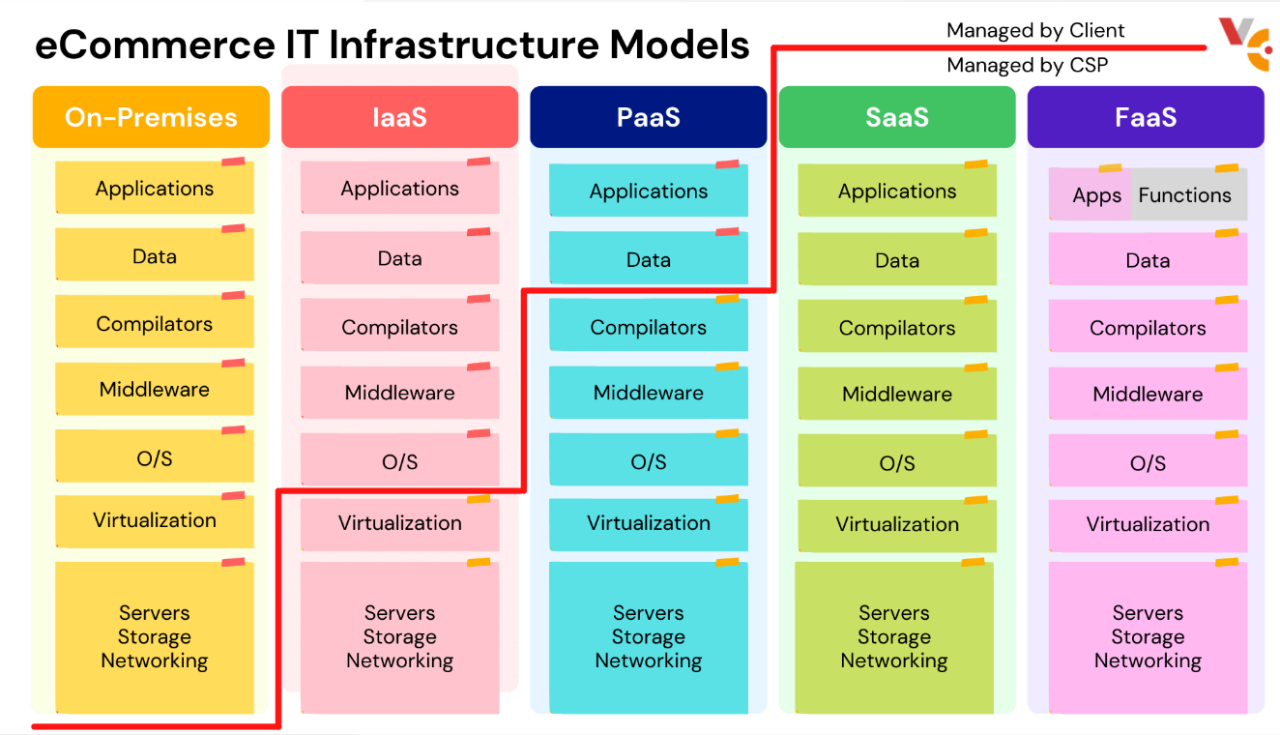
Running an HTTP server locally is good for development.

If you want to make your service accessible for everybody, i.e., if you want to have first version of *Chirp!* that is a proper social network in which you can read other's cheeps and not only your's, you have to deploy it to the "internet".

Brief introduction to hosting options.

