2023

ÁLLAMI PUBLIKUS FRONT-END ÉS MOBIL ÚTMUTATÓ

KÖVETELMÉNYEK ÉS AJÁNLÁSOK HONLAPOK MEGÚJÍTÁSÁHOZ,
MODERN MOBIL- ÉS WEBALKALMAZÁSOK FEJLESZTÉSÉHEZ

Tartalom

Előszó

Az útmutató célja

Az útmutató célja, hogy követelmények, ajánlások megfogalmazásával biztosítsa az új és megújuló webes felületek, mobil alkalmazások jó minőségben történő elkészítését a kor szellemének megfelelő, modern technológiák felhasználásával.

A leírtak a webalkalmazásokra (Web Apps), a progresszív webalkalmazásokra (PWA) és a mobil alkalmazásokra egyaránt vonatkoznak.

A követelmények, ajánlások kizárólag publikus, portál felületekre vonatkoznak. A belső használatú, jellemzően ERP, CMS vagy admin felületekre legfeljebb csak ajánlásként értendőek.

A modern front-end

A front-end fejlesztés az idők során egyre összetettebbé vált. A maradéktalan felhasználói élmény biztosítása érdekében, és a kor szellemének megfelelően mára már magában foglalja a dinamikus, interaktív, és reszponzív felhasználói felületek teljesítményoptimalizált működtetését.

A front-end definíciója

„A webalkalmazás azon része, amelyet a felhasználó közvetlenül lát és használ.”

- Oxford definition of front-end.

A front-end egy webalkalmazás felhasználói felületét jelenti a kliens oldalon, amely felelős a vizuális elemek, a tartalom megjelenítéséért, továbbá a felhasználói interakciók kezeléséért, valamint az abból következő vizuális frissítésekért, a háttérrendszerrel (back-end) való adatkommunikációért. Ezeket a HTML, CSS, és JavaScript technológiák használatával valósítja meg.

A dedikált front-end szakterület szerepe

A webes technológiák sokat fejlődtek és kifinomultabbak lettek. Világosabbá váltak a fejlesztés front-end (kliensoldali), és back-end (szerveroldali) szempontjaival kapcsolatos igények és elvárások.

A front-end a webalkalmazás felhasználói felületének létrehozásával, a felhasználói élmény (UX/UI) biztosításával foglalkozik, beleértve a vizuális elemek elrendezését, animálását és az interaktivitást. Ehhez speciális HTML, CSS és front-end specifikus programozási nyelvek ismeretére, a felhasználói viselkedés, és a tervezési elvek megértésére, valamint azok hatékony alkalmazására van szükség.

Egy projekt dedikált front-end (vagy a releváns kompetenciákkal rendelkező full-stack*) csapata biztosítani képes, hogy az alkalmazás vizuális, funkcionális, biztonsági aspektusai megkapják a megérdemelt figyelmet, ami a felhasználók számára jobb és innovatívabb webes élményhez vezet.

* Front- és back-end kompetenciával egyaránt rendelkező fejlesztő vagy csapat

Architektúra

Multi Page Application (MPA)

