Browser-based Applications

Today, browsers provide the required feature-set and performance to run sophisticated, web-based applications.

*Vorteile*: Client Backups werden nicht benötig, Plattform unabhängig, Keine Software Updates für Client notwendig, Software as a Service ist möglich;

*Nachteile*: Keine Hoheit über die Daten, Limitierter HW Access, Kein OS, Läuft überall (muss sehr flexibel, kompatibel sein), SEO, Kompatibilität (z.B. Schwerer bei SPS)

SPA (Single Page Application)

Schreibt Teile des DOMs um, statt ganze Seite. Logik wird von Server auf Client verschoben. Aller Code wird beim Initialen Aufruf geladen. Lazy Loading. Fühlt sich wie eine Desktop app an. Routing on Client-Side.

Diagram, schematic

Description automatically generated  
Vorteile:

Keine Page Reloads

* Fast Actions (Sofortiges Feedback)
* Komplexe Interaktionen einfach zu entwickeln
* Offline Usage, schneller Pageload

Nachteile:

* Langer und grosser initialer Download
* Abhängigkeit von einem Frontend und Updates
* Support von JavaScript (Crawlerprobleme)
* Back Button und Reload broken per Default
  + Kann in JS gefixt werden durch window.history oder nutzen von Anchors (#)

Ein Bild, das Text enthält.

Automatisch generierte BeschreibungEin Bild, das Text enthält.

Automatisch generierte Beschreibung

Bundling SPA (Development Setup)

* JS-Code wird kombiniert in ein JS und minifiziert. Teils auch Legacy-Browser Support wird geadded.
* Webpack ist bekannter Bundler. Konzepte:
  + **Entry**: Wo starten und dependency Graph folgen
  + **Output**: Wo das Ziel-JS-File platzieren
  + **Loaders**: Lädt module/Rule API und fügt diese zu
  + **Plugins**: Eigene Funktionen und Aktionen, Asset Management, Environment-Variablen
  + **Mode**: Minimizer, Optimierer

Dependency Injection

Reduziert verbindung zwischen Konsumer und Implementation. Erlaubt Kapselung und einfaches Testen, durch mocken der Dependencies.

React

React ist eine Libary, aber KEIN Framework. «The V in MVC», bietet minimales Featureset. Teilt alles in einfache Komponenten auf (wiederverwendbar, erweiterbar, wartbar, testbar, Aufgabenverteilung)

JSX (JavaScript ⚭ XML)

const menu = entries.map(entry =>  
 <**ListItem** as="a" to={`/${entry.path}`}>  
 <**h1**>{entry.title.toUpperCase()}</**h1**>  
 <**p**>{entry.subtitle}</**p**>  
 </**ListItem**>  
)

* Programmiersprache von React
* Kann an beliebiger stelle auftreten. {} = JS-Code
* React-Elemente beginnen mit Grossbuchstabe
* JS Keyword Ersatz: z.B (class 🡺 className)
* Style-Attribute nicht als String, sondern Objekt

**function** **Container**(props) {  
 **const** style = {display: 'flex', width: '100%', minHeight: 300};  
 **return** <div style={style}>{props.children}</div>;}

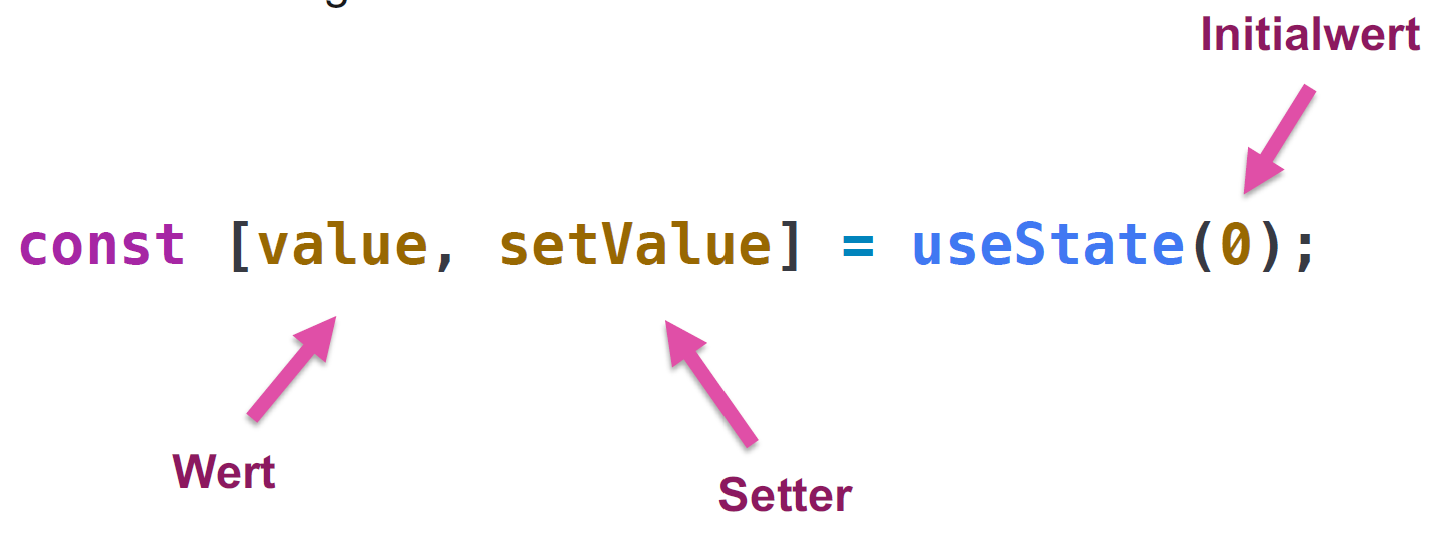
* Conditionals sind möglich, da Bool nicht gerendert
* JSX wird umgewandelt zu React.createElement. Deswegen: «import React from 'react'»
* Const-Definition oder auch Klasse möglich:

**function** **HelloMessage**(props) {**return** <div>Hello {props.name}</div>;}  
*// Äquivalent:*  
**const** HelloMessage = (props) => <div>Hello {props.name}</div>;  
**const** HelloMessage = ({ name }) => <div>Hello {name}</div>;  
**class** **HelloMessage** **extends** **React**.**Component** {  
 render() {  
 **return** <div>Hello {this.props.name}</div>  
 }}

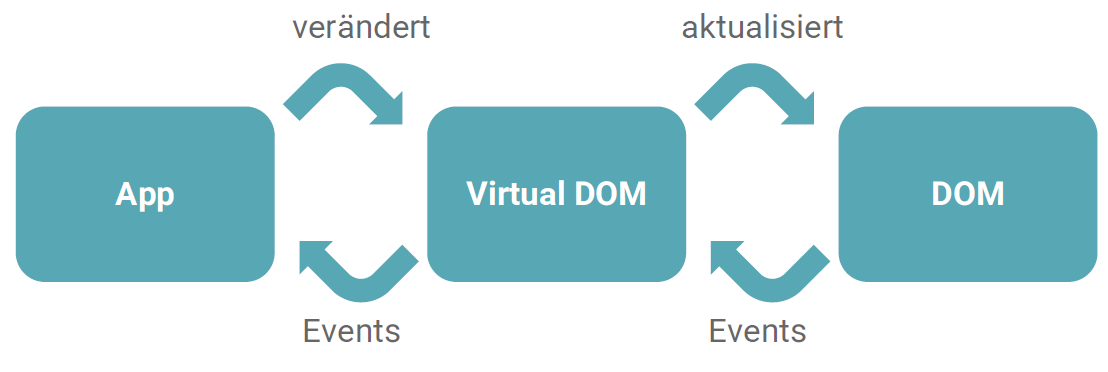
* Alle Parameter/Properties im props-Objekt
* Mounting mit

const root = ReactDOM.createRoot(document.getElementbyId('root'));  
root.render(<App />);

State:



* useState Hook wird benötigt.
* Mittels setXY kann Wert aktualisiert werden.
* React tut dann intelligent DOM aktualisieren (**Reconciliation**)
  + Vitueller DOM wird verglichen mit richtigem



* State einer Komponente ist immer Privat 🡺 Weitergabe via props, was auch mit Handler geht
* Props bevorzugen, State nur wenn nötig
* In Klasse mit: this.state, this.setstate

Formulare

* Event-Handler bei Inputs registrieren und Zustand ändern. 🡺 Controlled Components

<input value={username} **type**="text"  
onChange={e => setUsername(e.target.value)} />

* Formular Submit: event.preventDefault()

*Widget-Libaries*: Binden bekannte Frameworks ein als React-Komponenten (Reactstrap, Semantic UI React)

Komponenten Lifecycle

Mounting 🡺 Updating ⟳ 🡺Unmounting (🡺 Error)

🡺Bessere Lösung: **Effect Hooks**

**function** **ClockHooked**() {  
 **const** [date, setDate] = useState(**new** Date())  
 useEffect(() => { *// mount*  
 **const** timerID = setInterval(() => setDate(**new** Date()), 1000)  
 **return** () => {clearInterval(timerID)}}*// unmount* , [] *// dependencies*)  
 **return** (  
 <div>  
 <h2>It is {date.toLocaleTimeString()}.</h2>  
 </div>)}

Dependencies ([]) helfen um Effect nur auszulösen, wenn sich bestimmte Variablen ändern. Ebenso fetch with then ist möglich, da async.

React Router:

Komponentenbibliothek um Komponenten anzeigen, je nach URL.