- Functions-as-a-Service (FaaS)
- Software-as-a-Service (SaaS)
- Platform-as-a-Service (PaaS)
- Infrastructure-as-a-Service (IaaS)

For each of these options, there are many different providers. In this class, we use only Azure Service (a PaaS).
In case you are interested in IaaS, we will deepen that in next semester's elective *"DevOps, Software Evolution and Software Maintenance"*.
Image source: DevOps School



Task: Deploy to App Service Manually

- In a previous task, you created the `Chirp.CSVDBService` minimal Web API project.
- In a terminal switch to this directory `cd <path/to/Chirp.CSVDBService>`.
- Login to App Service. The following should open a browser window indicating that you are now logged in. (Use your ITU credentials to login).

```
az login
```

- After login, execute the following command. **OBS:** Replace `<no>` with the number of your group. This command deploys the current web application to Azure App Service.

```
az webapp up --sku F1 --name bdsagroup<no>chirpremotedb --os-type Linux --location westeurope --runtime DOTNETCORE:7.0
```

- Once the commands completes, it should return a URL for your web application. It is likely `https://bdsagroup<no>chirpremotedb.azurewebsites.net` (again replace `<no>` with the number of your group.)
- Now, either with your web browser or with `curl`, send an HTTP GET request to `https://bdsagroup<no>chirpremotedb.azurewebsites.net/cheeps`.
- Do you receive a JSON object that corresponds to the following `Cheep` object?

```
new Cheep("me", "Hej!", 1684229348)
```

Manually Deploy to Azure App Service

- Remember to take down the web service that you just deployed with `az webapp delete`, see `az webapp --help` in case you are in doubt of what you can do with a webapp on Azure App Service.
- What happened in the task?

Automatically Deploy to App Service

- In your project work, you will add a step to your CI pipeline on GitHub Actions that deploys to Azure App Service automatically.
- You decide if that happens on every push to main, on a tag of a new version, etc.
- Why can't you just have another CI step that looks like in the following?

```yaml
name: Deploy

on:
  push:
    branches: [ "main" ]

jobs:
  deploy:
    name: Deploy

    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
...
      - name: Deploy
        shell: bash
        run: |
          az login
          az webapp up --sku F1 --name bdsagroup<no>chirpremotedb --os-type Linux --location westeurope --runtime DOTNETCORE:7.0
          az logout
```

Automatically Deploy to App Service

In your project work, you will create a deployment workflow that will look similar to the following:

```yaml
...
  deploy:
    permissions:
      contents: none
    runs-on: ubuntu-latest
    needs: build>
    environment:
      name: 'Development'
      url: ${{ steps.deploy-to-webapp.outputs.webapp-url }}

    steps:
      - name: Download artifact from build job
        uses: actions/download-artifact@v3
        with:
          name: .net-app

      - name: Deploy to Azure Web App
        id: deploy-to-webapp
        uses: azure/webapps-deploy@v2
        with:
          app-name: ${{ env.AZURE_WEBAPP_NAME }}
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          package: ${{ env.AZURE_WEBAPP_PACKAGE_PATH }}
```

Source: GitHub Actions starter-workflows repository.

Design: The Pipeline Pattern

ASP.NET application's middle ware uses the pipeline pattern to handle HTTP requests.

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

// Configuration of the app's middleware pipeline
app.UseDeveloperExceptionPage();
app.UseStaticFiles();
app.UseRouting();

