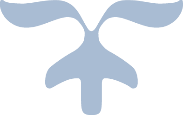


DATAMATIKER - NOTATER



SEPTEMBER 3, 2019

Indholdsfortegnelse

[Introduktion 7](#_Toc106866863)

[Softwaredesign/Software Engineering 7](#_Toc106866864)

[Målgruppe og persona 7](#_Toc106866865)

[Agil udvikling (Agile Samurai notater) 7](#_Toc106866866)

[Agil udvikling i bullet points 8](#_Toc106866867)

[DE 12 agile principper 9](#_Toc106866868)

[Inception deck 9](#_Toc106866869)

[The NOT list 10](#_Toc106866870)

[Elevator pitch 10](#_Toc106866871)

[Vision/Mission 11](#_Toc106866872)

[Product Box 11](#_Toc106866873)

[Burndown chart 12](#_Toc106866874)

[Risk list 12](#_Toc106866875)

[Kontekst diagram 12](#_Toc106866876)

[Scrum 13](#_Toc106866877)

[Extreme Programming (XP) 14](#_Toc106866878)

[Values 14](#_Toc106866879)

[Principles 14](#_Toc106866880)

[Newer XP Practices 15](#_Toc106866881)

[Old XP Practices 17](#_Toc106866882)

[Practices 17](#_Toc106866883)

[https://www.educba.com/scrum-vs-xp/ 18](#_Toc106866884)

[Planlæggelse til en succesfyldt udførelse 18](#_Toc106866885)

[Userstory 18](#_Toc106866886)

[Estimating (Bucket System Estimering/Planning Poker) 19](#_Toc106866887)

[UML Class diagram 19](#_Toc106866888)

[What is a class diagram? 19](#_Toc106866889)

[How do you make one? 19](#_Toc106866890)

[Why do you use it? 20](#_Toc106866891)

[Use case, sekvens diagrammer mv. 21](#_Toc106866892)

[Use case beskrivelser 22](#_Toc106866893)

[Brugervenlighed 23](#_Toc106866894)

[Gestaltlovende 23](#_Toc106866895)

[Jakob Nielsen’s 10 heuristics 23](#_Toc106866896)

[#1: Visibility of system status 23](#_Toc106866897)

[#2: Match between system and the real world 23](#_Toc106866898)

[#3: User control and freedom 23](#_Toc106866899)

[#4: Consistency and standards 23](#_Toc106866900)

[#5: Error prevention 24](#_Toc106866901)

[#6: Recognition rather than recall 24](#_Toc106866902)

[#7: Flexibility and efficiency of use 24](#_Toc106866903)

[#8: Aesthetic and minimalist design 24](#_Toc106866904)

[#9: Help users recognize, diagnose, and recover from errors 24](#_Toc106866905)

[#10: Help and documentation 24](#_Toc106866906)

[Dark Patterns 25](#_Toc106866907)

[Examples of dark patterns 25](#_Toc106866908)

[Trick questions 25](#_Toc106866909)

[Privacy Zuckering 25](#_Toc106866910)

[Roach Motel 25](#_Toc106866911)

[Confirmshaming 25](#_Toc106866912)

[SOLID design principles (food for thought) 25](#_Toc106866913)

[Single Responsibility Principle 25](#_Toc106866914)

[Open/Closed Principle Principle 26](#_Toc106866915)

[Liskov Substitution Principle 26](#_Toc106866916)

[Interface Segregation Principle 26](#_Toc106866917)

[Dependency Inversion Principle 27](#_Toc106866918)

[Other Principles 27](#_Toc106866919)

[Design Patterns 27](#_Toc106866920)

[Testing 28](#_Toc106866921)

[Unit tests (MS Test) 28](#_Toc106866922)

[Testing changing data 30](#_Toc106866923)

[Threading: improve the user experience 30](#_Toc106866924)

[Async 31](#_Toc106866925)

[REST API 31](#_Toc106866926)

[Components of a REST API 32](#_Toc106866927)

[REST API Example 33](#_Toc106866928)

[Using the REST API (consuming rest) 35](#_Toc106866929)

[CORS (who can access the API) 35](#_Toc106866930)

[How to implement CORS 35](#_Toc106866931)

[Swagger (API documentation) 36](#_Toc106866932)

[How to implement Swagger 36](#_Toc106866933)

[Databaser 36](#_Toc106866934)

[Normalisering 37](#_Toc106866935)

[Domain model 38](#_Toc106866936)

[Relational schema & Domain models (ER-Diagram) 39](#_Toc106866937)

[Teknik (Computer Networking) 40](#_Toc106866938)

[The Internet Layers (TCP/IP 5 layer model) 40](#_Toc106866939)

[Application layer 41](#_Toc106866940)

[HTTP and its construction 41](#_Toc106866941)

[Transport layer 42](#_Toc106866942)

[TCP 42](#_Toc106866943)

[TCP Segment Structure 43](#_Toc106866944)

[Example TCP Server: 44](#_Toc106866945)

[Example TCP Client: 45](#_Toc106866946)

[UDP 46](#_Toc106866947)

[Example UDP Senders: 47](#_Toc106866948)

[ 48](#_Toc106866949)

[Example UDP Receive: 48](#_Toc106866950)

[Domain Name System (DNS) 49](#_Toc106866951)

[Network layer 49](#_Toc106866952)

[IP Header 50](#_Toc106866953)

[IPv4 header summary: 51](#_Toc106866954)

[NAT (Network Address Translation) 52](#_Toc106866955)

[Statisk vs Dynamisk IP adresse 52](#_Toc106866956)

[Useful CMD commands 52](#_Toc106866957)

[Datalink layer 52](#_Toc106866958)

[Security & Cryptography 53](#_Toc106866959)

[Types of encryption today: 53](#_Toc106866960)

[Symmetric 53](#_Toc106866961)

[Asymmetric 54](#_Toc106866962)

[PGP - Pretty Good Privacy (Mixed asymmetric and symmetric) 54](#_Toc106866963)

[Certificates 55](#_Toc106866964)

[Firewalls 56](#_Toc106866965)

[Razor MVC 56](#_Toc106866966)

[Best Practice Architecture 56](#_Toc106866967)

[Folder Structure: 56](#_Toc106866968)

[Using the architecture 57](#_Toc106866969)

[Models, Forms and Validation 58](#_Toc106866970)

[Partials and ViewComponents 59](#_Toc106866971)

[Identity, Autherization and Authentication 60](#_Toc106866972)

[Scaffolding Identity (modifying the template) 61](#_Toc106866973)

[.NET Core Security 61](#_Toc106866974)

[Cross Site Request Forgery (CSRF/XSRF) 61](#_Toc106866975)

[Cross Site Scripting (XSS) 62](#_Toc106866976)

[Preventing XSS Attacks in Query Parameter 62](#_Toc106866977)

[Open-Redirect attack 62](#_Toc106866978)

[Secret Management (db connectionstring, api keys and the like) 62](#_Toc106866979)

[Exception handling 62](#_Toc106866980)

[Blazor 63](#_Toc106866981)

[JavaScript (EcmaScript) 63](#_Toc106866982)

[Array helpers 63](#_Toc106866983)

[Examples 64](#_Toc106866984)

[Remember these differences compared to C# 68](#_Toc106866985)

[JS Resources 69](#_Toc106866986)

[React 69](#_Toc106866987)

[How components work in basic react: 70](#_Toc106866988)

[Redux 71](#_Toc106866989)

[React-redux 73](#_Toc106866990)

[React-Router-dom 77](#_Toc106866991)

[React - with libraries 80](#_Toc106866992)

[Angular 80](#_Toc106866993)

[Data bindings 80](#_Toc106866994)

[Example: 81](#_Toc106866995)

[Angular CLI commands 82](#_Toc106866996)

[SEO (Search Engine Optimization) 82](#_Toc106866997)

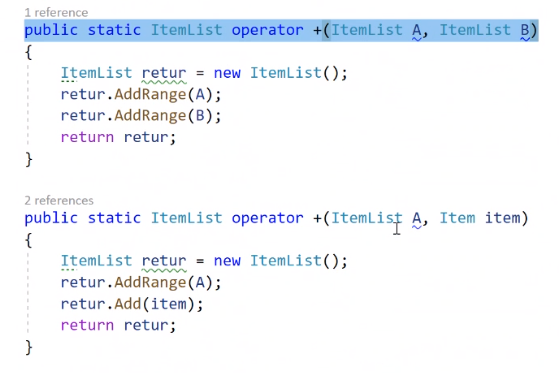
[Uncategorized notes 82](#_Toc106866998)

[Collections: Summary 82](#_Toc106866999)

[Microservices 82](#_Toc106867000)

[Making a class Enumerable C# 83](#_Toc106867001)

[Operator overload C# 83](#_Toc106867002)

[ 84](#_Toc106867003)

[SignalR (in a C# webapp with JavaScript) 84](#_Toc106867004)

[State Machines 87](#_Toc106867005)

[Example state table class: 88](#_Toc106867006)

[Threading: Technical details 90](#_Toc106867007)

[Entity Framework 91](#_Toc106867008)

[How to use EF (Code First) 91](#_Toc106867009)

[NuGet package manager console commands 92](#_Toc106867010)

[Azure 92](#_Toc106867011)

[Regular Expressions 93](#_Toc106867012)

[Computer komponenter 93](#_Toc106867013)

[Programmeringsdefinitioner 95](#_Toc106867014)

[Access types: public/private/protected 96](#_Toc106867015)

[MVVM Model 99](#_Toc106867016)

[XAML & Bindings 99](#_Toc106867017)

[How to do Commands 99](#_Toc106867018)

[Miscellaneous 100](#_Toc106867019)

[DOM (Document Object Model) 101](#_Toc106867020)

[NuGet Packages/useful 3rd party libraries 101](#_Toc106867021)

[Vue 102](#_Toc106867022)

[External resources 102](#_Toc106867023)

[Tools of the Trade 102](#_Toc106867024)

[Business & project management 102](#_Toc106867025)

[Design tools 102](#_Toc106867026)

[Programming 103](#_Toc106867027)

[Testing 103](#_Toc106867028)

[Other 103](#_Toc106867029)

# Introduktion

Dette dokument vil indeholde diverse notater fra datamatikerstudiet, heriblandt softwaredesign og softwarekonstruktion. Det er primært lavet så jeg kan læse op på de ting vi har arbejdet på igennem uddannelsen.

Det er ikke alle ting, som er skrevet til et tilstrækkeligt niveau. Hvis der er mangler eller noget bare ikke er godt nok bliver afsnittets rubrik markeret med Gul eller Grøn i mindre slemme tilfælde.

## Softwaredesign/Software Engineering

Softwaredesign blev ”født” i 1969, hvor Nasa skrev et dokument om hvad man kunne gøre for at få leveret fungerende software. Det er vigtigt, for at kunne komme igennem et projekt fra start til slut, især for større projekter.

# Målgruppe og persona

Målgruppe og persona er noget man ofte arbejde med når man skal lave et produkt, uanset hvad man arbejder med. En målgruppe kunne f.eks. være ældre personer, som har IT besværligheder. En persona er en fiktiv person som repræsenterer et eksempel på éns målgruppe. En persona har både et arbejde, en personlighedstype, bopæl, interesser mv.

Den bedste måde at tjekke om noget man har designet, er ordentligt er ved at teste det via en undersøgelse sendt til målgruppen.

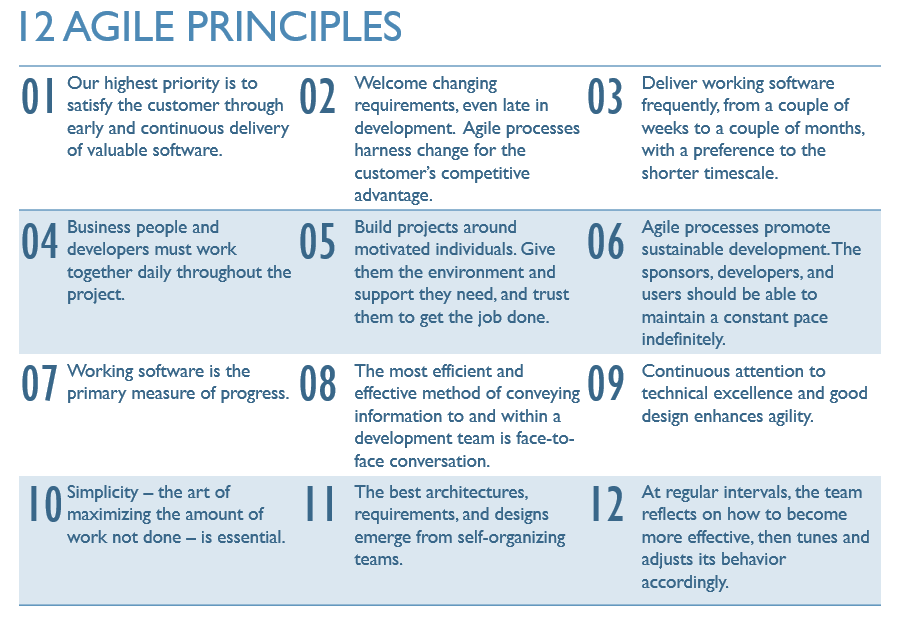
# Agil udvikling (Agile Samurai notater)

Den agile arbejdsmetode er en fleksibel arbejdsmetode, hvor man lægger fokus på at aflevere fungerende software frem for dokumentation og lignende.

## Agil udvikling i bullet points

* Fungerende software er første prioritet.
* Man bryder større opgaver ned i mindre dele (for at gøre dem overskuelige)
* Man tester regelmæssigt, for at sørge for at kriterierne til koden opfyldes.
* Kunden bestemmer hvad der skal prioriteres i projekt.
* Man skal være i dialog med kunden for at sørge for at man er på rette spor.
* Agil planlægning er smidig planlægning. Vær vaks på ændringer i specifikation og andet!
* Man afleverer i ”iterations”, som kan tage alt fra 1 uge til en måned. I en iteration laver man ”et stykke af puslespillet”
* I agil planlægning går man ud fra en ”To-do” list. Den bliver kaldt for en **master story list.**
* Master story listen består af **userstories** (individuelle funktioner)
* Den hastighed et team har med at lave userstories kaldes for Team Velocity.
* Opgaven er først færdig når koden opfylder test kriterierne.
* Krav ændrer sig og man kan ikke indsamle alle krav fra start. Derudover er der altid mere at lave end tiden tillader (prioriter det vigtige!)
* Der findes forskellige agile arbejdsmetoder, f.eks.: XP, Lean, SCRUM.
* Et agilt team skal helst sidde i samme rum.
* Et agilt team har brug for engagerede kunder.
* Et agilt team er selvorganiserede
* Kunden bestemmer alt, bortset fra fremgangsmetode.
* Medlemmer af et agilt team arbejder med lidt af hvert (generalister), men har dog ofte et specialefelt (analyst, programmør eller andet)

### DE 12 agile principper



## 

## 

## 

## 

## Inception deck

The inception deck is where you gather information related to the project (through short activities), so it is clear what the goal is and how to get there.

The content of the inception deck typically consists of the following: [[1]](#footnote-1)[[2]](#footnote-2)

* An elevator pitch.
* Product Box.
* NOT list.
* The major reasons why the project is being made. Identify the most important one.
* Who is a part of this project? (stakeholders)[[3]](#footnote-3)
* An outline of the solution to the project (“high level blueprint of the technical architecture”).
* An estimation of the project's scope (size), how long will it take?
* What is the team worried about?
* What are the priorities? (scope, budget, quality, time)
* What is needed to make the project a success? (time, team qualifications, cost)

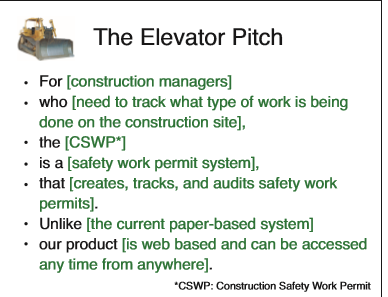
### The NOT list

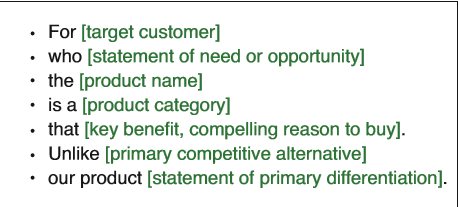
NOT listen består af 3 felter: in, out og unresolved. Med andre ord: det som skal laves, det som ikke skal laves og det som man ikke er sikker på.

Eksempel:

|  |  |
| --- | --- |
| IN: search, navigation menu, buy, buy-basket | OUT: chat, newsletter subscription, cookie login |
| Unresolved: wishlist | |

### Elevator pitch

Elevator pitch’en er til for at klargøre målet. Man skal kunne forklare hvad produkter er og hvorfor det er godt på 30 sekunder eller mindre. ”opskriften” på en elevator pitch ses nedenunder, med et eksempel til højre:



### Vision/Mission

Beskriver kundens mission, vision, værdier etc. for man kan forme produktet ud fra det.

Vision: noget man vil opnå, det ideelle mål.

Mission: det man gør for at opnå visionen.

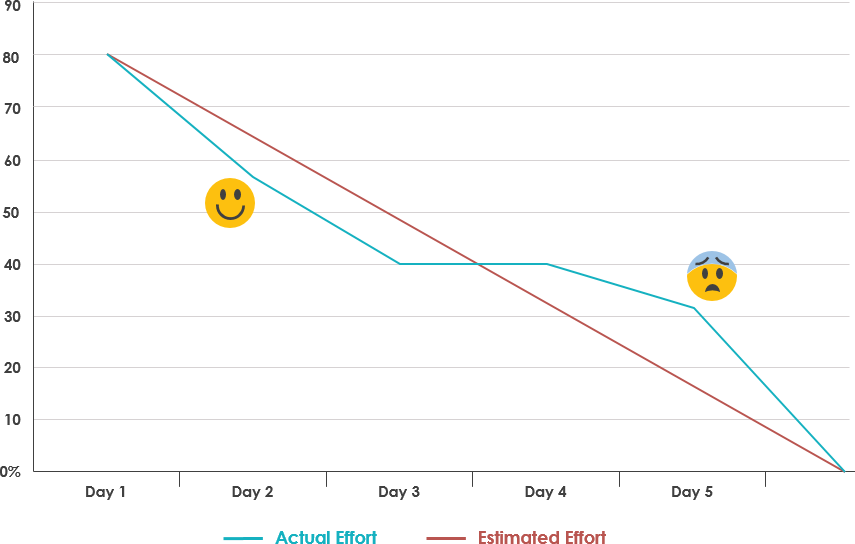
### Product Box

En produktboks er en klargørelse af slutresultatet ved at lave en boks til produktet ”hvordan ville den se ud?”

Indeholder typisk: slogan, produktkvaliteter (hvad gør er det godt til), et passende billede og produktets navn.

### Burndown chart

En graf, som viser den mængde arbejde der er tilbage (effort points) og den ideelle kurve på arbejde gjort.



### Risk list

Liste af risici, som kunne have en effekt på projektet.

### Kontekst diagram

Diagram med kommunikation med omverdenen. Hvem er med i dette projekt?

## 

## Scrum

Scrum is a framework for project management.



**Keywords:**

* Scrum Master
* Daily Scrum (meeting)
* Sprint review (all about the product)
* Sprint retrospective (all about the team)
* Sprint planning
* Definition of done
* Product backlog
* Scrum board
* Sprint (time period in which work is done. 1-4 weeks)
* Product owner

“In a nutshell, the sprint review is about the product, while the sprint retrospective is about the team. While the sprint review helps you to regularly meet customer expectations, retrospectives allows scrum teams to become faster, smarter, and even happier. And that’s just scratching the surface.The sprint review is about the product, while the sprint retrospective is about the team.”[[4]](#footnote-4) Sprint burndown er til retrospectives.