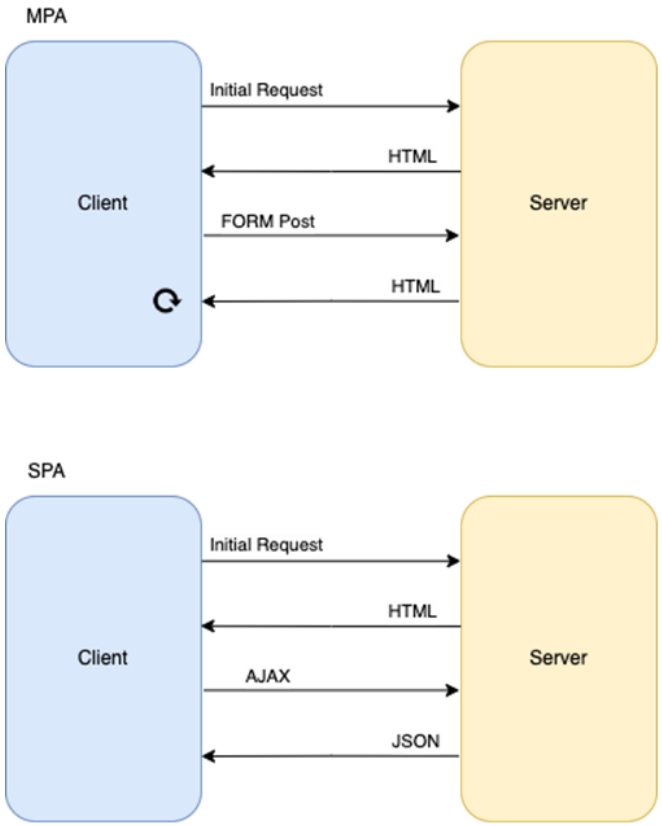
Az internet térhódítását követő időkben Multi Page Application-ök (MPA) készültek. Ezek a hagyományos webalkalmazások a webhely minden oldalához külön HTML-t töltenek be. Nem képesek a modern megközelítéssel már lehetséges magas szintű felhasználói élményt nyújtani. Fejlesztésükhöz nincs szükség a front-end keretrendszerek, és a JavaScript mélyreható megértésére.

2000-es évek elején minden weboldal HTML tartalma teljes mértékben a szerver oldalon volt renderelve és a fejlesztők szerver oldali scripting nyelvekre (pl. PHP, ASP) voltak utalva, hogy ezeket a tartalmakat le tudják generálni. Ebben az időben a JavaScript kevesebbet volt használva, tipikusan csak DOM elemek manipulálására.

Ezeket az alkalmazásokat Multi Page Application-nek nevezik, hiszen az egyes oldalnavigációk között teljes oldalbetöltés történik. Az adatok nem dinamikusan, a kliens oldalon kerülnek renderelésre.

Publikus oldalakat tilos MPA architektúrával fejleszteni!

Az MPA működése nem korszerű. Az oldalon belüli navigáció lassú, nem felel meg a jelenlegi piaci sztenderdeknek.



MPA vs SPA (kliens-szerver kommunikáció áttekintése)

Single Page Application (SPA)

A Single page application (SPA) technológia megjelenése új távlatokat nyitott az alkalmazások kliens oldali lehetőségeiben.

2006-ban az AJAX térhódításával párhuzamosan átalakultak a weboldalak és elterjedtek a SPA-k. Ezek abban különböznek elődjeiktől, hogy a fejlesztők dinamikusan tudnak adatokat lekérni a szerverről és a kapott adat alapján tudják renderelni az oldal egyes elemeit anélkül, hogy azt újra le kellene tölteni.

A SPA-k legnagyobb előnyei az MPA-kal szemben, hogy navigációknál, felhasználói interakcióknál nem kell teljes oldalbetöltést végezni, hiszen a kliens oldali alkalmazás manipulálja a DOM-ot.

Az SPA architektúra lehetővé teszi, hogy a webes alkalmazások reszponzivitása és felhasználói élménye a natív alkalmazásokéhoz közeli léptékű legyen.

Az architektúra lényege, hogy az oldal működéséhez nélkülözhetetlen böngésző által értelmezett utasítás-kód az oldal betöltésével letöltésre kerül, így a – „költségesnek" minősülő – a webalkalmazás teljesítményét hátrányosan érintő kliens-szerver kérések száma és a végpontokról érkező válaszok adatmennyisége is csökken.

Navigáció során a teljes oldal újratöltése nélkül csak az oldal releváns részei frissülnek dinamikusan. Ez zökkenőmentesebb, magasabb szintű felhasználói élményt, valamint jobb teljesítményt és rövidebb betöltési időt eredményez, mint a MPA esetében.

