1 SPA

What are the benefits of a browser based application? A browser based application (web application) has various benefits:

- You can work from anywhere at anytime
- It is platform independent (even mobile)
- No software update nor installation => easy maintenance
- Software can provided as a Services (SaaS)
- Can be cross-compiled to different ecosystems
  - Client app: electron.io
  - Moile app: Native Script / Ionic
  - Server app: "Universal" Compilation

What are the liabilities of a browser based application? Browser based applications do not have only benefits (What are the benefits of a browser based application?) but also downsides, such as:

- no data sovereignty
- limited / restricted hardware access (no OS access, may be less efficient)
- Search Engine Optimization (SE must execute JS)
- More complex deployment strategies
- Overhead

What is a Single Page Application? An SPA is a special kind of Web apps.

> A Single Page Application (SPA) is a web site [...] that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In an SPA, either all necessary code [...] is retrieved with a single page load or the appropriate resources are dynamically loaded and added to the page as necessary.
>
> SPAs use AJAX and HTML5 to create responsive Web apps, without constant page reloads. -- Wikipedia

The traditional web application architecture

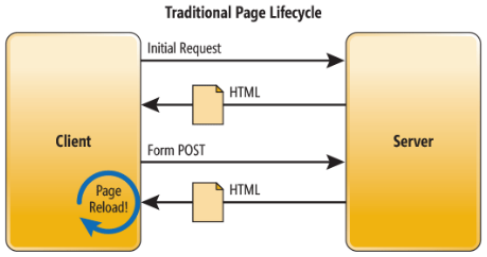

Figure 1: Traditional Architecture

The SPA architecture

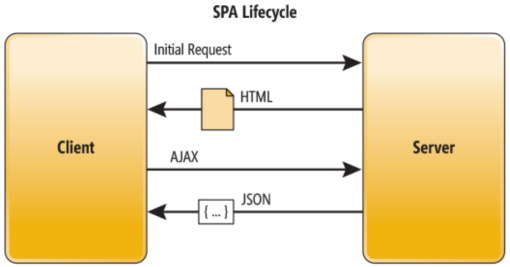

Figure 2: SPA Architecture

What are characteristics of an SPA? An SPA has the following properties:

- Plan HTML5 / CSS and JavaScript
  - no plugins like SilverLight or Flash
- no page reloads
- Working Back-Button
- Bookmarkable Links
- Provides (limited) offline functionality
- Uses (REST)-API services for data access

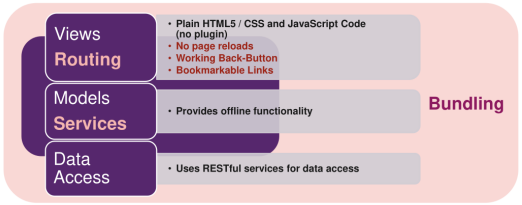

Figure 3: Logical Overview of an SPA

When would you prefer an SPA to a classic web application from the customer's point of view?

> As soon as a desktop (native) app with a similar user experience is required. The page feels like an application. An SPA also offers more options for complex web applications with lots of animations/-graphical elements.

What do you see as the technical benefits of an SPA?
The server application is separated from the display by a structured interface (e.g. REST / ODATA / WSDL). This opens up various advantages:

- Seperation of Concerns
- Better maintainability of the client code
- Division into different teams / competence centers

What does a typical layering in an SPA look like? The Views are connected using a routing in the browser (no new request to the server). The Business Logic provides data over servies and only the data layer will communicate directly with the server.

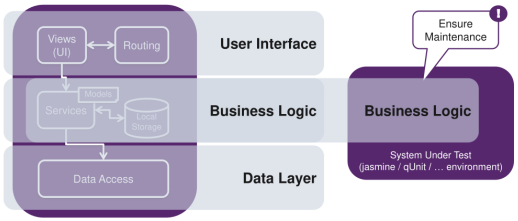

Figure 4: Layering in SPA

Why do we use bundling in an SPA?
An SPA may consist of many single JS files, which may or may not dependt on each other. To include them manually in your HTML is error pronce and tedious.
With bundling we achive the following things:

- All JS code must be delivered to the client over potentially metered/slow networks
- Bundling and minifying the source leads to smaller SPA footprint (e.g. using Tree Shaking)
- Bundling leads to a reliable dependency management
- Usage of pre and post processors during bundling

The initial footprint caused by bundling can be reduced by loading dependent modules on-demand.

2 React

What is JSX? JSX is an extension to JavaScript. It is used to write markup for an SPA. The JSX is transpiled during building into standard ECMAScript

> JSX is an XML-like syntax extension to ECMAScript without any defined semantics. It's NOT intended to be implemented by engines or browsers. It's NOT a proposal to incorporate JSX into the ECMAScript spec itself. It's intended to be used by various preprocessors (transpilers) to transform these tokens into standard ECMAScript. -- facebook.github.io/jsx/

How is JSX desugared in React?