## Extreme Programming (XP)

Extreme Programming er en arbejdsmetode med nogle principper, værdier og ting man gør i praksis når man programmere. De kan deles op i som følgende:

5 values, 14 principles, 24 practices (12 primary, 12 corollary)

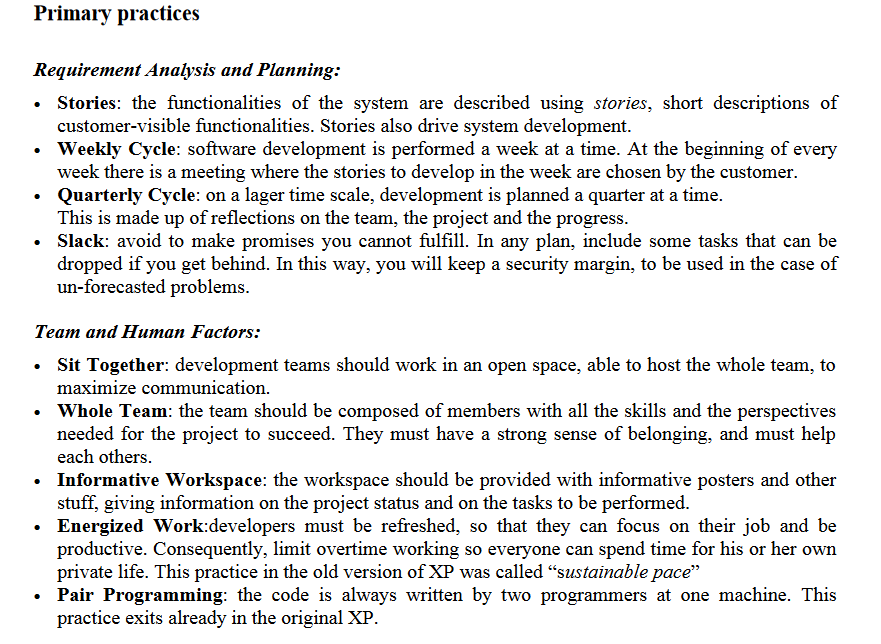
### Values

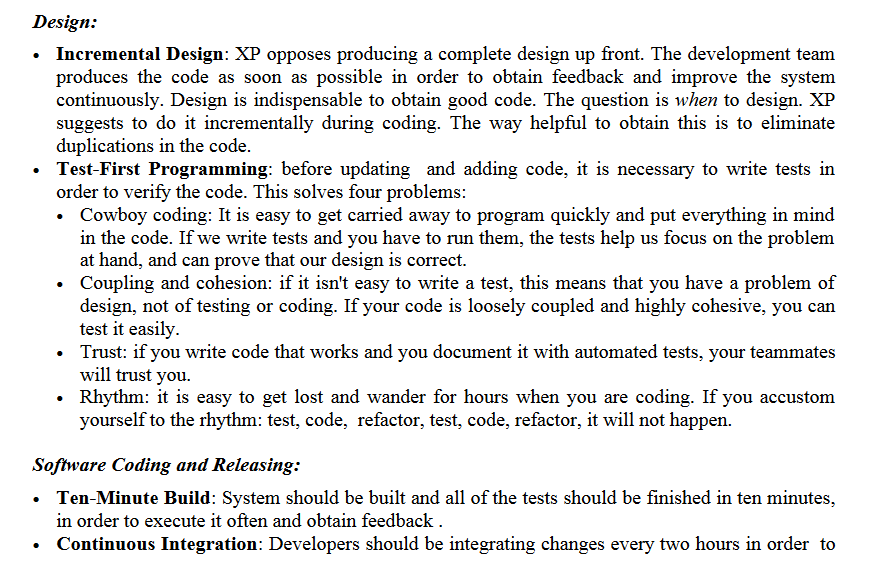
Communication (in team) - Feedback (communication with customer) - Simplicity - Courage - Respect

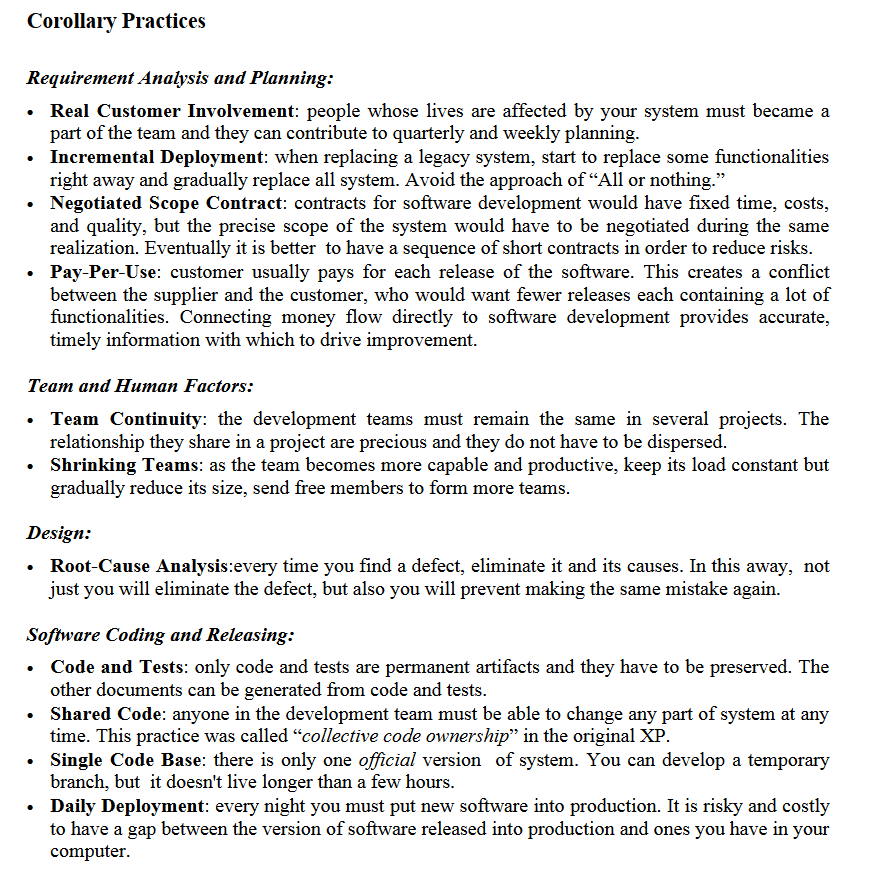
### Principles

* Mutual Benefit (team + business)
* Improve (continuously)
* Reflect (after each iteration)
* Fail (don’t freeze)
* Economics (Provide value. Don’t waste time on less important things.)
* Quality > Haste

### Newer XP Practices



 ease integration headaches.



### Old XP Practices

### Practices

* Incremental design (design only when needed, not ahead of time)
* Test-first programming (makes you think about what you’re trying to achieve)
* Slack (have minor functionality that you can drop, if you fall behind)
* Weekly cycle (plan weekly, drop things less important things for next cycle)
* Energised work (don’t overwork. Maintain constant pace)
* Sit together (for better communication)
* Root-cause analysis (if an error occurs, find out why and fix it instead of just fixing it)
* Single code base (don’t have branches that exists for too long)
* Code and tests (They are the most important and most up-to-date documentation. Write it well.)
* Daily deployment (deploy to production daily, **if** the team produces code that is not filled with bugs)
* Negotiated Scope Contract (do small contracts instead of larger ones to maintain flexibility in decision making)
* Incremental deployments (don’t release everything at once. Do it one step at a time.)

<http://www.javiersaldana.com/> A good resource that has the **principles**, **values** and **practices** of XP as part of it (albeit only the old ones)

## <https://www.educba.com/scrum-vs-xp/>

# Planlæggelse til en succesfyldt udførelse

## Userstory

En userstory er en kort beskrivelse af noget funktionalitet i et program. Det kan laves på nogle forskellige måder, men et eksempel på dette er:

As a [type of user]  
I want to [do something]  
because [some reason]

Ud over basis (ovenstående) kan man tilføje succes kriterier og tidsestimering til hver userstory, så man ved, hvad der skal til for at en userstory er færdig og hvor lang tid man forventer at det vil tage.

Extract from a site about writing good userstories:

* **“One action per a Story.** If you want to write something like “as a customer I want to browse items and add them to the cart” you’d better split it into 2 separate Stories.
* **Describe an intention, not a feature.** For example, instead of “I want to manage my profile” create a few Stories like “I want to be able to register”, “I want to upload my profile photo”, “I want to link my credit card to my profile” - each Story will have a different value.
* **Keep it short.** Users don’t care what library you will use to let them browse the list of items so leave all the tech details aside.
* **Avoid describing UI.** We’ve defined Stories as negotiable, remember? That's why all good User Story examples don't include any UI details. So don’t try to compose any special way to implement them (we’ll do this later).”[[5]](#footnote-5)

## Estimating (Bucket System Estimering/Planning Poker)

When you start estimating with the Bucket System you assign each user story two values: business value and effort value. You start by estimating what you think is the biggest effort and the lowest effort (same with business value) and estimate the rest of the user stories by comparing them to those two.

Business Value = how much value the business thinks they’ll get out of it (higher number = higher value)  
Effort value = how much effort it will take (lower number = lower effort)

To get an estimate of what to prioritize you divide business value with effort value:

Business Value / Effort Value = Priority

Those with higher Priority are completed first, unless a high priority task is dependent on another function being done first. Then the dependency is done first.

## UML Class diagram

Video: <https://www.youtube.com/watch?v=UI6lqHOVHic>

The above video covers it excellently.

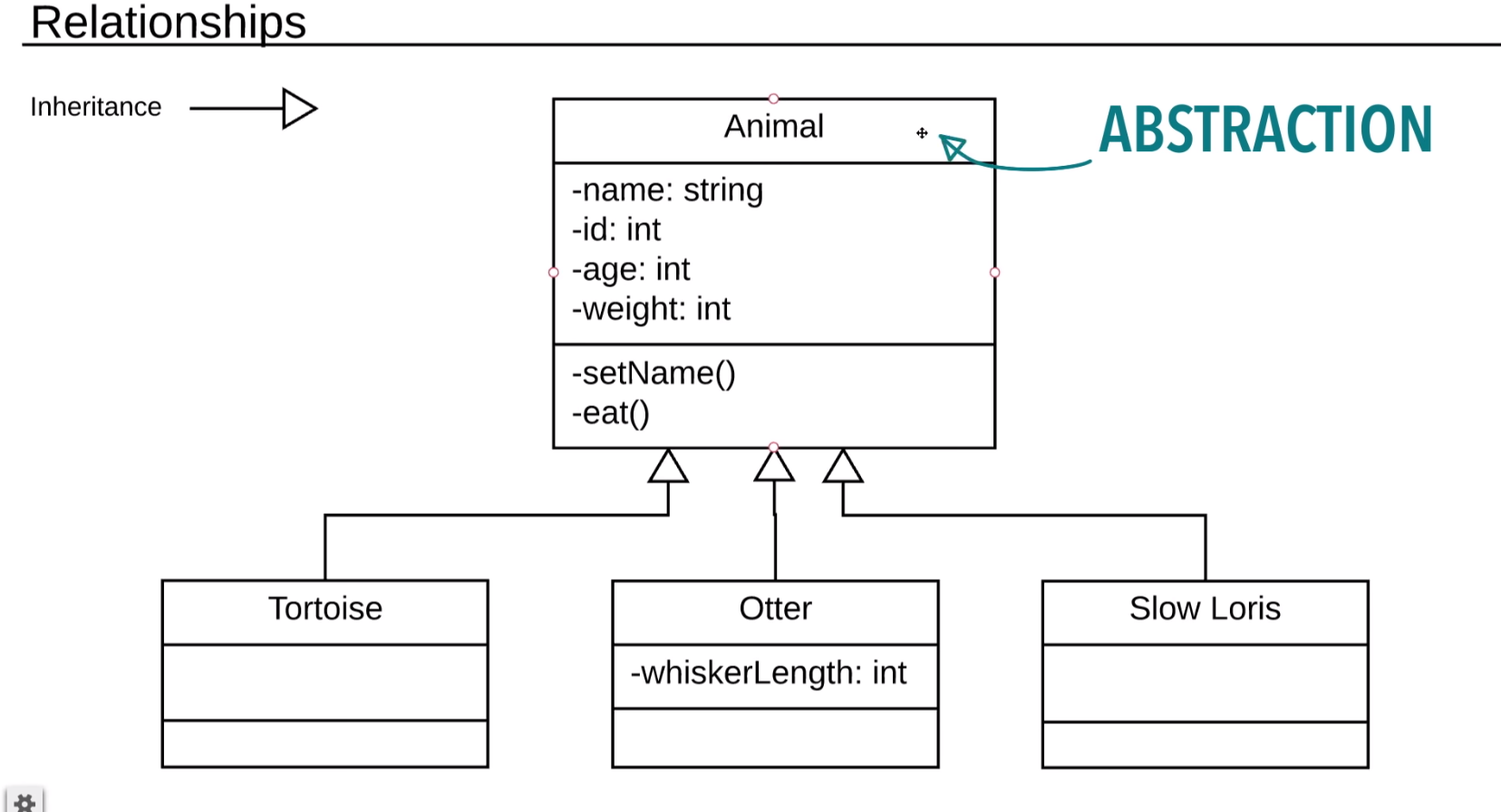
#### What is a class diagram?

A class diagram is a visual representation of how the classes in your program are structured and how they’re related.

#### How do you make one?

Visibility:

|  |  |
| --- | --- |
| + | Public |
| - | Private |
| # | Protected |
| ~ | Package |

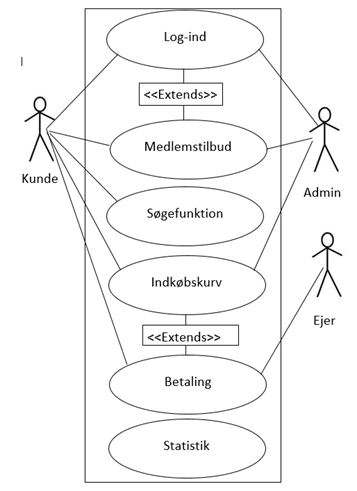


Missing core info

#### Why do you use it?

To get an overview/idea of how the structure of a project looks (or is going to look).

## Use case, sekvens diagrammer mv.

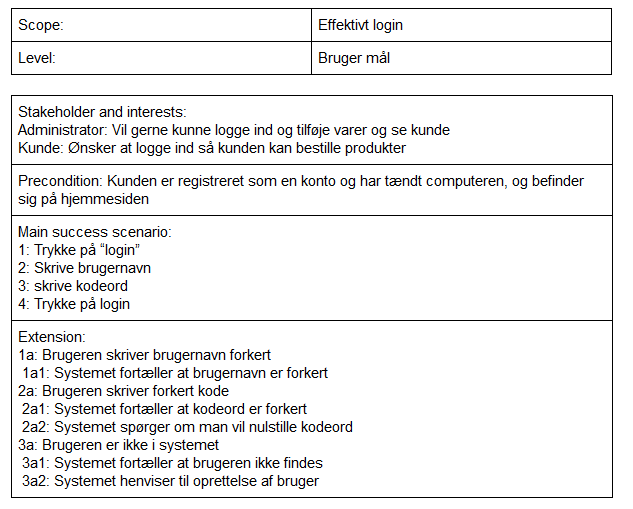
“A **use case** is a methodology used in system analysis to identify, clarify, and organize system requirements” 

Sidens interessenter. Kunden til venstre og Administrator til højre. Der er streger til hvad de begge skal kunne.

Indkøbskurv + betaling -> Extends. De er afhængige af hinanden. Man kan ikke have en indkøbskurv uden at betale.

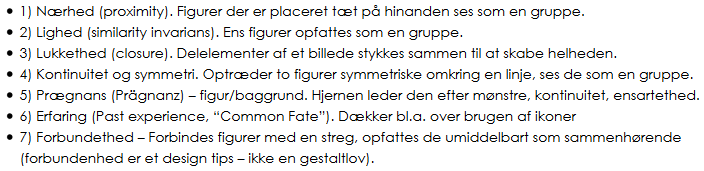
Sekvens diagram info

### Use case beskrivelser



# Brugervenlighed

## Gestaltlovende[[6]](#footnote-6)



*Source: 1http://informatik-gym.dk/glossary/gestaltlove/*

## Jakob Nielsen’s 10 heuristics[[7]](#footnote-7)

### #1: Visibility of system status

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

(Read full article on [visibility of system status](https://www.nngroup.com/articles/visibility-system-status/) and watch 3 min. [video on the visibility heuristic](https://www.nngroup.com/videos/usability-heuristic-system-status/).)

### #2: Match between system and the real world

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

(Read full article on the [match between the system and the real world](https://www.nngroup.com/articles/match-system-real-world/) and watch 3 min. [video on the real-world heuristic](https://www.nngroup.com/videos/match-system-real-world/).)

### #3: User control and freedom

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

(Watch 2-min. [video on the user control heuristic](https://www.nngroup.com/videos/usability-heuristic-user-control-freedom/).)

### #4: Consistency and standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow [platform conventions](https://www.nngroup.com/articles/do-interface-standards-stifle-design-creativity/).

(Watch 3-min. [video on consistency & standards](https://www.nngroup.com/videos/usability-heuristic-consistency-standards/).)

### #5: Error prevention

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

(Read full article on [preventing user errors](https://www.nngroup.com/articles/slips/) and watch 3 min. [video on error prevention](https://www.nngroup.com/videos/usability-heuristic-error-prevention/).)

### #6: Recognition rather than recall

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

(Read full article on [recognition vs. recall in UX](https://www.nngroup.com/articles/recognition-and-recall/) and watch 3 min. [video on recognition vs. recall](https://www.nngroup.com/videos/recognition-vs-recall/).)

### #7: Flexibility and efficiency of use

Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

(Watch 3 min. [video on Flexibility and Efficiency of Use](https://www.nngroup.com/videos/flexibility-efficiency-use/).)

### #8: Aesthetic and minimalist design

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

(Watch 2 min. [video on Aesthetic and Minimalist Design](https://www.nngroup.com/videos/aesthetic-and-minimalist-design/).)

### #9: Help users recognize, diagnose, and recover from errors

[Error messages](https://www.nngroup.com/articles/error-message-guidelines/) should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

(Watch 2 min. [video on helping users overcome errors](https://www.nngroup.com/videos/usability-heuristic-recognize-errors/).)

### #10: Help and documentation

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

## Dark Patterns

“Dark Patterns are tricks used in websites and apps that make you do things that you didn't mean to, like buying or signing up for something”.

### Examples of dark patterns[[8]](#footnote-8)

#### Trick questions

Sometimes during a questionnaire or something similar, you can get a question that sounds like one thing, at first glance, but is something else entirely when read carefully.

#### Privacy Zuckering

When you accidentally share more information about you than intended through a trick. Named after Facebook CEO Mark Zuckerberg.

#### Roach Motel

“You get into a situation very easily, but then you find it is hard to get out of it (e.g. a premium subscription).”

#### Confirmshaming

When you try to, for example, unsubscribe from a newsletter and the owners of the newsletter try to shame you into staying with something along the lines of “are you really sure you want to leave us all alone? :c”

# SOLID design principles (food for thought)

SOLID is an acronym for 5 different principles, which makes you think about your code and makes it easier to maintain and extend. Each letter is a different principle.

S = Single Responsibility Principle  
O = Open/Closed Principle  
L = Liskov Substitution Principle  
I = Interface Segregation Principle  
D = Dependency Inversion Principle

In the following subchapters I will go over each of them.

## Single Responsibility Principle

**Original definition:** “any single object in object-oriented programing ([OOP](https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP)) should be made for one specific function”[[9]](#footnote-9)

**My definition:** one class should not hold tons of code, which does a lot of different things. It should instead be dedicated to one purpose only, so each class only holds a minimal amount of code, which makes it easier to comprehend the class. (rule of thumb is that 1 class should not hold much more than 1 screen of code) unless there is a good reason for it to take up more space.

**Example:**

## Open/Closed Principle Principle

**Original definition:** “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”[[10]](#footnote-10)

**My definition:** you should not have to modify an already made class, but it should be easy to extend it via interfaces. Bottom line:

Use a lot of interfaces and add new functionality via classes that inherit from those, so you can implement new stuff without breaking the old stuff.

**Example:**

## Liskov Substitution Principle

**Original definition:** “Let *Φ(x)* be a property provable about objects *x* of type *T*. Then *Φ(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.”[[11]](#footnote-11)“

“Subtypes must be substitutable for their base types**.“**

**My definition:** when a class inherits from a base class you must implement all the base class methods and properties.

**Example:** if we take a humanoid super class and any of its subclasses (e.g. a human, elf or cyclops), they should be able to be exchanged with one another without breaking functionality.

## Interface Segregation Principle

**Original definition:** “Clients should not be forced to depend upon interfaces that they do not use.”[[12]](#footnote-12)

**My definition:** the class implementing the interface should not hold empty/not implemented elements from the interface. It should instead only inherit from an interface in which it can implement everything from it.

**Example:** [**https://medium.com/@radheshyamsingh\_83359/liskov-substitution-principle-solid-design-f9d48500c260**](https://medium.com/@radheshyamsingh_83359/liskov-substitution-principle-solid-design-f9d48500c260)

## Dependency Inversion Principle

**Original definition:**

* High-level modules should not depend on low-level modules. Both should depend on abstractions.
* Abstractions should not depend on details. Details should depend on abstractions.[[13]](#footnote-13)

**My definition:** use indirect methods of giving information, instead of directly.

**Example:** use a parameter of X instead of a local variable of X

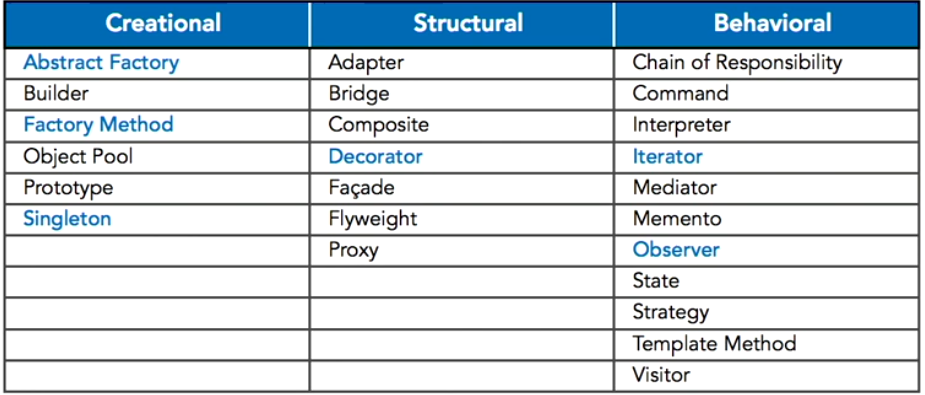
# Other Principles

KISS - Keep it Simple Stupid (keep the code as simple as possible)  
DRY - Don’t Repeat Yourself (refactor if you find yourself writing the same code 3 or more times)  
YAGNI - You Aren’t Gonna Need It (don’t write code you THINK you'll need)

# Design Patterns

Design patterns are templates, of sorts, that describe a way to solve a specific problem. Every pattern can be divided into 3 different categories:

* Creational (object creation mechanisms)
* Structural (class and object composition)
* Behavioral (communication between objects)



*1 Different Patterns Divided by Category*

**Do** ***not use them, just to use them****.* Use them to solve a problem you have stumbled upon.

<https://www.dofactory.com/net/design-patterns> here there are lots of good patterns with UML diagrams, frequency of use and more.

# Testing

Levels of testing:

* Validation of concepts and requirements (are domain models and userstories ok?)
* Validation of the design (are the diagrams, sketches and the like ok?)
* Component verification (unit tests and the like)
* System and integration validation (system wide testing)
* Operation verification (acceptance tests)

Equivalence partitions: defining partitions to make it easier to test (splitting up the area, which you test, so you test the lower values, middle values and higher values). For example:

|  |  |  |
| --- | --- | --- |
| Less than 4 | Between 4 and 10 | More than 10 |

Then you test the boundaries. E.g. in the above example you would test the values 3, 4, 7, 10, 11 (7 being a number in the middle, 3 being below 4 so it should give error, 4 being right on the boundary, 10 on the boundary, 11 above the boundary.).

## Unit tests (MS Test)

Annotations written in the unit tests before each part of the unit test:

* [TestClass] : set up the test.
* [TestMethod] : a test method to run.
* [TestInitialize] : run before each [TestMethod].
* [ClassInitialize] : run before the test starts.
* [AssemblyInitilize]: run once before starting a test run.

