10. Angular + IONIC

Dr. Bilicki Vilmos
Szegedi Tudományegyetem
Informatikai Tanszékcsoporth
Szoftverfejlesztés Tanszék
Classic Web Application Model

- Most user actions in the interface trigger an HTTP request back to a web server.
- The server does some processing and then returns an HTML page to the client. Processing includes
  - retrieving data
  - crunching numbers
  - talking to various legacy systems.
- This model is adapted from the Web’s original use as a hypertext medium.
- *But what makes the Web good for hypertext doesn’t necessarily make it good for software applications.*
What is the Problem?

- This approach makes a lot of technical sense.
- *But* it doesn’t make for a great user experience.
  - While the server is doing its thing, what’s the user doing?
    - Waiting.
    - And at every step in a task, the user waits some more.
Defining AJAX

- Ajax isn’t a single technology.
- It’s really several independent technologies coming together in new ways.

Ajax incorporates:

- standards based presentation using XHTML – HTML5 and CSS
- dynamic display and interaction using the Document Object Model
- data interchange and manipulation using XML and XSLT - JSON
- asynchronous data retrieval using XMLHttpRequest
- JavaScript binding everything together
Classic Model v AJAX

Classic web application model

- User interface
- JavaScript call
- HTTP request
- HTML+CSS data
- Ajax engine
- XML data

Ajax web application model

- User interface
- JavaScript call
- HTTP request
- HTML+CSS data
- Ajax engine
- XML data

HTTP request

Server-side systems

- Databases, backend processing, legacy systems
Classic Model v Ajax

**Classic Model**
- Client
  - User activity
  - Data transmission
  - System processing
- Time
  - Data transmission
- Server
  - Data transmission
  - System processing

**Ajax Model**
- Client
  - User activity
  - Browser UI
  - Ajax Engine
  - Client-side processing
- Time
  - Data transmission
  - Server-side processing
- Server
  - Data transmission
  - Server-side processing
DOM: What is it?

- An object-based, language-neutral API for XML and HTML documents
  - Allows programs and scripts to build, access, and modify documents
  - Supports designing of querying, filtering, transformation, formatting etc. applications on top of DOM implementations
- Instead of “Serial Access XML” could think as “Directly Obtainable in Memory”
The DOM tree
DOM structure model

Based on O-O concepts:

- **objects** (encapsulation of data and methods)
- **methods** (to access or change object’s state)
- **interfaces** (declaration of a set of methods)

Somewhat similar to the XPath data model (to be discussed with XSLT and XQuery) syntax-tree

- Tree structure implied by abstract relationships defined by the API; Data structures of an implementation may differ
Core Interfaces: Node & its variants

- **Node**
  - **Document**
  - **DocumentFragment**
  - **Element**
  - **Attr**
  - **CharacterData**
    - **Comment**
    - **Text**
    - **CDATASection**
  - **DocumentType**
  - **Notation**
  - **Entity**
  - **EntityReference**
  - **ProcessingInstruction**

“Extended interfaces”
< invoice form="00" type="estimated" >
   < addressdata >
      < name > John Doe </ name >
      < address >
         < streetaddress > Pyynpolku 1 
         < / streetaddress >
         < postoffice > 70460 KUOPIO 
         < / postoffice >
      < / address >
   < / addressdata >
   ...
Rich Internet Application

- Based on web browser plugins
- The goal is to mimic the desktop application
Responsive Web Application

- Web based but it is running on mobile phone
- Goal: should be used in mobile browser
- Browser based (JavaScript+HTML+CSS)
- 2001: audi.com
- 2010 significant need
Responsive Web Application

- Fluid grid based layout
  - Relative
  - Screen sizes (sx, md, lg, …)
- Flexible images (adapt to the size of the screen)
- Media queries
- Technology:
  - HTML5
  - AJAX
  - CSS (SASS, …)
Hybrid mobile application

- Mobil application is the goal
- Platform independent
- Web software stack
  - HTML5
  - AJAX
  - CSS
- Runtime
  - Cordova
Hybrid mobile application

- Unique elements
  - GUI
  - Native UI
- Build chain
- WebView as runtime
- Support for different peripherals
  - Smart watch, …
Hybrid mobile application
Hybrid mobile application
Progressive Web Application