app.MapGet("/", () => "Hello World!");
app.Run();
```

Example and image source: Andrew Lock *ASP.NET Core in Action, Third Edition*

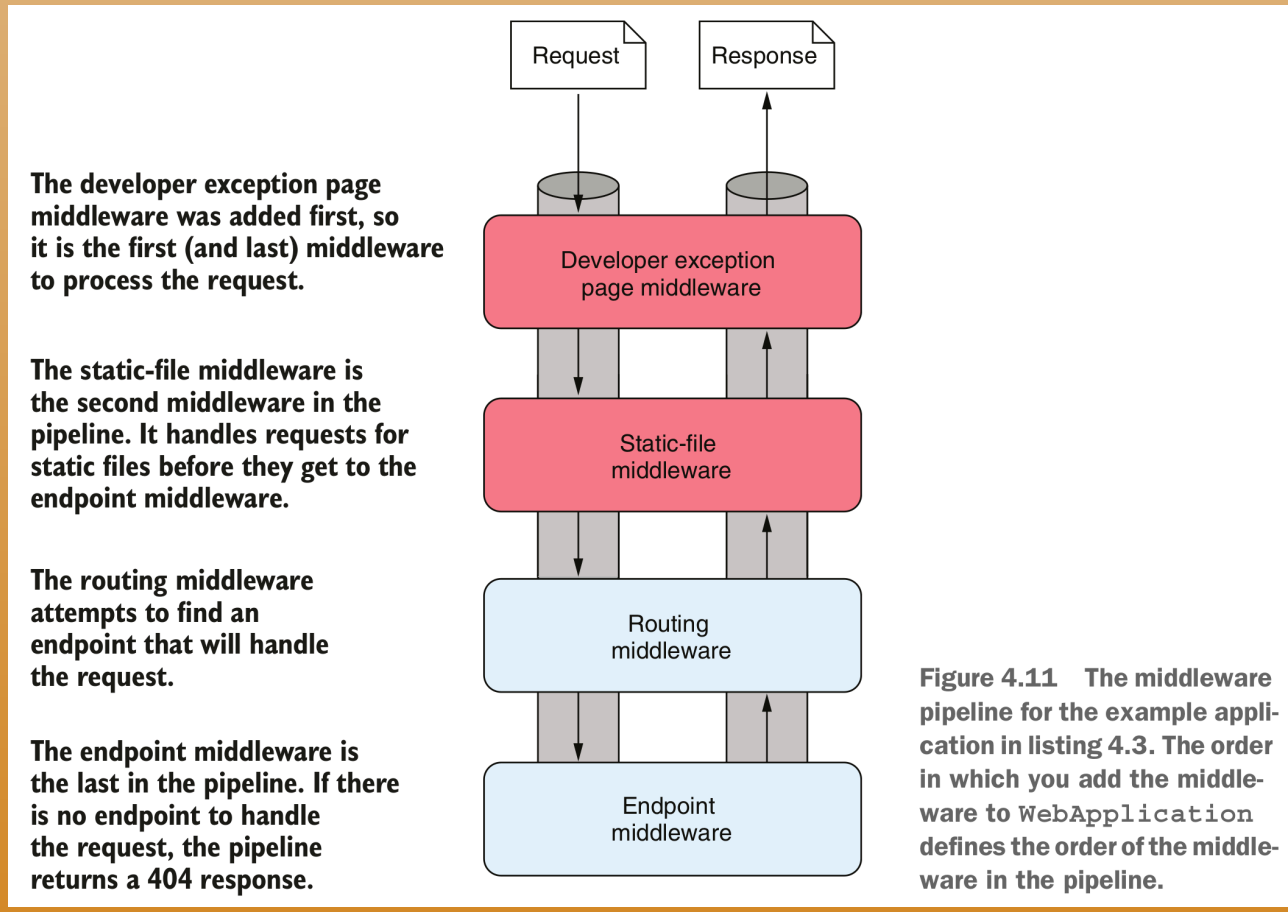
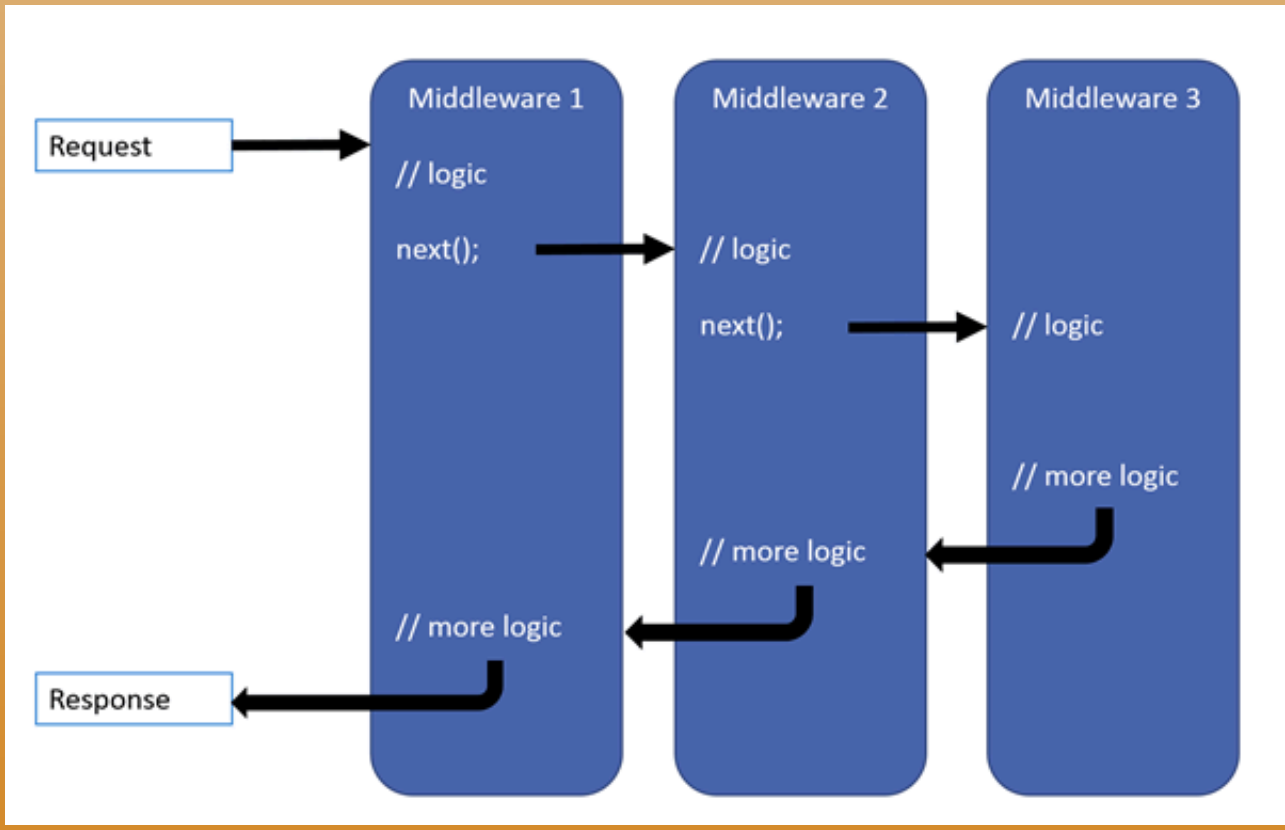


Figure 4.11 The middleware pipeline for the example application in listing 4.3. The order in which you add the middleware to `WebApplication` defines the order of the middleware in the pipeline.