Testing verification

* Assert.AreEqual(expected, actual)
* Assert.IsTrue(actual)
* Assert.ThrowsException<XXException>( ()=> [variable] = [value]

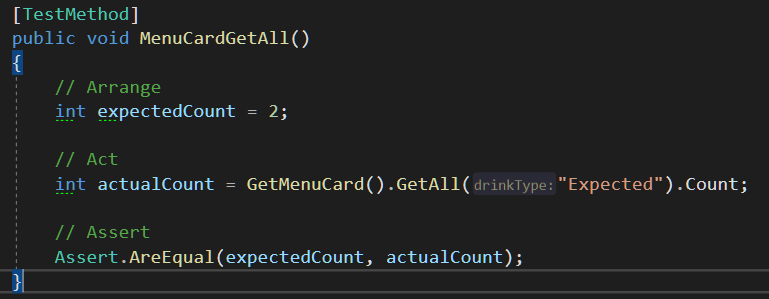
^ example of act part

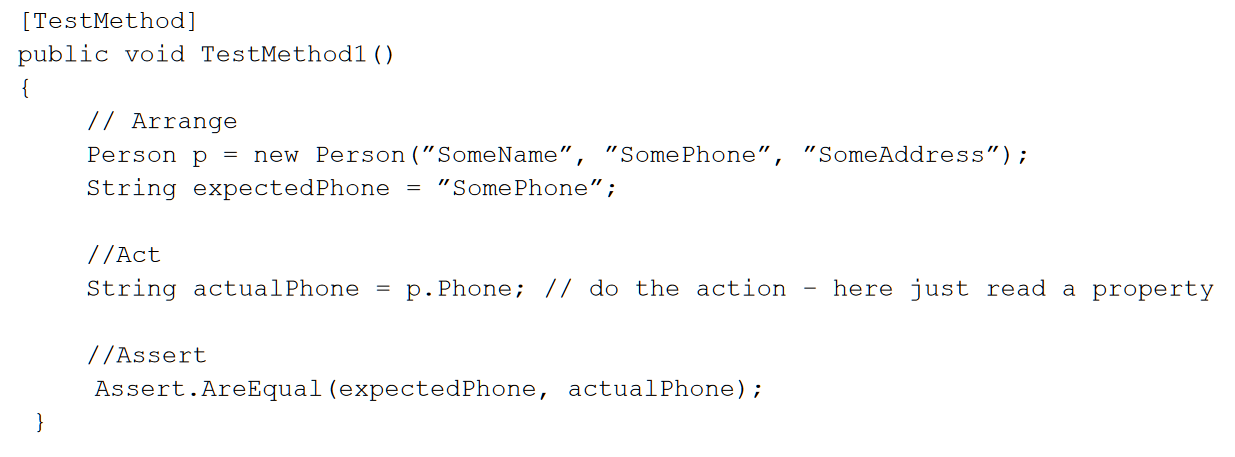
* Assert.KeyNotFoundException

Each part of the unit test consists of three things: arrange, act and assert.

* In the arrange part you
  + Set up the test data
* In the act part you
  + Do the action you’re trying to test
* In the assert part you
  + Check if the test ran as expected.

Example of unit test:



Note: The GetMenuCard returns an object of type MenuCard, which contains a List of Drinks and the method GetAll, which gets all drinks of a specific type.

## Testing changing data

When testing changing data you have a few options:

* Mocks (not something we’ve learned)
* Give all methods a return type, so you can test that (e.g. bool, ModelClass)
  + This doesn’t make for good tests, but it is fast.
* Make Test tables in your database, which you make a setup and cleanup on each time.

# Threading: improve the user experience

There are two types of operations you might want to use asynchronous programming for:

CPU heavy tasks (e.g. large calculations)

I/O heavy tasks (e.g. an API call)

* In C# the class **Task** is what is commonly used for multithreading (splitting work up between CPUs/cores).
* **Static fields are shared between all threads.**
* **Thread** is the basic keyword for parallelism in most languages, but Task is (in most cases) used instead in C#.
* Parallel starts multiple threads at once. Parallel.Invoke, Parallel.For/Foreach.
  + Parallel.For stops the thread while its doing its For loops. To avoid this, you wrap it in a task.

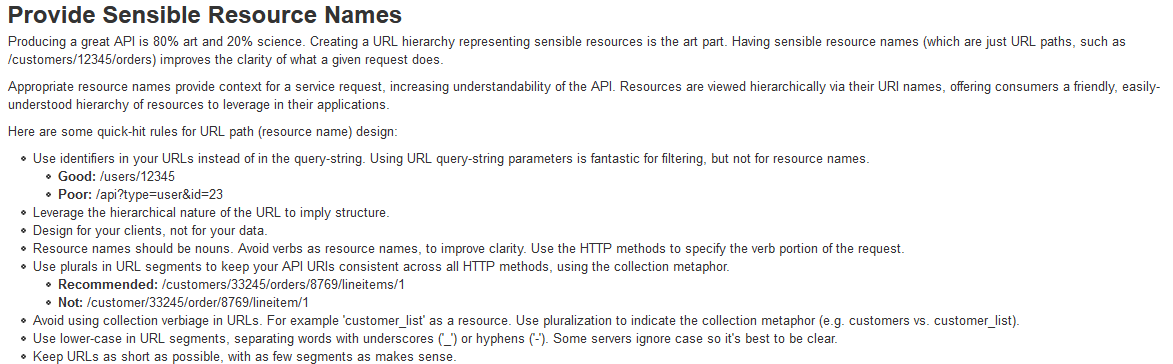
Keywords:

* **Task**
* **Task<T>**
* **Task.WaitAll**
* **Thread**
* **Async / Await[[14]](#footnote-14)**
* **Thread-safe collections e.g. ConcurrentBag<T>**
* **Parallel (advanced)**
* **Race condition:** when two or more threads work on the same thing and compete for access to it.
* **Critical region:** shared area (data) between multiple threads, which should be locked in some way.
* **Semaphore**: used to control an amount of threads allowed in a block of code (if more than 1 thread should be allowed)
* **Mutex:** a type of lock that doesn’t allow other threads to Release mutex.  
   **mutexObj.WaitOne()** and **mutexObj.ReleaseMutex();**
* **Lock** (to make things thread-safe) (a lock makes the locked area accessible to only 1 thread at a time.) You typically use an a static variable of type ‘object’ to lock with. (Lock takes 1 argument)
* Readonly (other way to make thread-safe)

## Async

* Async is a part of threading. It can make GUI apps Deadlock (a type of crash) if not handled carefully.
* Task.Result (because it waits for the result) will make the app *deadlock*, if run in e.g. a ViewModel. A solution to this is to make a class that checks tasks, a “task watcher” of sorts. One is implemented in this [article about using async with MVVM](https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/march/async-programming-patterns-for-asynchronous-mvvm-applications-data-binding).
* Beware of working with files, since if two threads try to access the same file it won’t work (generally)

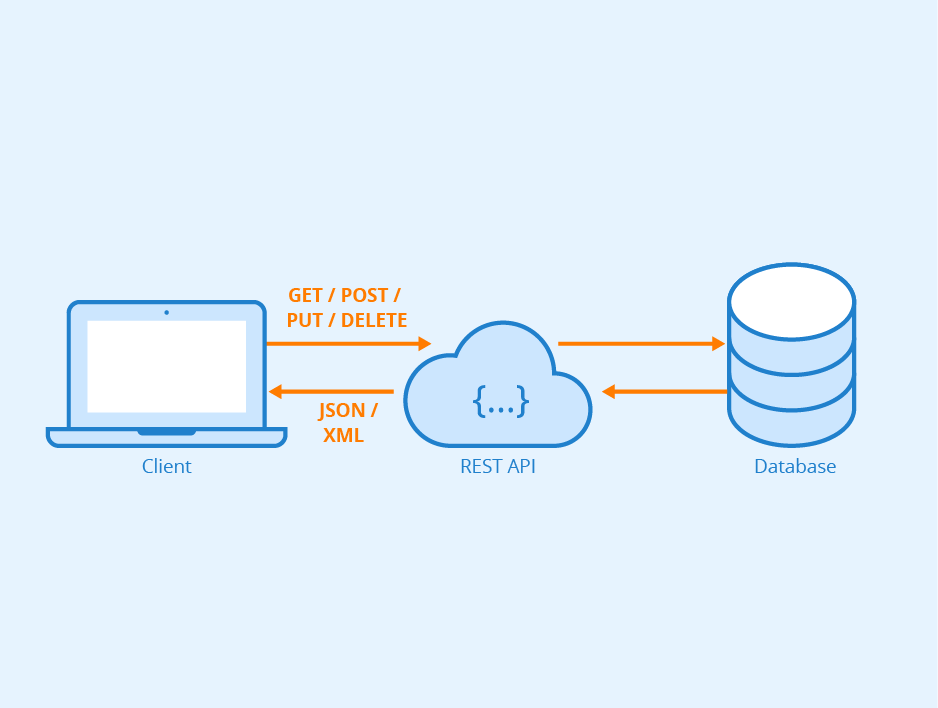
# REST API



More tips can be found here: <https://www.restapitutorial.com/> (pointers for good REST API design)

**REST:** **Re**presentational **S**tate **T**ransfer  
**API:** **A**pplication **P**rogramming **I**nterface.

A REST API (a type of Web Service) is a standard way to make what’s called a Web Service API. Essentially it is the binding link between the frontend application and the server/database.



Reasons for using an API include:

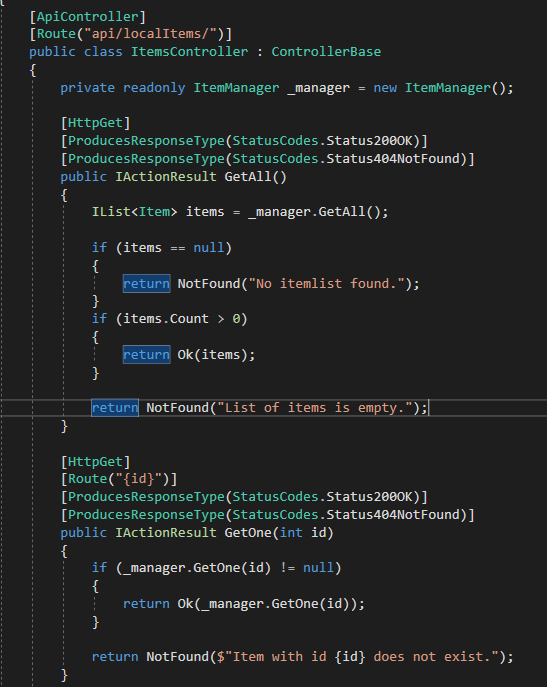
* It's reusable (since it's independent of the application/client)
* Easily scalable (can add more to it without problem)
* Independent of platform & language
* Makes the connection string more secure.

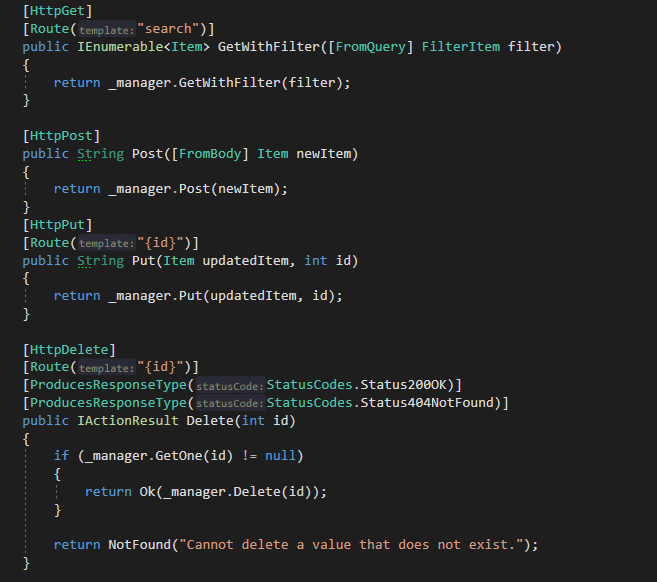
## Components of a REST API

“A REST/HTTP request from the client to the server usually consists of the following components:

* **URL Path** [https://api.example.com/user]
* **HTTP Method** [GET, PUT, POST, PATCH, DELETE]
* **Header** – (optional) additional information that the client needs to pass along in the request such as Authorization credentials, Content-Type of the body etc.
* **Parameters** – (optional) variable fields that alter how the resource will be returned.
* **Body** – (optional) contains data that needs to be sent to the server.”[[15]](#footnote-15)

## REST API Example



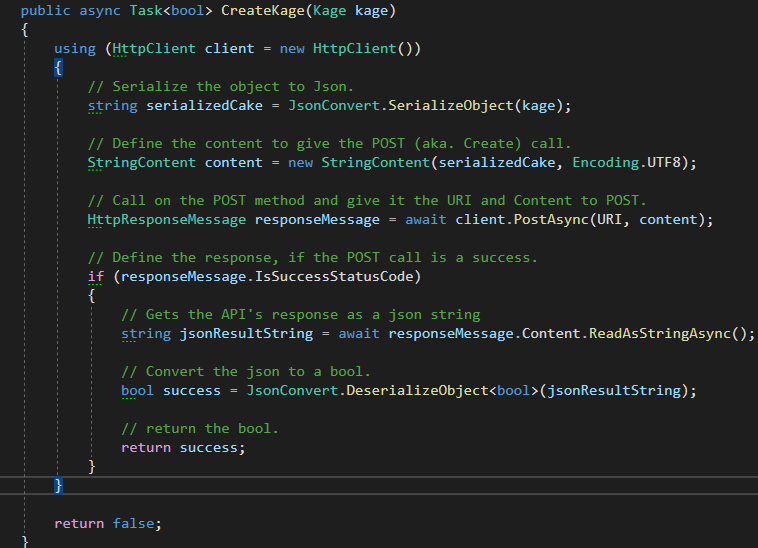


NOTE: the Post and Put can also return IActionResult. I just neglected to do so here.

Use annotations [HttpGet], [HttpPost]...  
 To retrieve info from HTTP body use [FromBody] in parameter. [FromQuery] is another annotation you can put in front of a parameter (used to get info from the URL).

## Using the REST API (consuming rest)

An example of using a REST API with the POST method (PUT and DELETE looks somewhat similar):



## CORS (who can access the API)

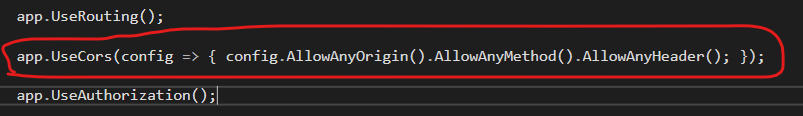
CORS stands for *Cross-Origin Resource Sharing.* It adds additional HTTP Headers to requests and responses, to allow or deny access to a resource (e.g. an API). It should be considered a minor security layer (which is necessary). Cors is used to slacken the rules of cross origin shenanigans.

CORS must always be a part of your application (if you use any kind of JavaScript)

### How to implement CORS

In Startup.cs:

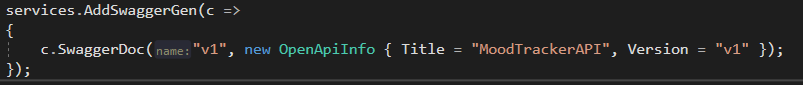




## Swagger (API documentation)

### How to implement Swagger

Config in Startup.cs



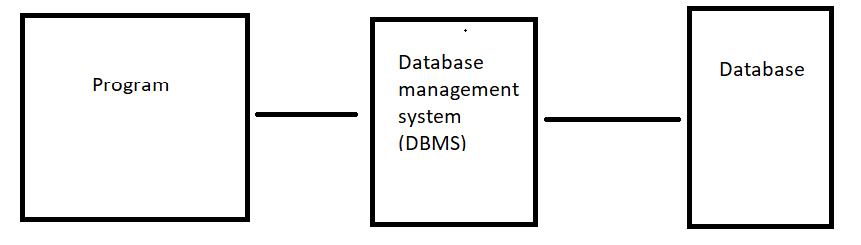


# Databaser

Der findes flere database sprog, men det vi fokuserer på, vil være Structured Query Language (SQL).

Sql Server har noget som hedder **SQL Server Data Tools** som kan bruges til at versionsstyre ens database. (den laver en “template” (via scripts) til hvordan databasen ser ud. Den kan man så importere som en rigtig database.

Når man skal sende noget til databasen, foregår det gennem et Database Management System (eksempler: MySql, Oracle, Sql Server, Microsoft Access).



CRUD = create, read, update, delete.  
RDBMG = Relational Database Management System  
Relation: A Table with columns and rows.  
Attribute: A named column of a table (relation)  
Domain: Set of valid values for an attribute.

Every table needs a primary key (typisk som [column\_name\_id])

Database relationer:

One to One  
One to Many  
Many to Many

**Keys**  
Superkey (deprecated)  
Candidate key (2 or more rows that must be unique. Cannot remove 1 without losing uniqueness)  
Primary key (Unique key that identifies the table. It is typically just an id that is incremented by 1 each entry, but a composite/candidate key could also be a primary key)

## Normalisering

Normalisering er et værktøj til at lave gode tabeller og (dermed gode) relationelle databaser De er inddelt i normalformer. Der findes 4 normal former:

* Første normal form
* Anden normal form
* Tredje normal form
* Fjerde normal form

**Normal form 1:**

* Kun en enkel værdi i et felt. (f.eks. kun et telefonnummer pr. entry)
* Alle kolonner i tabellen skal have et unikt navn.

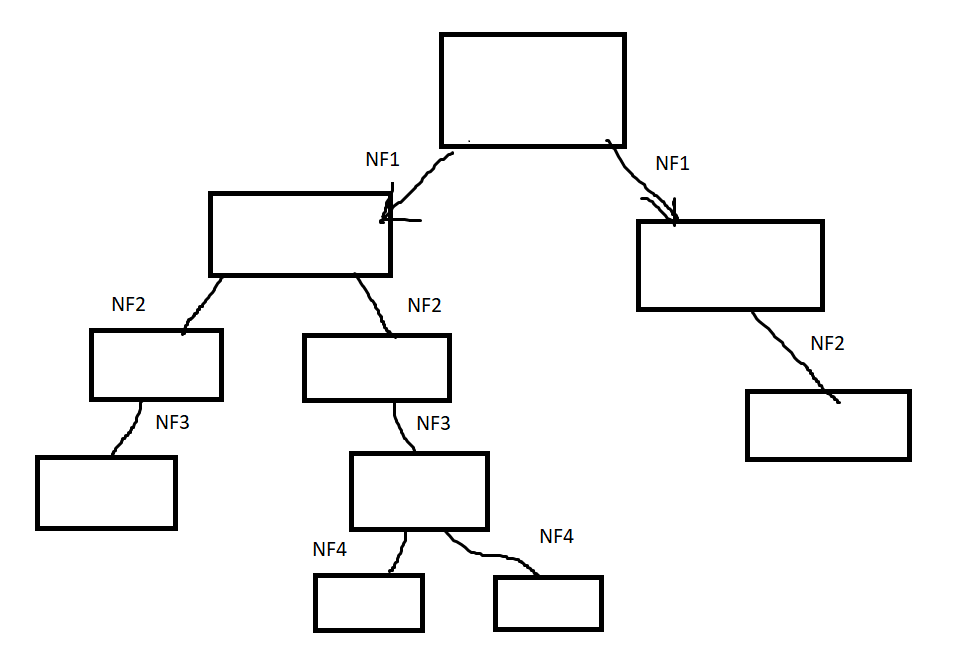
**NF 2:** kolonne B skal være afhængig af kolonne A. (og følger NF1)  
(f.eks. staff\_navn og staff\_tlf ville være afhængig af staff\_id). Med andre ord skal indhold i en tabel altid være afhængig af en form for primær nøgle.

(Afhængighed vil sige at, hvis man vælger alle hvor for en bestemt kolonne A værdi, vil man KUN finde én B værdig.  
(f.eks. SELECT staff\_name FROM Staff WHERE staff\_id = 1 skal kun returnere ét enkelt navn)

NF 3:

* NF1 og NF2 gælder for tabellen
* En værdi som ikke er en del af en primær nøgle (f.eks. staff\_name) må ikke være afhængig af en kolonne som ikke er en del af en primær nøgle (f.eks. supplier\_name, supplier address kunne ikke være i en kunde tabel, da supplier address er afhængig af supplier\_name)

**NF 4:** ikke dækket endnu (der må ikke være flere forskellige faste data i et felt (f.eks. flere forskellige typer newsletter subscriptions))

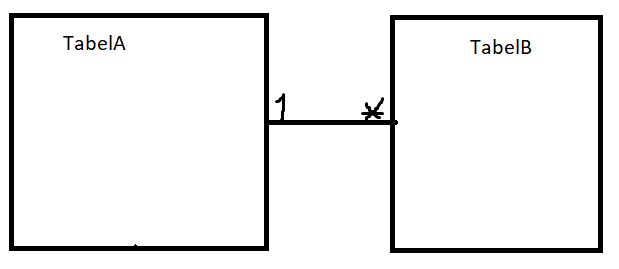
****

*For hvert led tjekker man om den ”næste” normal form gælder for tabellen. Hvis ikke den gør, splitter man tabellen, så man kan opfylde dens krav.*

Man bruger dem for at undgå dårlig database design, hvorved man kan undgå duplikeret data og større risiko for data uregelmæssigheder (problemer med at oprette, opdatere og slette fra databasen).

## Domain model

En domain model beskriver relationerne imellem forskellige tabeller med data, så man kan lave en ordentlig database. Tallet der står tættest på en tabel, er det antal der tæller for den tabel. Eksempel: Der skal være en TabelA til mange TabelB.

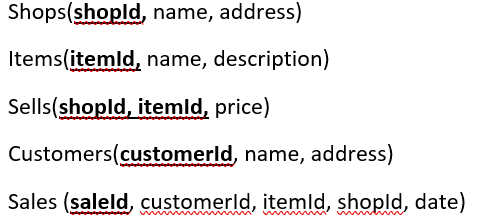


Når man skal oversætte en domain model til en database skal man:

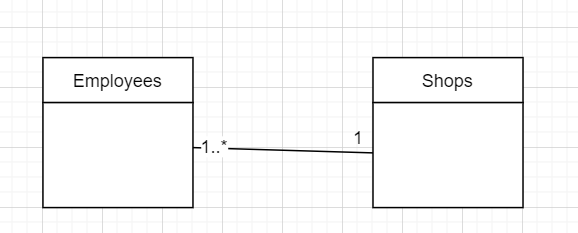
* Konverterer alle domain tabeller man har lavet til database tabeller
* Gennemgå relationerne på tabellerne og lav tabeller i forhold til det.
  + One to One relation:
    - kombiner til én tabel (medmindre man tænker at forholdet vil ændre sig)
  + One to Zero/One relation:
    - behold begge tabeller
    - Lav en foreign key
  + Zero/One to Zero/One relation:
    - behold begge tabeller
    - Lav en foreign key som kan være null.
  + One to Many relation:
    - Behold begge tabeller
    - Lav en foreign key
  + Zero/One to Many:
    - Behold begge tabeller
    - Lav en foreign key som kan være null.
  + Many to Many relation:
    - Behold begge tabeller
    - Lav en mellemtabel med foreign keys fra begge tabeller.