- Mobile applications with web capabilities

- Capabilities:
  - **Progressive** – Independent of the actual web browser
  - **Responsive** – Independent of the screen size
  - **Connection independent** – With the help of background processes it handles the slow network connections
  - **App like** – It provides the same look and feel
  - **Upt to date** – Background threads
  - **Safe** – HTTPS
  - **Discoverable** – Search engines can find it with the help of W3C descriptors
  - **Motivation** – Push, …
  - **Deployable** – One can put it to the home screen
  - **Linkable** – can be shared with URL
Progressive Web Application

Technology:
- HTML5
- AJAX
- CSS
- Web descriptor (W3C manifest)
- Background threads (worker)
- App Shell
  - Cache API
  - Fetch API
  - IndexeDB API
  - Notification API
  - …
Single Page Application (SPA)
The Challenge with SPAs

- DOM Manipulation
- History
- Module Loading
- Routing
- Caching
- Object Modeling
- Data Binding
- Ajax/Promises
- View Loading
Angular is a full-featured SPA framework.
What is Angular?

- Angular is a framework for developing mobile and desktop client side applications.
- The first version (1.x) was developed in 2009. Google plays an important role in its development.
- The second version (2.x) is already available for developers preview in beta version.
- The second version is kind of a new framework. Angular was rewritten from scratch.
- Apart of the same name there is nearly nothing in common.
The Web Convergence

- The hybrid application model simplifies the convergence of the web and the world of apps.
- The more we move forward with this convergence the more complex the code we write in JavaScript is.
- The angular framework assists us when dealing with complex code in JavaScript.
What is Angular?

- Angular 2 gives you the tools you need to build apps for
  - Desktop,
  - Mobile web,
  - Android, and iOS.
- Angular Universal provides for server-side rendering for fast initial views on mobile web.
- Ionic and NativeScript let you build hybrid and native UI mobile apps. Web worker support keeps your app UI fully responsive no matter how heavy the load.
Angular 2

- As of Angular 2, there are three programming languages we can use.

  - Dart
    - [www.dartlang.org](http://www.dartlang.org)

  - TypeScript
    - [www.typescriptlang.org](http://www.typescriptlang.org)

  - JavaScript
    - [developer.mozilla.org/docs/web/javascript](http://developer.mozilla.org/docs/web/javascript)
    - Can be either ES5 or ES6
Why AngularJS

“Other frameworks deal with HTML’s shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views”.

- Structure, Quality and Organization
- Lightweight ( < 36KB compressed and minified)
- Free
- Separation of concern
- Modularity
- Extensibility & Maintainability
- Reusable Components

“HTML? Build UI Declaratively! CSS? Animations! JavaScript? Use it the plain old way!”
jQuery

- Allows for DOM Manipulation
- Does not provide structure to your code
- Does not allow for two way binding
Hello HTML

<p>Hello World!</p>
Hello Javascript

<p id="greeting1"></p>

<script>
var isIE = document.attachEvent;
var addListener = isIE
    ? function(e, t, fn) {
        e.attachEvent('on' + t, fn);
    }
    : function(e, t, fn) {
        e.addEventListener(t, fn, false);
    };
addListener(document, 'load', function(){
    var greeting = document.getElementById('greeting1');
    if (isIE) {
        greeting.innerText = 'Hello World!';
    } else {
        greeting.textContent = 'Hello World!';
    }
});
</script>
Hello JQuery

<p id="greeting2"></p>

<script>
$(function(){
    $('#greeting2').text('Hello World!');
});
</script>
Hello AngularJS

<p ng:init="greeting = 'Hello World!'">{{greeting}}</p>
Other Javascript MV* Frameworks

- BackboneJS
- EmberJS
Angular approach

- Applications are composed from
  - HTML templates with Angularized-markup
  - component classes to manage templates
  - services implementing the application logic
  - Angular's bootstrapper top root component