How do you make props available for all child components? Some props must be availabel in all components (e.g. color scheme). It does not scale well, if you have to pass all props from the root component. To solve this problem, we can use contexts. However, you should only use contexts for read-only variables and limit the number of different contexts.

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee",
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222",
  },
};
const ThemeContext = React.createContext(themes.

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return (
    <button style={{
              background: theme.background,
              color: theme.foreground }}>
      {" "}I am styled by theme context!{" "}
    </button>
  );
}
```

How does Redux work? The state in Redux is represented as trees of objects. The tree is immutable. When you change something in the tree, a new tree will be created (Functional Programming).
A state action is communicated using a *Redux Action*. A *Reducer* takes the action and the current state an applies the action on the state to generate a new state.

3 Angular

Angular Angular is a SPA Framework:

- TypeScript based
- Reduces boilerplate code
- Comes with a Dependency Injection mechanism
- Provides fast, JS-optimized 2-way binding
- Clearly structured, supports information hiding principle on multiple levels of abstraction
- Increases the testability and maintainability of the client-side code
- Provides a framework that covers a wide range of topics from templating to data-services — but doesn't exactly specify how to wire them

  Angular is an advanced SPA framework… …which allows you to use the latest web engineering principles.

Angular Architectural Parts

ngModules A cohesive block of code dedicated to closely related set of capabilities
directives Provides instructions to transform the DOM
components A component is a directive-with-a-template; it controls a section of the view
templates A template is a form of HTML that tells Angular how to render the component
metadata Metadata describes a class and tells Angular how to process it
services Provides logic of any value, function, or feature that your application needs

Angular Module Declaration

- declarations: View Classes that belongs to this module
- exports: Subset of declarations that should be visible in other modules
- imports: Modules should be imported into this module
- providers: Creators of services that this module contributes to the global DI container
- bootstrap: Main application view (only in root module)

Binding Syntax

Directives Similar to a component, but without a template. Declared as a TypeScript class with an `@Directive()` function decorator.

- **Structural** directive: modifies structure of DOM
  - `*ngIf`
  - `*ngFor`
- **Attribute** directive: alters appereance / behavior of an existing element
  - `[ngStyle]`
  - `[ngClass]`

Services Provides any value, function, or feature that your application needs. Almost anything can be a service - it should do one thing and do it well.
The `@Injectable` decorater is used to register the service in the DI container. `@Injectable({ providedIn: 'root' })export class CounterService {}`

Forms

```
<form (ngSubmit)="doLogin(sampleForm)" #sampleFr
  <button type="submit" [disabled]="!sampleForm.
</form>
```

Listing 4: Angular Form Markup

```
@Component({})
export class SampleComponent {
  public doLogin(f?: NgForm): boolean {
    if (f?.form.valid) { /* do stuff */ return t
    return false;
  }
}
```

Listing 5: Angular Form Logic

Component Lifecycle **ATTACH**

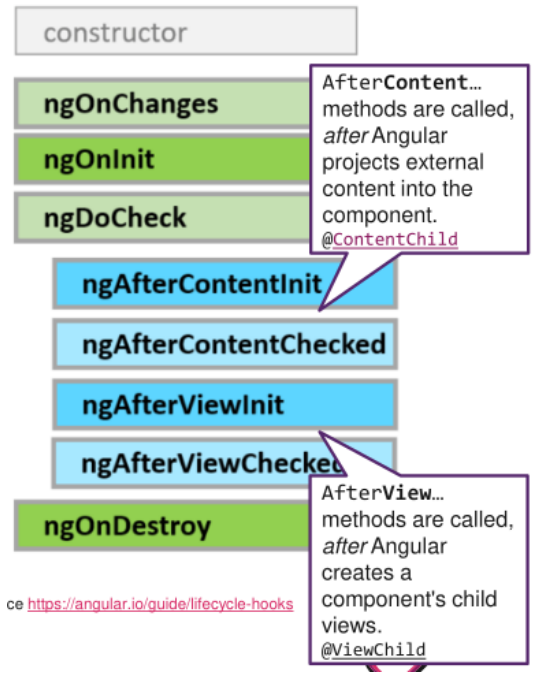

Figure 6: Component Lifecycle

Asynchronous Services Do not use RxJS in the UI. It is sufficient to use `EventEmitter`

RxJS

Routing

`.forRoot()` use EXACTLY once to declare routes on root (top) level

`.forChild()` use when declaring sub-routings (on all sub-levels)

Module Type / Example Architecture

Root / App Module provies the entry point for the app

Feature Modules clear boundaries between the application features

Shared Module Holds common components / directives / pipes

Core Module Keeps your Root Module clean

Feature Module (sub) Types

Domain Modules deliver a UI dedicated to a particular application domain

Routing Modules specifies the routing specific configuration settings of the Feature (or Root) Module

Service Modules provides utility services such as data access and messaging

Widget Modules makes components, directives, and pipes available to external modules

Lazy Modules (Routed Modules) represents lazily loaded Feature Modules.

RxJS - Observable Types

Hot Observables Sequence of events (such as mouse move events or stock tickers)

Cold Observables Observables start running on subscription (such as async web requests) and complete automatically

Flux Architecture The state in Redux is represented as trees of objects. The tree is immutable. When you change something in the tree, a new tree will be created (Functional Programming).

A state action is communicated using a *Redux Action*. A *Reducer* takes the action and the current state an applies the action on the state to generate a new state.

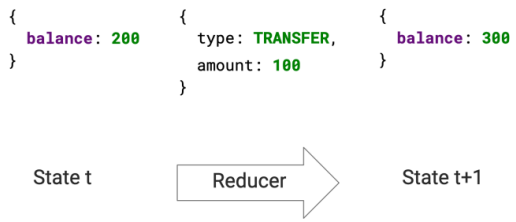

Figure 7: Action Reducer State change

Pipes

Pure Pipes Pure pipes are executed when it detects a 'pure change' to the input expression (DatePipe)

Impure Pipes Impure pipes are executed on every component change detection cycle (AsyncPipe)

4 ASP.NET

PWA

- PWA

For what is Firbase used? To develop a PWA we used the following features from Firebase:

- Cloud Firestore / Realtime (DB)
- Cloud Functions (Serverless Functions)
- Authentication
- Hosting / Cloud Storage

Front Controller The Front Controller is the Entry Point and executes logging and routing (actions that are required for all incoming requiest). After that, the front controller forwards the request the the responsible controller.

MVVM

- MVVM

Middleware

source `https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0`

Middleware is a "function block" and has a certain task. When a middleware has finished its task it will call the next following middleware or finishes the request.
Using `app.Run()` a terminating middleware is registerd. After this middleware the request is finished.