* *<BrowserRouter>*: (Alle Routen müssen Teil davon sein>
* *<Route>*: Wird nur gerendert wenn path matcht
* *<Link>:* statt a zu verwenden bei internen Links.

function App() {  
 return (   
 <BrowserRouter>  
 <div>  
 <ul>  
 <li><Link to="/">Home</Link></li>  
 <li><Link to="/list">List</Link></li>  
 <li><Link to="/sap">SAP</Link></li>  
 </ul>  
 <hr />  
 <Routes>  
 <Route index element={<Home/>} />  
 <Route path="/list" element={<List/>} />  
 <Route path="/sap" element={<SAP/>} />  
 </Routes>  
 </div>  
 </BrowserRouter>)}

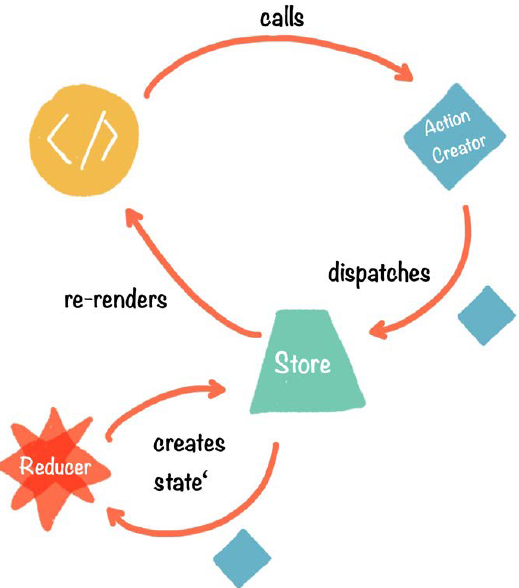
Typechecking

* Flow: Erweitert JavaScript um Typenannotionen, Zukunft ungewiss
* Typescript: Superset von JavaScript, vollwertige Sprache, besser von Libaries und IDEs supported
* React Context:Ermöglicht Props für alle Unterkomponenten zur Verfügung stellen. (Globale Read-Only-Props)

**const** themes = {…}  
**const** ThemeContext = React.createContext(themes);  
**function** **App**() {  
 **return** (  
 <ThemeContext.Provider value={themes.dark }>  
 <Toolbar />  
 </ThemeContext.Provider>);}  
function Toolbar() {  
 const theme = useContext(ThemeContext);  
 return ( <p style={{color: theme.foreground}}>…</p>)}

Ideal für globale read-only Props. Wird schnell unübersichtlich. So eingeschränkt nützlich, evtl. bei Libaries.

Redux



* Libary für Statemanagement. React-Unabhängig.
* State als Tree von Objekten, immutable im Store
* Actions benötigt für Statusänderung
* Reducer-Funktion führt Status durch. Erstellt t+1
  + Jeder Reducer Teil-Tree, Verbunden in Root-Reducer
* React-Toolkit für einfache implementation in React
  + Reducer und Actions via createSlice
  + Store Initialisierung voa configureStore
  + Einbinden der Store in Root-Komponente via Provider
  + Dispatch der Aktionen mit useDispach
  + Zugriff auf State mit UseSelector
  + AsyncThunk

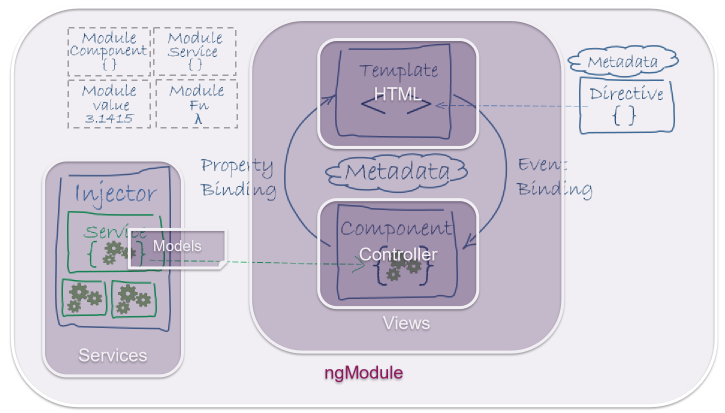
*// Slice*  
**const** balanceSlice = createSlice({  
 name: "balance",  
 initialState: { value: 0 },  
 reducers: {transfer: (state, action) => {  
 state.value += action.payload.amount;},},});  
**export** **const** {transfer} = balanceSlice.actions;  
*// configure Store*  
**import** { configureStore } **from** "@reduxjs/toolkit";  
**import** { balanceReducer } **from** "./redux/balanceSlice";  
**const** store = configureStore({reducer: { balance: balanceReducer}});  
**export** **default** store;  
*// Include in React*  
**import** store **from** "./store";  
render(  
 <Provider store={store}>  
 <App />  
 </Provider>,  
 document.getElementById('root'))

**ReduxThunk** 🡺 Im Slice via extraReducers und builder mit cases für pending, fulfilled und error unterschiedliche Actions. Zusätzlich state in Sliche aufnehmen.

**Fazit**: Redux bietet bessere Übersicht, da Zustand an einem Ort, seralisiert möglich und auch Server Rendering. Einfaches Debugging durch Time-Travelling.

Angular Architektur

Flexibles SPA Framework für CRUD Applikationen. TypeScript-Basiert, Reduziert Boilerplate, code. Hat Dependency Injection, Gute Performance durch 2-Way-Binding. Gut für langlebige Applikationen, aber hart zu lernen.