- Angular takes over, presenting our application content in a browser and responding to user interactions according to the instructions we provided.
Angular Approach
Angular architecture
Hello World

```html
<!DOCTYPE html>
<html>
<head>
  <script src="https://code.angularjs.org/2.0.0-alpha.26/angular2.sfx.js"></script>
  <script src="main.js"></script>
</head>
<body>
  <my-application></my-application>
</body>
</html>
```

```
3 import { bootstrap } from "angular2/platform/browser";
4 import { Component } from "angular2/core";

5 @Component(
6  { selector: 'hello-world',
7    template: `Hello world
8    <div>
9    <div>
10   class HelloWorld {
11     }
12  `)
13 })
14 bootstrap(HelloWorld);
```
Module

- Angular apps are composed of modules.
- Modules export things — classes, function, values — that other modules import.
- We prefer to write our application as a collection of modules, each module exporting one thing.
- A typical module is a cohesive block of code dedicated to a single purpose. A module exports something of value in that code, typically one thing such as a class.
Example

- The import statement tells the system it can get an AppComponent from a module named app.component located in a neighboring file.

- The **module name** (AKA module id) is often the same as the filename without its extension.

```typescript
app/app.component.ts (excerpt)
export class AppComponent {}

app/main.ts (excerpt)
import {AppComponent} from './app.component';
```
Library Modules

- Some modules are libraries of other modules.
- Angular itself ships as a collection of library modules called "barrels". Each Angular library is actually a public façade over several logically related private modules.
- The angular2/core library is the primary Angular library module from which we get most of what we need.
- There are other important Angular library modules too such as angular2/common, angular2/router, and angular2/http.
The Component

- **A Component** controls a patch of screen real estate that we could call a *view*.
- We define a Component's application logic - what it does to support the view - inside a class.
- The class interacts with the view through an API of properties and methods.
- Angular creates, updates, and destroys components as the user moves through the application. The developer can take action at each moment in this lifecycle through optional Lifecycle Hooks.
Example

```typescript
app/hero-list.component.ts

1. export class HeroListComponent implements OnInit {
2.   constructor(private _service: HeroService){ }
3.   
4.   heroes:Hero[];
5.   selectedHero: Hero;
6.   
7.   ngOnInit(){
8.     this.heroes = this._service.getHeroes();
9.   }
10. 
11.   selectHero(hero: Hero) { this.selectedHero = hero; }
12. }
```
The Template

- We define a Component's view with its companion **template**. A template is a form of HTML that tells Angular how to render the Component.

- A template looks like regular HTML much of the time ... and then it gets a bit strange.

- We can mix ... and will mix ... our custom components with native HTML in the same layouts.

- And in this manner we can and will compose complex component trees to build out our richly featured application.
Example

app/hero-list.component.html

1. `<h2>Hero List</h2>`
2. 
3. `<p><i>Pick a hero from the list</i></p>`
4. `<div *ngFor="#hero of heroes" (click)="selectHero(hero)">`
5. `{hero.name}`
6. `</div>`
7. 
8. `<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>`
Angular Metadata

- Metadata tells Angular how to process a class.
- The easy way to attach metadata in TypeScript is with a decorator.
- A decorator is a function. Decorators often have a configuration parameter.
- The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.
Example

app/hero-list.component.ts (metadata)

1. @Component({
2.   selector: 'hero-list',
3.   templateUrl: 'app/hero-list.component.html',
4.   directives: [HeroDetailComponent],
5.   providers: [HeroService]
6. })
7. export class HeroesComponent { ... }

Template

Metadata

Component

{ }
The `@Component`

- selector - a css selector that tells Angular to create and insert an instance of this component where it finds a `<hero-list>` tag in parent HTML. If the template of the application shell (a Component) contained
- templateUrl - the address of this component's template
- directives - an array of the Components or Directives that this template requires.
- providers - an array of dependency injection providers for services that the component requires.
- The `@Component` function takes the configuration object and turns it into metadata that it attaches to the component class definition. Angular discovers this metadata at runtime and thus knows how to do "the right thing".
- The template, metadata, and component together describe the view.
- We apply other metadata decorators in a similar fashion to guide Angular behavior. The `@Injectable`, `@Input`, `@Output`, `@RouterConfig` are a few of the more popular decorators
Data Binding

- Without a framework, we would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates.
- Writing such push/pull logic by hand is tedious, error-prone and a nightmare to read as the experienced jQuery programmer can attest.
- Angular supports **data binding**, a mechanism for coordinating parts of a template with parts of a component. We add binding markup to the template HTML to tell Angular how to connect both sides.
- There are four forms of data binding syntax. Each form has a direction - to the DOM, from the DOM, or in both directions - as indicated by the arrows in the diagram.
Data Binding

- Interpolation: With interpolation, we put the property name in the view template, enclosed in double curly braces: {{myHero}}.