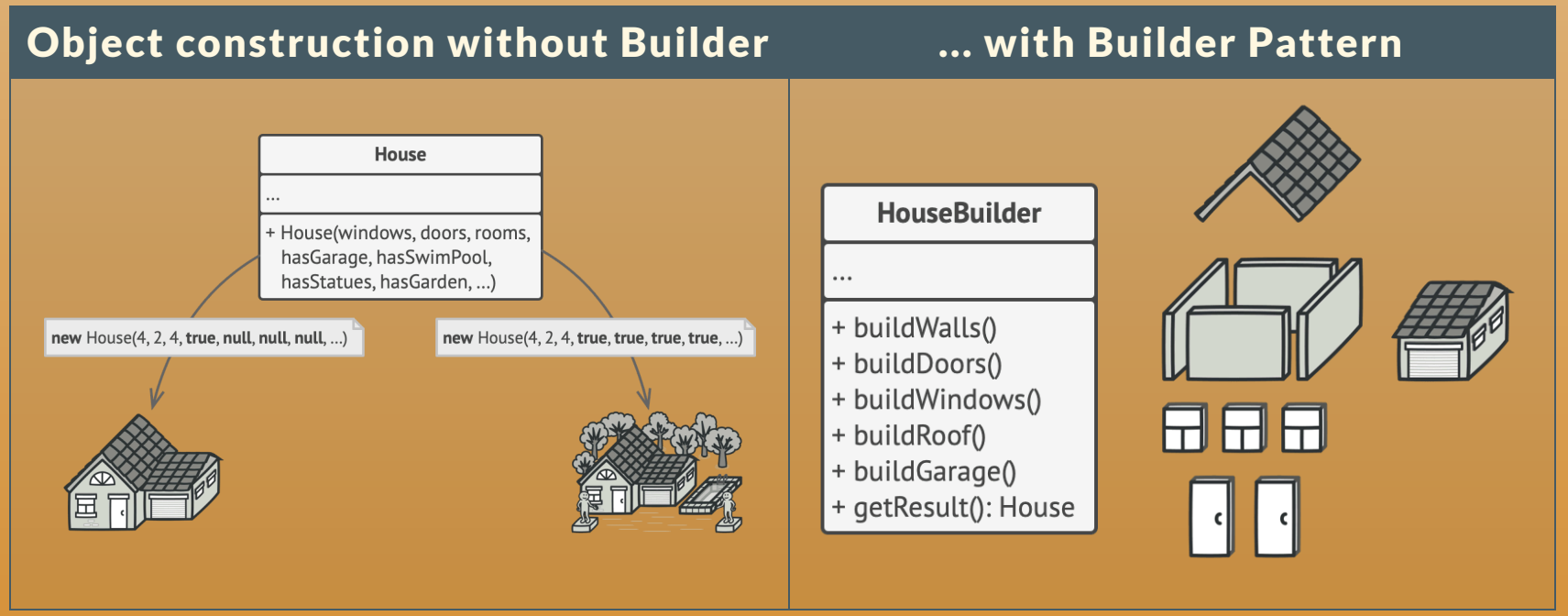
Design: The Pipeline Pattern

> Middleware is software that's assembled into an app pipeline to handle requests and responses. Each component:
>
> - Chooses whether to pass the request to the next component in the pipeline.
> - Can perform work before and after the next component in the pipeline.



Source and image source: R. Anderson et al. *ASP.NET Core Middleware*

Design: The Builder Pattern