Headless architektúra

A headless architektúra egy olyan webalkalmazás architektúra, ahol a front-end és a back-end külön-külön alkalmazásokként működnek és API segítségével kommunikálnak egymással. A front-end csak megjeleníti az adatokat, amelyeket a back-end biztosít, így a front-end könnyen cserélhető, míg az adatmodell és a back-end megvalósítás megmarad. Ezzel a megközelítéssel a fejlesztők külön kezelhetik a front-end és a back-end részeket, lehetővé téve az adatkezelés és a felhasználói felület fejlesztésének szétválasztását.

A headless architektúrának számos előnye van. Ezek közül a legfontosabbak:

Skalázhatóság: a front-end és a back-end külön alkalmazásként működik, így a két rész könnyen skálázható, ami javítja a rendszer teljesítményét.

Szabadság: a headless architektúra lehetővé teszi a fejlesztőknek, hogy a legjobb eszközöket használják mind a front-end, mind a back-end fejlesztéséhez anélkül, hogy korlátozva lennének egy adott technológia használatában.

Fejlesztési agilitás: a két alkalmazás külön fejleszthető, ami növeli a fejlesztési agilitást és gyorsabb fejlesztési ciklusokat tesz lehetővé.

Egyszerűbb karbantartás: a headless architektúra lehetővé teszi, hogy a front-end és a back-end külön legyen karbantartva, ami csökkenti a karbantartás komplexitását és növeli a rendszer fenntarthatóságát.

Több platform támogatása: egy back-end ki tud szolgálni több front-end alkalmazást is. (Például: webalkalmazás, PWA, natív mobil alkalmazás stb.)

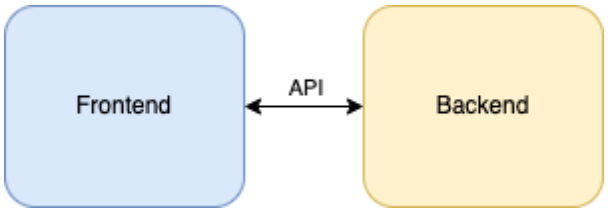
Architektúrális követelmények

A rendszert kötelezően headless architektúrális szemlélettel kell kialakítani!

A tartalmi és a megjelenítési réteg kódjának elkülönítésére a lehető legnagyobb mértékben törekedni kell. A front-end és a back-end alkalmazás közötti kommunikációt API rétegen keresztül kell megvalósítani a megjelenítési rétegtől elválasztva.

Az API réteg használata kifejezetten indokolt esettől (pl. ritkán változó statikus elemek) eltekintve kötelező.

A megjelenítési rétegnek csak a megjelenítési és felhasználói oldal működési logikáját kell tartalmaznia, a tartalmi adatokat az API rétegtől kell lekérni.



Renderelési minták

SPA-k renderelésére számos technika alkalmazható.

Client Side Rendering (CSR)

Kliens oldali renderedés fő ismérve, hogy a szerverről az oldal JavaScript kódján és a stylesheet-eken kívül csak egy HTML shell-t kap. Minden renderelést a kliens oldali JavaScript kód végez.

Előnyök

- szerver minimális terhelése
- a kód telepítés előtt, build folyamat során előre optimalizálható és akár egy statikus fájlmegosztással is lehet üzemeltetni
- könnyen cache-elhető

Hátrányok

- az oldal betöltődési folyamata során kezelni kell az egyes állapotokat, különben szélsőséges esetben a felhasználó csak egy fehér képernyőt lát ezalatt
- nem minden keresőmotor tudja indexelni a JavaScript által generált tartalmat

Ajánlott alkalmazási terület

Belső dashboard-, és kezelőfelületek, ahol nem szükséges a keresőmotorok általi indexelés.