- Property binding: We write a template property binding when we want to set a property of a view element to the value of a template expression.

  ```html
  <img [src] = "heroImageUrl">
  ```

- Event binding: User input triggers DOM events. We listen to those events with event bindings that funnel updated values back into our components and models.

  ```html
  <button (click)="onClickMe()">Click me!</button>
  ```

- Two-way data binding: combines property and event binding in a single notation using the ngModel directive. In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

  ```html
  <input [(ngModel)]="hero.name">
  ```
Data Binding

- Angular processes *all* data bindings once per JavaScript event cycle, depth-first from the root of the application component tree.
The Directive

- Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by a *directive*.

- A directive is a class with directive metadata. In TypeScript we'd apply the `@Directive` decorator to attach metadata to the class.

- There are two *other* kinds of directives as well that we call "structural" and "attribute" directives.
  - **Structural** directives alter layout by adding, removing, and replacing elements in DOM.
  - **Attribute** directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.
The Service

- "Service" is a broad category encompassing any value, function or feature that our application needs.
- Almost anything can be a service.
- A service is typically a class with a narrow, well-defined purpose.
- Angular itself has no definition of a service. There is no ServiceBase class.
- Yet services are fundamental to any Angular application.
Component vs Service

- Components depend upon services to handle most chores.
  - They don't fetch data from the server
  - They don't validate user input,
  - They don't log directly to the console.
  - They delegate such tasks to services.

- A component's job is to enable the user experience and nothing more.

- It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a "model").

- A good component presents properties and methods for data binding.

- It delegates everything non-trivial to services.
Example

```typescript
export class Logger {
    log(msg: any) { console.log(msg); }
    error(msg: any) { console.error(msg); }
    warn(msg: any) { console.warn(msg); }
}
```
Dependency Injection

- "Dependency Injection" is a way to supply a new instance of a class with the fully-formed dependencies it requires.
- Most dependencies are services.
- Angular uses dependency injection to provide new components with the services they need.
Dependency Injection

- When Angular creates a component, it first asks an **Injector** for the services that the component requires.

- An Injector maintains a container of service instances that it has previously created.

- If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular.

- When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.
Dependency Injection: provider

- A provider is something that can create or return a service, typically the service class itself.
- We can register providers at any level of the application component tree. We often do so at the root when we bootstrap the application so that the same instance of a service is available everywhere.
Example

```typescript
app/hero-list.component (constructor)

constructor(private _service: HeroService){ }
```

```typescript
app/main.ts (excerpt)

bootstrap(AppComponent, [BackendService, HeroService, Logger]);
```

```typescript
app/hero-list.component.ts (excerpt)

@Component({
    providers: [HeroService]
})
export class HeroesComponent { ...
```
Basic directives

- **NgIf**
- **NgSwitch**
- **NgStyle**
- **NgClass**
- **NgFor**

```html
14 .bordered {
15 border: 1px dashed black;
16 background-color: #eee;
17 }
7 <div [ngClass]="{bordered: false}">This is never bordered</div>
8 <div [ngClass]="{bordered: true}">This is always bordered</div>

<table class="ui celled table">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="#p of item.people">
      <td>{{ p.name }}</td>
      <td>{{ p.age }}</td>
    </tr>
  </tbody>
</table>
```
Change Detection

- The basic task of change detection is to take the internal state of a program and make it somehow visible to the user interface.