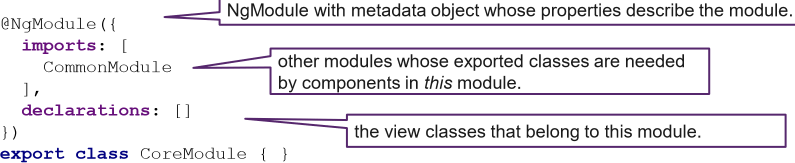
Architectural Overview

* ***ngModules***: A cohesive block of code dedicated to closely related set of capabilities. TS: module
* ***Directives***: Provides instructions to transform the DOM. TS: class, wie namespace oder packet
* ***Components***: A component is a *directive-with-a-template*; it controls a section of view. TS: class
* ***Templates***: A template is a form of HTML that tells Angular how to render the component. HTML/CSS
* ***Metadata***: Metadata describes a class and tells Angular how to process it. TS: decorator
* ***Services***: Provides logic of any value, function, or feature that your application needs. TS: class

Angular Modules

Basis für Modularität. App besteht aus mind. einem root-Module. Module können aus N Module bestehen. Enthalten und exportieren Klassen, Funktionen.

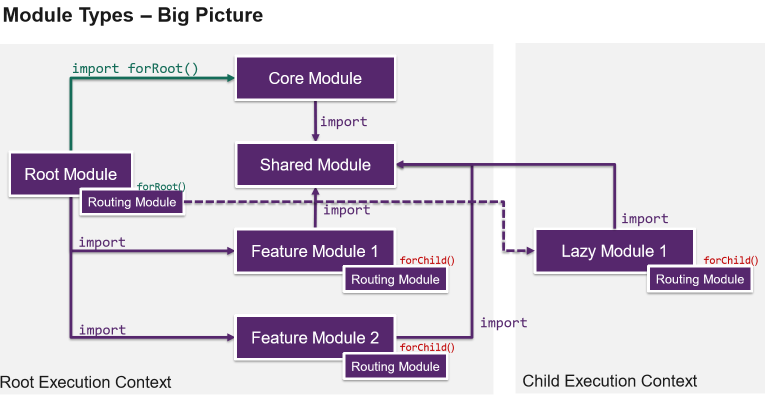
ngModule linkt mehrere TypeScript-Module.

  
Möglichst flache Ordnerstruktur. Metadaten:

* *declarations: [ Type1, Type2, Type3, …]:* The view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes
* *exports: [ Type1, Type2, … Module1, Module2 ]:* The subset of **declarations** that should be visible and usable in the component templates of other modules. Can re-export other modules, which are automatically included when importing module.
* *imports: [ Module1, Module2, …]:* Specifies the modules which **exports/providers** should be imported into this module.
* *providers: [ Provider1, Provider2, …]:* Creators of services that this module contributes to the global collection of services (Dependency Injection Container); become accessible in all parts of app.
* *bootstrap: [ Component ]:* The main application view, called the root component. Only the root module should set this property.

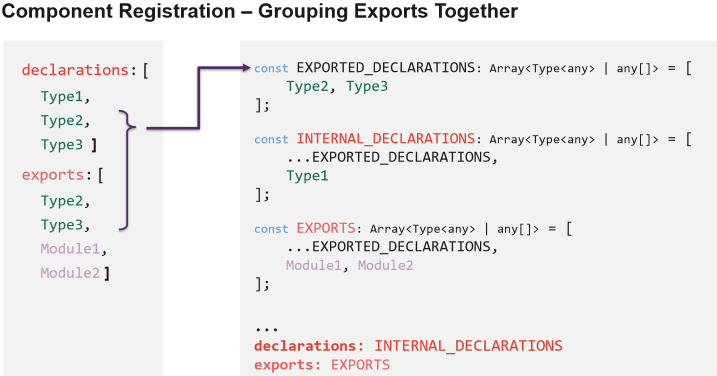
Overview Application Modules *Core Module*: Basis Services & Components

*Shared Module*: Globale UI Sammlung, gemeinsame Components & Pipes



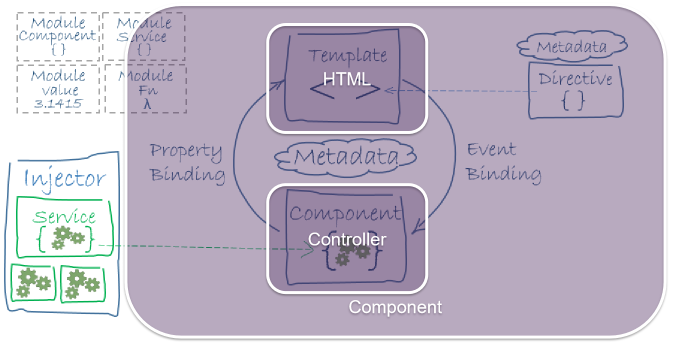
Angular Components

* Besteht aus *TS Controller* für Applikationslogik der View, *@Component* TS class, *HTML File* für Darstellung mit Mustache-based Templates, *CSS/SCSS File* für Styles
* Separation von UI Logik und Repräsentation und kann nested sein 🡪 Component Tree
* *Information Hiding*: Each Component declares a part of the UI 🡪 testbar, wartbar und wiederverwendbar
* Components sind Bestandteil eines ngModules, welches mehrere Components deklariert und diese exportieren, damit sie in anderen Modulen verwendbar werden**. Selector** des Components wird für Subcomponent registriert.