Server-Side Rendering (SSR)

A SPA applikáció a teljeskörű keresőoptimalizálás biztosítása érdekében kiegészíthető SSR-el. Ilyenkor a szerver egy kész HTML tartalmat biztosít a kliens számára az első oldallekéréskor. Ezt a tartalmat a böngésző azonnal meg tudja jeleníteni, a keresőmotorok által működtetett robotok pedig azonnal fel tudják térképezni. A további navigáció már kliens oldalon, dinamikusan történik. Az SSR segítségével a SPA előnyeit megtartva tudunk a keresőmotorok számára könnyen fogyasztható tartalmat előállítani, így a kereső találati listájában előkelőbb helyezést elérni.

Ajánlott SSR technológiák: Angular Universal, Next.js.

Előnyök

- a keresőmotorok könnyen tudják indexelni az oldalt
- a felhasználó nem egy üres képernyőt lát betöltéskor, hanem egy tartalommal teli oldalt
- könnyen beállítható közösségi oldalakon való megosztáskor különböző kép és leírás
- a szerver átveszi a renderelési terhelést, így teljesítménynövekedés érhető el
- növeli a keresőmotoroknál a helyezést és az oldal teljesítményutatóját a lényegesen gyorsabb betöltődés miatt

Hátrányok

- nagyobb szerver oldali terhelés
- nem lehet statikus fájlmegosztással üzemeltetni

Ajánlott alkalmazási terület

Olyan publikus weboldalaknál érdemes használni, ahol a tartalom rendszeresen változik.

Static Site Generation (SSG) / Prerendering

Az SSG eredményeként statikus HTML fájlok jönnek létre a webalkalmazás összes oldalához, amelyek könnyen tárolhatóak különféle, klienseket kiszolgáló platformokon.

Az SSG lényege, hogy az alkalmazás még a kiszolgálóra való telepítés előtt a build folyamat során HTML, CSS, JS, egyéb statikus forrásfájlokra generálódik le a használt útvonal struktúrának megfelelően. Ezek után az alkalmazást egyszerűen statikus fájlmegosztással lehet üzemeltetni.

Alkalmazható SSG technológiák: Angular Universal, Next.js.

Előnyök

- statikus fájlmegosztással üzemeltethető
- könnyen cache-elhető
- keresőmotorok könnyen tudják indexelni az oldalt
- közösségi oldalakon való megosztáskor oldalanként különböző OG adatok beállíthatók
- költséghatékonyan üzemeltethető
- a generálás akár CI/CD folyamatból is indítható automatikusan
- A kiszolgált tartalom előre optimalizált

Hátrányok

- a tartalom változása esetén újra kell generálni az oldalakat

Ajánlott alkalmazási terület

Olyan publikus oldalaknál érdemes használni, ahol a tartalom nem, vagy nagyon ritkán változik.

Angular vagy React

Az aktuális iparági front-end standard az Angular és a React. Ez a két eszköz megbízható megoldás Single Page Application (SPA) webalkalmazások készítéséhez.

Forrás: https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe

Mindkettőt előszeretettel használja több nagyvállalat, és kis startup egyaránt. Jól használható dokumentációval, nagy, és aktív közösséggel rendelkeznek. Mögöttük egy-egy tech-óriás áll: a Google (Angular) és META/Facebook (React).

Applikáció fejlesztéséhez kizárólag Angular, React, vagy ezek valamelyikén alapuló technológia választható!

Styling

Cascading Style Sheets (CSS)

A CSS lehetővé teszi, hogy a fejlesztők elkülönítsék egy weboldal megjelenítési stílusát a tartalomtól, így az könnyebben karbantartható és frissíthető. Használatával szabályozható egy weboldal elemeinek elrendezése, betűtípusa, színe, az egyes elemek különböző vizuális paramétere.

A komplex weblapok kialakítása nagy méretű CSS fájlokat eredményez. A növekvő komplexitásra válaszul elterjedt a különböző CSS preprocesszorok használata.

> Összetett webalkalmazások fejlesztéséhez CSS helyett az SCSS használata ajánlott.