- Application state change could be caused by:
  - **Events** - click, submit, …
  - **XHR** - Fetching data from a remote server
  - **Timers** - setTimeout(), setInterval()
Change Detection

- Based on the NgZones onTurnDone event
- In Angular 2, each component has its own change detector
- A change detector tree is built. This tree can also be viewed as a directed graph where data always flows from top to bottom.
- With immutable objects we can fine tune the change detection
Component Router

- With the Component Router service, users can navigate a multi-screen application in a familiar web browsing style using URLs.
- Using routing we maintain separation between the different parts of our app, easily maintain the state of our app and protect parts of the app based on specific rules.

```javascript
@RouteConfig( [

    {path: '/', name:'root', redirectTo ['/Home'],
    {path: '/about', name:'About', component: AboutComponent, ...

] )
```
Routing

- We are on client side
  - One URL should be enough
  - but
    - We can not refresh the application and keep the state
    - We can not save the URL
    - We can not share the URL

- We can divide the application to separate areas
- We use separate URLs for these areas
Routing

- Solutions with inner link

```html
1 <!-- ... lots of page content here ... -->
2 <a name="about"> <h1> About </h1> </a>

http://something/#about
```

- # Hash based routing

```
http://something/#/about
```

- HTML5 – based history and URL manipulation
  - `pushState`
Routing

- Component dependent paths

- Elements
  - RouteConfig
    - path, name, component/redirectTo
  - RouterOutlet
    - Hova szeretnénk az oldalt kihelyezni
  - RouterLink
    - Hova menjen (oldal betöltés nélkül)
Routing

- Events

- Handling the protected pages

```typescript
16 @CanActivate(
17 (nextInstr: any, currInstr: any) => {
18   let injector: any =
19     Injector.resolveAndCreate([AuthService]);
20   let authService: AuthService =
21     injector.get(AuthService);
22   return authService.isLoggedIn();
23 })
```
Events

- The DOM raises events. So can components and services.
- Angular offers mechanisms for publishing and subscribing to events including an implementation of the RxJS Observable proposal.
- Observable type represents one of the fundamental protocols for processing asynchronous streams of data.
- It is particularly effective at modeling streams of data which originate from the environment and are pushed into the application, such as user interface events.
Forms

- A form creates a cohesive, effective, and compelling data entry experience.

- An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.

- It provides the following tools:
  - Controls (ControlGroups) encapsulate the inputs in our forms and give us objects to work with them
  - Validators give us the ability to validate inputs, any way we’d like
  - Observers let us watch our form for changes and respond accordingly
Examples

```javascript
let personInfo = new ControlGroup({
  firstName: new Control("Nate"),
  lastName: new Control("Murray"),
  zip: new Control("90210")
})
```

```typescript
export class DemoFormSkuBuilder {
  myForm: ControlGroup;

  constructor(fb: FormBuilder) {
    this.myForm = fb.group({
      'sku': ['ABC123']
    });
  }

  onSubmit(value: string): void {
    console.log('you submitted value: ', value);
  }
}
```
HTTP

- Angular comes with its own HTTP library which we can use to call out to external APIs.
- When we make calls to an external server, we want our user to continue to be able to interact with the page.
- That is, we don’t want our page to freeze until the HTTP request returns from the external server.
- To achieve this effect, our HTTP requests are asynchronous.
- It is based on Observable pattern
Example

getRandomQuote() {
    this.http.get('http://localhost:3001/api/random-quote')
        .map(res => res.text())
        .subscribe(
            data => this.randomQuote = data,
            err => this.logError(err),
            () => console.log('Random Quote Complete')
        );
}

logError(err) {
    console.error('There was an error: ' + err);
}
Hybrid Apps with Angular & Ionic Framework

- Motivation:
  - Native apps:
    - Platform specific
    - Respective development tools
    - Best performance
    - Time consuming
    - Expensive development
Hybrid approach

- Platform independent (iOS, Android etc.)
- HTML5, CSS3 & JS
- Limited performance
- Quick development
- Direct access to native APIs with Cordova
Architecture

1. Web View
2. Cordova
3. Ionic
Architecture - Web View

- UI component
- There can be any number of them
- Any size including full screen
- Displays a single web page
  - Which can be loaded remotely or locally
Architecture - Web View - A Non-Hybrid Example

Twitter’s Web View
Architecture - Web View - Implications

- Two Implications:
  - Modern Hybrid Apps are not hacks
  - Hybrid Apps are not necessarily slow
Architecture - Apache Cordova