## Relational schema & Domain models (ER-Diagram)

A **relational schema** looks like this:



A **domain model** looks like this:



*1 or more* employees can work at *1 shop*.

A domain model is also a type of ER-diagram called the conceptual ER diagram.

3 types of ER-diagrams exist[[16]](#footnote-16):

* Conceptual (basic structure with relations and table names)
* Logical (what conceptual has + column names and additional tables to enable functionality)
* Physical (logical + datatypes)

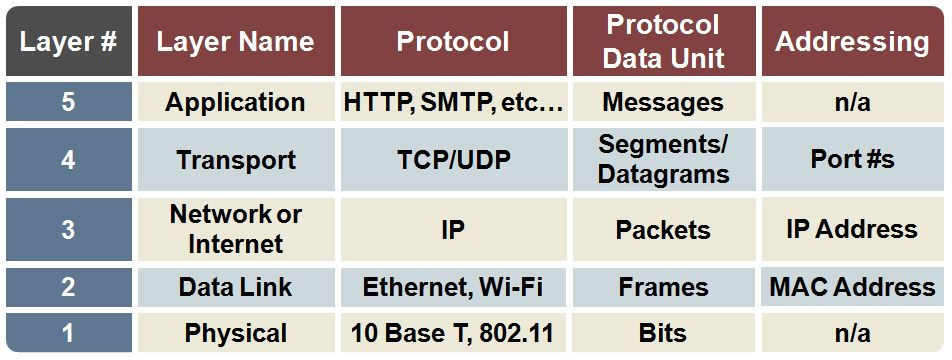
# Teknik (Computer Networking)

Circuit based network

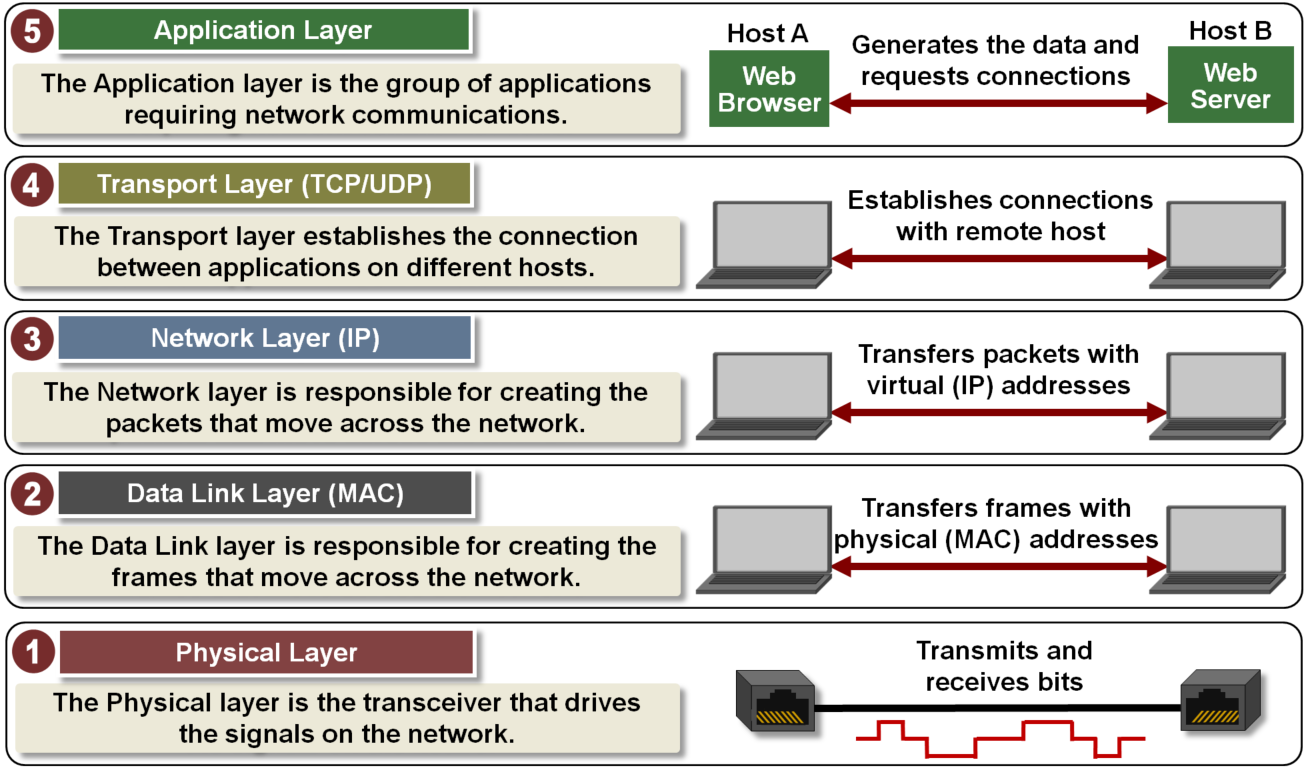
Packet based network (most common)

## The Internet Layers (TCP/IP 5 layer model)

The internet of today consists of 5 layers. From the top down we have the Application-, transport-, network-, datalink- and finally the physical- layer. When data is sent through the network it goes through these layers from the top and down, this is called ***vertical communication***. For each layer an additional small amount of data is attached to the data packet being sent. This is called a header. It gives info to the receiving layers (application to application, transport to transport) and is called ***horizontal communication***. The (datalink or network?) attaches a “footer”, in addition to a header.



<https://microchipdeveloper.com/tcpip:tcp-ip-five-layer-model> for more detailed info.



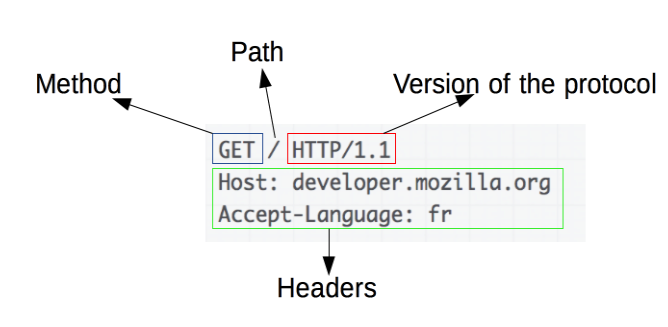
### Application layer

#### HTTP and its construction

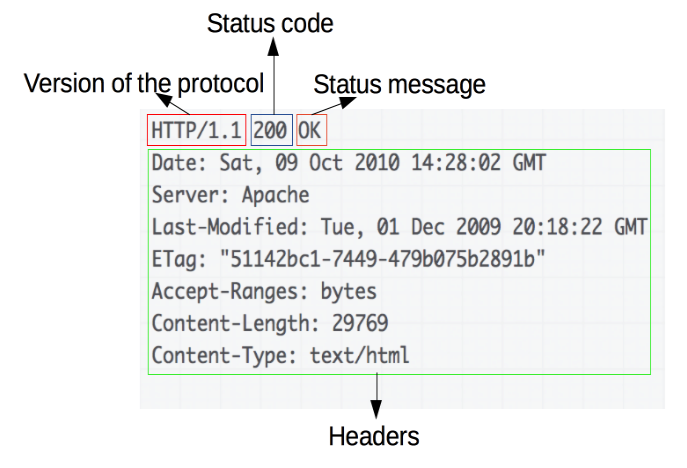
HTTP versions: 1.0, 1.1, 2.0, 3.0

In 1.0 = 1 request, 1 response: close conn.  
In 1.1 = persistent connection.  
In 2.0 = mostly the same as 1.1, but increased speed and has backwards compatability w. 1.1  
In 3.0 = in the making (as of 2020)

**HTTP Request:**



**HTTP Response:**



### Transport layer

#### TCP

**TL;DR:** TCP stands for Transport Control Protocol and is Connection Oriented and reliable.

Connection oriented (constant connection, no need to reconnect for new messages since you’re already connected)

“C*onnection-oriented* describes a means of transmitting data in which the devices at the end points use a [protocol](https://searchnetworking.techtarget.com/definition/protocol) to establish an end-to-end connection before any data is sent. Connection-oriented protocol service is sometimes called a "reliable" network service, because it guarantees that data will arrive in the proper sequence. Transmission Control Protocol ([TCP](https://searchnetworking.techtarget.com/definition/TCP)) is a connection-oriented protocol.

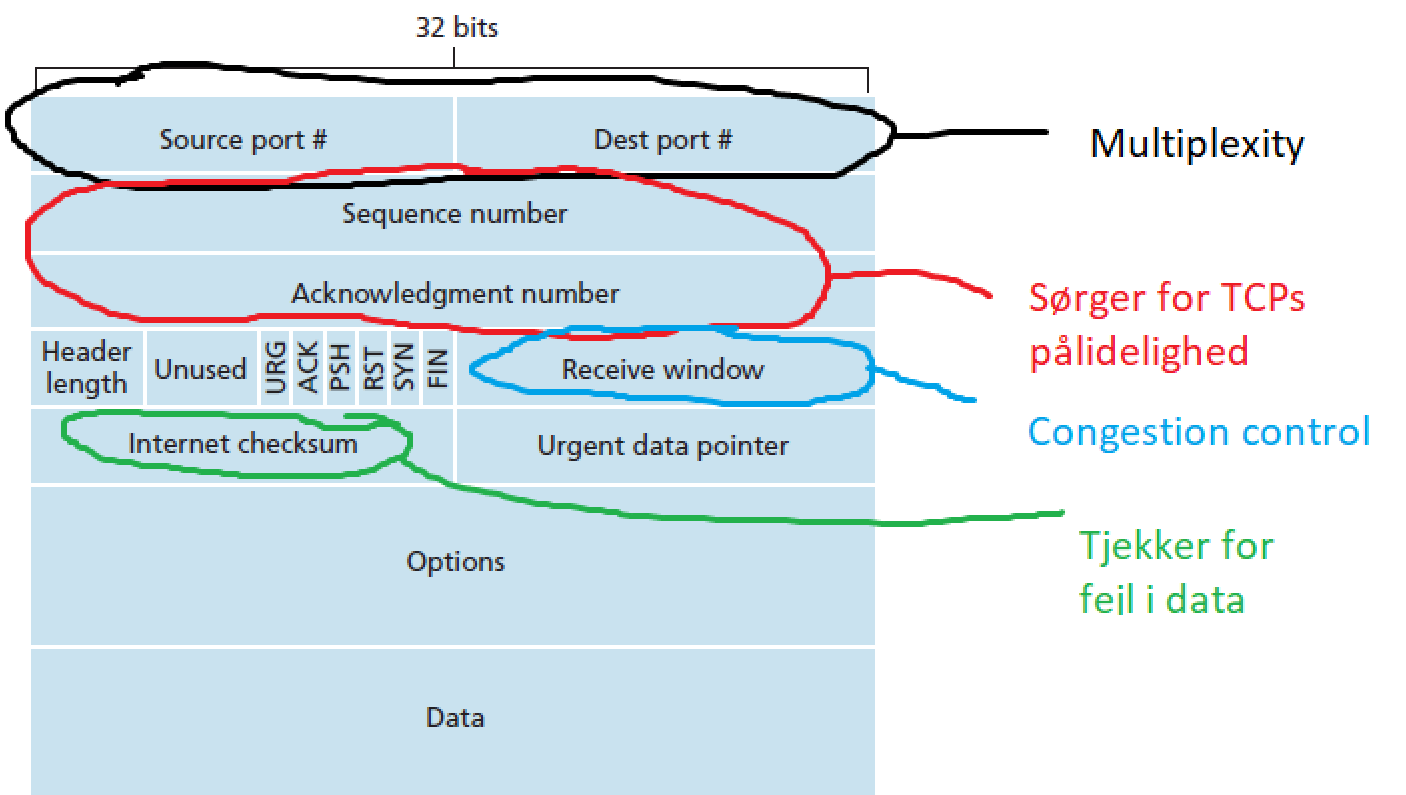
For connection-oriented communications, each end point must be able to transmit so that it can communicate. The alternative to connection-oriented transmission is the [connectionless](https://searchnetworking.techtarget.com/definition/connectionless) approach, in which data is sent from one end point to another without prior arrangement. Connectionless protocols are usually described as [stateless](https://whatis.techtarget.com/definition/stateless) because the end points have no protocol-defined way to remember where they are in a "conversation" of message exchanges. Because they can keep track of a conversation, connection-oriented protocols are sometimes described as stateful.”

Tcp listener/tcp client

(checksum is something used to check for corrupt data and are a part of both UDP and TCP segment header)

#### TCP Segment Structure

Multiplexity er et fancy ord for holde orden på forbindelsen mellem to computere



**3-way handshake med TCP i simpel udgave:**

1. Sequence numbers are generated randomly to prevent hijacking of the connection.
2. Client opretter forbindelse til serveren (pkt/seq) sequence
3. serveren svarer ok (ack) acknowledgement
4. client svarer at den har modtaget serverens ok

**Using TCP:**

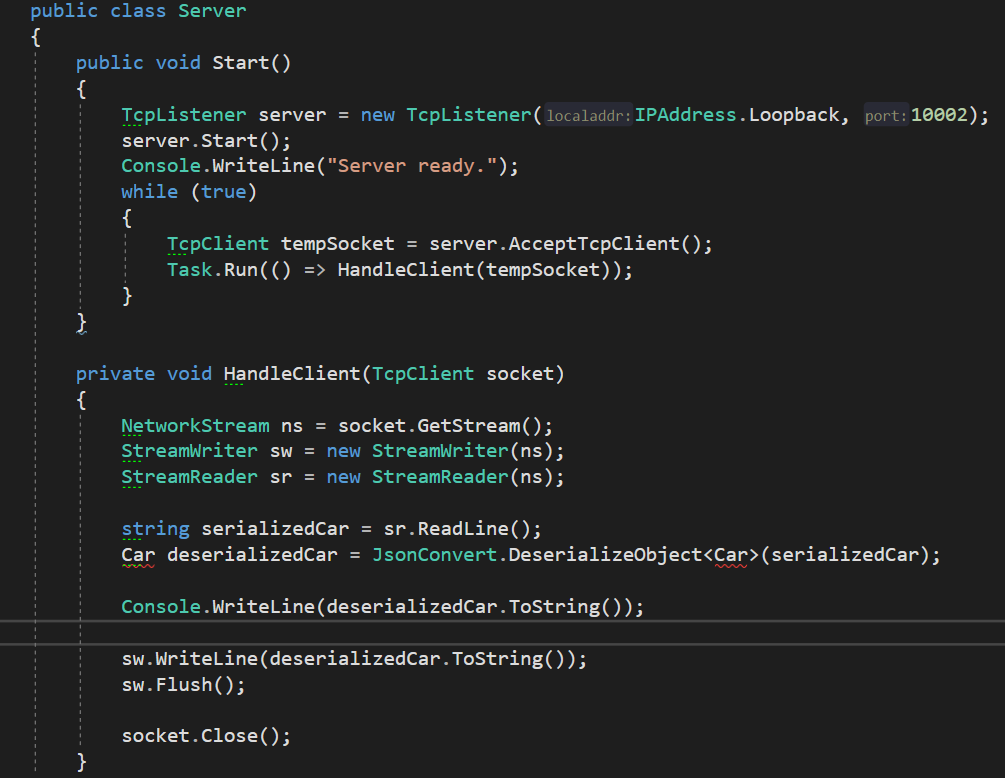
A server uses TCPListener, while client uses TCPClient. Example of server:

TCPClient(IP Address, Port). The IP address specifies the machine to connect to, while the port specifies the application.

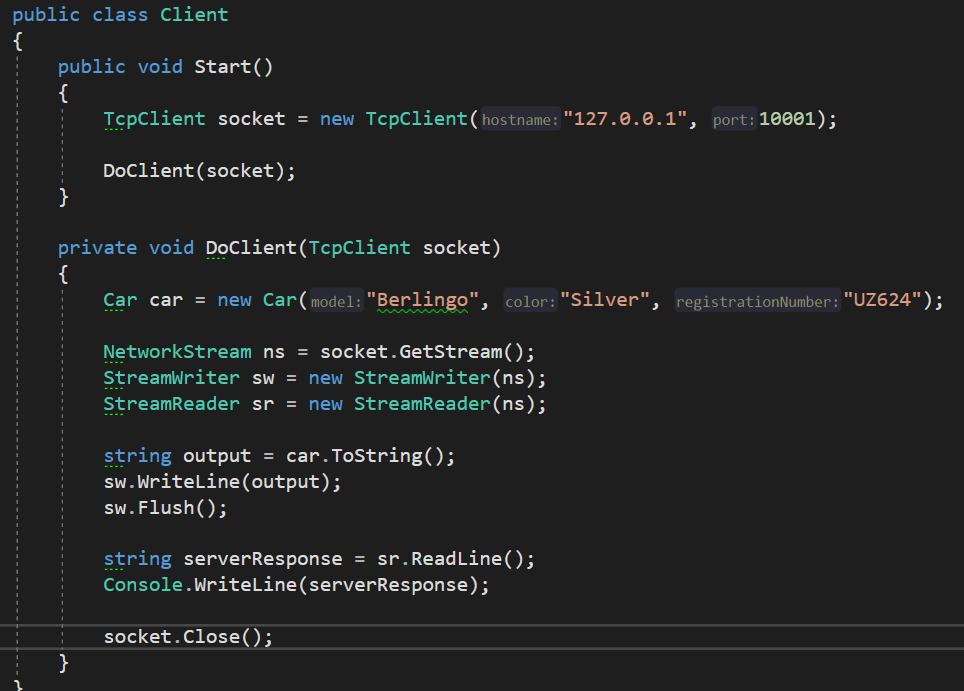
Socket indeholder Stream, modtagers ip/port, afsenders ip/port.

**Note:** the following examples are not secure. No encryption.

##### Example TCP Server:



##### Example TCP Client:

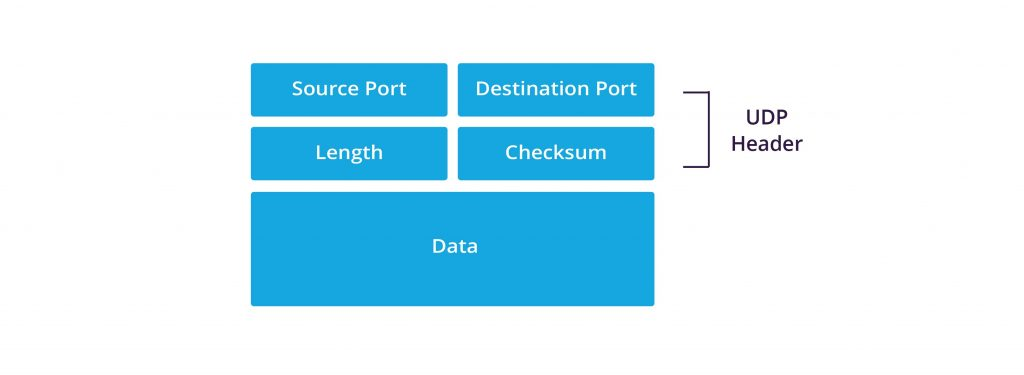


#### UDP

UDP is unreliable, fast and Connection Less.

**Connection Less** means that messages are sent without a handshake event (like in TCP), so it could start sending packets (aka. datagram) without the server being available. Moreover data might be lost on the way, because if the packet is not received it is not resent (by default). In addition the packet might arrive in a different order. Lastly, it must reconnect for a new message to be sent.  
C# classes: Udp client.

UDP does not employ a “listener”. It only uses UDPClient. The client can receive and send data.



##### 

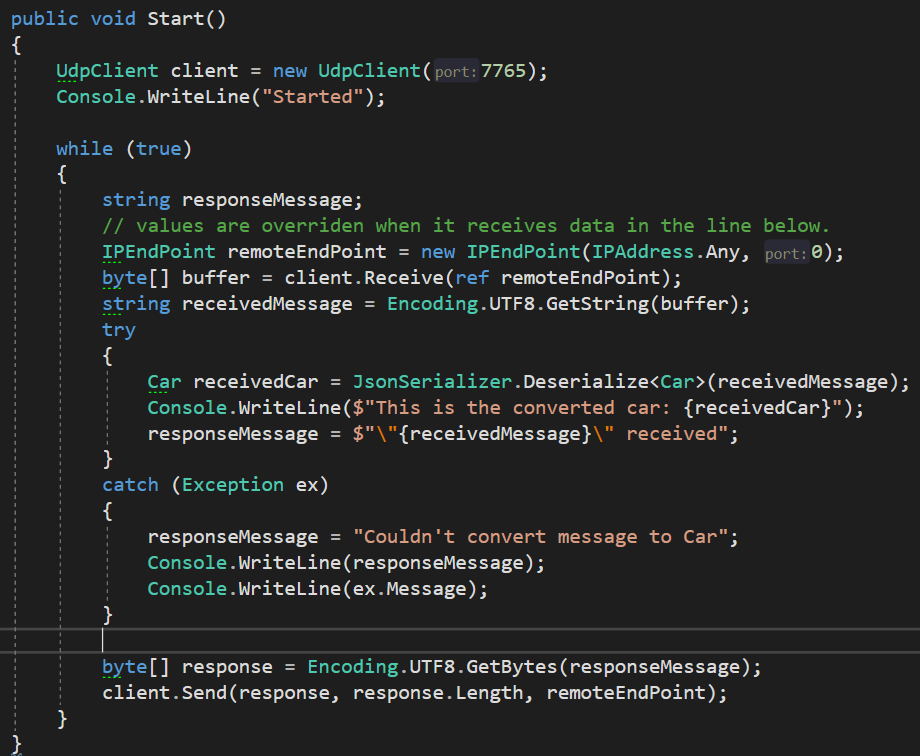
##### Example UDP Senders:

Python3 broadcast:



##### 

##### Example UDP Receive:



## Domain Name System (DNS)

DNS’ primære opgave ligger i at oversætte logiske adresser til IP Adresser. DNS system kan generelt betegnes i en træ-struktur som følgende:

* **Root**
  + **Top Level domain:** dk.
    - **Authoritive domain:** zealand.
      * **Lokalt:** www.

