ASP.NET Blazor 8

Ali Ilci

B. A.

Einleitung

- Kursablauf & Organisatorisches
- Räumlichkeiten
- Vorstellungsrunde
  - Hintergrund, Motivation und Vorkenntnisse
- Bei Unklarheiten: Möglichst sofort fragen!

Über mich – Ali Ilci

- Freiberuflicher IT - Trainer (MCT)
- Fachinformatiker
- Erziehungswissenschaftler
- Themenschwerpunkte:
  - ASP.NET Core / MVC / WebApi / Blazor
  - C#
  - Html5 & Css3
  - JavaScript / TypeScript

- www.ilci.de

.NET Core

Was ist .NET Core

- Plattformübergreifendes Framework mit dem Ziel so universell wie möglich eingesetzt zu werden
- Open Source
- Bibliotheken werden durch NuGet verteilt
- Typische Einsatzszenarien sind zurzeit ASP.NET Webanwendungen, Konsolen-Apps sowie UWP-Apps
- Ist keine Untermenge des .NET Framework
- Besitzt ein Command Line Interface (CLI)

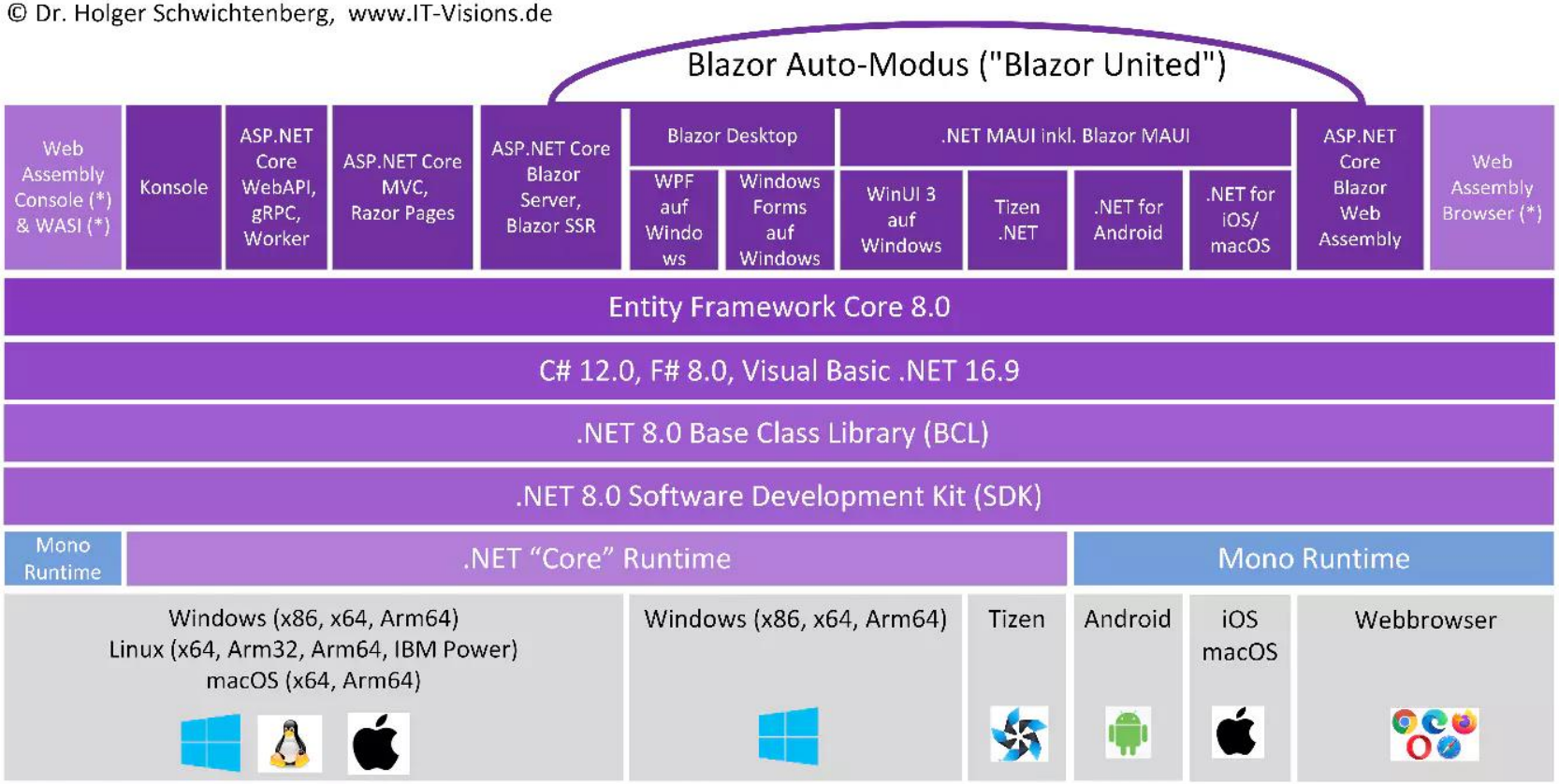
Full .NET Framework vs. .NET Core

Full .NET Framework	.NET Core
Vollständiges „etablierte“ Framework	Modulare Version des .NET Framework
Nur Windows	Cross-Platform
	Keine Untermenge vom .NET Framework
	Windows, Linux, Mac
	Implementation des .NET Standard

.NET Familie 2023

Aufbau von .NET 8.0

© Dr. Holger Schwichtenberg, www.IT-Visions.de

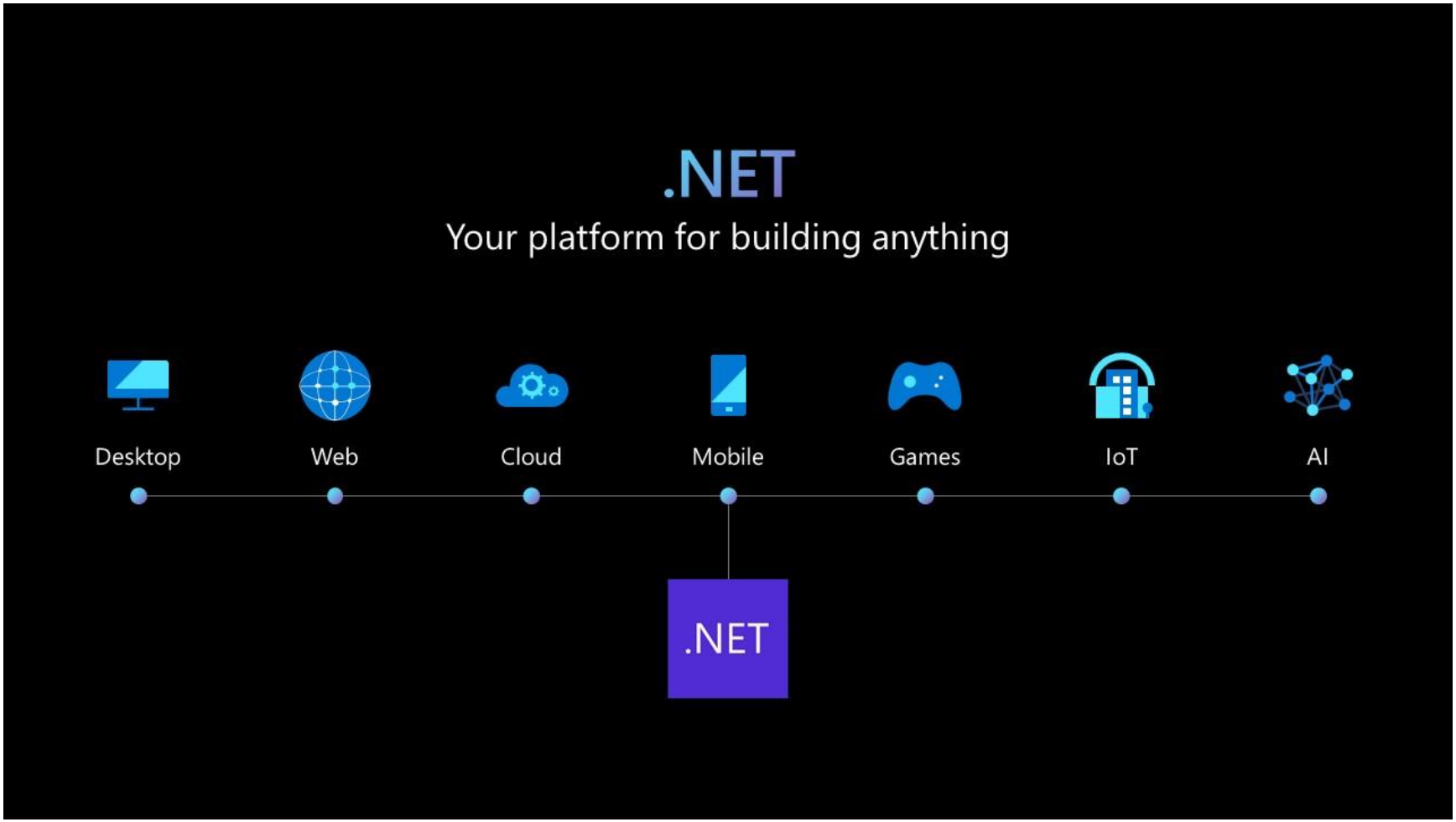


(*) Experimentelle Implementierungen ohne Support seit .NET 7.0 und auch noch in .NET 8.0

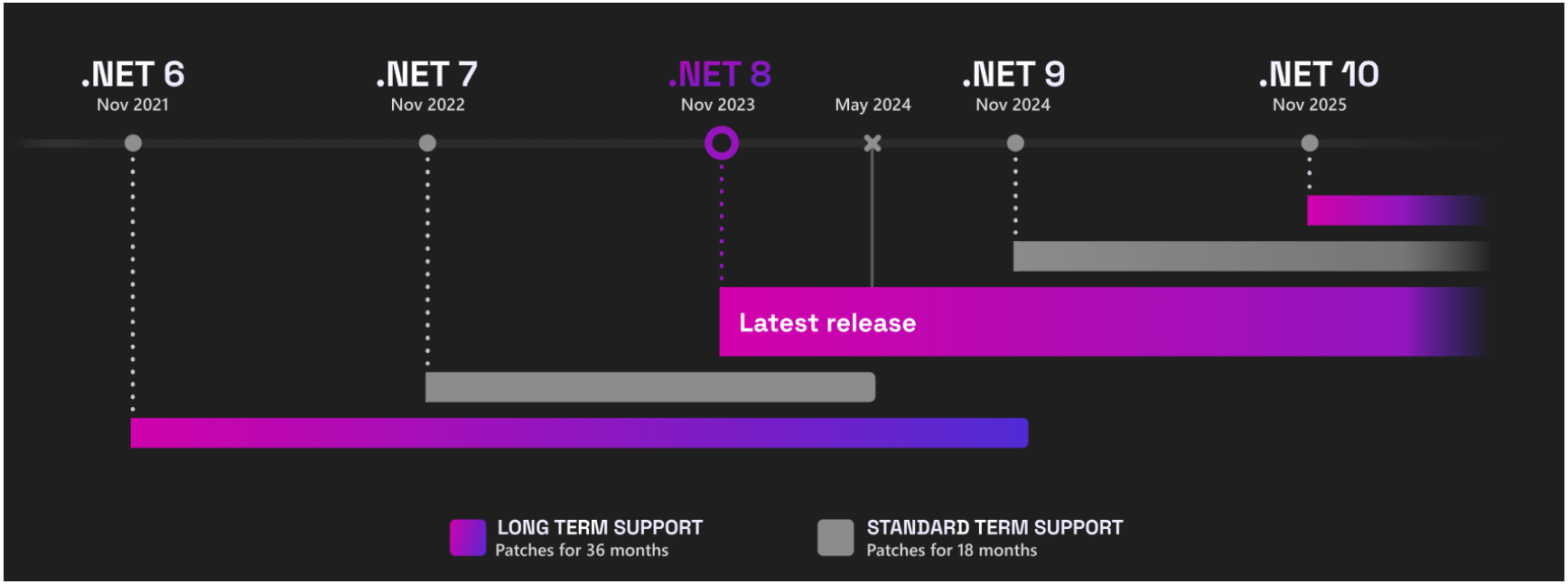
https://www.heise.de/blog/Neu-in-NET-8-0-2-Neue-Anwendungsarten-9581213.html

Holger Schwichtenberg Stand: 22.Dezember 2023

.NET 8 Platform



.NET Schedule



https://dotnet.microsoft.com/en-us/platform/support/policy

.NET CLI

- Command Line Application, die zum Entwickeln von Anwendungen auf allen Plattformen verwendet wird
- Führt die App aus und hosted die CLR
- OpenSource / SDK
- Erweiterbar (z.B. Entity Framework Befehle)
- Dokumentation
  - https://docs.microsoft.com/de-de/dotnet/core/tools/

.NET CLI Befehle „dotnet“ (Auszug)