Component Lifecycle

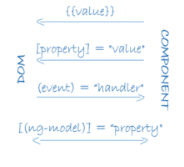
* *ngOnInit():* Creation event – **hydration**, Setup the component and initially fetch data from an underlying data source.
* Update



* *ngOnDestroy():* Destruction event- **dehydration**, Use this method to detach event handlers to avoid memory leaks.
* **View Encapsulation**: CSS gilt nur für Komponente selbst. So kann direkt div und p gestylt werden. :host

Angular Templates

**Template ist die View bei MVC**. HTML mit Angular Template Syntax, definiert wie Components gerendert werden.

**Binding**: Mechanism for coordinating how UI dependent information is managed.

Angular provides multiple data binding facilities:

* One-way *in* from data source to **view** target   
  🡪 **Property Binding** <p>... {{counter.team}} ..</p>
* One-way *call* from **view** target to data source   
  🡪 **Event Binding** <button (click)="counter.eventHandler($event)">
* Two way 🡪 **Property and Event Binding**<input type="text" [(ngModel)]="counter.team">

Content Projection <ng-content select=’css’> um Subkomponenten in Komponente Einbinden

**Pipes** (|) werden verwendet, um kleine Transformationen an Resultaten einer Expression durchzuführen. Können **gechained** werden mit mehreren Parametern.

Pure vs. Impure

Pure Pipes werden nur auf **bound expressions** ausgeführt, sind darum restriktiv, aber schnell.

Impure Pipes reagieren auf jede Component Änderung (Mouse Move, Keystroke etc.), darum **gefährlich**, wenn long-running.

**FilterPipes**

Es gibt keine FilterPipes in Angular 🡪 Verlagerung der Filter- und Sortierlogik in die Components selbst.

**Asyncrone Pipe**: Bound Observable direct to UI with AsyncPipe

Angular Directive

Ähnlich zu Komponenten, ohne Template

* *Attribute Directive*: Ändert das Aussehen oder Verhalten eines vorhandenen Elements, Components oder anderer DirectiveEin Bild, das Text enthält.

  Automatisch generierte Beschreibung
* *Structural Directives*: Modifiziert Struktur des DOM. Verantwortlich für HTML Layout. Mit \* Ein Bild, das Text enthält.

  Automatisch generierte Beschreibung

Angular Services

Bietet jeden **Value**, jede **Funktion** oder jedes **Feature**, das die App benötigt. Im Module unter Providers exportieren.

Angular verwendet DI, um neue Komponenten mit den benötigten Services auszustatten. Darum müssen die **Services innerhalb des DI-Containers registriert** werden.

TS **@Injectable** Decorator

Als providedIn: ‘root’ registrieren und im aufrufenden Component/Service in Ctor injecten oder im ngModule deklarieren und in forRoot() platzieren für multi-providers

Angular Forms

Nutzen vom NgModule namens FormsModule. Es gibt:

* *Template-Driven forms*: Nutzt Template-Syntax. Benötigt weniger Code, aber Logik der Validation im HTML.   
  🡺 Intutitiver, weniger Code, Gut für kleine Forms
* *Reactive (or model-driven) Forms*: Benötigt ReactiveFormsModule. Form wird im Controller gebaut (FormBuilder). Logik der Validation im Controller. Asynchrone Validation.   
  🡺 Validierung testbar, Async, Für everyday Use.

Template-Driven Forms

* Two-Way Binding mit [(ngModel)] Directive in <input>, HTML5-validatior wie pattern und required.
* Somit automatisch validiert. Referenz möglich im Template: Ein Bild, das Text enthält.

  Automatisch generierte Beschreibung

Angular Kommunikation (Asynchrone Services)

**Component (UI Layer)**

Um auf Updates eines Events zu hören, wird auf dem unterliegenden **Business Service subscribed** (hydration). Damit Daten auf dem UI Level refreshed werden.

**Business Service (Business Layer)**

Subscribes auf DAL Observable.

Angular ermöglicht es mit der **EventEmitter-Klasse** Components über Data Updates zu informieren (**Observer Pattern**). Der Component registriert seine Events auf dem Service. Bei Updates der Daten werden alle registrierten UI Components informiert.

**Data Resource Service (DAL)**

Liefert Observable Interface.

Kommuniziert über Angular **HttpClient** mit Backend API. HttpClient implementiert Asynchronität mit RxJS (**Observable Pattern**). **HttpClient Methoden == Cold Observable**

*Hot Observable:* Sequenz von Events (Mouse Clicks), Shared für alle Subscriber

*Cold Observable*: Startet sobald subscribed (Http Requests). Nicht geshared, Geschlossen sobald Task fertig.