Hvilket samlet ville blive til [www.zealand.dk](http://www.zealand.dk). Denne adresse vil blive oversat til en ip adresse, når man har indtastet den i browseren.

Top Level domain administerer de authoritative domæner, mens at de authoritative domæner administrerer de lokale. Et andet eksempel kunne være: portal.azure.com.   
portal = lokal, azure = authoritative, .com = toplevel

Root domænet er en ‘tom string’ og ses ofte ikke i vores URL. Nogle gange repræsenteres den med et punktum (.) for enden af URLen. Den bliver dog sat ind selv om det ikke står der.

Det sekundære formål for DNS:

* kan bruges som mailserver (MX: mail exchange)
* logisk navn til nameserver (NS: name server )
* konvertere IP Adresse til logisk adresse (ved at være deprecated) (PTR)
* Alias (CNAME), gør så man kan vidredirigere fra en adresse til en anden.

Root, toplevel, og authoritative domæner har hver især en name server (NS).

Cache funktion bliver brugt (med udløbsdato på requests) for ikke at overloade systemet med requests.

Handy cmd command: nslookup [logisk adresse] (ip address)

## Network layer

The network layer is all about navigating through the internet. It happens, in broad strokes, like this:

*private network (default gateway) => public network(s) => private network.*

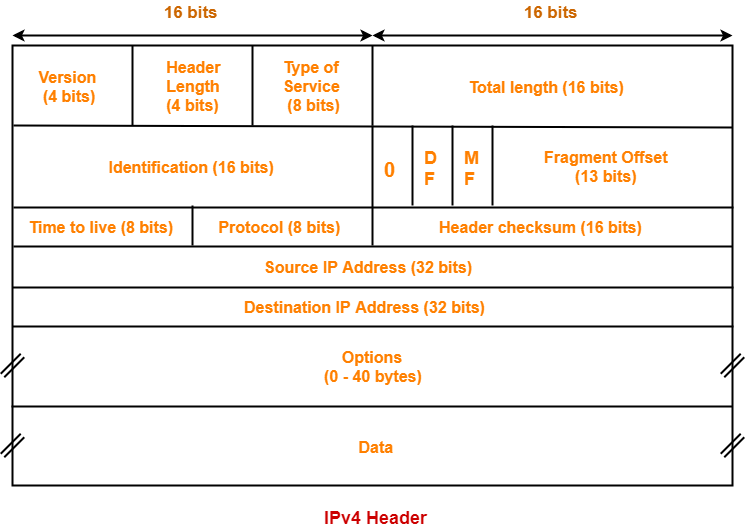
Det sker mere konkret med IP Adresser og Forwarding Tables (aka. Routing Tables), which uses some routing algorithms.

En IP address indeholder 32 bit (4 byte), som i alt giver ca. 4 millarder adresser. En byte kan indeholde et tal imellem 0 og 255. Det betyder at når man laver en ip address kan tallene imellem hvert punktum ikke kan være mere end 255. (255.255.255.255 er maks). Dette format hedder i øvrigt ”dot notation”. Hvis den sidste byte er 255 står den til at broadcaste, imens 0 er reserveret til netværks id (host id? :thinking:).

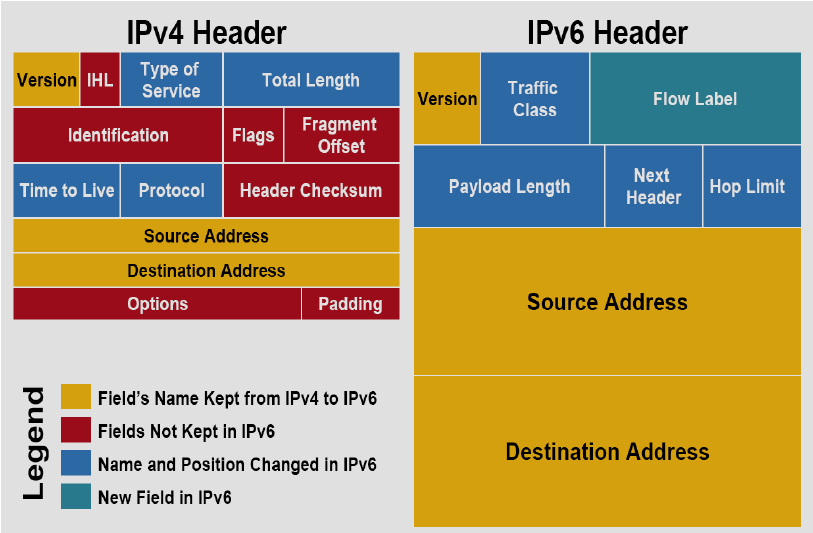
De 3 første tal i en ip addresse er netværks id’en. Det resterende er til host id.

Man bruger en subnet mask til at sige hvor meget der er en del af netværks id’en.

### IP Header



**Note**: dette taler om IPv4. IPv6 er meget anderledes og er så småt ved at blive inkorporeret)



Brugbart: alle ip addresser på det samme netværk har de samme første 3 cifre. Det sidste skal være unikt i forhold til de andre på netværket. Routerens sidste tal er ofte 1.

subnet masken er en form for formular til ens ip address. Subnet mask 255.0.0.0 siger at den første byte skal være imellem 1 og 255 og resten skal være 0. Tag med et gran salt (min opfattelse).

F.eks.: 10.200.180.5 og 10.200.180.6 er to enheder på samme netværk med to forskellige ip adresser.  
Man kunne afprøve derhjemme med ens IPv4 adresse (i ipconfig).

#### IPv4 header summary:

**IP version:** 4 eller 6  
**head len(gth):** header length in bytes  
**length:** total length, incl. data  
**time to live:** how many routers the packet can go through before it gets deleted  
**upper layer:** tells what type of data it is (udp, tcp, other)  
**header checksum:** checks for errors  
**32 bit ip source:** the IP from the origin of the packet.  
**32 bit ip destination:** the IP of the place it should arrive.

### NAT (Network Address Translation)

NAT sørger for at skifte mellem private og offentlige adresser og er det som den typiske hjemme-router sørger for.

NAT oversætter en lokal ip adresse (f.eks. 10.0.0.2:3345) til en global tilladt ip adresse (f.eks. 138.76.29.7:5001) aka. en WAN (Wide Area Network) adresse.

Alle fra samme LAN oversættes til samme ip adresse med et unikt portnummer, så man på tilbagevejen kan vide hvilken computer skulle have response pakken.

### Statisk vs Dynamisk IP adresse

**Statisk**

* Ændrer sig ikke
* Godt til servere (inklusive spil servere)
* Sættes manuelt

**Dynamisk adresse aka. DHCP (Dynamic Host Configuration Protocol)**

Gælder ALTID:

* Man får automatisk en IP adresse når man tilslutter
* Samt en subnetmask
* Tidsbegrænset (udløber, så man får en ny ip efter noget tid (timer/dage oftest)
* Godt til klienten

OFTE har man også

* Primær DNS
* Default Gateway

Derudover kan man også have nogle optional ting. Eg. sekundær DNS.

Man får tildelt en dynamisk ip via et broadcast (255.255.255.255) med et port-nummer til en DHCP server. (MAC adresser er også involveret, somehow)

### Useful CMD commands

* ipconfig
* ipconfig /all
* route
* arp

## Datalink layer

Address Resolution Protocol (ARP) sørger for at oversætte IP adresser til MAC adresser.

Also has addressing. MAC addresses are 48-bit.

* Understand principles behind the layer services
* Error detection, correction
* sharing a broadcast channel
* link layer addressing (IEEE 802.xx)
* LAN: Ethernet, VLANs
* Instantiation, implementation across various layer technologies (Ethernet, fast ethernet, Wireless, ppp)

## Security & Cryptography

4 elementer man skal sikre:

* Er dataen man sender sikret mod uvedkommende lyttere? (eavesdropping)
* Er dataen blevet ændret undervejs?
* Ved man at det er den rigtige afsender? (masquerade)
* Er systemerne sikret mod nedbrud (DDOS)

Sikring mod DOS/DDOS sker ved Firewall niveau, så man kan ikke gøre meget ved det.

De andre ting kan man løse med kryptografi.

Cryptography is about sending messages between two parties that a 3rd party cannot understand.

plain text => encrypt (algorithm + key) => send the cipher => decode (algorithm + key) => read

### Types of encryption today:

#### Symmetric

* Uses the same key to encode and decode.
* Using 128 bit for the encryption is the standard (as per 2020)
* Goals fulfilled:
  + Confidentiality (yes)
  + Authentication (yes)
  + Integrity (yes)
  + Operational Security (no)
* Types
  + DES (Data Encryption Standard), not hard to crack
    - To make it harder, you encrypt it with DES (64 bit) multiple times.
  + IDEA (International Data Encryption Algorithm) - Tough to brute force(trillions of years)
  + RC5, tough to brute force (trillions of years)
* Problems
  + How to exchange the key used, securely
  + (needs) a Key Distribution Center (KDC)
* Fast encoding

#### Asymmetric

* + Uses a public key and a private key.
  + Encodes with one key and decodes with another.
  + Goals fulfilled:
    - Confidentiality (yes)
    - Authentication (no)
    - Integrity (no)
    - Operational Security (no)
  + Types:
    - RSA (most known), 1024 bit (2048 bit)
      * 512 bit approx. brute force time: 5 months
  + Problems:
    - Can’t verify if the sender is authentic, without certification
    - Must put trust in certification authority (eg. VeriSign, Thrust, TDC)
    - Get the public key from the certification authority
      * This should solve the authentication problem
  + Encode two times. Once with private key A and once with public key A before the receiver gets the message. Another way is to use MD.
  + MD = message digest, which is a hashed version of the message and works like a checksum to see if the message has been tampered with. Used in combination with the message when encoded with a public key.
  + Slow encoding (uses prime numbers for the encryption)
  + Public/private keypair. When encoding with a public key the associated private key can decrypt it. If you encrypt with the private key it's the public key that can decrypt it.

Der er 2 måder at lave asymmetrisk på.

1. Man encoder med afsenders private key og afkoder med afsenders public key
2. Man encoder med modtagerens public key og afkoder med modtagerens private key

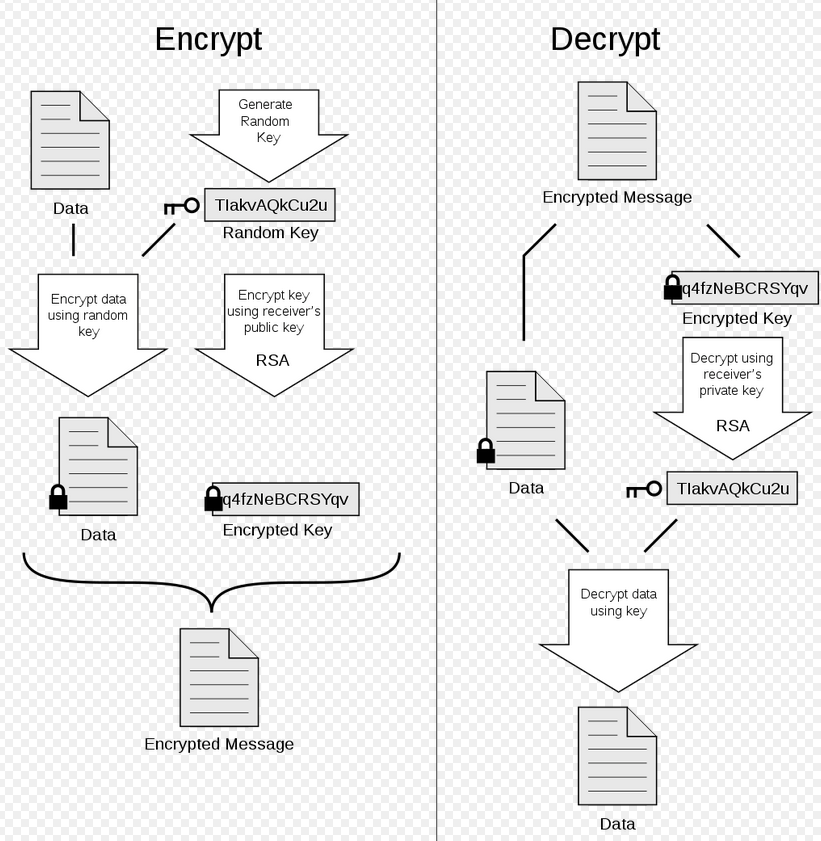
Hash = transforming the sent message into a fixed length using an algorithm (e.g. SHA-2)

Digital signature => (besked => *hash* => encode *w. private key*)

Message Authentication Code bruges til at give asymmetrisk authenticity

#### PGP - Pretty Good Privacy (Mixed asymmetric and symmetric)

* Uses asymmetric to send a symmetric key
* For the rest of the communication the symmetric key is used.



*Model that showcases how PGP works.*

### Certificates

To verify if the sender is the correct sender… Secure Socket Layer (SSL). TLS (Transport Layer Security) is the ‘successor’ to SSL, but they are largely similar. TLS is made by ISO[[17]](#footnote-17)

When you connect to a website for the first time (with SSL) you send a message requesting their certificate (and public key). You (the client) then sends back a symmetric key encrypted with the public key received. This enables fast encrypted messages between the two.

Master secret is shared between sender and receiver. It is [insert what the master secret is composed of]

### Firewalls

The purpose of the firewall is to allow “proper” traffic (non-hostile) and discard the rest.

The firewall itself should have the following characteristics:

* All traffic must pass the firewall.
* Can allow and block traffic.
* The Firewall itself should be immune to attacks.

The things it should control:

* Services (web, ftp, mail) from #port number
* IP Addresses (hosts)
* 3 (4) other areas

Types of firewall:

* Hardware: e.g. a screening router
* Software: computer based (from Operating System)
* 3rd thing

The firewall cannot protect you from allowed traffic and attacks from inside the LAN. As a general rule of thumb it doesn’t protect against viruses and other user-activated malware e.g. trojans.

Intrusion Systems are common as well. Intrusion Detection- and Intrusion Prevention- System.

# Razor MVC

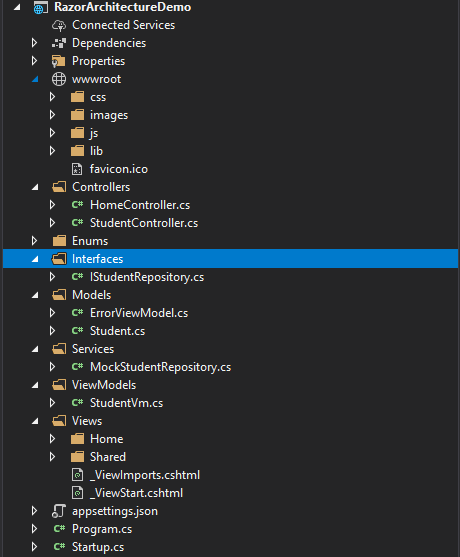
## Best Practice Architecture

The standard architecture for a loosely coupled application uses SOLID principles. This chapter will describe how the standard Razor architecture looks and how to use it.

### Folder Structure:

Notes:

* Enum folder was not a part of the lecture
* If you have View Components you make 2 Components folders. 1 in Shared and 1 in root in the project. Each Component has its own subfolder in the Component folder.
* Partials are put in the Shared folder and are prefixed with \_ and surfixed with Partial.



### Using the architecture

To use this architecture correctly you have to follow a series of steps (to have dependency injection). These steps show how to add/implement a new model into the system/architecture.

Step 1. Make a model you need to use  
Step 2. Make an interface and a service that implements the interface for the model.  
Step 3. Make a controller for the model (eg. StudentController)  
Step 4. Implement Dependency Injection (2 steps: define and register)  
 Step 4. a) Make a constructor in the controller with a parameter for the interface you   
 made in step 2. Additionally make a private field with the interface as the type.  
 Thereafter you can initialise the private field in the constructor.  
 Step 4. b) Register the dependency injection in the Startup.cs.ConfigureServices()  
 by writing:

**services.AddSingleton<ISomeInterface, ImplementationOfInterface>()** note: AddTransient and AddScoped are alternatives to AddSingleton

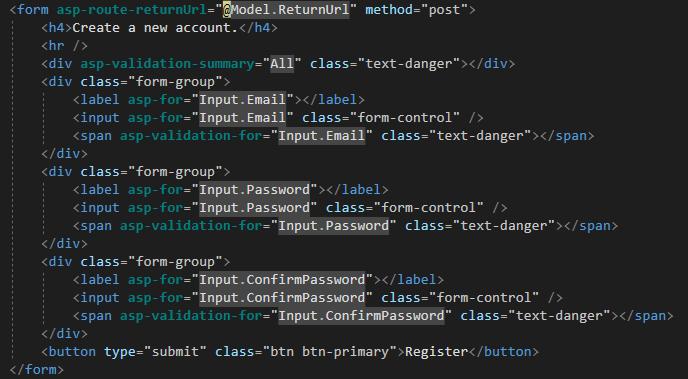
Step 5. Create a ViewModel with properties that needs to be accessed in the view (handles complex data)  
Step 6. Pass the ViewModel into the View in the controller.

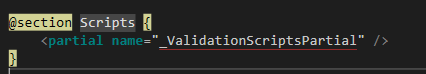
Step 7. use the @model in one of the Views attached to your controllers actions to make use of the ViewModels properties.

That concludes, how to use the basic functionality of the architecture.

To use query parameters, use a <form> tag with <input> and a submit button. (equivalent to [FromQuery] attribute). Good for searching.

## Models, Forms and Validation



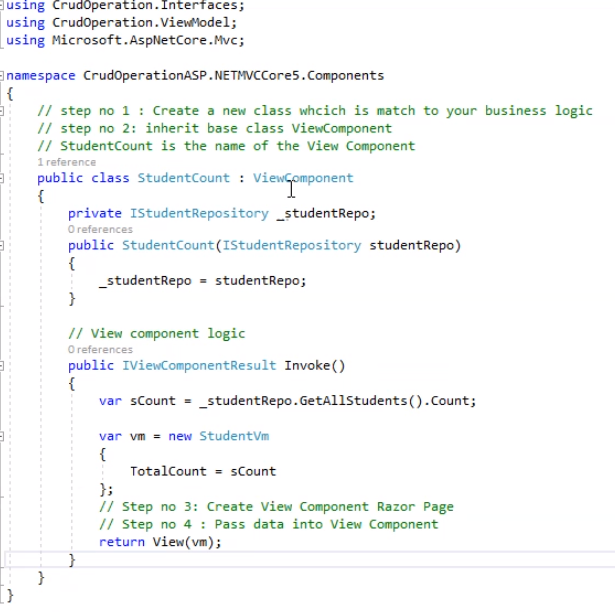


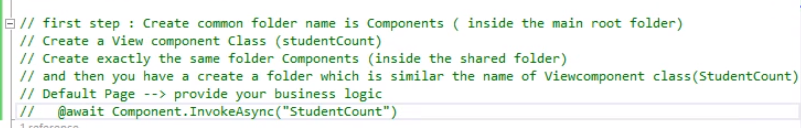
* The tag-helper **asp-for** automatically detects datatype for the input field, in addition to add validation to the field (according to the models DataAnnotations), when posting/updating content.
* For Validation messages to display you need to use the **asp-validation-for** and **asp-validation-summary** tag helpers.
* \_ValidationScriptsPartial must be added at the bottom via Scripts section for the default validation to work:

## Partials and ViewComponents

ViewComponent are primarily used, when you have dynamic content in the \_Layout.

ViewComponent can only have Invoke() or InvokeAsync().





To call the ViewComponent you use **@await Component.InvokeAsync(“nameOfComponent”)**

## Identity, Autherization and Authentication[[18]](#footnote-18)

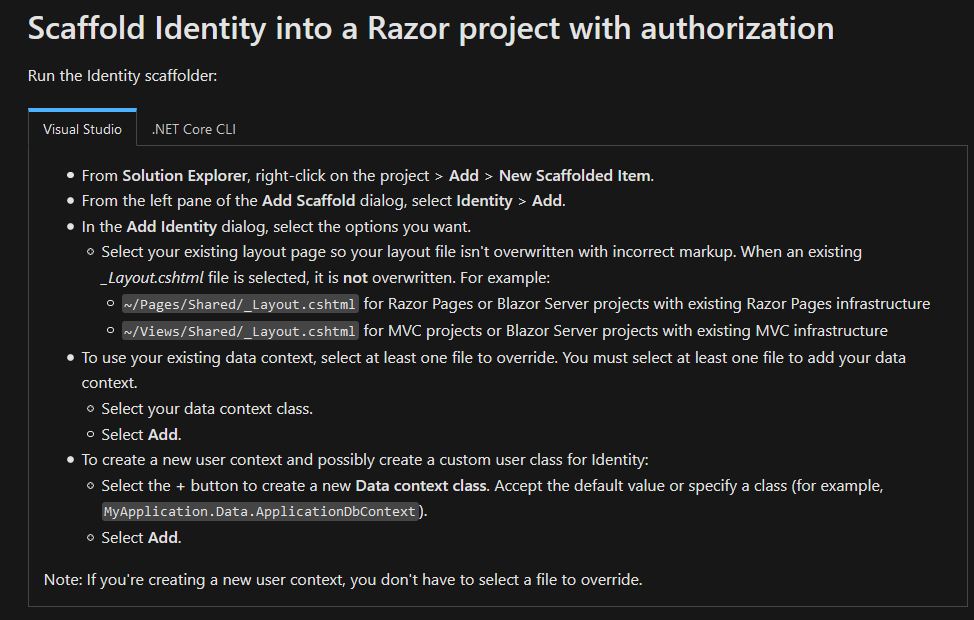
In ASP.NET there are some pre-build libraries for managing authentication and authorization (login and manage access level). This is not exclusive to MVC, but can be used in eg. Blazor as well.

AspNetCore.Identity, AspNetCore.Authentication, AspNetCore.Authorization

There is a lot of functionality in it out of the box, but if you for example want to add more information about a user you need to make a new model.