SCSS

Az SCSS támogatja változók használatát, a selector-ok egymásba ágyazását, a kód duplikáció nélküli újrahasznosíthatóságát mixinek által. Bővítve a CSS által elérhető funkcionalitást, az SCSS segítségével elérhető az importálás és szelektor öröklődés. Használatával komplex és karbantartható CSS írható.

Az SCSS funkciói

Variables

SCSS Sass

```scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

CSS

```css
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}
```

A többször használt értékeket külön CSS vagy SCSS változóban kell tárolni!

SCSS változókban olyan színeket, betűkészletet vagy más CSS-értéket szokás tárolni, amelyet több helyen is fel szeretnénk használni a kódban.

Az oldalra vonatkozó font family-t globálisan adjuk meg! Ha egy komponens elemére vonatkozóan másik font-ra van szükségünk, akkor ott kivételként szerepeljen!

A globálisan használt változókat külön SCSS fájlba kell kiszervezni és onnan használatba venni.

A többször felhasznált változókat ajánlott kategória szerint csoportosítva fájlokba rendezni és importálni őket.

Forrás: https://sass-lang.com/documentation/variables

Nesting

SCSS Sass

```scss
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

CSS

```css
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

Összetett selector-ok esetén az egy sorban történő megadás helyett bontsuk azt logikai alkotóelemekre (pl. a DOM felépítés vagy komponensek alapján). Ezeket hierarchikus módon hozzuk létre, hogy az a lehető legjobban tükrözze a szükséges komponensek egymásba ágyazottságát.

Kerüljük el a túlzott mélységű egybeágyazást, valamint a szükségtelen mennyiségű osztály és selector-összetettség használatát!

Maradjon könnyen olvasható az SCSS! A túlzott egybeágyazás átláthatatlanná teszi a kódot, illetve nehezebb rajta módosítást végrehajtani.

Forrás: https://sass-lang.com/documentation/style-rules/declarations#nesting

Partials

Lehetőségünk van részleges SCSS-fájlokat létrehozni, amelyek CSS-részleteket tartalmaznak. Ezek más SCSS-fájlokba is beilleszthetőek. Ez egy egyszerű módja a CSS modularizálásának, és megkönnyíti a kód karbantartását.

A logikailag összetartozó (akár globális) CSS kódrészleteket érdemes külön fájlokba tárolni.

Forrás: https://sass-lang.com/guide#topic-4

Modulok

SCSS Sass

```scss
// _base.scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

```scss
// styles.scss
@use 'base';

.inverse {
  background-color: base.$primary-color;
  color: white;
}
```

CSS

```css
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```

A külön kigyűjtött kódrészleteket (lásd Partials) a `@use` paranccsal tudjuk felhasználni.

Forrás: https://sass-lang.com/guide#topic-5

Mixinek

SCSS Sass

```scss
@mixin theme($theme: DarkGray) {
  background: $theme;
  box-shadow: 0 0 1px rgba($theme, .25);
  color: #fff;
}

.info {
  @include theme;
}
.alert {
  @include theme($theme: DarkRed);
}
.success {
  @include theme($theme: DarkGreen);
}
```

CSS

```css
.info {
  background: DarkGray;
  box-shadow: 0 0 1px rgba(169, 169, 169, 0.25);
  color: #fff;
}

.alert {
  background: DarkRed;
  box-shadow: 0 0 1px rgba(139, 0, 0, 0.25);
  color: #fff;
}

.success {
  background: DarkGreen;
  box-shadow: 0 0 1px rgba(0, 100, 0, 0.25);
  color: #fff;
}
```

A mixinek segítségével hozzunk létre olyan CSS-deklaráció csoportokat, amelyeket újra felhasználunk a weboldalon.

> Használjunk mixin-t, amennyiben szeretnénk újra felhasználható paraméterezett CSS szabályt alkalmazni.

Forrás: https://sass-lang.com/guide#topic-6

Extend/Inheritance

SCSS Sass ⇒ CSS