Befehl	Beschreibung
new	Erstellt ein neues Projekt, eine Konfigurationsdatei oder eine Lösung auf Grundlage der angegebenen Vorlage.
restore	Stellt die Abhängigkeiten und Tools eines Projekts wieder her
build	Erstellt ein Projekt und alle seine Abhängigkeiten
publish	Packt die Anwendung und ihre Abhängigkeiten in einen Ordner für die Bereitstellung auf einem Hostsystem
run	Führt Quellcode ohne explizite Kompilierungs- oder Startbefehle aus.
test	.NET-Testtreiber, der verwendet wird, um Komponententests auszuführen.
pack	Packt den Code in ein NuGet-Paket
clean	Löscht die Ausgabe eines Projekts.
sln	Ändert eine .NET Core-Projektmappendatei.
store	Speichert die angegebenen Assemblys im Laufzeitpaketspeicher

Themen

Repository

Tools

Bootstrap

Model

Web Technologien

JS Interopt

Komponenten

Theme

Logging

Authentifizierung / Autorizierung

DataAnnotations

Routing

Validation

Databinding

RCL

Http-Client

Razor

Custom Elements

Fehlerbehandlung

Dependency Injection

Asynchronität

Data Annotations

Best Practices

Responsive Web Design

C# 10/11/12

C# 10 / 11 / 12

Exkurs

Implicit & Global Using Statements

- Vermeidet wiederholte Auflistung von Usings
- Aktivierbar in Projektdatei (Default-Einstellung)
  `<ImplicitUsings>enable</ImplicitUsings>`
- Globale Usings um Namespace projektweit bekannt zu machen

```
//Vorher
using System;
using System.Collecions.Generic;
using Microsoft.AspNetCore.Http

//Neu mit Implicit Using Statements
-

//Neu mit Global Using Statements in separater CS-Datei
global using Microsoft.AspNetCore.Http
```

File Scoped Namespaces

- Einrückung der Namespaces
- Konfigurierbar für gesamte Solution mit .editorconfig

```
//.editorconfig
csharp_style_namespace_declarations = file_scoped
dotnet_diagnostic.IDE0161.severity = suggestion
```

```
namespace MeinNamespace.Services
{
    public class MeineKlasse
    {}
}

//File Scoped Namespaces
namespace MeinNamespace.Services;

public class MeineKlasse
{}
```

MaxBy/MinBy & DateOnly and TimeOnly

- **DateOnly** bzw. **TimeOnly** repräsentieren entweder das Datum **oder** die Uhrzeit im Gegensatz zu DateTime
- **MinBy/MaxBy** geben ganzes Objekt zurück

```
//DateOnly & TimeOnly
DateOnly date = DateOnly.MinValue; //01.01.0001 ohne Zeit
TimeOnly time = TimeOnly.MinValue; // 12:00

//MaxBy/MinBy
List<Person> people = new List<Person>
{
  new Person { Name = "John Doe", Alter = 25},
  new Person { Name = "Jane Doe", Alter = 23}
}

var person = people.MaxBy(c => c.Alter); // John Doe
```

Raw String Literal / Required Member

- Unformatierte Zeichenfolgen-Literale können beliebigen Text enthalten, ohne dass Escapezeichen erforderlich sind
- Required – Modifizierer gibt an, dass das Feld / Eigenschaft von allen Konstruktoren oder via Objektinitializer initialisiert werden muss

```
//String Literal vor C#11
var message = "Der folgende Text ist sehr \"wichtig\" ".

//ab C#11
var message = """Der folgende Text ist sehr "wichtig" """.


//Required Member
public required string FirstName {get; init;}
```

Primärkonstruktoren C# 12

- Verkürzte Schreibweise von Konstruktoren. Achtung parameterloser Konstruktor ist nicht mehr vorhanden und muss separat deklariert werden. Dadurch Aufruf des Primärkonstruktors

```
public class Kunde(Guid kundeId, string name, float preis)
{
   public Guid KundeId { get; set; } = kundeId;
   public string Name { get; set; } = name;
   public Kunde() : this(Guid.Empty, "") { }

   public override string ToString()
   {
    return $": {Name} {preis}";
   }
}
```

Vereinfachte Initialisierung von Mengen, Spread, Opt. Parameter in Lambdas C# 12

- Syntax mit eckigen Klammern wie in Javascript
- Spread-Operator möglich (Array aus anderen Arrays)
- Optionale Parameter in Lambdas

```
// Vorher
string[] fruechte = new string[] {"Banane", "Orange"};

//Nachher
string[] fruechte = ["Banane", "Orange"];

//Spread-Operator
string[] alleFruechte = [.. fruechte, "Apfel"];

//Lambda mit opt. Parameter
var bru = (decimal x, decimal mw = 1.19) => (x * mw);
```

ASP.NET Core 8

Überblick ASP.NET Core 8

ASP.NET Core 8
(MVC + WebAPI + MinimalApi + Razor Pages + SignalR + gRPC + Blazor (SSR, Server, Blazor WebAssembly)

C# 12

.NET 8 Base Class library

.NET Core
Windows & Linux & MacOS

ASP.NET Blazor

Was ist Blazor?

ASP.NET Core Blazor ist Microsofts Webentwicklungsframework zur Entwicklung von interaktiven Single-Page-Applications (SPA)

mit C# und HTML

Zurzeit drei Hauptarten von Blazor

- *Blazor WebAssembly* (aka Client-Side)
- *Blazor Server* (aka Server-Side)
- Blazor Static Server-side-Rendering (aka SSR seit .NET 8 als MPA)

Vorteile von Blazor

- Entwickeln mit C# statt JavaScript
- Nutzung des .NET-Ökosystems von .NET Bibliotheken
- Seit Version 6 Native Dependencies

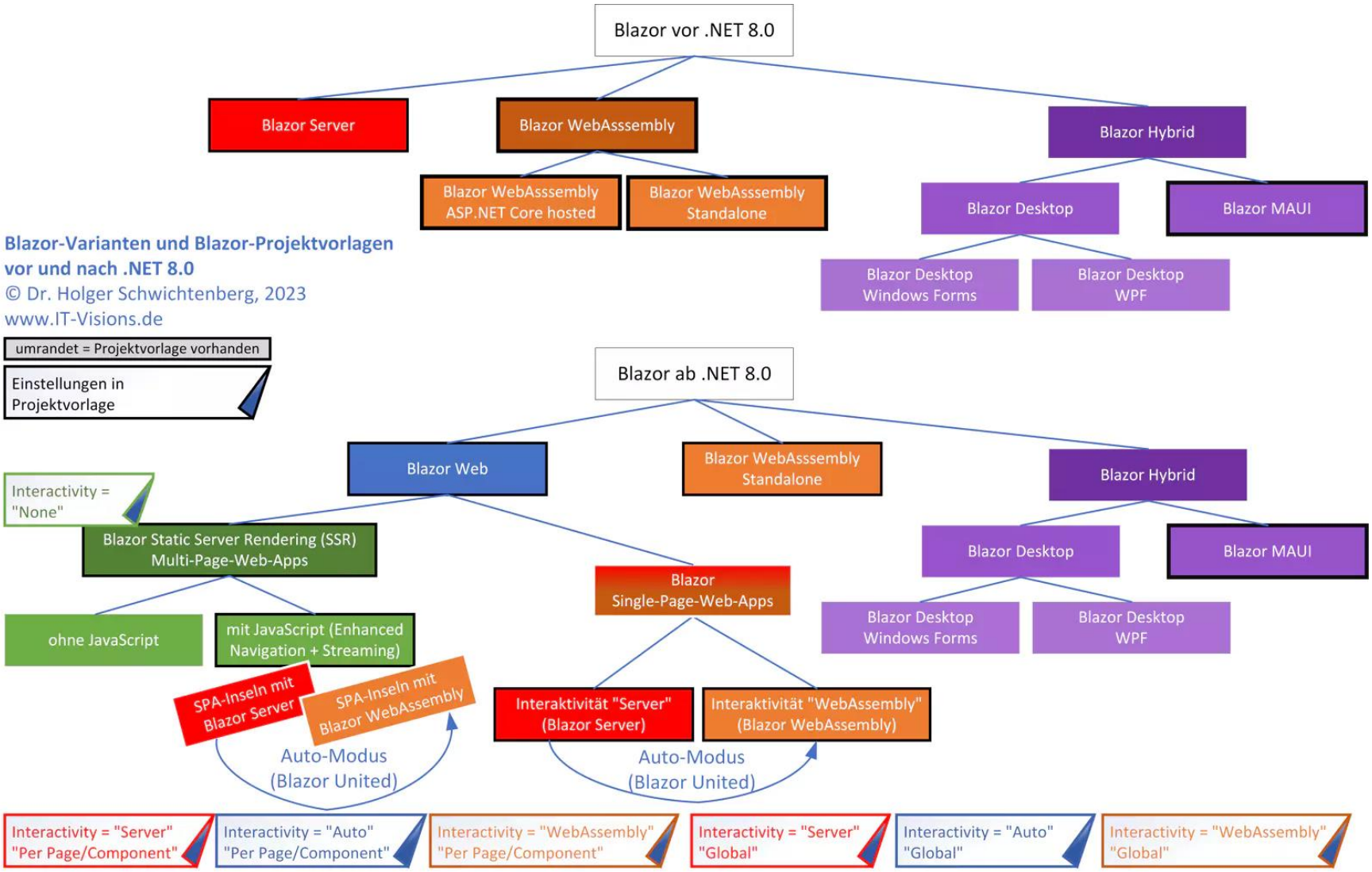
WebAssembly

WebAssembly (wasm) ist ein offener Standard, der vom W3C festgelegt wurde. Er definiert einen Bytecode zur Ausführung von Programmen innerhalb von Webbrowsern, kann aber auch außerhalb von diesen genutzt werden. Ziel der Entwicklung war es, leistungsstärkere Webanwendungen als bisher zu ermöglichen, sowohl was die Ladezeiten als auch die Ausführung betrifft