**Interceptors**

Mit Interceptors können http Headers pro Request gesetzt werden. Aus Sicherheitsgründen kann es erforderlich sein, den Authorization Header (bearertoken) pro Request einzufügen. Interceptors müssen im DI Container registriert werden.

Angular Routing

Mit Angular Router kann zwischen Views navigiert werden, Routes werden in **Routing Module** definiert. Jedes ngModule sollte sein eigenes Routing Module enthalten. Initiale Top-level Routes müssen mit **forRoot()** registriert werden. Angular verwendet die **History.pushState** des Browsers für die Navigation.

*forRoot():* use forRoot()-import **EXACTLY** once to declare routes on root (top) levels. Contains all the directives, the given routes, and the **router service** itself.

An Angular app has **one singleton instance of the Router service**.

*forChild():* use forChild()-import when declaring sub-routings (on all sub-levels)

Routes von **oben gegen unten definieren**! Damit nicht **pagenotfound** immer gematched wird.

**Router Outlet**

RouterOutlet is a directive from the Router module.

Legt fest, wo der Router die Views anzeigen soll. Kann in jedem Subcomponent definiert werden.

**Lazy Modules**

Lazy Modules werden geladen, sobald die Route das erste Mal angefragt wird. Lazy Module sollten nicht direkt von eagrly loaded Modules referenziert werden, sonst werden muss das Modul bei Applikationsstart verfügbar sein.

**Guards:**canLoad: nur Laden, wenn Guard zutrifft.  
 canActivate: Guard um route zu disablen/blocken (security), wenn nicht zutrifft.

Angular Architecture Pattern

Flux und Redux Pattern (wie bei React) bei grösser werdenden & komplexeren Applikationen einsetzten, sonst MVC oder MVC+S.

MVC+S Pattern

RxJS wird durch alle 3 Layer gezogen 🡪 Async UI Sections.

*Business Service*: Liefert Daten im streaming style. Hot Observale mit RxJS Subject also EventEmitter.

*Data Service*: Liefern Cold Observable konvertiert für BS in Hot Observable und muss diesen über Change notifien.

Flux Pattern

Action wird von Dispatcher an alle Stores gesendet, dieser updated sich und löst Change Event für View aus.

Redux Pattern

Analog Redux React mit Store und Reducers. Modul ngrx und ngrx/@data

PWA

PWA nutzen Web-API mit Progessive Enhancement um **cross-plattform Web Apps** zu erstellen.

Eigenschaften

* *Safe*: Die neuen Features wie Push-API sind nur über HTTPS möglich. Chrome erlaubt diese Features auch auf Localhost für die Entwicklung.
* *Responsive*: Das UI soll für alle Grössen passen. Wie Media Queries. (Desktop, Tablet, Smartphone, Brillen, Uhren)
* *Progressive*: Die neuen API’s nutzen um die bestmögliche UX zu erreichen, ohne die älteren Browser zu ignorieren   
  🡪 Progressive Enhancement
* *Linkable*: Teilen von Inhalten ohne das der Sharing-Partner eine App installieren muss.
* *Discoverable*: Die Webseite soll über Suchmaschinen auffindbar sein 🡪 Web App Manifest
* *Installable*: Die Web Applikation mit Icon auf den Startbildschirm speichern 🡪 Web App Manifest
* *Network independent*: Die App soll mit schlechter, langsamer oder sogar gar keiner Verbindung funktionieren.
* *Re-engageable*: Es soll möglich sein das die Verbindung zum Benutzer der Seite wieder aufgenommen werden, obwohl der Benutzer die Seite nicht geöffnet hat.   
  🡪 Service Workers, Push-API

ASP.NET (Core)

Framework für Server-Side-Apps (Vergleichbar Express.js)

* **Convention over configuration**: Unterstützt Entwickler beim Entscheiden und vereinfacht Entwicklung massiv.
* **Multi-Threading**: Thread-Pool. Jeder Request nutzt eigenen Thread (Bleibt blockiert bis Request geschlossen).
* **Front**-**Controller**: Entry point eines http-Request, Routing
* **Middleware**: Request durchläuft Stack von Middlewares. Jede middleWaware kann terminieren oder next(). Kann auch bei Rückgabe Dinge tun.
  + Registrierung (Startup.cs)
    - 1. Neue Middleware

app.Use(async(context, next) => {  
 System.Diagnostics.Debug.WriteLine("Handling request");  
 awaitnext.Invoke();  
System.Diagnostics.Debug.WriteLine("Finishedhandlingrequest.");});

* + - 2. Erzeugen Verzweigung für Anfragepfad

app.Map("/logging", builder=> {  
 builder.Run(async(context) => {  
 awaitcontext.Response.WriteAsync("HelloWorld!");});  
});

* + - 3. Terminiert den Request, keine weiteren Middlewares werden aufgerufen

app.Run(async(context) => {  
awaitcontext.Response.WriteAsync("HelloWorld!");});

* + - 4. Custom Middleware (Klasse)

Dependency Injection

Reduziert hohe Koppelung zwischen verschiedenen Klassen und verbessert Testbarkeit der Applikation. Wird nach Registrierung über Ctor injected. *DBContext ist nicht Multi-Threading safe!*

**Lebensdauer:**

* *Transient*: Transient lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.
* *Scoped*: Scoped lifetime services are created once per request.
* *Singleton*: Singleton lifetime services are created the first time they are requested (or whenConfigureServices is run if you specify an instance there) and then every subsequent request will use the same instance.