```scss
/* This CSS will print because %message-shared is extended. */
%message-shared {
  border: 1px solid #ccc;
  padding: 10px;
  color: #333;
}

// This CSS won't print because %equal-heights is never extended.
%equal-heights {
  display: flex;
  flex-wrap: wrap;
}

.message {
  @extend %message-shared;
}

.success {
  @extend %message-shared;
  border-color: green;
}

.error {
  @extend %message-shared;
  border-color: red;
}

.warning {
  @extend %message-shared;
  border-color: yellow;
}
```

A kód duplikáció elkerülése végett az `@extend` paranccsal örököltessük más CSS class-ok tulajdonságait.

Forrás: https://sass-lang.com/guide#topic-7

Operátorok

SCSS Sass

```scss
@use "sass:math";

.container {
  display: flex;
}

article[role="main"] {
  width: math.div(600px, 960px) * 100%;
}

aside[role="complementary"] {
  width: math.div(300px, 960px) * 100%;
  margin-left: auto;
}
```

CSS

```css
.container {
  display: flex;
}

article[role="main"] {
  width: 62.5%;
}

aside[role="complementary"] {
  width: 31.25%;
  margin-left: auto;
}
```

Használjunk SCSS operátorokat, pl: `+, -, *, math.div(), %`

Forrás:
https://sass-lang.com/guide#topic-8
https://sass-lang.com/guide
https://www.geeksforgeeks.org/what-is-the-difference-between-css-and-scss
https://blog.logrocket.com/the-definitive-guide-to-scss
https://www.upwork.com/resources/what-is-scss

React és Styling

A CSS/SCSS mellett a React-el használható styling megoldások közé tartoznak a CSS modulok, illetve a CSS-in-JS könyvtárak használata.

CSS-in-JS

A legnépszerűbb CSS-in-JS könyvtárak: a Styled-components, az Emotion, a Linaria és a Glamorous. Ezek közül a projekt egyedi igényeitől és korlátaitól függ, hogy melyik CSS-in-JS könyvtár a legjobb választás.

Közös jellemzőjük, hogy a stílusokat JavaScript-ben írhatjuk és kezelhetjük, illetve automatikusan a komponensekhez rendelhetjük őket. Ezáltal a stílusok meghatározása a formázandó komponenshez közel történik. Ezen felül pedig a prop-ok és state-ek alapján dinamikus styling-ra van lehetőség.

A felsorolt könyvtárak támogatják a szerveroldali-renderelést is, továbbá mindegyik használható más CSS-in-JS könyvtárakkal és eszközökkel (például: Material UI).

Forrás
https://styled-components.com
https://emotion.sh
https://linaria.dev
https://glamorous.rocks
https://mui.com

TypeScript

A TypeScript az Angular alkalmazások fejlesztésének elsődleges nyelve, használata React esetében is kötelező!