Quelle: https://de.wikipedia.org/wiki/WebAssembly

Webseite: https://webassembly.org/

Überblick ASP.NET Core Blazor 8

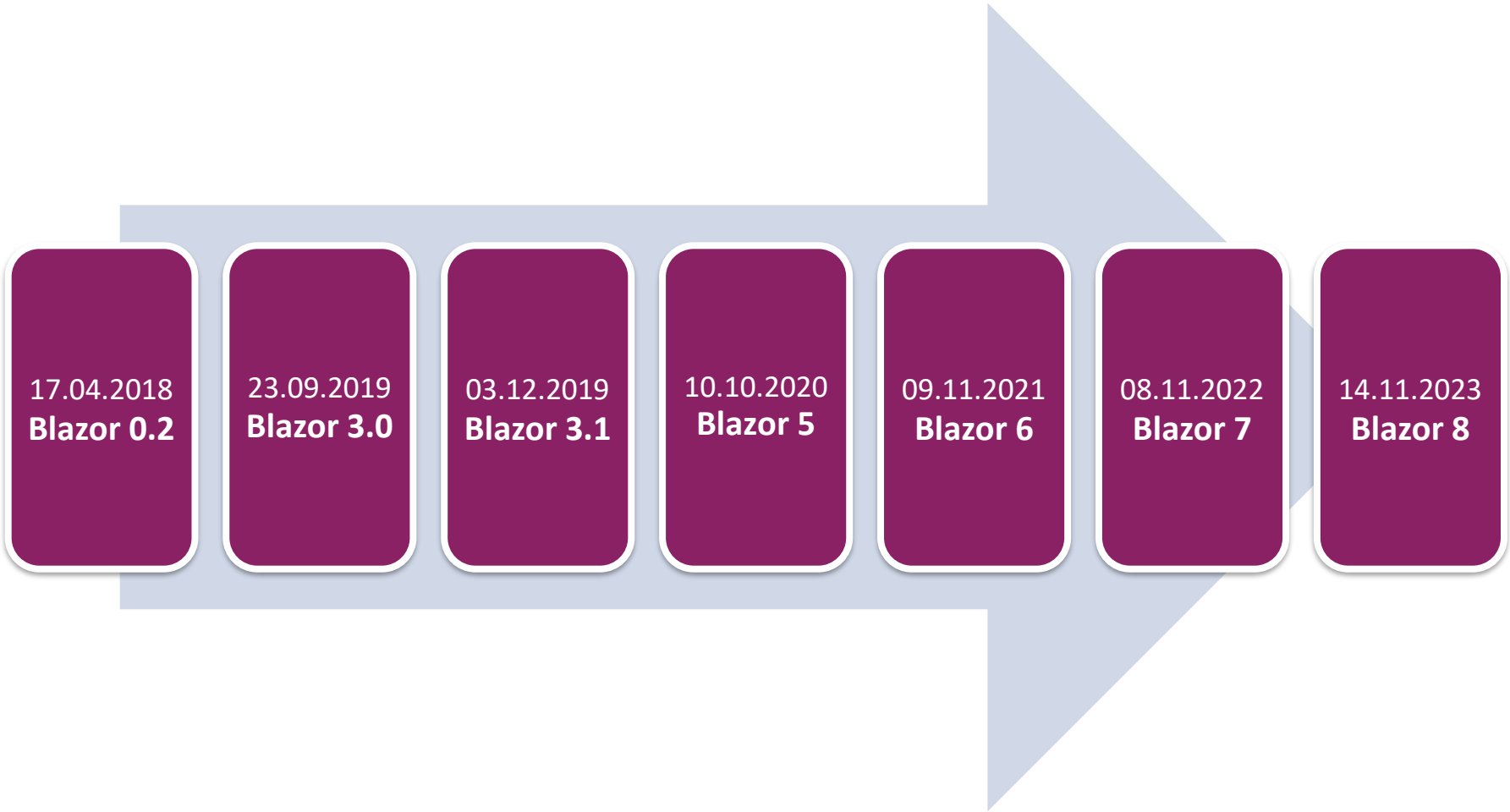


NEU in ASP.NET Core Blazor 8

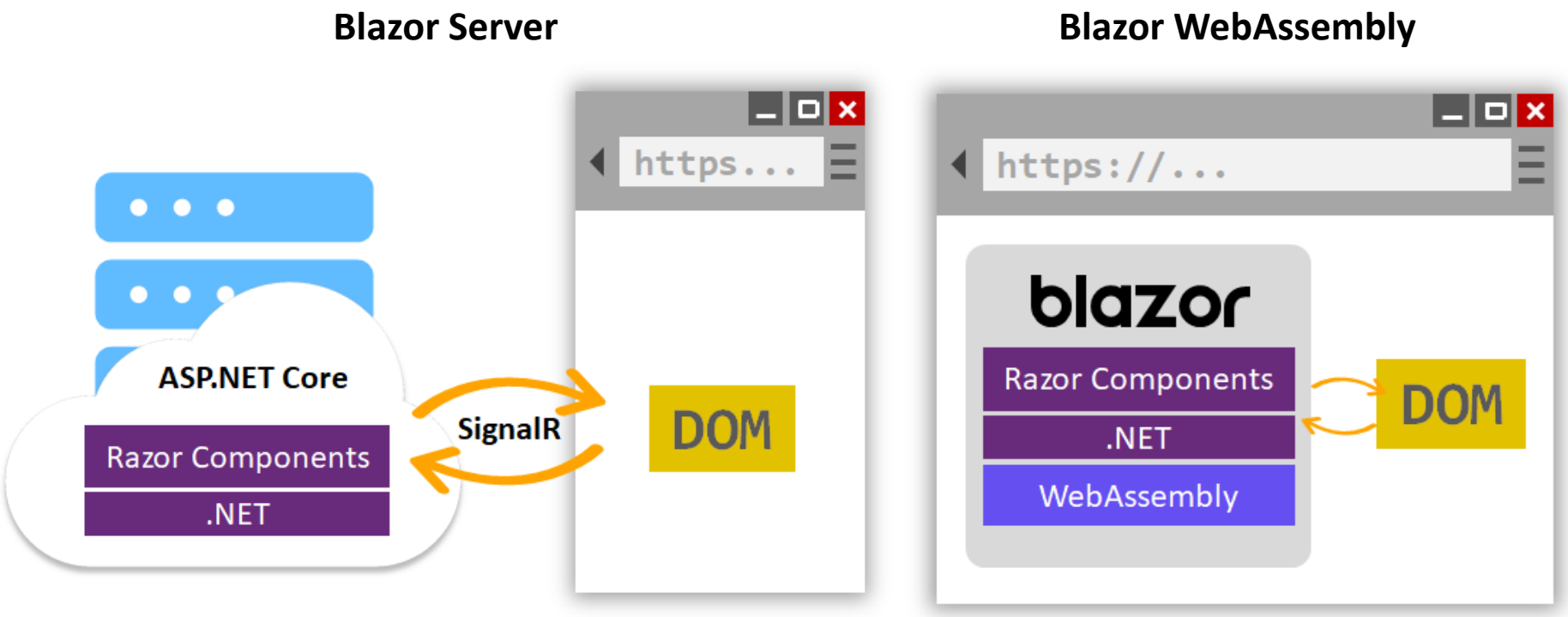
- Unterstützung C# 12
- Auto-Rendering-Modus
- Static-Server-Rendering
- Full-Stack WebUI
- Hot Reload Verbesserungen
- Unterstützung Fluent UI
- Keyed Services
- Sektionen
- QuickGrid
- AOT
- Blazor Identity UI
- HTTP/3

https://learn.microsoft.com/en-us/aspnet/core/release-notes/aspnetcore-8.0?view=aspnetcore-8.0#blazor

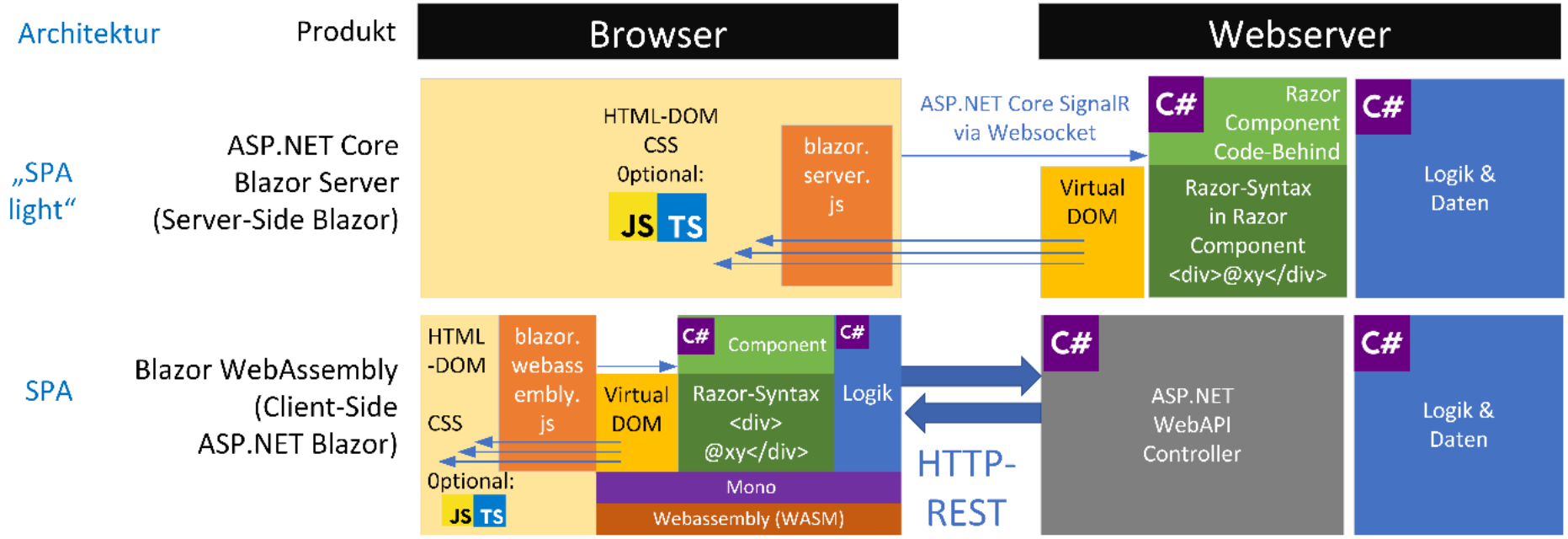
Versionshistorie Blazor



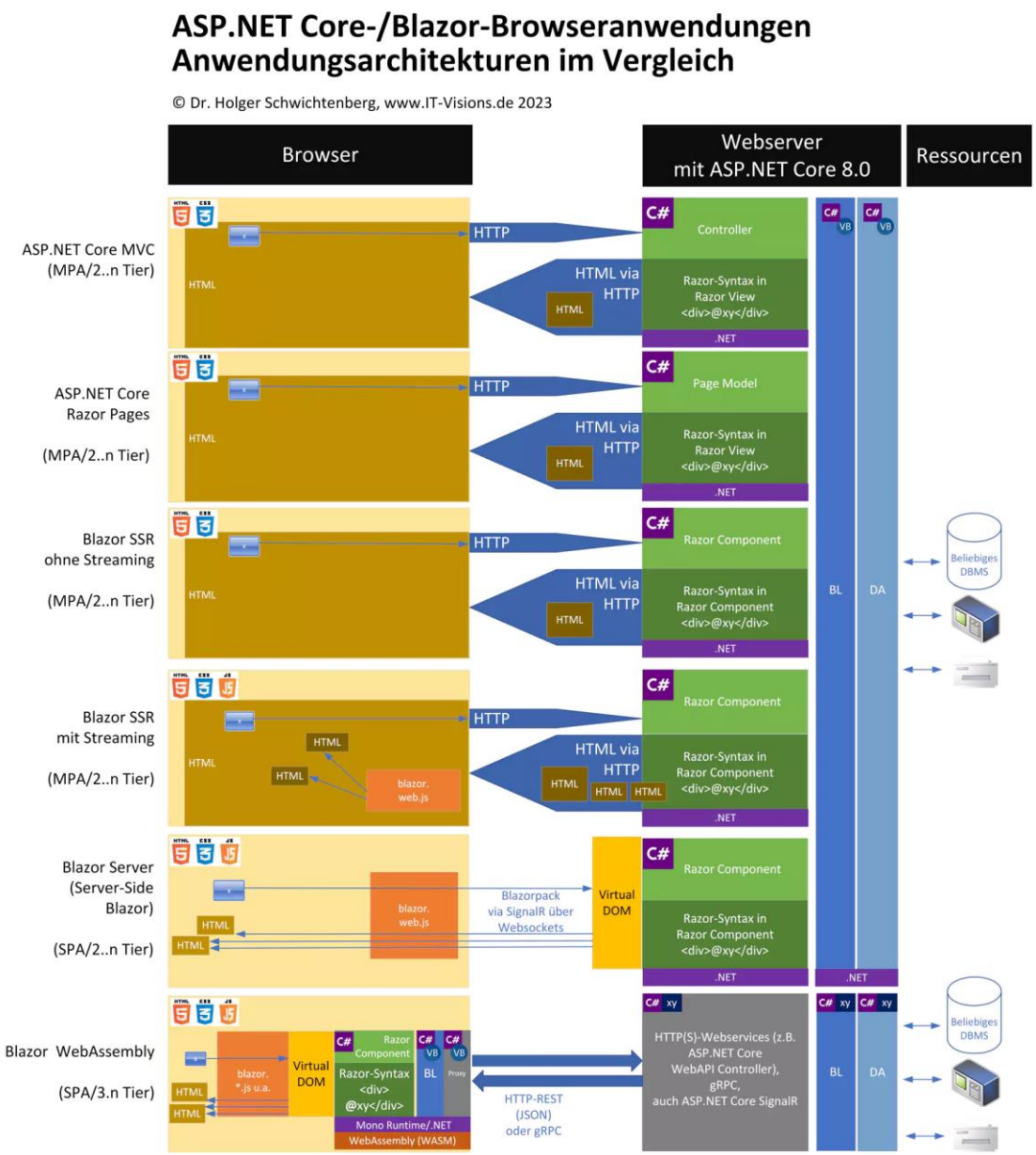
Blazor Server vs. Blazor WebAssembly



ASP.NET Core Blazor: Architekturalternativen



Architektur .NET 8



Unterschiede Blazor Server & Blazor WASM

Merkmal	Blazor WebAssembly	Blazor Server / SSR
Entwickeln mit C#	✓	✓
Kleine Downloadgröße	.NET Runtime & Abhängigkeiten müssen heruntergeladen werden	✓
Hohe Ausführgeschwindigkeit	✓	Latenz
Serverless	✓	Server nötig
Browserunabhängig	Neuere Browser Kein IE	✓
Static File Deployment CDN	✓	Server nötig
Offline-Fähig	✓	Nein

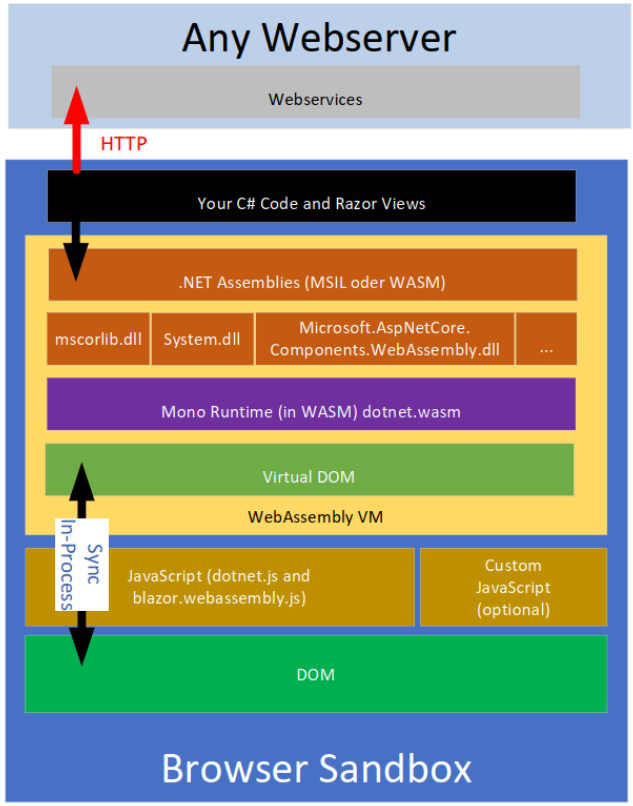
Unterschiede Blazor Server & Blazor WASM Fort.