- The simplest hybrid app is a full screen web view
- But the OS communication is platform dependant
- So are the build processes
- This is where Cordova comes in
Architecture - Apache Cordova - Overview

- Originally an Open Sourced version of Phone Gap
- Now managed by the Apache Software Foundation
- Supports Android, IOS, Windows Phone, ...
- Phone Gap now is built on Cordova
Architecture - Apache Cordova - As Middleware
Architecture - Apache Cordova - Details

- Cordova provides 2 platform independent interfaces:
  a. Using device functionality
  b. Building to multiple platforms
Architecture - Apache Cordova - Device Interface

- Contains a set of plugins for native OSs
- Web View accessible via global JS objects
- There is a plugin storage website at https://cordova.apache.org/plugins/
- For example: File Plugin
Apache Cordova

### Platforms
- Amazon Fire OS
- Android
- Bada
- Blackberry
- FirefoxOS
- iOS
- Mac OS X
- Qt
- Tizen
- Ubuntu
- WebOS
- Windows (desktop)
- Windows Phone 7
- Windows Phone 8
- Browser

### Plugins
- Battery Status
- Camera
- Contacts
- Device
- Device Orientation
- Dialogs
- File Transfer
- Geolocation
- Globalization
- In-App Browser
- Media Capture
- Network Information
- Splashscreen
- Statusbar
- Vibration
Architecture - Apache Cordova - Build Interface

- Just wraps device APKs
  - So you still have to download each APK
- Installed as a command line NPM package
- This package is used globally when building applications
Architecture - Apache Cordova - End Result

Just a blank screen
Architecture - Ionic - What it Gives You

1. A UI Framework
2. Tools for the Development Process (optional)
Architecture - Ionic - Piecing it All Together

1. Cross-Platform Builds
2. How The UI Runs
Architecture - Ionic - 1. Cross-Platform Build Tools
IONIC + CORDOVA

- Ionic CLI
- Cordova
  - pl. --livereload
  - iOS SDK
  - Android SDK
  - Windows 10 SDK
Architecture - Ionic - 2. How The UI Runs
Ionic - What it Gives You

- Cross-Platform Build Tools
- A UI Framework
- Tools for the Development Process (optional)
Ionic - UI Framework - MVC
Ionic - UI Framework - In addition to MVC

- Directives: An HTML element or attribute with custom functionality. Can use full MVC
- Services: Wrappers over JavaScript object that use and provide Dependency Injection
- Router: Maps app urls (routes) to controllers - view sets
Ionic - UI Framework - Example
Ionic VS Angular

- Ionic wraps the Angular framework architecture with:
  - A Set of UI components
  - Mobile Specific Optimizations
Ionic VS Angular - 1. A Set of UI Components

All optional

Ionic Components (CSS Styles)

Ionic Components (Directives & Services)
Ionic VS Angular - 2. Mobile Specific Optimizations

1. Directive native optimizations
2. Built in practices to help with mobile
3. Crosswalk
Ionic - Tools - Ionic View

Ionic Servers

My apps

Barkpark
ID: J3983534
SIZE: 32MB
MODIFIED: FEB 11 2014

Weather tracker
ID: J3983534
SIZE: 32MB
MODIFIED: FEB 11 2014

Snapcat
ID: J3983534
SIZE: 32MB
MODIFIED: FEB 11 2014

Slappy Bird
ID: J3983534
SIZE: 32MB
MODIFIED: FEB 11 2014
Navigation simplicity

```javascript
pushSettings() {
  this.nav.push(Settings);
}

goBack() {
  this.nav.pop();
}

resetHome() {
  this.nav.setRoot(Home);
}
```
OS Adaptive design

- **Design modes**
  - iOS
  - Android
  - Windows Phone - uses Android styles
  - Other platforms - uses iOS

```html
<ion-list [attr.no-lines]="isMD ? ' ' : null">
```

OS Adaptive design

- SASS styles splitted based on the platform
  - app.core.scss - Global styles
  - app.ios.scss - iOS specific styles
  - app.md.scss - Android specific styles
  - app.variable.scss - Full list
OS Adaptive design

- Ionicons based on the platform
- Android icons following the Material Design Guide
- iOS styled icons