> A TypeScript használata kötelező!

A TypeScript egy Microsoft által kifejlesztett, nyílt forráskódú programnyelv, amely JavaScript-re épül – a JavaScript szuperszettje, és statikus típusrendszerrel javítja a fejlesztési folyamat biztonságát és hatékonyságát, valamint kibővíti a nyelvet abban (még) nem létező fejlesztést és hibafelderítést segítő szerkezetekkel, jelölésmódokkal. Minden érvényes JavaScript kód egyben érvényes TypeScript kód is.

Mivel kompatibilis a JavaScripttel, lehetővé teszi a fejlesztők számára, hogy lépésről lépésre migrálják a kódjukat TypeScript-re anélkül, hogy teljesen át kellene írni azt.

A TypeScript a JavaScript statikus típusait használja, így a fordító képes a hibákat jelezni a fejlesztés során, mielőtt az alkalmazás futna, illetve éles környezetbe kerülne. Ezáltal javul a kódminőség, és elkerülhetőek a futásidőben felmerülő hibák. A TypeScript típus annotációi pedig könnyebbé teszik a kód megértését és a dokumentáció-készítést is.

Mivel a TypeScript alapvetően objektumorientált, de lehetőséget biztosít funkcionális koncepciók megvalósítására is, ezért segít a komplex alkalmazások és azok egyes részeinek (pl. a UI komponensek) könnyebb átláthatóságában és karbantarthatóságában.

Főbb jellemzői, és előnyei

Erős típus-ellenőrzés: Típusok adhatók meg a változókhoz és a függvényekhez, ami segíti a hibák kiszűrését és javítja a kód minőségét és karbantarthatóságát.

> Erősen ajánlott kerülni az `any` típus használatát. Ahol ez nem lehetséges, ott inkább az `unknown` típust szükséges előnyben részesíteni.

Jelölt típusok: Jelölt típusokat kínál a JavaScript objektumokhoz, ami segít a kód olvashatóságában és áttekinthetőségében.

Fejlesztőeszköz-támogatottság: Kiváló támogatással rendelkezik a modern fejlesztői eszközökben, (Visual Studio Code, WebStorm, stb.), ezáltal megkönnyítve a kód írását, hibakeresését és karbantartását.

Interfészek: A fejlesztők interfészeket hozhatnak létre, melyek lehetővé teszik, hogy a közös kommunikációs felület szabályai (típusmegjelölései, paraméterei, stb.) mentén több, egymással kompatibilis, szükség szerint csereszabatos megvalósítás is létrejöhessen biztosítva azok statikus típusellenőrzését is.

Támogatja a **klasszikus objektum-orientált** programozási funkciók használatát, így például az öröklődést, absztrakciót, interfészek és implementációk szétválaszott alkalmazását, generikusokat.