Merkmal	Blazor WebAssembly	Blazor Server / SSR
Kapselung der Geschäftslogik nötig	✓	Nein aber möglich
Restriktionen durch Browser-Sandbox	✓	Zugriff auf .NET APIs
Netzwerkprotokolle	Http, Https, gRPC	alle
Gute Skalierbarkeit	✓	
Multi-Threading	Nur UI-Thread	✓
Schutz des eigenen Codes	In Client einsehbar	✓

Browser Sandbox

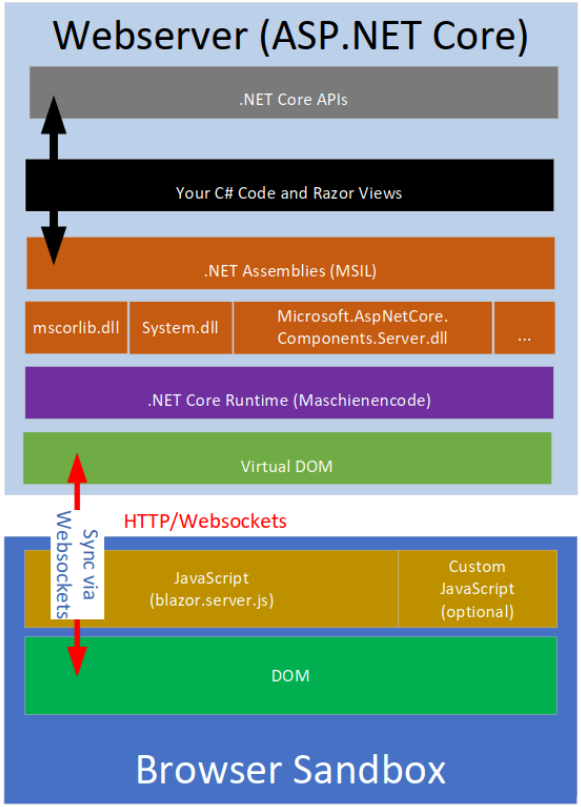
Blazor WebAssembly

Author: Dr. Holger Schwichtenberg, www.IT-Visions.de



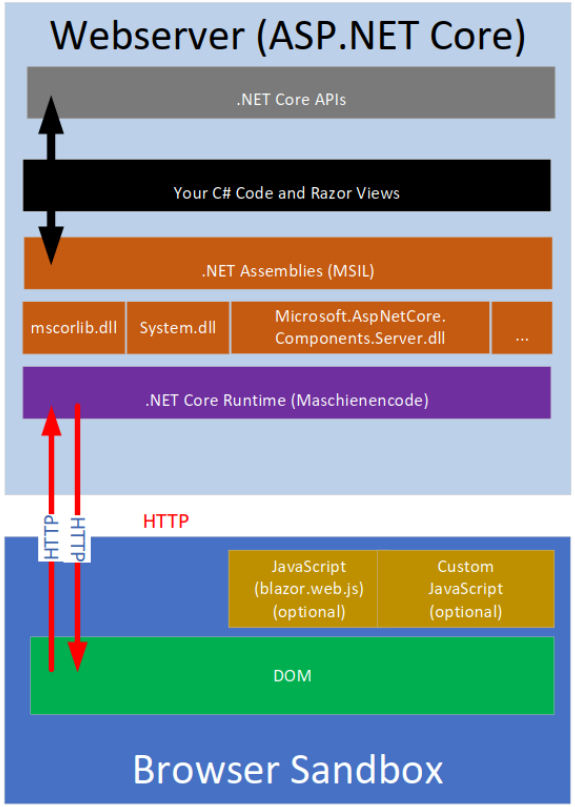
Blazor Server

Author: Dr. Holger Schwichtenberg, www.IT-Visions.de



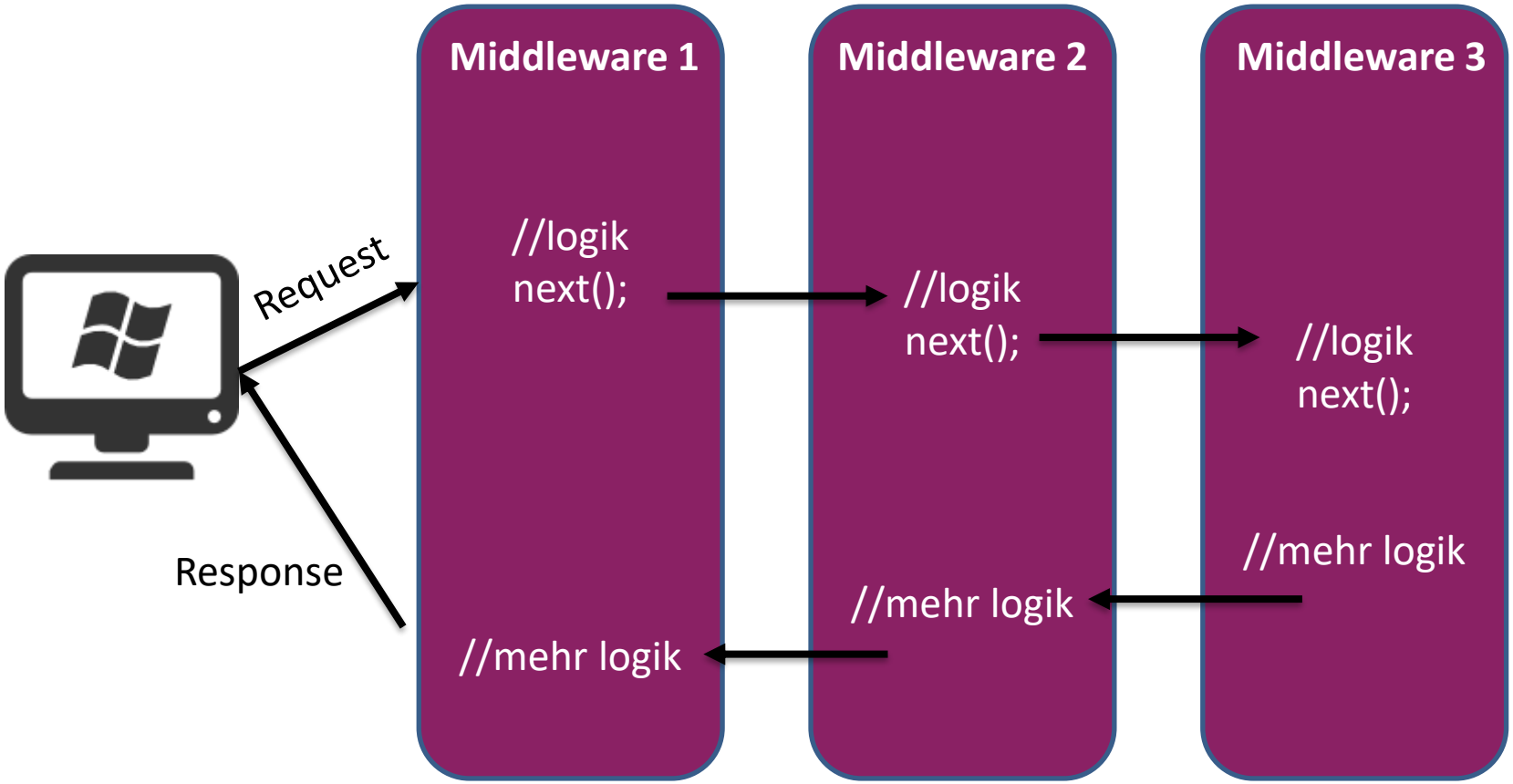
Blazor SSR

Author: Dr. Holger Schwichtenberg, www.IT-Visions.de



ASP.NET Core Request Pipeline & Middleware

- Querschnittsfunktion ähnlich HttpModulen & MessageHandler
- Erste Anlaufstelle bei der Bearbeitung der Anfragen.



Middleware

- Software, die zu einer Anwendungspipeline zusammengesetzt wird (Delegate)
- Anwendung durch Extension-Methoden des I**ApplicationBuilder** Objekts
- Die **WebApplicationBuilder** stellt den Mechanismus bereit die Request-Pipeline zu konfigurieren
- Built-In-Middleware
  - Mvc, StaticFiles, CORS, Routing, DeveloperExceptionPage etc.
- Custom Middleware möglich
- Reihenfolge des Aufrufs relevant

Start bzw. Program

- **Program**
  - Verantwortlich für die Konfigurierung/Ausführen der Anwendung
  - Einstiegspunkt der Webapplikation
- **WebApplicationBuilder** verantwortlich für Configuration und Service Registration
- **IServiceCollection** wird genutzt um Services dem Container hinzuzufügen und zu konfigurieren
- **Middlewares** um Pipeline zu konfigurieren
- **App.razor** ist Startkomponente der Anwendung, die auf die **Routes.razor** verweist, die widerum die **MainLayout.razor** lädt
- **<script src="_framework/blazor.web.js"></script>** in App.razorstellt SignalR-Verbindung zum WebServer her

Blazor WebAssembly

Blazor WebAssembly-Apps (WASM) werden auf Clientseite im Browser in einer auf WebAssembly basierenden **Mono** .NET-Runtime (WebAssembly-Bytecode) in die Browser-Sandbox geladen und ausgeführt. WASM hat keinen direkten Zugriff auf das DOM des Browsers. Stattdessen wird Mithilfe eines Virtual DOM per Javascript das Virtual DOM mit dem echten DOM synchronisiert

- Einstiegspunkt ist die „**index.html**“ (Hosting Page) im www-Ordner JavaScript-Datei „**blazor.webassembly.js**“ lädt Abhängigkeiten (IL-code bzw. dlls/.wasm), die .NET-Runtime (**dotnet.wasm**) herunter und initialisiert die Runtime
- Program.cs verweist auf Startkomponente **App** ("#app"). Sie beinhaltet die Router-Funktionalität.
- App-Komponente definiert Standard-Layout ("MainLayout"), die die diversen Komponenten implementiert und in den @Body Platzhalter lädt

Die WebAssembly-Anwendung wird mithilfe eines in WebAssembly implementierten Interpreters (JITerpreter) für die .NET-Zwischensprache im Browser zur Laufzeit ausgeführt. Die Ahead-of-Time Kompilierung (AOT) ermöglicht optional die direkte Kompilierung in WebAssembly zur nativen Ausführung durch den Browser.

Hands On

Projekt „Qotd"

Razor-Komponenten

Razor

Was ist eine Razor-Komponente?

- Zentrale UI-Elemente einer Webanwendung
- Templatesprache HTML, CSS und einer speziellen CodeSyntax bzw. View-Engine namens RAZOR @ sowie C#-Code
- Dateien haben die Endung „razor“ und erben implizit von ComponentBase
- Name muss mit Großbuchstaben starten
- Zwei Möglichkeiten Code zu erstellen
  - Mixed Ansatz mit @code
  - Partielle C# Klasse
- Können mit der @page Direktive direkt aufgerufen werden
- CSS und JavaScript-Isolation möglich
- Einbinden als Tag => <NameComponente />

Komponenten-Beispiel

```
@page "/counter"
<p>Current count: @currentCount</p>
<button @onclick="IncrementCount">Click me</button>

@code{
   int currentCount = 0;

    void IncrementCount()
    {
       currentCount++;
    }
}
```

Adresse

Variable

Click-Ereignis

Mixed Ansatz
Counter.razor

```
public partial class Counter
{
   …
}
```

C# Klasse
Counter.razor.cs

Razor-Syntax

Syntax	Beschreibung

Code Block	`@code {`
	`  int x = 666;`
	`  string y = "Dies ist ein Text";`
	`}`

Funktionen	`@functions {`
	`  string GetInfo(string message) { return … }`
	`}`
	
	`@GetInfo("Hallo")`

Ausdruck (Nicht HTML encoded)	`@((MarkupString) message)`

Kombinieren Text und markup	`@foreach(var item in items)`
	`{`
	`  <span>@item.Eigenschaft</span>`
	`}`

Code and Klartext mischen	`@if (foo)`
	`{`
	`  <text>Klartext</text>`
	`}`

Razor-Syntax II

Syntax	Beschreibung

Code and Klartext mischen (alternativ)	`@if (foo)`
	`{`
	`  @:Plain Text is @bar`
	`}`

Escape @	`Hi philha@@example.com`
Kommentar	`@* Dies ist ein Kommentar *@`
Generische Methode aufrufen	`@(MyClass.MyMethod<AType>())`
Datenbindung an Stuerelemente	`<input type="text" @bind="Nachname" />`
Ereignisbindung	`<button @onclick="Methode">Klick hier</button>`
Ereignisbindung mit Parameter	`<button @onclick="() => Methode(param)">Klick hier</button>`

Hands On

Razor Syntax

Parameter

Parameter, Arbitary, Cascading

Parameter

- Parameter können von Eltern-Komponente übergeben werden via Attribute
- Parameter **muss** *Property* und *public* sein
- Zusätzlich als Pflichtparameter [**EditorRequired**] dem Child-Parameter hinzugefügt werden, damit Entwickler den Parameter setzt

Index.razor

```
@page  "/"

<Home Title="@title"></Home>

@code {
    private string title = "Homepage";
}
```

Home.razor

```
<h1>@Title</h1>

@code {

    [Parameter, EditorRequired]
    public string Title { get; set; }
}
```

Arbitrary Parameter

- Zusätzliche Parameter können gesammelt übergeben werden

```
@page  "/"

<Home Title="@title" alt="Bild des Autors" width="150"></Home>

@code {
     private string title = "Homepage";
}
```

```
<h1>@Title</h1>
<img src="…" @attributes="AdditionalAttributes" />

@code {
    [Parameter] public string Title { get; set; }

    [Parameter(CaptureUnmatchedValues = true)]
    public Dictionary<string, object> AdditionalAttributes { get; set; }
}
```

Cascading Parameters

- Parameter werden von Eltern- zu allen Kindkomponenten übergeben

```
@page  "/"
<CascadingValue Value="@_color">
    <Home></Home>
</CascadingValue>

@code {
      private string _color = "red";
}
```

```
<h1 style=„color: @Color">@Title</h1>

@code {
  [Parameter] public string Title { get; set; }

  [CascadingParameter]  //optional mit Name
  public string Color { get; set; } // Setter vorhanden & public  & Type muss gleich sein
}
```

Render Fragment Parameters

- Inhalt an Kind-Komponenten senden

```
@page  "/"
    <Home>
        Dieser Inhalt wird in der Kind-Komponente angezeigt
        <AlternativContent>
            Alternativer Inhalt wird angezeigt
        </AlternativContent>
    </Home>
```

```
<div>@ChildContent   @AlternativContent</div>

@code {
    [Parameter]
    public RenderFragment ChildContent { get; set; } //ChildContent od NamedTag in Eltern-Komp

    [Parameter]
    public RenderFragment AlternativContent { get; set; } //alt. Named Tag in Eltern-K
}
```

Hands On

Parameter

ASP.NET Blazor Models

Überblick

- Was ist ein Model?
- Erstellung von Models
- Data Annotations
- Validation

Was ist ein Model?