```
app.Run(async (context) => {
    await context.Response.WriteAsync("Hello");

    app.Run();
});
```

Context contains all information for the request as well for the response. Using `app.Use(<lambda>)` a middleware is registred, that calls the next middleware (does not terminate)

```
app.Use(async (context, next) => {
    await context.Response.WriteAsync("Hallo");
    next.Invoke();
});
```

```
app.Use(async (context, next) => {
    await context.Response.WriteAsync(" Welt");
    next.Invoke();
});

app.Run();
```

Extension Method as middleware

```
// Extension Method als Middleware
app.UseMyMiddlware();


app.Run();


static class MyMiddlwares
{
    public static IApplicationBuilder UseMyMiddlware(this WebApplication app)
    {
        return app.Use(async (context, next) =>
        {
            await context.Response.WriteAsync(", Hello from Extension Method");
            await next.Invoke();
        });
    }
}
```

Class as middleware. Is often used when the middlware becomes larger than 2-3 lines of code.

```
app.UseMiddleware<MyClassMiddleware>();

app.Run();

class MyClassMiddleware
{
    private readonly RequestDelegate _next;
    public MyClassMiddleware(RequestDelegate nex
    {
        if (next == null)
            throw new ArgumentNullException(name

        this._next = next;
    }

    public async Task InvokeAsync(HttpContext co
    {
        await context.Response.WriteAsync(", Hel
        await this._next.Invoke(context);
    }
}
```

Middleware Registration

- app.Use()
- app.Map()
- app.Run()
- app.UseMiddleware<>()

Dependency Injection - Lifetime Register services with different lifetimes:

- Transient, are created every time when used
- Scoped, are created once for every request (used most often)
- Singelton, it's just a singelton (only for Readonly, configs, options)

Register a transient service: `builder.Services.AddTrans`

Captive dependency The term "captive dependency" refers to the misconfiguration of service lifetimes, where a longer-lived service holds a shorter-lived service captive.

- Transient requires Singelton -> OK

- Singelton requires Transient -> Not OK
  - When the singleton is used a 2nd time (e.g. parallel) it uses the same reference to the transient service.
  - Transient services are normally **NOT** thread-safe and an error can occuer when used in a not thread safe context

Entity Framework

- Entity Framework

Web Assembly File Type

- *.WASM (WebAssembly)
  - Compiled Web Assembly
  - Can be transformed back to WAT
- *.WAT
  - Text Based Web Assembly
  - Can be compiled to WASM
- Web Assembly
  - Is a stack machine (similar to the JVM)
  - Evaluation Stack

Key Concepts

- Module
  - Represents a WebAssembly binary that has been compiled by the browser into executable machine code
  - Stateless
  - Import / Export
- Memory
  - shared memory section between JS and web assembly
- Table
  - A resizable typed array of references (e.g. to functions) that could not otherwise be stored as raw bytes in Memory (for safety and portability reasons).
- Instance
  - A Module paired with all the state it uses at runtime including a Memory, Table, and set of imported values.