A screenshot of a computer

Description automatically generated with low confidence**Wichtig**: Komponenten dürfen sich nur Komponenten mit gleicher oder längerer Lebensdauer Injection lassen.

ASP.NET Core Webapp

Besteht aus Pages & MVVM

*Pages***:** Alternative und **vereinfachte Variante vom MVC**, Router muss nicht konfiguriert werden. **Best-Practices für Serverseitiges-Rendering.**

*Routing Pages*: 1. Eine Web Applikation generiert anhand von der URL eine Antwort. 2. Eine URL wird auf eine Aktion "gemappt". 3. Bei einem Aufruf wird im Folder /pages/ nach einer Page gesucht und ausgeführt.

*@page "/test/{id?}":* Der Parameter überschreibt die Default-Routing-Informationen

*MVVM*: Pages hat 2 Files: View (.cshtml), View Model (.cshtml.cs)

*RAZOR Files*

* *\_Layout.cshtml*: Definiert generelle Layout für die App.
* *\_ViewStart.cshtml*: \_Layout hier definieren
* *\_ViewImports.cshtml*: TagHelpers registrieren
* *Appsettings.json*: DB Connection, Logging, usw.

**TagHelpers**

Tag Helpers ermöglichen C# Code an HTML Tags zu binden. 

**Partial**: dummer Component

**ViewComponents**: bessere Partials (Component wie bei Angular oder React)

**ViewData**: Daten werden im LayoutTree verfügbar

**TempData:** Wert wird in Cookie gespeichert, Refresh OK

**Index.cshtml**

**AJAX**

Query Unobtrusive AJAX"-Library ermöglicht es Form-Requests ohne eigenen JavaScript Code zu definieren.

Beispiel ASP.NET Page Ajax

**@page**   
**@model** **IndexModel** *//VM Type -> var Model*  
 **ViewData**["Title"] = "Home page";  
}  
<form asp-page="Bmi" data-ajax="true" data-ajax-method="GET" data-ajax-mode="replace" data-ajax-update="#result">  
 <div **class**="form-row">  
 <div **class**="form-group col-md-6">  
 <label asp-**for**="@Model.Bmi.Height"></label>  
 <input asp-**for**="@Model.Bmi.Height" name="height" **class**="form-control">  
 </div>  
 <div **class**="form-group col-md-6">  
 <label asp-**for**="@Model.Bmi.Weight"></label>  
 <input asp-**for**="@Model.Bmi.Weight" name="weight" **class**="form-control"/>  
 </div>  
 </div>  
 <button **type**="submit" **class**="btn btn-primary">**Submit**</button>  
</form>  
<div id="result"></div>  
**@section** **Scripts**{<script src="https://cdn.jsdelivr.net/npm/jquery-ajax-unobtrusive@3.2.6/dist/jquery.unobtrusive-ajax.min.js"  
script>}

**Index.cshtml.cs**

**public** **class** IndexModel : PageModel  
{ *//Aufbereitung der Daten für View*  
 **public** Bmi Bmi { **get**; **set**; } = new Bmi(){Height = 170, Weight = 80};   
 **public** void OnGet(){ }}

**Bmi.cshtml**

@page  
@model BmiRechner.Pages.BmiModel  
@{  
 ViewData["Title"] = "Bmi";  
 Layout = null;  
}  
@**if** (Model.WrongData){  
 <strong>Falsche Daten!</strong>}  
**else** {  
double value = Model.BmiValue;  
**var** data = **new**[]{  
**new** {Text = "starkes Untergewicht", Max = 16.0}};  
 <table class="table">  
 <caption>Ihr BMI: <span class="badge">@value</span></caption>  
 <thead>  
 <tr>  
 <th>Beschreibung</th>  
 </tr>  
 </thead>  
 @{  
 double old = 0;  
 foreach (var entry in data)  
 {<tr class="@(value > old && value <= entry.Max ? "active" : "")">  
 <td>@entry.Text</td>  
 </tr>old = entry.Max;}}  
 </table>}

**Bmi.cshtml.cs**

**public** **class** **BmiModel** : **PageModel**{  
 **private** **readonly** IBmiService \_bmiService;  
 **public** **BmiModel**(IBmiService bmiService){  
 \_bmiService = bmiService;}  
 *// Properties werden 1:1 übergeben -> ModelBinding*  
 [**BindProperty(SupportsGet = true)**]  
 **public** Bmi Bmi { **get**; **set**; }  
  