Eine Klasse

Wer nutzt das Model?

- Benutzer
  - Mit welchen Daten wird interagiert?
- ASP.NET Blazor
  - Eingabesteuerelemente
  - Validation

Model

Author
- Id : Guid
- Name : string
- Description : string
- BirthDate: DateOnly

```
public class Author
{
 public Guid Id { get; set; }

 public string Name { get; set; }

 public string Description {get;set;}

 public DateOnly? BirthDate {get;set;}
}
```

Daten- und Event-Binding

Datenbindung

- Anzeigen und die Verarbeitung von Daten bzw. Ereignissen als zentrale Funktionalität
- One-Way-Datenbindung
- Two-Way-Datenbindung
- Ereignisbehandlung

One / Two-Way-Bindung

- One-Way ist Bindung einer C#-Variablen an ein DOM-Element. Bei Änderung des Variablenwerts spiegelt es sich im UI wieder
- Two-Way ist bidirektional d.h. wenn Wert in UI-Element ändert, aktualisiert sich die C#-Variable und umgekehrt. Standardmäßig erfolgt die Synchronierung im **onchange**-Event.

```
<div>@title</div> // One-Way
<input @bind="_name" /> // Two-Way mit C# Feld
<input @bind="Name" /> // Two-Way mit C# Property Kurzversion
<input @bind-value="Name" @bind-value:event="onchange" /> //Two-Way mit C# Langversion

// Two-Way XXL - Langversion
<input value="@Name" @onchange="@((ChangeEventArgs e) => Name = e.Value.ToString()) "

// Two-Way-Binding mit oninput-Event
<input @bind-value="Name" @bind-value:event="oninput" />

@code {
   private string title = "Mein Titel";
   private string _name;
   private string Name { get; set; }
}
```

Ereignisbehandlung

Die Behandlung von Html-DOM-Ereignissen erfolgt über das Muster **@on[Event]**

```
<button @onclick="ErsteMethode">Klick mich</button>
<button @onclick="() => ZweiteMethode(item)">Klick mich 2</button> //mit Parameter als Lambda
<button @onclick="() => { item++ }">Klick mich 3</button> //Code direkt Abkürzung

<input @onkeypress="KeyPress" /> //Standardereignis mit Argument
<input @onkeypress="(e) => KeyPress(e)" /> //Explizit mit Argument zwingend bei zusätzl. Parameter

<a @onclick="OnClickHandler" @onclick:preventDefault>Klick</a> //Standardereignis unterbinden
<p @onclick="OnClickHandler" @onclick:stopPropagation></p> //Ereignisweitergabe unterbinden


@code {
    private void ErsteMethode() {}
    private void ZweiteMethode(string item) {}
    public void KeyPress(KeyboardEventArgs args) {}
}
```

Ereignisweitergabe

Komponenten können Ereignisse auslösen und z.B. Elternkomponenten informieren

```
<button @onclick="() => EineMethode(item)">Klick mich</button>

@code {
    [Parameter]
    public EventCallback<int> OnEventCallback { get; set; }  //Rückgabe Typ integer

    public void EineMethode(int item)
    {
        OnEventCallback.InvokeAsync(item);  //Auslösen des Ereignises
    }
}
```

```
<KindComponente OnEventCallback="MeineMethode"></KindComponente>

@code {
    public void MeineMethode(int item)  {}
}
```

Hands On

Data-Binding

Lebenszyklus

Lebenszyklus

Methode	Beschreibung
OnInitialized & OnInitializedAsync	Diese Methode wird ausgelöst, sobald die Komponente initialisiert wird und alle Parameter von der übergeordneten Komponente erhält. *Achtung*: Blazor Server ruft die Methode bei ServerPrerendered die Startkomponenten zweimal auf.
OnParametersSet & OnParametersSetAsync	Diese Methode wird ausgelöst, nachdem die Komponente ihre Parameter erhalten hat und ihre Werte ihren jeweiligen Eigenschaften zugewiesen wurden. Diese Methode wird *wiederholt* ausgelöst, sobald der Wert des Parameters aktualisiert wird
OnAfterRender & OnAfterRenderAsync	Sobald die Komponente das Rendern beendet hat, wird diese Methode aufgerufen. Es wird in dieser Phase verwendet, um zusätzliche Initialisierungsschritte unter Verwendung des gerenderten Inhalts durchzuführen. Bei dieser Methode ist der firstRender-Parameter nur bei der *ersten Ausführung* der Methode auf "true" gesetzt. Jedes andere Mal ist der Wert falsch

Lebenszyklus II

Methode	Beschreibung
ShouldRender	Diese Methode kann verwendet werden, um die Aktualisierung der Benutzeroberfläche zu unterdrücken. Diese Methode wird jedes Mal aufgerufen, wenn die Komponente gerendert wird. Unabhängig davon, ob wir diese Methode so einstellen, dass sie "false" zurückgibt, wird die Komponente immer anfänglich gerendert
SetParametersAsync	Diese Methode wird zuerst ausgelöst, sobald die Komponente erstellt wird. Alle an die Komponente gesendeten Parameter werden im ParameterView-Objekt gespeichert. Falls einige asynchrone Aktionen ausgeführt werden sollen, bevor ein Parameterwert zugewiesen wird
StateHasChanged	Diese Methode wird immer dann aufgerufen, wenn Blazor benachrichtigt werden soll, dass sich etwas in der Komponente geändert hat, und diese Komponente neu gerendert werden soll.

Hands On

Lebenszyklus

Routing & Navigation

Routenvorlage App-Razor

- **App.razor** definiert die Router-Komponente
- Assembly wird gescannt um Routeninformationen für die Komponenten der Anwendung zu erfassen
- **Routes**-Komponente empfängt Route-Data-Objekt mit Routenparameter
- Rendert die angegebene Komponente mit deren Layout
- **FocusOnNavigate** setzt den UI-Fokus auf eine Element
- Falls zur angeforderten Route keine Komponente gefunden wird **<NotFound>** dargestellt
- Durch **AdditionalAssemblies** können zusätzliche Assemblies bei der Suche nach Komponenten für das Routing eingebunden werden
- **BaseUri** wird in App.razor gesetzt

@Page Direktive

- @page definiert eine Routing Regel (oder mehrere) damit Blazor die richtige Komponente aufruft
- Case-Insensitiv
- Müssen eindeutig sein für Komponenten
- Ohne @page Direktive kann Komponente nicht über URL aufgerufen werden, sondern kann nur in andere Komponenten eingebettet sein
- Können Parameter beinhalten mit optionalen Parameter-Typ-Einschränkung

```
@page "/"
@page "/home"
@page "/kunde/autrag"

@page "/autor/details/{name}"          //Routen-Parameter => Parameter-Typ string ist Standard
@page "/autor/auftrag/{name?}"         //Routen-Parameter => Parameter-Typ optional
@page "/autor/details/{AutorId:guid}" //Routen-Parameter mit Parameter-Typ-Einschränkung
```

Routing Einschränkungen

Einschränkung	Beschreibung	Beispiel
bool	Boolean (true, false)	{active:bool}
datetime	DateTime	{dob:datetime}
decimal	Matches a decimal value.	{price:decimal}
double	Double	{weight:double}
float	Float	{weight:float}
guid	GUID value.	{id:guid}
int	32-bit integer.	{id:int}
long	Long	{ticks:long}

Navigation

- Per Html-Hyperlink kann zu einer Komponente navigiert werden
- Zuätzlich ist eine Navigation auch im Code möglich mit dem NavigationsManager

```
//Html Variante
<a href="/autors">Liste der Autoren</a>
```

```
//Code Variante
public partial class Home
{
    [Inject]
    public NavigationManager NavManager { get; set; }

    public void Methode()
    {   NavManager.NavigateTo("/kunde");   }
}
```

Dependency Injection

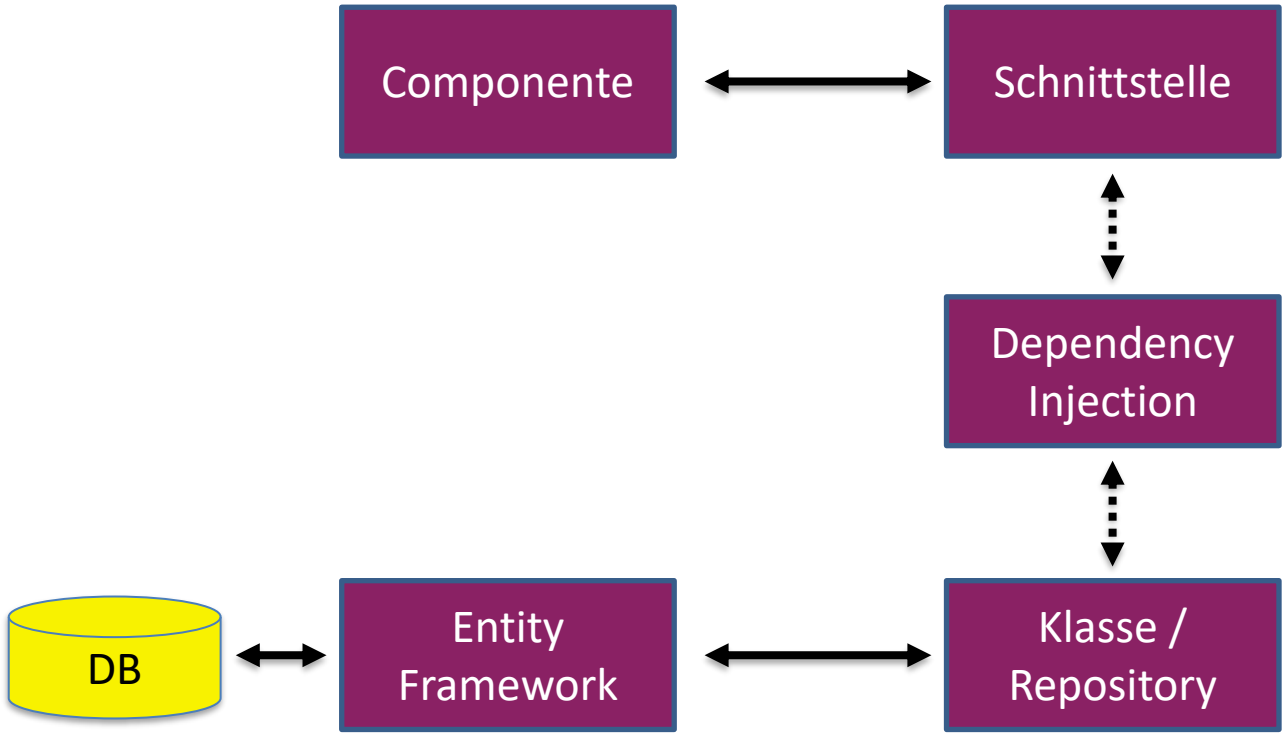
IoC

IoC / Dependency Injection

- ***I**nversion **o**f **C**ontrol* delegates the function of selecting a concrete implementation type for a class‘s dependencies to an external component

- *Dependency Injection* is a specialization of the IoC Pattern. The Dependency Injection pattern uses an object – the container – to initialize objects and provide the required dependencies to the object

Zugriff mit Dependeny Injection



Dependency Injection

- Ziel ist die Flexibilität und Testbarkeit zu steigern
- Erleichtert das Austauschen & Testen von Abhängigkeiten
- Built-In-ASP.NET Core
- Populäre DI-Frameworks
  - AutoFac

DI Scopes

- Builder.Services in Program verantwortlich für die Bereitstellung der Services

Scope	Bedeutung	Erläuterung
Transient	**vorübergehend**	Bei jeder Verwendung (Anforderung einer Klasse beim Dependency Injection-Container) erhält man eine eigene Instanz
Scoped	**bereichsbezogen**	**Blazor Server** Der Bereich (Scope) bezieht sich bei Blazor Server auf eine Websocketsverbindung, also die Verbindung eines Browserfensters mit der Blazor Server-Anwendung ("Circuit"). Dies entspricht in der Regel einer Benutzersitzung. **Blazor WebAssembly** Dies entspricht in der Regel einer Benutzersitzung (Browser/Tab). Innerhalb einer Benutzersitzung erhält man bei einem Scoped Service bei jeder Verwendung dieselbe Instanz.