Make a new User class (typically called ApplicationUser), which inherits from IdentityUser to add additional information to a user.[[19]](#footnote-19) In addition, you need to replace any references to IdentityUser with ApplicationUser AND the dbContext needs to inherit from IdentityDbContext<ApplicationUser>.

## Scaffolding Identity (modifying the template)[[20]](#footnote-20)



## .NET Core Security

### Cross Site Request Forgery (CSRF/XSRF)[[21]](#footnote-21)

To protect against Cross Site Request Forgery (CSRF):

* Use SameSite cookies, at least lax but preferably strict
* Use antiforgery tokens (ValidateAntiForgeryToken annotation)
* Always use HTTP Post with forms

SameSite cookies is setup in the Startup.cs, along with other cookie settings:

Response.Cookies.Append("UserId", "1234", new CookieOptions {   
 SameSite = SameSiteMode.Lax   
});

### Cross Site Scripting (XSS)[[22]](#footnote-22)

Preventing XSS attacks is pretty simple if you follow these best practices:

* Validate every user input, either reject or sanitize unknown character, for example, < or > which can be used to create [scripts]
* Test every input from an external source
* Use HttpOnly for cookies, so it is not readable by Javascript (therefore an attacker can’t use Javascript to read your cookies)
* Use Markdown instead of HTML editors

#### Preventing XSS Attacks in Query Parameter

@JavaScriptEncoder.Encode([YourQueryValueHere])

### Open-Redirect attack

When redirecting, always check if the url is local. **Url.IsLocalUrl(theUrlToCheck)**

TODO: write the important parts from the below articles into this document:

### Secret Management (db connectionstring, api keys and the like)

<https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-5.0&tabs=windows>

Common security issues and best practices against them: <https://cheatsheetseries.owasp.org/cheatsheets/DotNet_Security_Cheat_Sheet.html#asp-net-mvc-guidance>

## Exception handling

[Exception handling in ASP.NET MVC (6 methods explained)](https://www.codeproject.com/Articles/850062/Exception-handling-in-ASP-NET-MVC-methods-explaine)

1. Use standard try-catch
2. Override OnException
3. Using ‘HandleError’ Attribute
4. Inheriting from ‘HandleErrorAttribute’
5. Handling HTTP Errors
6. Global error handling (override the “Application\_Error” event)

Best practice is apparently a combination of method 4 and 6.

# Blazor

See my [Blazor synopsis](https://drive.google.com/file/d/1TC0LcFb1YXRJ9EAqNAQjVTQCJ4-y826k/view?usp=sharing).

# JavaScript (EcmaScript)

JavaScript (aka. EcmaScript/ES) is a rich language used broadly in web development. It is quite different from other languages, such as C#, C++ and Java.

**3 key concepts in JavaScript (my own definition):**

1. Selection (getting the component you want to change or get value from)
2. Binding (make a connection between the html and javascript)
3. Event handling (listen to events, e.g. an on-click, so you know when it happens)

Aside from this, you must understand how JavaScript works behind the scenes. Here are 3 keywords necessary for that purpose.

**Keywords:** syntax parser, lexical environment, execution context, closure

**Syntax parser:** interprets our code and turns it into something the machine can read.

**Lexical environment:** where something is written.

**Execution context:** the context of the execution. Is it happening in the global execution context or more locally, in eg. a function.

**Closure:** “A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.”[[23]](#footnote-23)

## Array helpers

In the later versions of JavaScript (ES6+) JS has some methods you can use on your array to get the results you want.

* forEach (does something for each element in the array)
* map (makes a new array based on values from an existing array)
* filter (filters values into a new array)
* find (finds a single value from an array)
* reduce (performs a callback function (aka. it does something) on each element and puts it all into a single value.)
* every (checks if every value in the array fulfill a criteria)
* some (checks if some of the values in the array fulfill a criteria)
* sort (sorts an array based on a predicate)

### Examples

**forEach example 1:**

let names = ["Henrik", "JAMshid", "AndERS", "EBBe", "pER", "MicHAel", "PETEr"];

names.forEach((name, i, names) => {

if (name === "Henrik" || name === "pER") {

names[i] = name.toUpperCase();  
 }

else {

names[i] = name.toLowerCase();  
 }  
});

**Map example 1:**

let cars2 = [

{ brand: 'VW', model: 'Passat', fuel: 'diesel', owner\_tax: 5550 },

{ brand: 'VW', model: 'Passat', fuel: 'gasoline', owner\_tax: 460 },

{ brand: 'VW', model: 'Passat', fuel: 'hybrid', owner\_tax: 150 },

{ brand: 'BMW', model: '320i', fuel: 'diesel', owner\_tax: 4280 },

{ brand: 'BMW', model: '320i', fuel: 'gasoline', owner\_tax: 430 },

{ brand: 'BMW', model: '320i', fuel: 'hybrid', owner\_tax: 210 },

{ brand: 'Tesla', model: 'S', fuel: 'electric', owner\_tax: 0 }

];

// Extracts the model values

let carModels = cars2.map(x => x.model);

console.log(carModels);

// Extracts the brand values and puts them into objects with the property/key 'Mærke'.

let carBrands = cars2.map(c => { return { Mærke: c.brand } });

console.log(carBrands[0].Mærke);

// Same as the first, but with brands (just to display difference between carBrands and carBrands2)

let carBrands2 = cars2.map(x => x.brand);

**Map example 2:**

****

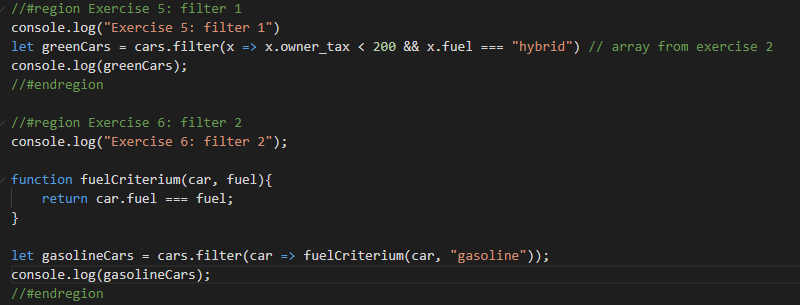
// Deserialize json to js (JSON.stringify = serialise)

let drivers = JSON.parse(driversJson);

// Extracts info from the deserialized array and returns a new array with only the extracted information (in a new type of object).

let basicInfoAboutDrivers = drivers.map(driver => { return { Kode: driver.code, Fornavn: driver.givenName, Efternavn: driver.familyName } });

**Filter**

****

**Reduce example 1**

let trips = [{distance : 48}, {distance : 12}, {distance : 6}];

// The reduce method does something to each object in the array you use it on and accumulates it into a single value.

// In this case it adds the distance from each trip together into a total distance value.

// reduce syntax:

// array.reduce(currentTotal, object) => { do something with object and currentTotal }, currentTotalInitialValue);

// currentTotalInitValue is often 0.

let totalDistance = trips.reduce((totalDistance, trip) => {

return totalDistance + trip.distance;

}, 0);

**Reduce example 2**

var desk = [  
 { type: 'sitting' },  
 { type: 'standing' },  
 { type: 'sitting' },  
 { type: 'sitting' },  
 { type: 'standing' }  
];

var deskTypes = desk.reduce((totalAccumulator, desk) => {  
 totalAccumulator = {sitting: totalAccumulator.sitting, standing: totalAccumulator.standing}

if(desk.type === "sitting"){  
 totalAccumulator.sitting += 1;  
 }

if(desk.type === "standing"){  
 totalAccumulator.standing += 1;  
 }  
 return totalAccumulator;

}, { sitting: 0, standing: 0 });

**Find**

let tesla = cars.find(c => c.brand === "Tesla");

// returns the first object fulfilling the criteria

**Every**

**Some**

## Remember these differences compared to C#

=== in JS is the same as == in C# (strict comparison), while == is a more lenient comparison.

Example:

“22” == 22 = true

“22” === 22 = false

callback function parameters:



If the parameter is enclosed in brackets [ ] it is optional.

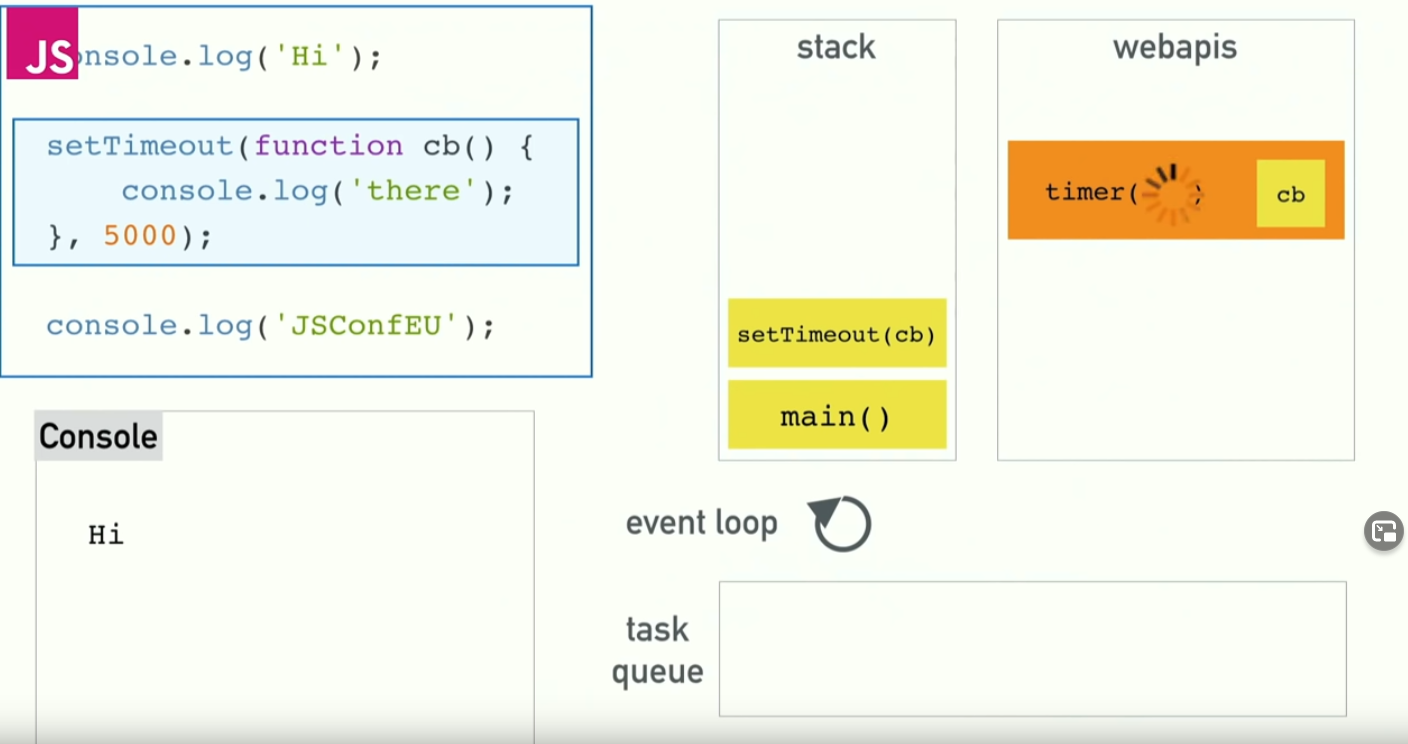
Callback functions are basically equal to (parameters) => { // do something }, altså en function som parameter

expression = returns something (has a return type)

statement = doesn’t return anything (void)

To make something run “asynchronously” use a setTimeout function with a callback function with a timer of 0 (genereally):

**setTimeout(function callback() { // do something }, 0)**

****

## JS Resources

To use JavaScript you need to understand DOM manipulation.

A good resource to learn JS from scrap: <https://www.w3schools.com/js/>

Resource for JS and DOM manipulation: <https://www.w3schools.com/js/js_htmldom_methods.asp>

<https://www.youtube.com/watch?v=8aGhZQkoFbQ> event loop (understanding asynchronous in js)

**Bundler:** browserify (extensible bundler. commonly used with watchify)

## React

**setState** giver *this*.*state* en ny værdi OG kalder render() lifecycle funktionen.

To create a react project use the command **npx create-react-app [app-name]**

React is a library not a framework, therefore it needs a lot of extensions to work optimally.

The typical extensions used are: redux, react-redux, redux-thunk and axios.

**Two-way databinding in React doesn’t exist.** Use javascript.

setSkillDescription(value){

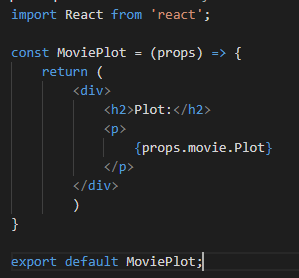
this.SkillDescription = value;  
 }

<input className="form-control" type="text" value={this.SkillDescription} onChange={e => this.setSkillDescription(e.target.value)}/>

### How components work in basic react:

There are 2 types of React components:

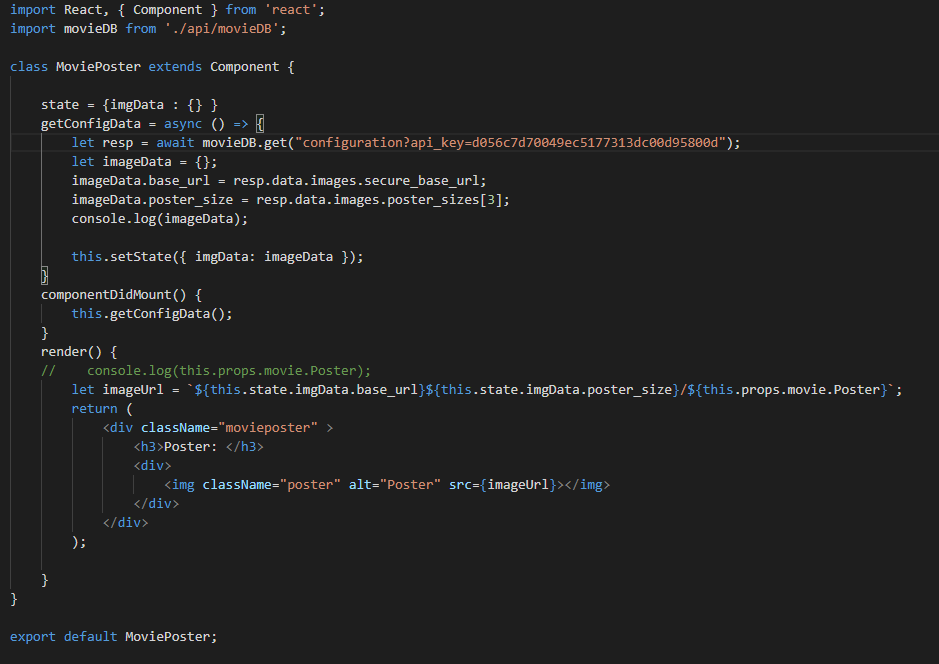
**Function component:**



A function component is basically a normal arrow function with some jsx (javascript html) that should be rendered. The props is a parameter for the function. In this case the props were passed a movie object (from another .js file) which has the property Plot (a string).

This is the simplest form of component.

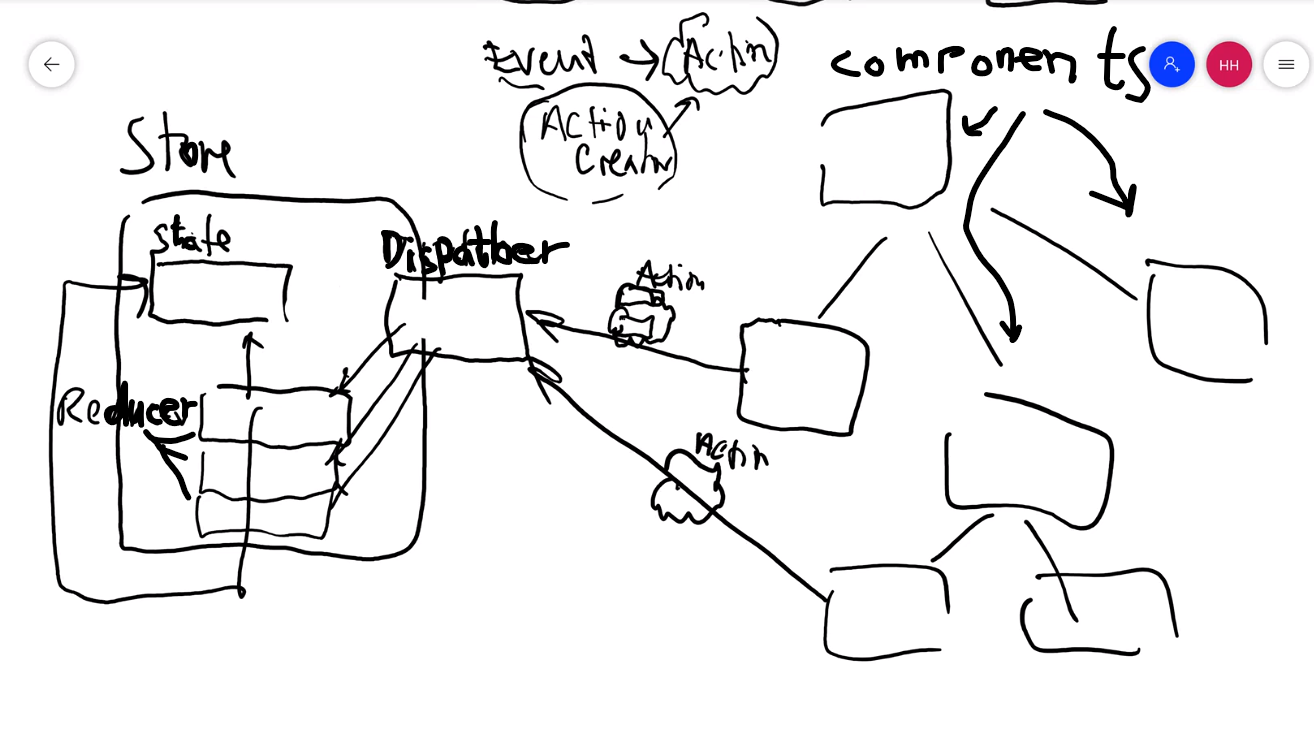
**Class component:**



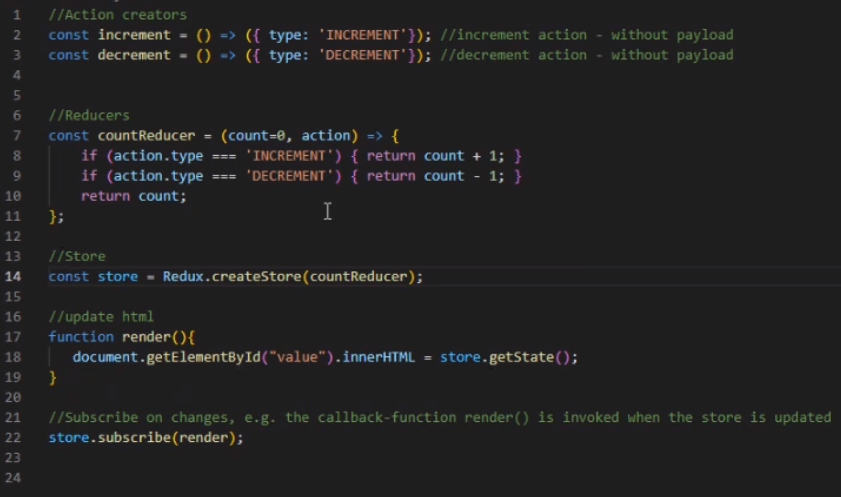
When a function component doesn’t cut it, you use a class component. You import React’s component (to extend/inherit from it). “props” is by default implemented, when you extend React’s component and you refer to it by this.props.

### Redux

Redux theory/how does it work:

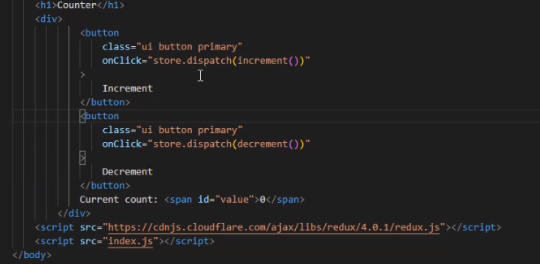


Redux (vanilla javascript, no libraries) eksempel:



Note: store.getState() is only used in pure redux (not in react-redux)

The action creators must have a property of “type” in them.



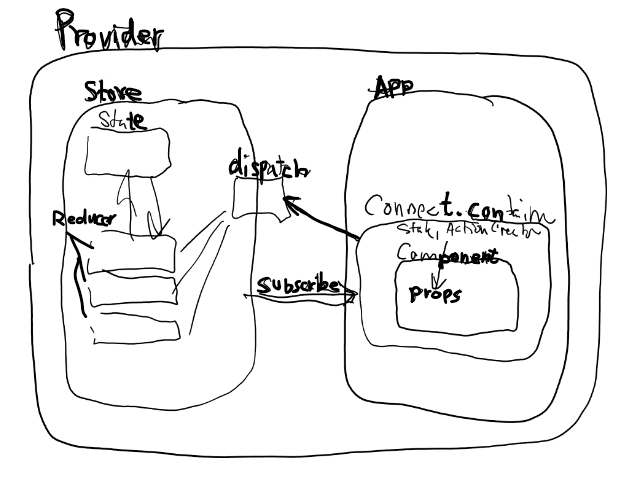
*Take note of the onClick() syntax.*

When having more than one reducer you use combineReducers in the redux lib to bundle them into one object.

### React-redux

More commonly you combine these two libraries.

**Note:** Create folders for actions, components and reducers in your architecture



Connect container (the wrapper for the component)

To wrap the component in the code you:

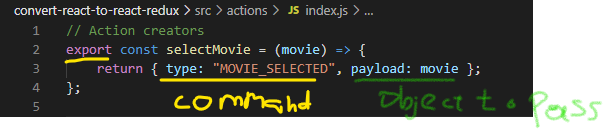
**export default connect(arg1, arg2)(ComponentName);**

export default connect(mapStateToProps, mapDispatchToProps)(MovieList);

Your state and dispatches (functions) are stored in the store.