 **public** **bool** WrongData { **get**; **set**; }  
 **public** **double** BmiValue { **get**; **set**; }  
  
 **public** **void** **OnGet**(){  
 *// Validation des Models*  
 **if** (ModelState.IsValid){  
 BmiValue = \_bmiService.Calculcate(Bmi);}  
 **else**{WrongData = true;}}}

**Startup.cs** (Mit Swagger und Routing für API)

public void ConfigureServices(IServiceCollection services){   
 services.AddSwaggerGen() ;  
 services.AddMvc();  
 services.AddTransient<IBmiService, BmiService>();  
}   
public void Configure(IApplicationBuilder app, IWebHostEnvironment env){   
 app.UseStaticFiles();  
 app.UseSwagger();  
 app.UseSwaggerUI(options =>  
{options.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");});  
 app.UseRouting();  
 app.UseEndpoints(endpoints =>{endpoints.MapControllers(); endpoints.MapRazorPages();});}

**\_Layout.cshtml**

**<!DOCTYPE html>**  
<html><head><title>@ViewData["Title"]</title></head>  
<body>@RenderBody()</body></html>

**Validation**

[**Display(Name = "Pizza Name")**]   
[**StringLength(20, MinimumLength = 3)**] [Required]  
**public** **string** Name { **get**; **set**; }

ASP.NET EF Core

OR-Mapper, Code First, Sehr viel Konvention.

DbContext in Startup für DI registrieren mit Optionen die auf ConnectionString in appsettings.json verweisen.

**Migrations**

Migrations kann das Datenbank Schema inkrementell aktualisiert werden. Prod. Deploy. ohne ganze DB abreissen

dotnet ef migrations add init 🡪 erstellt Migrations

Ausführen: in Konsole, Skript oder im Startup für Development.

ASP.NET Identity Management

Authentication (Wer bin ich? Identifikation)

[Authorize] = User muss authentifiziert sein. (Action oder Controller).

[AllowAnonymous] = Action ok für anonymer Zugriff

**User.Identity**

**this.User** = eingeloggter User (**ClaimsPrincipal**)

Claim beinhaltet Statement über User

**User**.FindFirstValue(**ClaimTypes**.NameIdentifier);

**Authorization (Was darf ich?)**

Lösung 1 Attribute:

[Authorize(Roles = "Admin,PowerUser")]  
[Authorize(Policy = "OlderThan18,Founders")]

Lösung 2 Services:

var user = await \_userManager.GetUserAsync(User);  
var isInRole = await \_userManager.IsInRoleAsync(user, "Admin");  
var result = await \_authorizationService.AuthorizeAsync(User, null, "Founders");

Lösung 3 Claims:

**User**.HasClaim(ClaimTypes.**Role**, "Admin")

ASP.NET Unit Testing

xUnit als Framework im seperaten Projekt. ([Fact])  
Einiges muss gemockt werden. Bei Datenbank wird meist einfach inMemory-DB genutzt anstelle mocken.

Integration Test: Klasse SUT mit Test-Web-Host.(Startup.cs anpassen möglich für Test, wie auch eigenes appsettings.

ASP.NET Web API

***API-Routing****:* Funktioniert über Attribute, [Route] definiert einen neuen Eintrag im Router, [HttpMethod] bei Actions ist required

Die Route für eine Action kann man direkt bei dieser definieren. Timeline

Description automatically generated

**BmiController.cs**

[**Route("api/[controller]")**]  
[**ApiController**]  
**public** **class** **BmiController** : **ControllerBase** {

**private** **readonly** IBmiService \_bmiService;  
 **public** **BmiController**(IBmiService bmiService) {\_bmiService = bmiService;}  
 [**HttpGet**]  
 **public** **double** **Calculate**([FromQuery]Bmi data)  
 {**return** \_bmiService.Calculcate(data);}  
 [**HttpGet("{id:int}")**]  
 **public** **double** **GetById**(**int** id)  
 {**return** \_bmiService.Get(id);}  
 [**HttpPost**]  
 **public** **double** **AddResult**([FromBody]Bmi data){**return** \_bmiService.Add(data);}}

***Swagger***: Spezifikation für Dokumentation von REST-API. Kann automatisch generiert werden durch Service SwaggerGen. UI ebenfalls integrierbar via Middleware SwaggerUI.

***Rest Hateoas****:* Verlinkte Dateien als Links zur verfügung stellen. Lösung: UrlHelper injecten über Startup- RestResource erstellen (erbt von Ressource), Enitäten erben Restresource welche AddSelfLink() oder AddLink() für Routen haben. Bevor return in Controller sollten die Methoden aufgerufen werden.

***Exception Handling:*** 1. Exception, welche notwendige Daten sammelt, 2. globalen Errorhandler, der Exception für Client aufbereitet, 3. Custom Exception beim ungültigen Zustand.  
***JWT Token:*** Token um Claims darzustellen, welche ausgetauscht werden. JSON based standard, via http-Header.