DI Scopes

Scope	Bedeutung	Erläuterung
Singleton	**übergreifend**	**Blazor Server** Es gibt nur eine einzige globale Instanz für alle Verwendungen übergreifend innerhalb eines Prozesses. Singleton bedeutet bei Blazor Server, dass alle Benutzer die dieselbe Dienst-Instanz erhalten, da es ja nur einen Prozess für alle Benutzer gibt. **Blazor WebAssembly** Bei Blazor WebAssembly bedeutet Singleton, dass jeder Benutzer in jeder Instanz der Webanwendung (d.h. pro Browser-Tab) eine eigene Instanz des Services erhält. Da es bei Blazor WebAssembly nur eine Benutzersitzung in einem Prozess gibt, verhält sich die Lebensdauer "Scoped" bei Blazor WebAssembly genau wie "Singleton"

DI in Komponenten

- In Razor-Template mit **@inject** oder Property-Injektion **[Inject]**
- Im Code-behind mit Property-Injektion [Inject]
- In anderen Klassen mit Konstruktorinjektion
- Custom Dienste müssen in der ServiceCollection registriert werden mit einem von drei Lebensdauerkonzepten (Scopes)

```
@inject NavigationManager NavManager //Razor Template Variante


public partial class Home
{
    [Inject]  //Code Behind Variante
    public NavigationManager NavManager { get; set; }
}
```

```
//Service Collection in Program.cs
builder.Services.AddScoped<ISchnittstelle, Schnittstelleimplementierung>();
```

Dependency Constructor-Injection

```
public class AuthorService
{
  private readonly IAuthorRepository _authorRepository;

  protected AuthorService(IAuthorRepository authorRepository)
  {
    _authorRepository = authorRepository;
  }
}
```

Interface, *keine* konkrete Implementierung

Konstruktor Injection

Hands On

DI

WebServices/API via Http

Http-Client

- System.Net.Http.HttpClient in Projektvorlage vorhanden
- In Razor via [Inject] bzw. Konstruktorinjektion

```
//Service Collection in Program.cs
builder.Services.AddScoped(sp => new HttpClient { BaseAddress = new
Uri(builder.HostEnvironment.BaseAddress) });
```

```
//Razor Komponente
@inject HttpClient Client

//Codebehind bei Razor Komponente
[Inject] HttpClient Client {get; set;}

//Konstruktor Injektion bei regulären Klassen
public class Service {
   private HttpClient _client;
   Service(HttpClient client)
   { _client = client; }
}
```

HttpFactory & TypedClient

- Möglichkeit den Http-Client durch IHttpClientFactory zu erstellen (Empfehlung)

```
builder.Services.AddHttpClient("MeineAPI", opt =>
{
    opt.BaseAddress = new Uri("https://localhost:5011/api/");
});

//---------- Komponente ----------------------------------------------------------
[Inject] private IHttpClientFactory HttpClientFactory { get; set; }

protected override async Task OnInitializedAsync()
{
    var client = HttpClientFactory.CreateClient("MeineAPI");
}
```

Typed HttpClient

- Weitere Möglichkeit den Http-Client als TypedClient zu nutzen

```
builder.Services.AddHttpClient<IMeinService,MeinService>(opt =>
{
    opt.BaseAddress = new Uri("https://localhost:5011/api/");
});

//---------- MeinService.cs ----------------------------------------------------
public class MeinService
{
    private readonly HttpClient _client;

    MeinService(HttpClient client)
    {
        _client = client;
    }
}
```

Http-Client Methoden (Auszug)

Methode	Beschreibung
GetAsync(string requestUri)	Sendet einen GET-Request an die Uri
GetStringAsync(string requestUri)	Sendet eine GET-Request an den angegebenen Uri und gibt den Antworttext als Zeichenfolge zurück
PatchAsync(string requestUri, HttpContent content)	Sendet einen PATCH-Request an die Uri mit Content
PostAsync(string requestUri, HttpContent content)	Sendet einen POST-Request an die Uri mit Content
PutAsync(string requestUri, HttpContent content)	Sendet einen POST-Request an die Uri mit Content

Erweiterungsmethode	Beschreibung
GetFromJsonAsync<T>(string requestUri)	Sendet einen GET-Request an die Uri und deserialisiert die Antwort als JSON-Objekt
PostAsJsonAsync<T>(string requestUri, T value)	Sendet einen POST-Request an die Uri mit serialisiertem Content als JSON im Body
PutAsJsonAsync(string requestUri, T value)	Sendet einen PUT-Request an die Uri mit serialisiertem Content als JSON im Body

HttpClient-Beispiel

```
public partial class Home
{
    [Inject ]private HttpClient _client {get; set;}
    private readonly JsonSerializerOptions _options;
    public Qotd Qotd {get; set;}

    protected override async Task OnInitializedAsync()
    {
        _options = new JsonSerializerOptions { PropertyNameCaseInsensitive = true };
        var response = await _client.GetAsync("https://localhost:1234/api/qotd");
        var content = await response.Content.ReadAsStringAsync();
        if (!response.IsSuccessStatusCode) {
            throw new ApplicationException(content);
        }
       Qotd = JsonSerializer.Deserialize<Qotd>(content, _options);

       //Alternative Kurzversion
       Qotd = await _client.GetFromJsonAsync<Qotd>("https://localhost:1234/api/qotd");
    }
}
```

Formulare

Formulare

- Komponente EditForm als zentrales Webformular
- Eröffnet einen EditContext, den alle untergeordneten Eingabe- und Validierungskomponenten zum Datenaustausch verwenden
- Built-In Eingabe-Komponenten
- Validation via DataAnnotations des Models
  - Built-In-Validation
  - Custom ValidationsAttribute
- Handler für Formularübermittlung

EditForm

- Bindung an Model **oder** EditContext

```
<EditForm Model=@author>….</EditForm>

@code {   private Author author {get; set;} = new();}
```

```
<EditForm EditContext=@editContext>….</EditForm>

@code {
   private Author author {get; set;} = new() { Name = "Simpson"};
   private EditContext? editContext {get; set;};

   protected override void OnInitialized()
   {
        editContext = new EditContext(author);
   }
}
```

EditForm-Formularübermittlungsereignisse

- EditForm besitzt drei Ereignisse
- Nicht alle drei Ereignisse dürfen gleichzeitig Handler haben (OnValidSubmit/OnInvalidSubmit oder OnSubmit)

Ereignis	Beschreibung
OnSubmit	Wird ausgelöst, wenn Formular übermittelt wird
OnValidSubmit	Wird ausgelöst, wenn Formular übermittelt wird und alle Validatoren erfolgreich geprüft haben
OnInvalidSubmit	Wird ausgelöst, wenn Formular übermittelt wird und mindestens ein Validator einen Fehler meldet

Eingabe-Komponenten

Komponente	HTML-Repräsentation
<EditForm>	<form>
<InputText>	<input type="text">
<InputTextArea>	<textarea>
<InputSelect>	<select>
<InputNumber>	<input type="number">
<InputCheckbox>	<input type="checkbox">
<InputRadio><InputRadioGroup>	<input type="radio">
<InputDate>	<input type="date"
<InputFile>	<input type="file">
<DataAnnotationsValidator>	-
<ValidationSummary>	<ul class="validation-errors"> <li class="validation-message">…
<ValidationMessage>	…

Datenbindung & Validierung

- Eingabekomponenten werden via Two-Way-Datenbindung eingebunden
- Validation via ValidationMessage-Komponente und DataAnnotationValidator

```
<EditForm Model="@autor" OnValidSubmit="@HandleValidSubmit">

  <DataAnnotationsValidator />
  <ValidationSummary />

  <div>
    <InputText id="Name" @bind-Value="@autor.Name"></InputText>
    <ValidationMessage For="@(() => autor.Name)" />
  </div>
…
</EditForm>

@code {}
```

Validierung

Data Annotations

- Steuern die Anzeige und die Validation von Modeleigenschaften (Properties)
- System.ComponentModel.DataAnnotations
- Built-In-Data-Annotations für Anzeige

Anzeige Attribute	Beschreibung
DataType	Legt den Datentyp fest

Model - Validierungsmöglichkeiten

- 3 Möglichkeiten der Model-Validierung
  - Built-In-Attribute
  - Benutzerdefinierte Attribute
  - Selbstvalidierendes Model

Built-In-Validierungsattribute

Validation Attribute	Beschreibung
Compare	Prüft, ob zwei Eigenschaften denselben Wert haben. (z.B. Passwort)
CreditCard	Prüft, ob der zu validierende Wert eine Kreditkartennummer ist
EmailAddress	Prüft auf Email-Format
MaxLength	Prüft auf eine maximale Länge
MinLength	Prüft auf eine minimale Länge
Range	Prüft, ob sich der zu validierende Wert in einem bestimmten Wertebereich befindet
RegularExpression	Validiert die Eigenschaft mit regulären Ausdrücken
Required	Markiert die Eigenschaft als Pflichtfeld

Model mit Validierungsattribute

Author
- Id : Guid
- Name : string
- Description : string
- Birthdate: DateOnly

```
public class Author
{
 public Guid Id { get; set; }

 [Required(ErrorMessage="Bitte …")]
 [StringLength(50)]
 public string Name { get; set; }

 [StringLength(50)]
 public string Description {get;set;}

 public DateOnly? BirthDate {get;set;}
}
```

Benutzerdefinierte Validierungsattribute

- Ableitung von Basisklasse **ValidationAttribute** und überschreiben von "**`IsValid`**" Methode

```
public class NoAdminAttribute : ValidationAttribute
{
  protected override ValidationResult IsValid(object value,
                          ValidationContext validationContext)
  {
   string name = (string) value;
   if(name.ToLower() != "admin" && name.ToLower() != "administrator")
     {
       return ValidationResult.Success;
     }
     var error = "Der Wert darf nicht Admin oder Administrator sein";
     return new ValidationResult(error);
  }
}
```

Benutzerdefinierte Validierungsattribute

Autor
- Id : Guid
- Name : string
- Description : string
- Birthdate: DateTime

```
public class Author
{
 public Guid Id { get; set; }

 [Required(ErrorMessage="Bitte …")]
 [StringLength(50)]
 [NoAdmin]
 public string Name { get; set; }

 [StringLength(50)]
 public string Description {get;set;}

 public DateOnly? BirthDate {get;set;}
}
```

Selbstvalidierendes Model

- Vorteil von vereinten Validierungen in einer Klasse
- Nachteil von nicht mehrfach verwendbaren Validierungen
- Schnittstelle **`IValidateObject`** muss implementiert werden

Selbstvalidierendes Model

```
public class Author : IValidateObject
{
     …
     [StringLength(50)]
     //[NoAdmin]
     public string Name { get; set; }
     …
}

public IEnumerable<ValidationResult> Validate(
                                          ValidationContext validationContext)
{
    List<ValidationResult> errors = new List<ValidationResult>();
    if (Name.ToLower()=="admin" || Name.ToLower()== "root")
    {
        errors.Add(new ValidationResult("Der Nachname darf nicht Admin
          oder root sein"));
    }
    return errors;
}
```

Hands On

Formular - Validierung

Interoperabilität JavaScript

.NET ⇔ Javascript

Interoperabilität

- Blazor kann nicht direkt auf das DOM zugreifen
- Zentrale Schnittstelle ist die **IJSRuntime**, welche mit [Inject] injiziert wird
- Möglichkeit 3rd-Party-Bibliotheken zu nutzen
- JavaScript-Dateien werden im **wwwroot** abgelegt und müssen in der **index.html** verwiesen werden
- Alternativ kann man die JS-Isolation als Modul in der Komponente nutzen {**KomponentenName}.razor.js**
- Komponenteninstanz kann JS übergeben werden
- JavaScript kann statische wie auf Instanzmethoden aufrufen

.NET nach JavaScript

- **InvokeVoidAsync** => keine Rückgabe von JS
- **InvokeAsync<T>** => mit Rückgabe von JS

```
//JS in Datei example.js in wwwroot/js
function showAlert(message) {  alert(message); }
function confirm(message) { return window.confirm(message); }

//index.html
<script src="js/example.js"></script>

//Komponente
[Inject] IJSRuntime JsRuntime { get; set; }

public async Task EineMethode()
{
    //Syntax => InvokeVoidAsync("Funktionsname","Parameter")
    await JsRuntime.InvokeVoidAsync("showAlert", "JS function called from .NET");
    var bool = await JsRuntime.InvokeAsync<bool>("confirm", "Wollen Sie wirklich …? ");
}
```