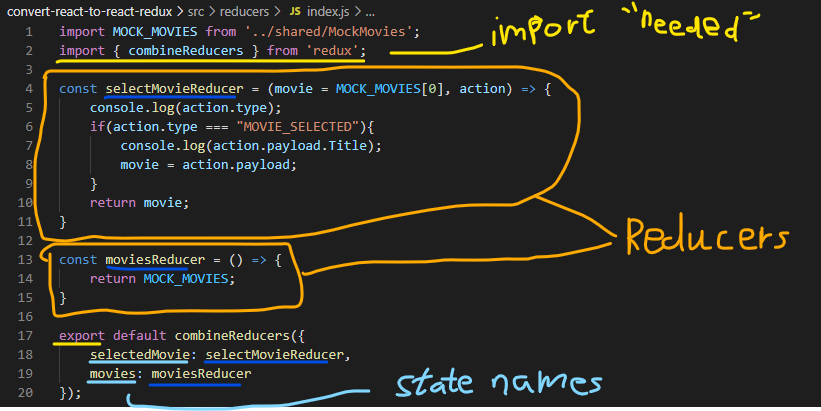
To give the **state** and **dispatches** to the **store** you need **reducers** and **actions**. Actions specify the “command”/”type” you want to take and the reducers react to the action type, which are given to the store through the combineReducers in the redux lib.

**Example action index.js:**



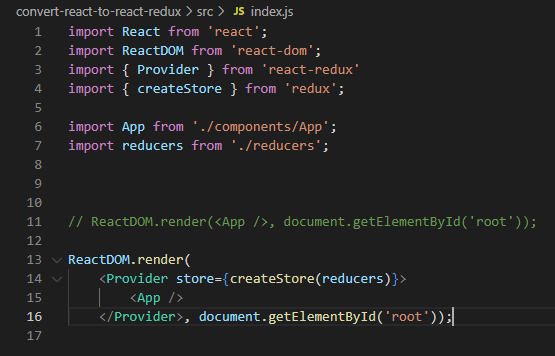
note: payload is optional ^

**Example reducer index.js:**



**Example component with react-redux:**



To make it all work you need to set it up in the index.js (src folder) likeso:

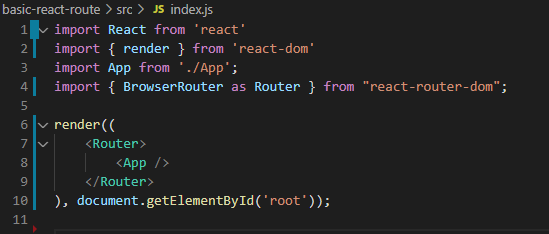
This wraps the App in the Provider, which contains the redux store.

### React-Router-dom

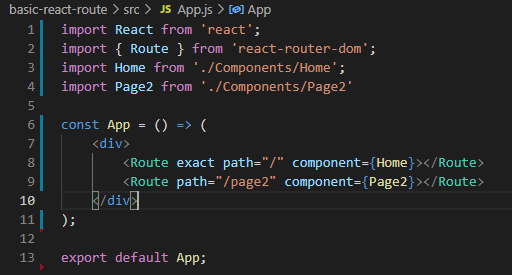
<https://www.youtube.com/watch?v=QUz3k2O3ZJU> **(The Net Ninja - Complete React Tutorial (& Redux) #25, the React Router)**

**react-router-dom** is the import name.

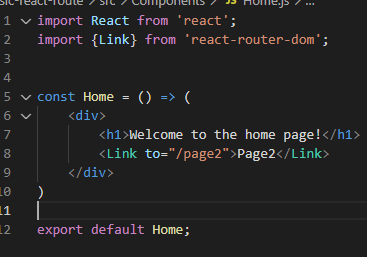
Step 1. Wrap App in the BrowserRouter (here renamed to Router)



Step 2. Use it in App (Route defines the relative path which activates the component you pass to the route as an argument). If ***exact*** is not defined in the case of the path”/” it would display that component on every page.

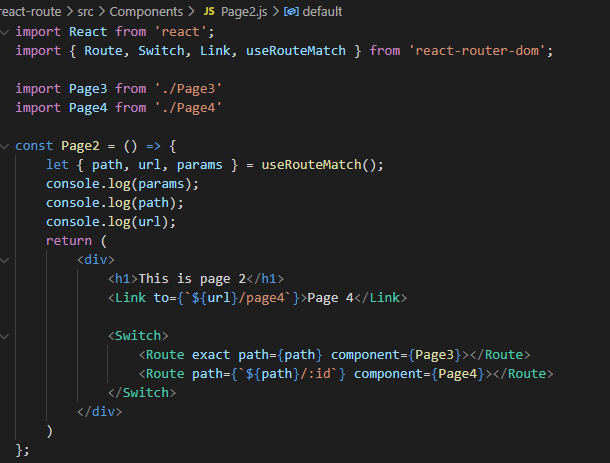


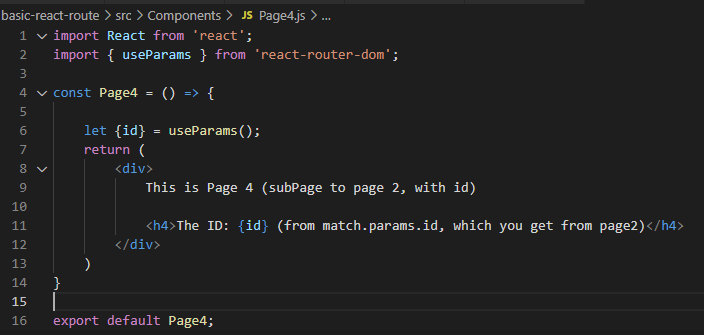
Step 3. <a> tags are replaced with <Link to=”path''>Path</Link>



**Step 4. Switch and Route-Parameters.**

Switch selects ONE of the nested routes.





### React - with libraries

Libs: react, react-redux, react-router-dom, redux-thonk, axios,

# Angular

Lots of things are imported from @angular/core (eg. Component, OnInit, Input, Output, EventEmitter.

Angular CLI is a useful tool to generate components, services etc.

Keywords:

* Pipe: used for transforming data. Eg. slice a string.

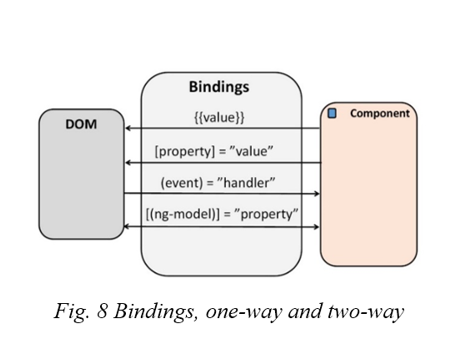
All Angular CLI commands start with “ng”

## Data bindings

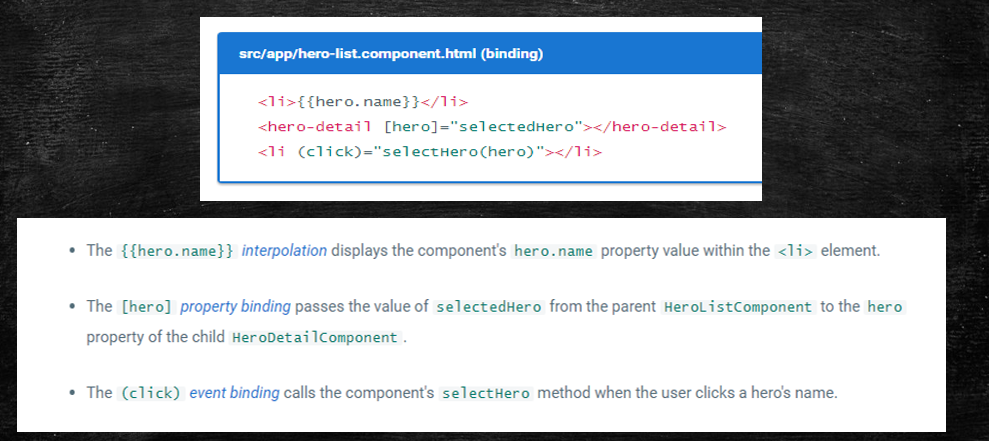
In Angular there are 5 types of binding:

* **template interpolation:**  {{nameOfProperty}}
* **property binding**: [property] = “value”
* **event binding**: (event) = “handler” (event could e.g. be: click)
* one-way binding: property-, event-, or interpolation- binding.
* **two-way binding:** [(ngModel)]=”propName” aka. banana-in-a-box syntax.

Side note: @Input, @Output annotations/decorators are put on properties in different situations.



### Example:





## Angular CLI commands[[24]](#footnote-24)

All Angular CLI commands start with “ng ”

“new [projectName]” : makes a new project (including creating of directory)

“server -o” : starts server on localhost:4200 and opens the browser on that page.

“generate component|service|otherthings [name]” : generates the thing specified (surfixed with the type you’re generating). Eg. ng generate component MovieList generates as MovieListComponent.

“g c [name]” : Shorthand for generating a component.

# SEO (Search Engine Optimization)

Search Engine Optimization often just called SEO is a what you do to optimize your ranking for search engines. There are many practices in this regard, however I will only cover a few notes about it in this chapter.

The following is a list of tips I’ve found:

* Links in user generated content - such as comments and posts - should be marked with rel=”ugc nofollow” to prevent it from affecting site SEO.

# Uncategorized notes

## Collections: Summary

Lists are fast when you need to access an element by index, but searching for an item in a list is slow since it requires a linear search.

Dictionaries provide fast lookups by key. Keys should be unique and cannot be null.

HashSets are useful when you need fast lookups to see if an element exists in a set or not.

Stacks provide LIFO (Last-In-First-Out) behaviour and are useful when you need to provide the user with a way to go back.

Queues provide FIFO (First-In-First-Out) behaviour and are useful to process items in the order arrived.[[25]](#footnote-25)

## Microservices

RabbitMQ - message broker (manages messages between services)

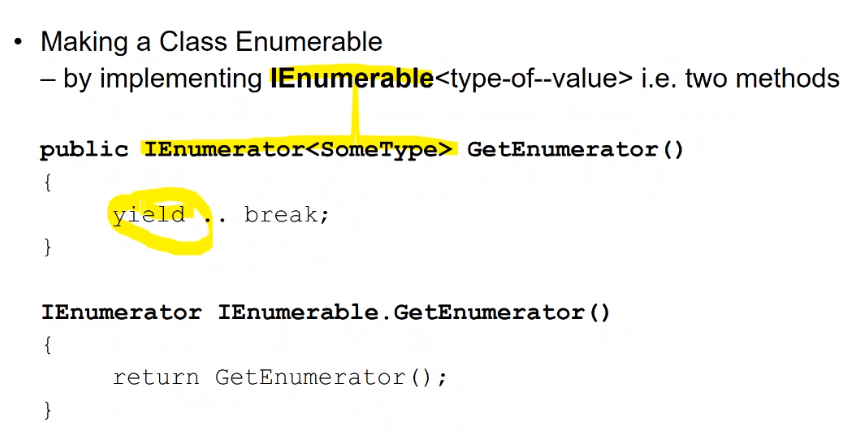
**CQRS** stands for Command Query Responsibility Segregation: most common pattern today (2021)

Mediator pattern (MediatR - a library for making it simpler)

Microservice theory: <https://www.programmingwithwolfgang.com/microservices-getting-started/>

Microservices practice: <https://www.programmingwithwolfgang.com/microservices-getting-started/>

## Making a class Enumerable C#



note:  
public IEnumerator<SomeType> GetEnumerator()  
{  
 foreach(SomeType item in SomeCollection){  
 yield return item;  
 }  
}

## Operator overload C#

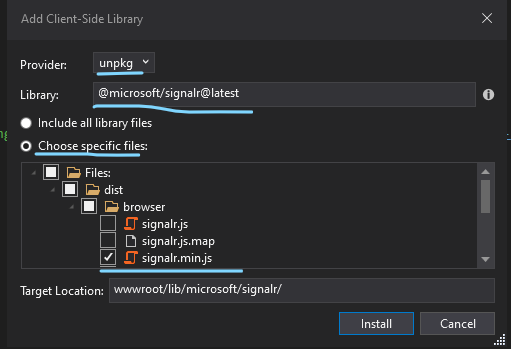
Its possible to override the operators in c# (so you can make custom behaviour for them) An example would be ‘+’ which can be seen below:

## 

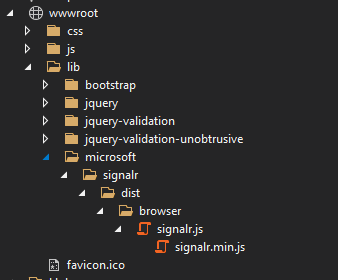
It's worth noting that, when you override the == operator you must override != as well. Probably Equals and ToHashCode as well.

## SignalR (in a C# webapp with JavaScript)

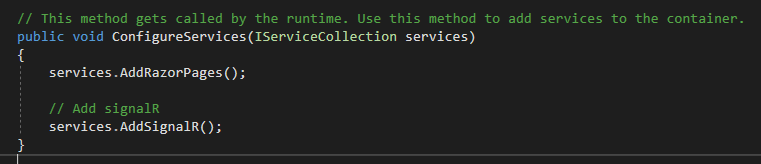
1. Get the client side signalR library:



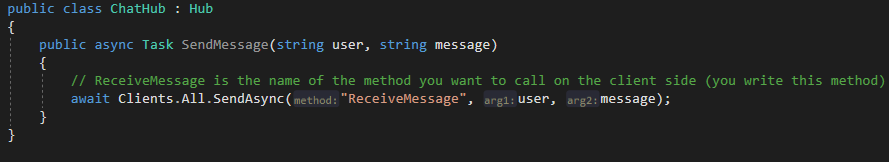
Should look similar to this when done:



1. Add SignalR server side to the Startup.cs (in ConfigureServices method)



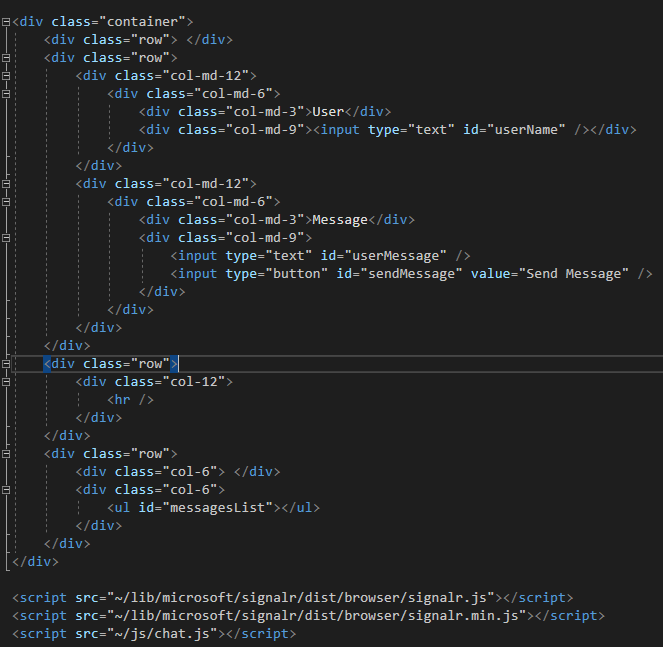
1. Make a Hub (configures which methods to call on the client side)



1. Write the code to trigger through signalR:



1. The HTML associated with the above JS:



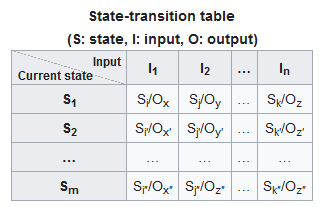
## State Machines

**Big word alert:** state machine

**Definition:** some code that keeps track of your current state and reacts to input according to the state you are currently in.

There are a few types of State Machines. In C# there is the Big Five’s State pattern[[26]](#footnote-26), which makes a lot of classes to represent each state.

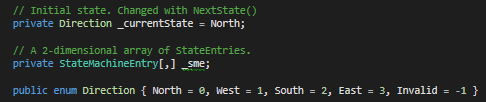
Contrarily, you can make a State Table, which holds all the possible combinations in a single class.

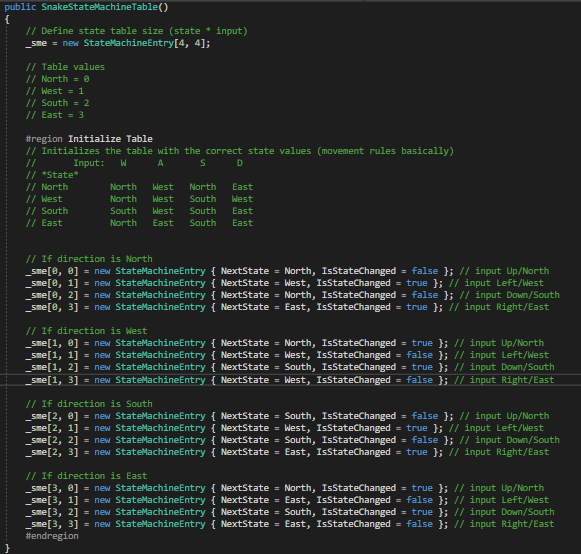


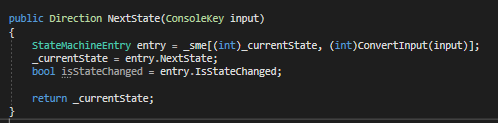
In the above table you look at the state and the input and look where they meet. That’s the change in state you see (and the output).

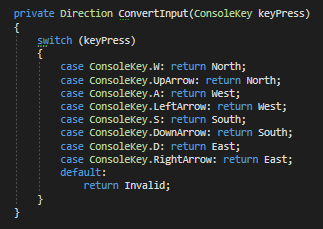
The below section illustrates a state machine for the classic Snake game.

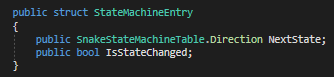
### Example state table class:





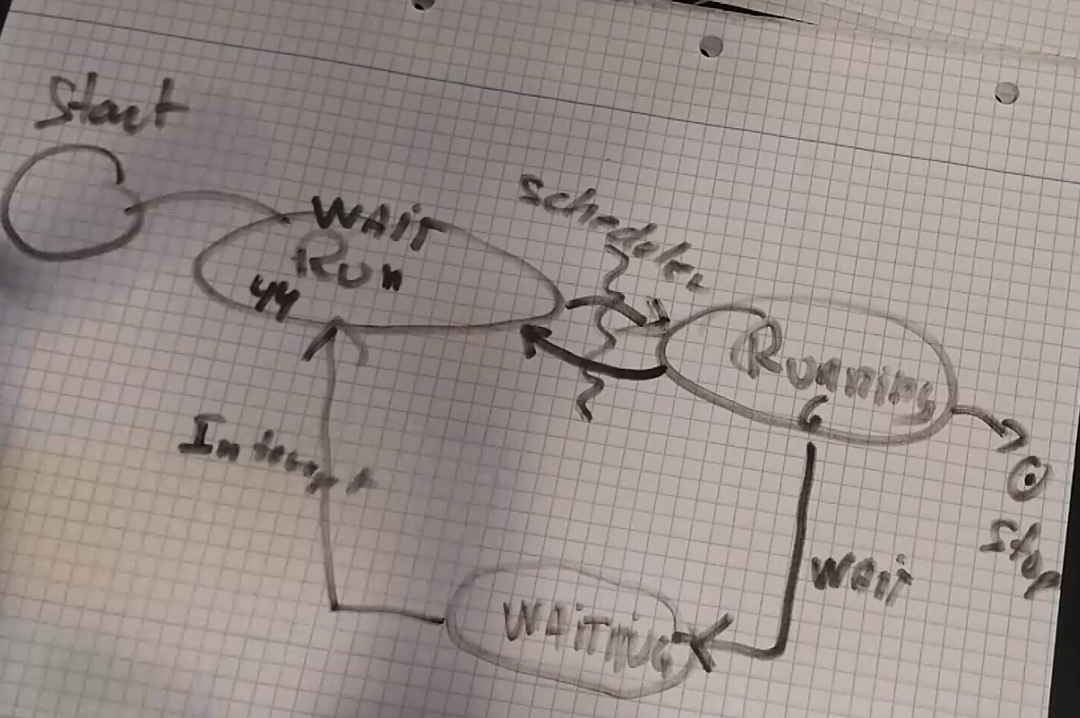




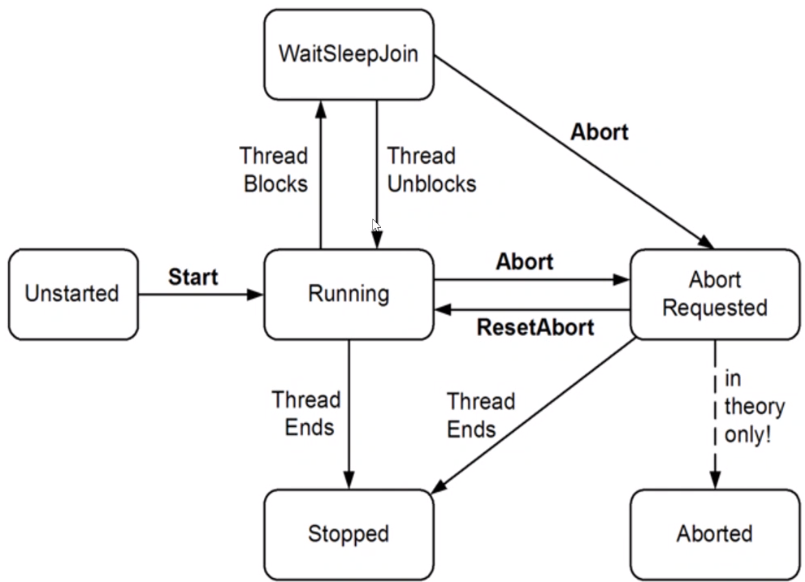
(outside the class)

## Threading: Technical details

**Simplified Process Loop[[27]](#footnote-27):** (process being something running on the computer)



**Thread life cycle:**

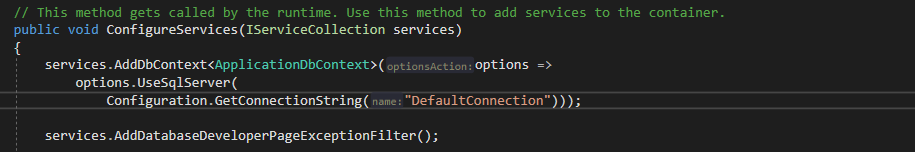


## Entity Framework

Entity Framework (EF) is a framework that works with data. It generates database tables based on your c# models. Instead of SQL queries you use LINQ queries to access data.