```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
WebApplication app = builder.Build();
```

More on that, the next time...

Image source: A. Shvets *Design Patterns*

Recommendation: Write *Short* Units of Code

The 80/24 Rule

Write small blocks of code.

In C-based languages like C#, Java, C++, or JavaScript, consider staying within a 80 × 24 character box. That corresponds to an old terminal window.

Don’t take the threshold values 80 and 24 too literally. I picked them for three reasons:

- They work well in practice
- Continuity with tradition
- Mnemonically, it sounds like the Pareto principle, also known as the 80/20 rule

You can decide on other threshold values. I think the most important part of this rule is to pick a set of thresholds and consistently stay within those limits.
Source: Mark Seemann *"Code That Fits in Your Head"*

Recommendation: Separate Kinds of Code

> **Command Query Separation**
>
> Separate Commands from Queries. Commands are procedures that have side effects. Queries are functions that return data. Every method should be either a Command or a Query, but not both.
> Source: Mark Seemann *"Code That Fits in Your Head"*

Recommendation: Hierarchy of Communication

> **Hierarchy of Communication**
>
> Write code for future readers; it may be yourself. Favour communicating behaviour and intent according to this prioritised list:
>
> 1. Guide the reader by giving APIs distinct types.
> 2. Guide the reader by giving methods helpful names.
> 3. Guide the reader by writing good comments.
> 4. Guide the reader by providing illustrative examples as automated tests.
> 5. Guide the reader by writing helpful commit messages in Git.
> 6. Guide the reader by writing good documentation.
>
> The items on the top of the list are more important than the items at the bottom.
> Source: Mark Seemann *"Code That Fits in Your Head"*

What to do now?

- If not done, complete the Tasks (blue slides) from this class
- Check the reading material
- Work on the project

-If you feel you want prepare for next session, read chapters 13, 14, and 15 Andrew Lock *ASP.NET Core in Action, Third Edition*