.NET nach JavaScript als Modul

- Globaler window namespace bleibt unberührt
- Import in index.html ist nicht nötig. Laden bei Bedarf
- Variablen / Funktionen müssen mit **export** markiert werden
- Auch als Komponenten-Datei möglich (Komponente.razor.js)

```
//JS in Datei example.js in wwwroot/js
export function showAlert(message) {  alert(message); }
export function confirm(message) { return window.confirm(message); }

//Komponente
[Inject] IJSRuntime JsRnt { get; set; }
private IJSObjectReference _jsModule;

public async Task EineMethode()
{
    _jsModule = await JsRnt.InvokeAsync<IJSObjectReference>("import", "./js/example.js");
    _jsModule.InvokeVoidAsync("showAlert", "Js Function aufgerufen von .NET");
}
```

JavaScript nach .NET

- JS kann statische oder .NET Instanzmethoden aufrufen
- **[JSInvokable]** Attribut markiert, dass Methode durch JS aufgerufen werden kann. Methode muss public sein
- **DotNet**-Objekt in JS um statische C# Methoden aufzurufen

```
//Komponente C#
[JSInvokable] //Methodenname muss unique sein oder mit Name [JSInvokable("cqw")]
public static string CalculateQuadratWurzel(int number) =>
    $"Die Quadratwurzel von {number} beträgt {Math.Sqrt(number)} ";

//JavaScript
export function calculateQuadratwurzel() {
    //DotNet.invokeMethodAsync("Name Assembly", "Methode" , Nutzdaten)
    DotNet.invokeMethodAsync("BlazorWasmExample", "CalculateQuadratWurzel", 10)
      .then(result => {
          var el = document.getElementById("eineid");
          el.innerHTML = result;
      });
}
```

JavaScript zu C# mit Instanz

- DotNetObjectReference kann JS übergeben werden
- invokeMethod bzw. invokeMethodAsync verfügbar

```
//Komponente C#
public async Task SendDotnetInstanceToJs()
{
    var dotNetObjRef = DotNetObjectReference.Create(this);
    await JsRuntime.InvokeVoidAsync("showOnlineStatus", dotNetObjRef);
}

[JSInvokable]
public void SetOnlineStatus(bool isOnline) { … }

//JavaScript
export function showOnlineStatus(dotNetObjRef) {
      //C# Methodenname und Parameter
      dotNetObjRef.invokeMethodAsync("SetOnlineStatus", navigator.OnLine)
}
```

Hands On

JavaScript

Razor Class Library (RCL)

RCL

- Ziel der projektübergreifenden Wiederverwendbarkeit und Kapselung von Komponenten
- Normale .NET Assemblys
- Referenzieren Microsoft.AspNetCore.Components.Web
- Importieren des Namespace in **_Imports.razor** des Blazor-Projekts via @using
- CSS-Dateien aus wwwroot werden mit **<link href=„_content/{Package ID}/{Pfad und Datei-Name}** **/>** in _index.html eingebunden
  - PackageId ist normalerweise der AssemblyName
  - Einbindung in index.html nicht nötig bei CSS- oder JS-Isolation

Beispiel

```
//RCL
//Komponente
namespace MeineLibrary.Components;
public partial class OnlineStatus {…}

//CSS in wwwroot der Komponente
//Datei meinekomponente.css

//Blazor - Anwendung
//Datei _Imports.razor
@using MeineLibrary.Components

//Index.html
<link href="_content/MeineLibrary/meinekomponente.css" rel="stylesheet" />

//z.B. MainLayout.razor
<OnlineStatus />
```

Hands On

RCL

Tracing / Logging

Tracing / Logging

- Built-In-Tracing/Logging durch ILoggerFactory der Microsoft.Extensions.Logging bzw. ILogger<T> mit DI
  - Aktivierung in `Configure`
    - `loggerFactory.AddConsole();`
    - `Feature Dependency Injection`
  - Benutzerdefiniertes Logging möglich durch `3rd Party-LoggingProvider`
- Loggingmöglichkeiten
  - NLog
  - Serilog

Hands On

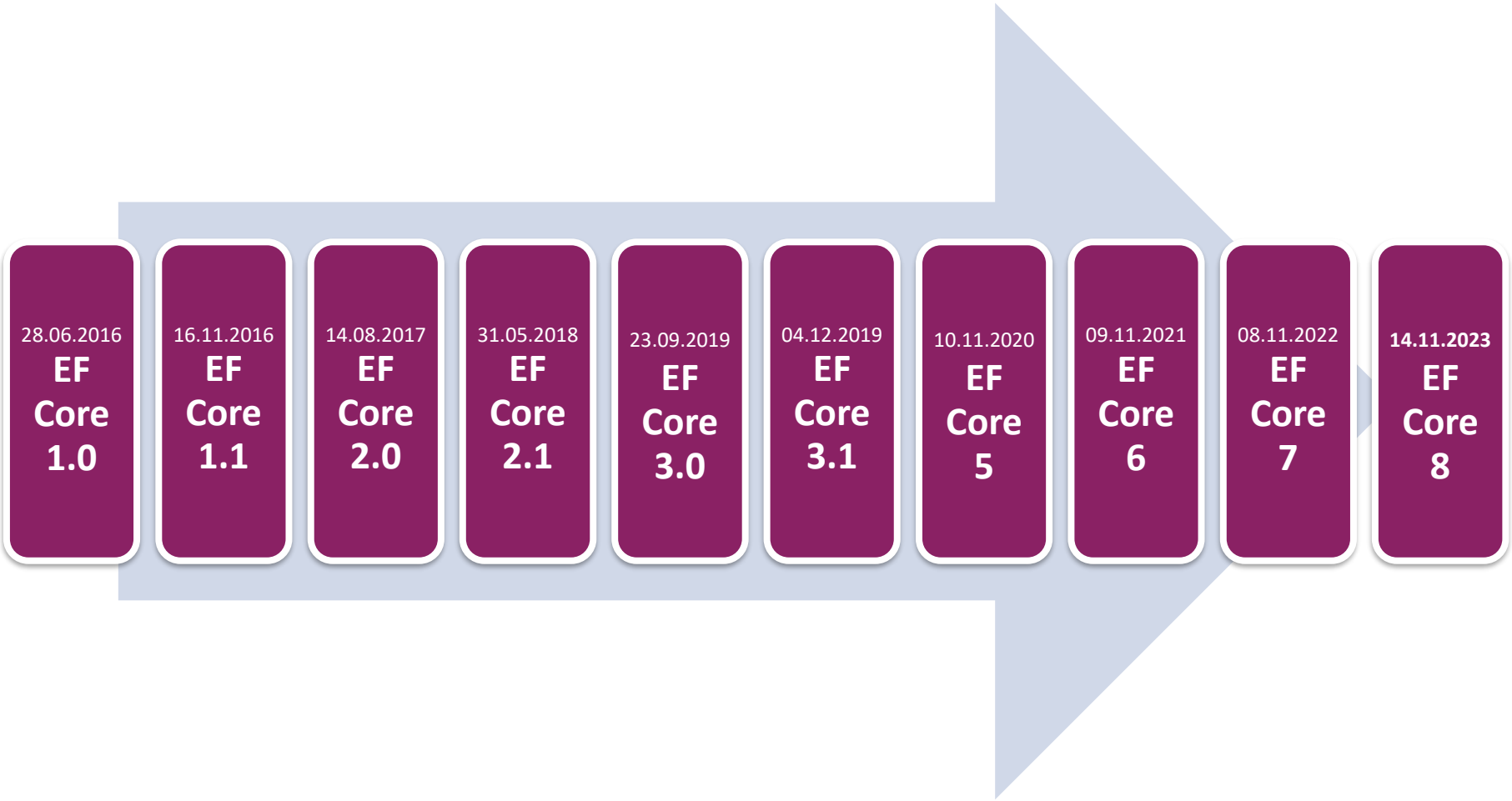
Logging mit SeriLog

Exkurs: Entity Framework Core

ADO.NET Entity Framework

- ORM-Framework
  - Bildet Objekte auf eine relationale DB ab
  - Blazor, Web Forms, MVC, WPF, WCF, Web API
  - Unabhängig von verwendeter Datenbank

Versionshistorie Entity Framework



Ohne Entity Framework

- Fehleranfällig, Unsicher, Kompliziert

```
string con = "Data Source=.\\;Initial Catalog=AutorDB;IntegratedSecurity=True";
string cmd= "INSERT INTO Author (LastName,FirstName) VALUES (‘Twain', ‘Mark')";

using (SqlConnection connection = new SqlConnection(connString))
{
   using(SqlCommand com= new SqlCommand(cmdString))
   {
      com.Connection= con;
      connection.Open();
      com.ExecuteNonQuery();
   }
}
```

Mit Entity Framework

- Objekte <> Tabellen Abbildung
- Schutz vor SQL-Injections
- Vermeidung von Syntaxfehlern
- ConnectionString in Konfigurationsdatei

```
using (var db = new AuthorDBContext())
{
   db.Authors.Add(new Author { LastName = "Twain", FirstName = "Mark" });
   db.SaveChanges();
}
```

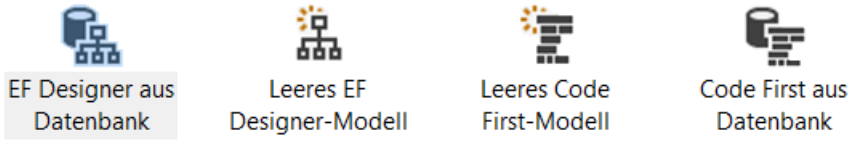
EF Begrifflichkeiten

- Domain-Model
  - Konzeptionelles Model, welches ein Geschäftsproblem darstellt, aber noch keine technische Lösung
- Objekt – Modell
  - Eine Klasse, die die Eigenschaften eines realen oder abstrakten Objektes beschreibt
- ER – Modell
  - Visualisiertes Datenmodell
- DB – Modell
  - Datenbankdiagramm einer physischen Datenbank

Aufbau Entity Framework

- Entity
  - = Tabelle in Datenbank
  - Eigenschaften = Spalten der Tabellen
- Relationship
  - Fremdschlüsselbeziehungen in der Datenbank
- Context
  - Repräsentiert den Zugriff auf die Datenbank
  - CRUD

Model Workflows EF 6 im klassischem .NET



Vorgehensweise	Datenbank vorhanden	Beschreibung
Code First	**Nein**	Objekte und deren Beziehungen werden in Klassen beschrieben. Daraus wird dann mit dem EF die DB erstellt
Model First	**Nein**	Grafisches Datenmodell aus dem die DB erstellt werden kann
Database First	**Ja**	Entity Data Model Designer entwickelt aus einer DB ein Objekt-Modell, das grafisch im Designer dargestellt wird
Reverse Engineering Code First	**Ja**	Generierung von Entity Framework Klassen aus der DB

Model Workflows EF Core

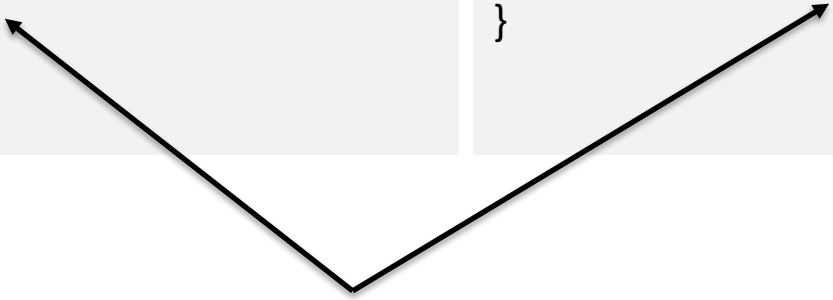
Vorgehensweise	Datenbank vorhanden	Beschreibung
Code First	**Nein**	Objekte und deren Beziehungen werden in Klassen beschrieben. Daraus wird dann mit dem EF die DB erstellt
Reverse Engineering Code First	**Ja**	Generierung von Entity Framework Klassen aus der DB

Code First Entitäten

- Für jede Entity eine Klasse erstellen

```
public class Author
{
   public Guid AuthorId { get; set; }
   public string Name { get; set; }
   public string Description { get; set; }
   public virtual List<Quote> Quotes { get; set; }
}
```

```
public class Quote
{
   public Guid QuoteId { get; set; }
   public string QuoteText { get; set; }
   public int AuthorId { get; set; }
   public virtual Author Author { get; set; }
}
```



Beziehung 1 : n

Code First Data Context

- Erstellen eines EF Data Context
- Data Context => DB
- Jede Entity **T** eine Eigenschaft **DbSet<T>** hinzufügen

```
public class QuoteContext : DbContext
{
    public QuoteContext(DbContextOptions<QuoteContext> options) :
                                base(options)
    {}

    public DbSet<Author> Authors => Set<Author>();

    public DbSet<Quote> Quotes => Set<Quote>();
}
```

Verwenden EF in Controller

- DataContext instanziieren
- Gewünschte Entitäten abfragen (Filtern möglich)
- Als Model an View übergeben

```
public partial class Index
{
    [Inject]
    private QuoteContext db {get; set;}

    public IEnumerable<Author> Authors { get; set; }

    protected override async Task OnInitializedAsync()
    {
        Authors = db.Authors.ToList();
    }
}
```