* Code first approach (generates database tables based on code models)
* Database first approach (generates code models based on database tables)

### How to use EF (Code First)

1. Make a folder called models and make the models you need.
2. Make a Context class for the models you need data access to (inherit from **DbContext**)
3. Define the “tables”/**DbSet<Model>** you want. (each DbSet creates a table based on model class)
4. Inject the Context class into Startup.cs with **services.AddDbContext<Context>**
5. (services.AddDatabaseDeveloperPageExceptionFilter();) for dev exceptions related to Db****

DefaultConnection is defined in appsettings.json

1. Use add-migration command to generate the database (the way it is generated is shown in the Data/Migrations folder). If it looks correct, you use the update-database command. Otherwise use the remove-migration command in Packet Manager Console and review your model classes **(hint: use attributes to define max-size, nullable etc. ([Required], [MinLength] etc.)**
2. Add controllers with scaffolding (add new scaffolded item) MVC with view and EF
3. Choose the model to scaffold for and the DbContext
   1. When you Add it creates a controller with and 5 views (CRUD)
4. Now you can modify the controllers/views as you need it.

#### NuGet package manager console commands

(EFCoreTools package is needed)

**add-migration [InsertChangeName]** (check if the new migration is correct before proceeding to update)

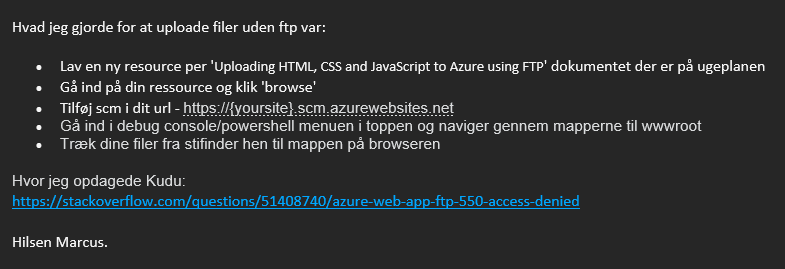
**remove-migration** (removes the latest migration, assuming you haven’t updated the database)

**update-datebase** (run this command after you’ve made changes via migrate add)

(and there are many more)

## Azure

Kudo tools (scm in url)



<https://stackoverflow.com/questions/51408740/azure-web-app-ftp-550-access-denied>

## Regular Expressions

Links til læring om regular expressions:

Nice presentation (though not C#):<https://cs.lmu.edu/~ray/notes/regex/>

Microsoft - regular expressions:<https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions>

Interactive tutorial:<https://regexone.com/>   
Slides: [RegularExpressions.pdf](https://pele-easj.dk/2021f-AvanceretCsharp/materiale/RegularExpressions.pdf) (Peters slides fra undervisning)

## Computer komponenter

**CPU:** central processing unit. Det er den, som sørger for udregninger og er den som bestemmer computerens hastighed, generelt.

CPUen fungerer ved en cyklus fra nogle inputs (register) (R1, R2, Rn…) til en ALU (beregner), som sender resultatet tilbage til R’erne. Cache er den del af CPUen som er tilsluttet bundkortets (motherboardets) BUS, som forbinder en computers komponenter.

GHz = svingninger i sekundet (1 GHz = 1 millard svingninger)

**RAM:** random access memory. Gemmer f.eks. åbne programmer når man bruger ”sleep” mode. (indeholder maskinkode) DDR3 hurtigere overførsel til CPU end DDR2. *RAM memory wall* keyword.

**I/O (input/output):** skærm, tastatur, SSD, HDD, netkort, lydkort etc.

De fleste senere programmeringssprog foregår på disken, eg. C#, python. Det kompileres om til bytekode, som RAM laver om til maskinkode via en interpreter (fortolker), som CPU’en kan læse.

**GPU:** graphical processing unit. Sørger for grafikken. De fleste CPU’er har et integreret grafikkort nu om dage, men hvis man skal bruge krævende grafik, skal man have fat på et eksternt grafikkort.

Binært system: 2-tals system. 20, 21, 22, 23, 24…

210 = 1024  
Så tallet 30250 vil have (fra højre til venstre) 0\*20, 5\*21, 2\*22, 0\*23, 3\*24 eller i andre ord fra højre til venstre fordobles tallet der tælles. (første tal er 1’ere, andet tal er 2’ere, tredje tal er 4’ere etc.)

Et KB (kilobyte) = 1024 bytes  
Et MB (megabyte) = 1024 KB  
Et GB (gigabyte) = 1024 MB  
TB (terabyte)…  
PT (petabyte)…

Bogstaver og deres konvertering til tal, så computeren kan læse dem:

Der findes nogle forskellige systemer. De mest brugte er: ASCII-tabellen (USA), UTF8, USO-889x-1, Unicode.

ASCII-tabellen kan udvides for at inkludere bl.a. æøå.

## Programmeringsdefinitioner

Summary Instance fields, variables, methods/functions, parameters, arguments, properties, classes, …

An **instance field** is a variable, but unlike normal variables instance fields have an underscore (\_) prefix, if you follow the standard naming norms. In addition, they have the accessor ‘private’ in front of their type. **private [type] [\_name];** (aka. private fields)

A **property** is also a type of variable, but instead of holding a value itself, it **references** an instance field. Properties can either be read or written, depending on how the property is set up. You access instance fields through properties. **public [type] [Name]{get{return \_instanceField; }}**

**Constructor:** a constructor is what you use, when you declare a new object. E.g.:

ClassName**(**string name**)**{  
 \_name = name;  
}

**Parameter:** parameters are written in parentheses after a method definition and have a type and a name each.  
SomeMethod**(*string name, int age, bool isMale*)**{  
 // the functionality of the method  
} **Argument:** when you initialize something, it may require some arguments, which adhere to the *parameters* specified.  
**SomeMethod(“Børge”, 23);**

***get { return \_someInstanceField; }***makes you able to read it. The similar ”set” enables you to change the value.  
***set{ \_someInstanceField = value; }***

**Logical expression:** an expression that can result in either “true” or “false”. e.g.: *10 > 5 = true*

**while loop:** a loop that runs until its logical expression is no longer true or a “break;” is used. **while(true){ run this code if(x == 2){break;}}**

**for loop**: similar to the while loop, except its iterated a number of times and is written like this: **for(int i = 0; i > someSize; i++){}**

**foreach loop:** foreach of the elements in a collection (List, array or similar) some code will be executed.

**Sentinel-controlled loop:** loops until outside influences it to stop (typically a while loop)

**Counter-controlled loop:** an example of counter-controlled loop is the foreach loop.

**if-statement**: if(*logical expression*){*code block*}

**if/else statement:** if(*logical expression*){*code block*} else{*code block*} (the else is executed if the if statements expression is not true.

**else if-statement:** an extra “if” statement, that is executed if its logical expression is true and the previous if-statement’s expression is not.

**Object:** An object is a variable of sorts, which holds a reference to one or more values. In object-oriented programing an object is made from a “template” (constructor), which is defined in a Class and contains values according to the template.

**Collection:** a group of objects. Collections come in various forms, each specialised to a certain purpose. The 5 most important collection types (in C#) are: **List, Dictionary, HashSet, Stack, Queue**.

**Class:** A Class is file, which holds C# code. In object-oriented programming it works as a template for one or more objects.

**Static**: when you define something as static you cannot make a new object of it (meaning there is only 1 instance of that object)

**Operator:** used in calculations and logical expressions. Examples separated by comma: >, +, <, -, %, /, \*, ??, ??=, etc.

#### Access types: public/private/protected

**public:** all classes have access.

**private:** nothing outside the class has access.

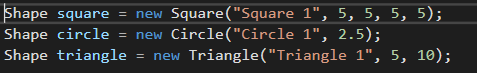
**protected:** only derived classes have access.

**Base class:** a class that gives code to other classes.

**Derived class:** a class that inherits code from a base class. Example of how to inherit from a base class: **public DerivedClassName : BaseClassName**

**Virtual:** when you make a method virtual[[28]](#footnote-28), you say enable it to be overridden in another class.

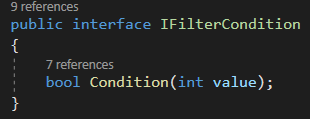
**Override:** when you override a method[[29]](#footnote-29), you replace a method (of the same name) with the override code block.

**Polymorphic behavior:** when something is polymorphic it can be created as another type. Examples: 

A square, a circle and a triangle is created with the type of Shape. This means they can all be added to the same Generic Collection: *(List<Shape> shapes = new List<Shape>();* )  
shapes.Add(circle)  
 shapes.Add(triangle)

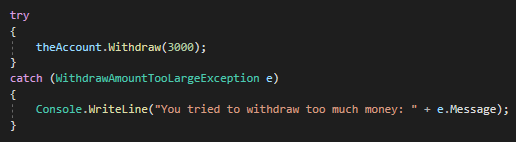
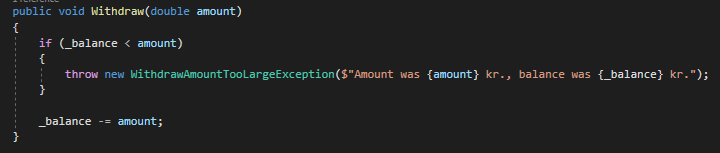
**Abstract (keyword):** when something is abstract, it means it does not have any body. Only its definition (datatype and name) is defined.

**Interface:** an interface is a class that is fully abstract, meaning all everything in it is by default abstract (and public). Starts with a capital I. Moreover, no attributes/variables are “allowed”. Example:



**Exception Handling:** handling of unwanted happenings. Exceptions are handled in 4 stages: detection, throwing, catching, handling. See BankWithExceptions solution for basics. (you can make your own exceptions, if they inherit from Exception base class)

**try-catch:** the try code block holds the code you’re trying to run, while the catch does something when an exception is thrown (the exceptions are thrown in the methods used, in this case in the Withdraw method)

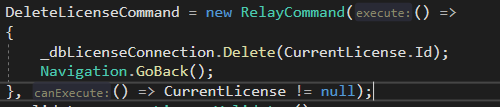
**GRASP patterns:** GRASP stands for **General Responsibility Assignment Software Patterns** and “are guidelines for assigning responsibilities to classes & objects”.

**Singleton:** a class that can only have ONE instance, which enables the data it holds to be shared in multiple files. Note: Gang of Four Singleton Pattern is considered bad practice (but single instances are not)

**Observer:** makes gui updates possible without refreshing (temp. description).

**Predicate:** Predicate in general meaning is a statement about something that is either true or false. In programming, predicates represent single argument functions that return a boolean value.

**Anonymous function:** a function without a name. Example:



The parentheses after “execute:” declares the anonymous function. You declare any parameters in there. You only need the curly brackets if the function is more than one line of code.

**Reflection**: at få information om typer når man kører programmet (runtime) eg.

*Type t = someObject.GetType();*Types used in reflection: ***Type****,* ***PropertyInfo****,* ***MethodInfo***

## MVVM Model

MVVM is a way to structure code. You have a folder structure, which consists of a Model, ViewModel and View folder. In the View folder you put the XAML files (the visual), in the Model folder you put the C# classes (models), and in the ViewModel you find c# files that connect the two (View and Model). This structure is meant to separate the View (Markup language files) from the Model files (templates for objects). Instead the ViewModel works as a liaison between them. You can make more folders to complement this. Example of other folders: Common, [FunctionSpecificFolder], Commands.

### XAML & Bindings

To make a binding between a controller in the XAML markup you do like so:

**<TextBlock Text=”{Binding Name}”></TextBlock>**

To enable this behavior, you need to set the View’s Datacontext to reference the corresponding ViewModel. When that’s done, you need to make the Name information available via a property in the ViewModel, which in turn, usually, uses Model classes.

### How to do Commands

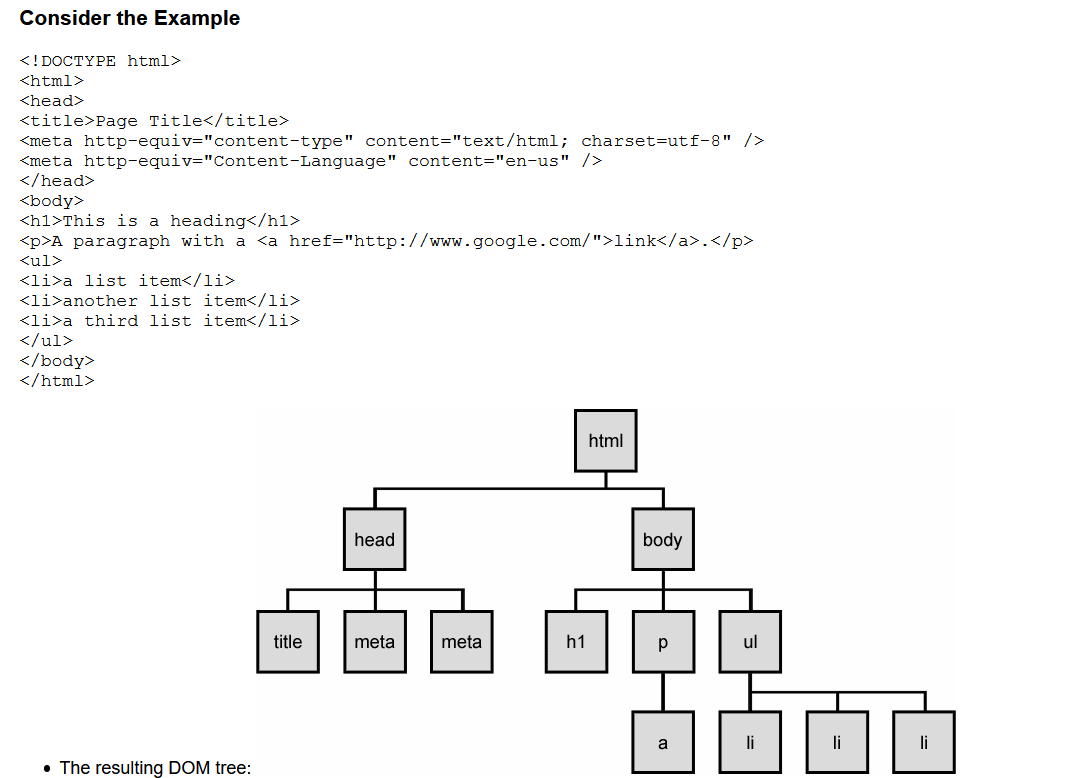
* Make Command class that implements ICommand
  + It needs a property of type Action, in addition to the methods.
* Make Command property in ViewModel
* Make the method you want the Command to execute.
* Initialize the Command in the ViewModel with the method you want it to execute.
* Bind the command to a button or the like in the View.

# Miscellaneous

* Applikationer har et begrænset område, som KUN den app kan bruge til deres filer.
* Hvis noget kan risikere at tage mere end 50ms burde man opdele opgaven til en anden CPU
* Når man henter en NuGet package er det tilsvarende til at hente en DLL-fil og lægge den ind i ens projekt.
* Når man opretter et objekt ( [objekt] [navn] = **new** [object](); ) allokere man RAM til det nye objekt.
* SignalR is something, which sends only the CHANGED data to eg. a web page instead of a whole page. (used in Blazor Server, so you don’t have to send a lot of bytes between server and client)

### DOM (Document Object Model)

Below is an example of a DOM. Self-explanatory. It is used in connection with JavaScript.



### NuGet Packages/useful 3rd party libraries

* Newtonsoft.Json (to make it easier to read/write to a json file)
* **AvaloniaUI (cross-platform version of WPF)**
* Behavior (used for view navigation, Blend thing for UWP)
* ImageResizer (fra Web Integrator)
* Entity Framework (database access, automatic db table generation based on models)
* **MSTest (c# testing framework)**
* **Selenium (blackbox testing for websites)**
* **ASP.NetCore.Cors (Cors handling for .NET Core)**
* **SwashBuckle.AspNetCore**
* <https://github.com/psf/requests> (HTTP consumption, Python)
* Stylet (a framework for WPF apps to enable easier MVVM.)
* RabbitMQ (used in connection to microservices and/or docker)

## Vue

Vue is a framework for JavaScript, which makes databinding, and more, easier.

<https://www.vuemastery.com/pdf/Vue-Essentials-Cheat-Sheet.pdf> (quick reference for Vue)

Intro to learning Vue: <https://www.vuemastery.com/courses/intro-to-vue-3/intro-to-vue3>

## External resources

<https://devdocs.io/> (contains guides to a lot of different 3rd party packages & frameworks, such as Vue, WebPack and many more)

<https://microchipdeveloper.com/> (contains a lot of technical information, such as info about each layer on the 5-layer model)

<https://www.dofactory.com/net/state-design-pattern> (lots of design patterns for C# explained in detail)

<https://www.youtube.com/watch?v=QUz3k2O3ZJU> **(The Net Ninja - Complete React Tutorial (& Redux) #25, the React Router)**

[**https://dotnetketchup.com/?hide=fsharp%2Cvbnet**](https://dotnetketchup.com/?hide=fsharp%2Cvbnet) **(C# / .NET news site)**

### Tools of the Trade

#### Business & project management

* Trello (project management)
* Asana (project/work management across teams)
* Azure Devops (project management and repository)
* GitHub (project repository)
* <https://downgit.github.io/> (Download single folder from github project)

#### Design tools

* DB Designer ([www.dbdesigner.net](http://www.dbdesigner.net)). Tool for designing ER diagrams.
* Draw.io (program to make various diagrams)
* Xtensio (persona maker)
* <https://meyerweb.com/eric/tools/color-blend/#12406E:FF9900:4:hex> (Making a color palette to your design)
* <https://htmlcolorcodes.com/color-picker/> (useful for picking colors for a design)
* <https://favicon.io/favicon-generator/> (generates favicons in different sizes based on image or text)
* Photoshop (image editing) (Alternative: Gimp)
* <https://www.remove.bg/> **(automatically removes the background from an image/picture)**
* <https://figma.com> (free design tool a bit similar to AdobeXD)
* <https://fonts.google.com/>

#### Programming

* **ReSharper (extension to help with writing code and more)**
* <https://www.youtube.com/watch?v=9BGWtTahGnw>Yaml Crash Course (JSON alternative)
* [charts.js](https://www.chartjs.org/docs/latest/) (line charts, pie charts, bar charts…)
* Keycloak (Identity Provider you can use with OAuth2)

#### Testing

* <https://validator.w3.org/> (Validate HTML)
* <https://jigsaw.w3.org/css-validator/> (Validate CSS)
* <https://whichloadsfaster.zomdir.com/>(page load time comparison)
* [https://www.fastorslow.com**/**](https://www.fastorslow.com/)(global website performance test)
* <https://gtmetrix.com/> (web vitals and more)
* <https://webpagetest.org/> (web vitals and more)
* <https://tools.pingdom.com/> (page load time and more)
* <https://ahrefs.com/> (SEO and “performance crawl”)

#### Other

* <https://www.netlify.com/> (deploying to web, looks like its worth checking out)
* <https://databox.com/> Samling af data i et dashboard. Integreres med f.eks. Google Analytics, Zapier og Asana.
* <https://ngrok.com/> (make your localhost on a specific port available, temporarily and securely, for anyone to see)

1. Rasmussen, J., ”The Agile Samurai” p. 51-53 [↑](#footnote-ref-1)
2. <https://trello.com/c/vnF8VDe5/4-agile-inception-deck> [↑](#footnote-ref-2)
3. <https://manifesto.co.uk/scrum-roles-scrum-team/> [↑](#footnote-ref-3)
4. <https://geekbot.com/blog/sprint-review-vs-sprint-retrospective-the-critical-difference/> - great article about the difference between review & retrospective + suggestons for completing each. [↑](#footnote-ref-4)
5. <https://stormotion.io/blog/how-to-write-a-good-user-story-with-examples-templates/> [↑](#footnote-ref-5)
6. <http://informatik-gym.dk/glossary/gestaltlove/> [↑](#footnote-ref-6)
7. <https://www.nngroup.com/articles/ten-usability-heuristics/> [↑](#footnote-ref-7)
8. <https://www.darkpatterns.org/types-of-dark-pattern> [↑](#footnote-ref-8)
9. <https://whatis.techtarget.com/definition/Single-Responsibility-Principle-SRP> [↑](#footnote-ref-9)
10. <https://stackify.com/solid-design-open-closed-principle/> [↑](#footnote-ref-10)
11. <https://stackify.com/solid-design-liskov-substitution-principle/> [↑](#footnote-ref-11)
12. <https://stackify.com/interface-segregation-principle/> [↑](#footnote-ref-12)
13. <https://stackify.com/dependency-inversion-principle/> [↑](#footnote-ref-13)
14. <https://www.youtube.com/watch?v=il9gl8MH17s&t=632s> Async/Await (example and deep dive) [↑](#footnote-ref-14)
15. <https://rapidapi.com/blog/api-vs-web-service/> [↑](#footnote-ref-15)
16. <https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/> [↑](#footnote-ref-16)
17. The International Organization for Standardization [↑](#footnote-ref-17)
18. <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-5.0&tabs=visual-studio> [↑](#footnote-ref-18)
19. <https://www.youtube.com/watch?v=NV734cJdZts> (Extend IdentityUser) [↑](#footnote-ref-19)
20. <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/scaffold-identity?view=aspnetcore-5.0&tabs=visual-studio> [↑](#footnote-ref-20)
21. <https://www.programmingwithwolfgang.com/cross-site-scripting-in-asp-net-core/> [↑](#footnote-ref-21)
22. <https://www.programmingwithwolfgang.com/cross-site-request-forgery-csrf-in-asp-net-core/> [↑](#footnote-ref-22)
23. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures> [↑](#footnote-ref-23)
24. <https://www.sitepoint.com/ultimate-angular-cli-reference/> [↑](#footnote-ref-24)
25. <https://programmingwithmosh.com/net/csharp-collections/> [↑](#footnote-ref-25)
26. <https://www.dofactory.com/net/state-design-pattern> [↑](#footnote-ref-26)
27. Process Loop is a self-defined keyword [↑](#footnote-ref-27)
28. public virtual int CalcSomething(){// base functionality} [↑](#footnote-ref-28)
29. public override int CalcSomething(){// new (specialized) functionality} [↑](#footnote-ref-29)