Database Migrations

- Löst das Problem Code und Datenbank synchron zu halten
- Möglichkeit von Snapshots
  - Generierung von Scripten um Code/Datenbank zu synchronisieren
  - Migrierung & Scripting SQL diverser Snapshots
- NuGet-Konsole
  - **`Add-Migration`** fügt neue Migration ein
  - **`Update-Database`** synchronisiert Modell mit Datenbank
  - **`Script-Migration`** erstellt SQL-Script

EF Core Commands

Installation des Tools

dotnet tool install --global dotnet-ef

https://docs.microsoft.com/de-de/ef/core/miscellaneous/cli/dotnet

Package Manager Console	dotnet CLI	Beschreibung
add-migration <migrationsname>	**dotnet ef migrations add <migrationsname>**	Erstellt eine Migration mit Migrations Snapshot
Remove-migration	**dotnet ef migrations remove**	Entfernt die letzte hinzugefügte migration
Update-database	**dotnet ef database update**	Aktualisiert die
Script-migration	**Dotnet ef migrations script**	Generiert ein SQL Script mit allen Migrationen

Bootstrap 5

Die „K“-Frage für Webentwickler

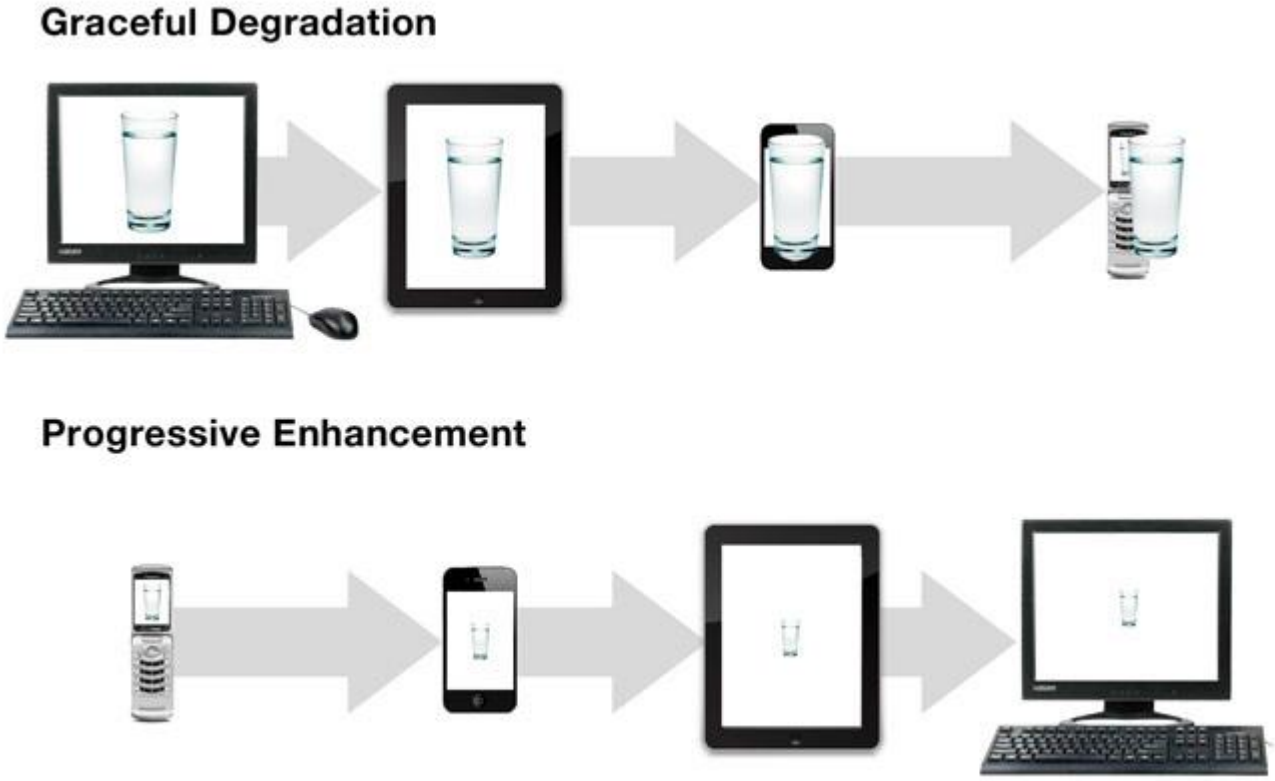
Künstler

Klempner

Was ist Bootstrap?

- Open Source Front-End Framework von Twitter
- Mobile First Responsive Web Design
- Modular
- Besteht aus CSS und JavaScript
- Vielfalt an Designkomponenten
- Standard-Designvorlage seit MVC 5

Mobile First Responsive Design

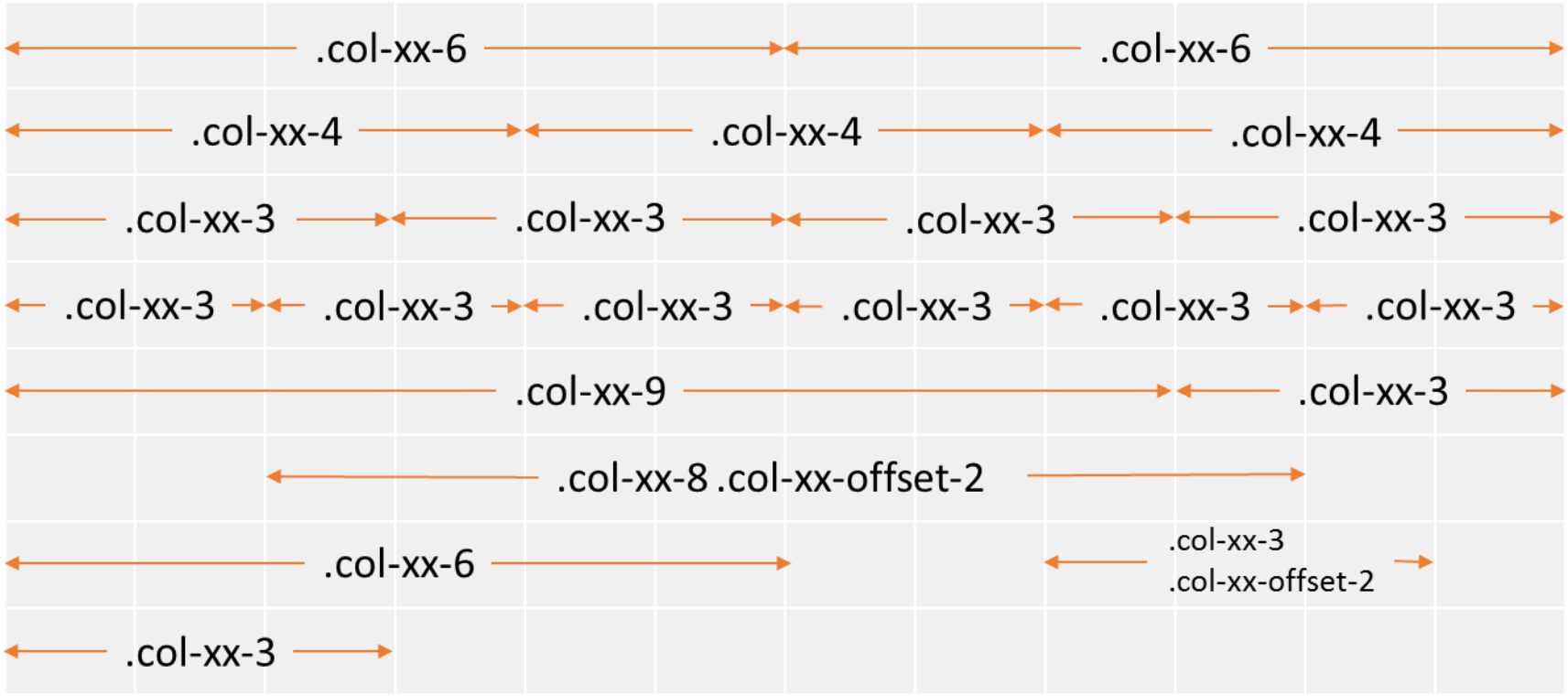


Quelle: http://www.deepblue.com

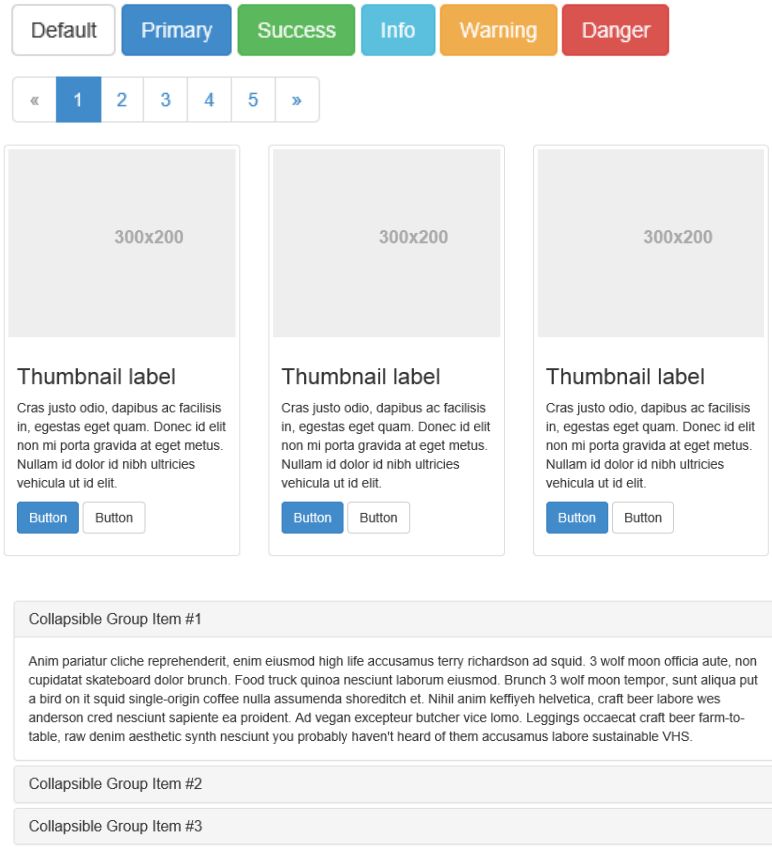
Grid System

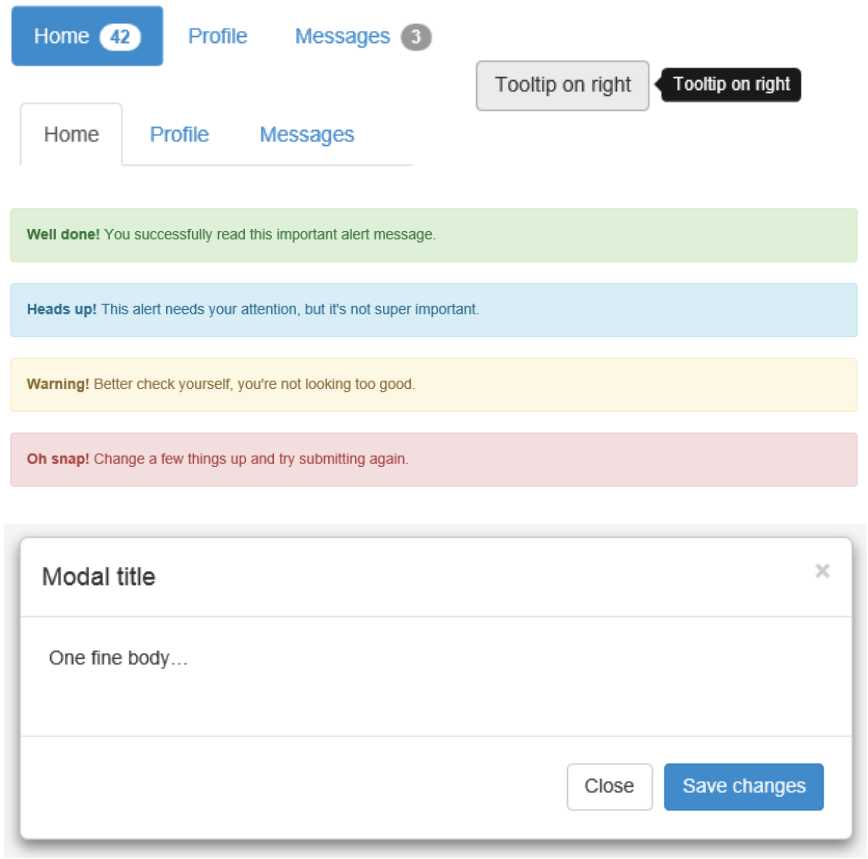
	xs <576px	sm ≥576px	md ≥768px	lg ≥992px	xl ≥1200px	xxl ≥ 1400
Max container width	None (auto)	540px	720px	960px	1140px	1320px
Class prefix	.col-	.col-sm-	.col-md-	.col-lg-	.col-xl-	.col-xxl-
# of columns	12					
Gutter width	1.5rem (.75rem on left and right)					
Custom gutters	Yes					
Nestable	Yes					
Column ordering	Yes					

12 Column Grid System



Bootstrap Komponenten Auszug





Links

- Bootstrap 5
  - http://getbootstrap.com/
- Themes
  - http://bootswatch.com/ (Free)
  - https://wrapbootstrap.com/ (Kommerziell)
- Online Editor
  - http://bootply.com/
- Fuel UX
  - http://earmbrust.github.io/dieselui/#