Könnyen áttekinthető kód: A TypeScript segítségével a fejlesztők könnyebben áttekinthetik a kódjukat, mert a típusok és a jelölések a kód részét képzik.

Elérhetőek az ES6 és ES7 funkcionalitások mielőtt azok megkapnák a széleskörű támogatást a főbb böngészőktől.

A JavaScript kódra történő fordítás során megadható a böngésző támogatottság, így alacsonyabb beépített támogatással rendelkező böngészők számára is futtathatóvá válnak az egyébként ott nem támogatott nyelvi szerkezetek, tulajdonságok, programozási módszerek. Támogatja a legtöbb új JavaScript funkcionalitást, beleértve a destructing-et, arrow function-öket és template literálokat.

Beépített language service-szel rendelkezik, ami támogatást nyújt kód kiegészítéshez, hibajelzéshez, figyelmeztetésekhez és automatikus javításokhoz (IntelliSense) a legelterjedtebb fejlesztői eszközökön.

Nagy, és növekvő fejlesztői közösséggel rendelkezik, akik hozzájárulnak a fejlesztéséhez, és használják projektjeikben.

Forrás

https://www.typescriptlang.org - weboldal

https://github.com/microsoft/TypeScript - forráskód

https://www.typescriptlang.org - a TypeScript, és előnyei részletesebben

Angular

Bevezetés

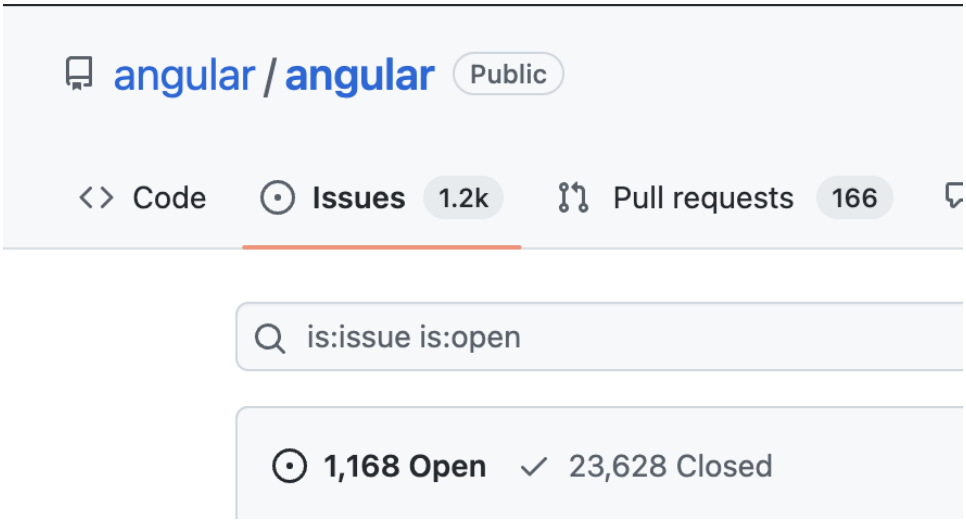
Az Angular egy modern, „*Open Source*” front-end keretrendszer, mely mögött az egyik legnagyobb tech-óriás: a Google áll. A karbantarthatóságot, a tesztelhetőséget, és a stabilitást tekintve is kiállta a próbát és bizonyított. Alkalmas és ajánlott keretrendszer nagyvállalati alkalmazásra. Használatával összetett webalkalmazások fejleszthetők.

Támogatás

Az Angular népszerűsége és támogatottsága 2010 óta folyamatosan nő.

A fejlesztési platformot több tízezer fejlesztő hasznosítja nap mint nap, ebből adódóan a kód és mögötte rejlő irányelvek fejlődése folyamatos. Több, mint 23.000 lezárt thread [hibajegy] (2023. januárig) jegyzi a fejlesztői közösségben rejlő erőt.

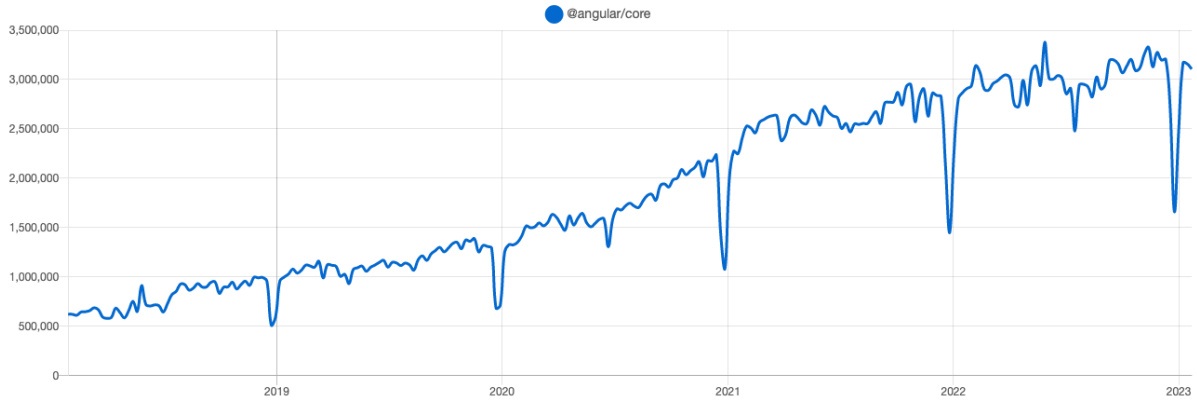
Fontos megemlíteni, hogy az Angular mögött álló nemzetközi szintű közösség számos helyi szervezettel rendelkezik.



Forrás
https://angular.io/resources?category=community
https://github.com/angular/angular/issues

Népszerűség

Az Angular növekvő népszerűségét jelzi az *@angular/core* npm csomag letöltéseinek száma:



Forrás: https://npmtrends.com/@angular/core

Esettanulmányok

Az Angular számos jól ismert webalkalmazás alapja, mint pl. Microsoft Office, Deutsche Bank, Mixer, Santander, Gmail, Forbes, UpWork, PayPal, Grasshopper, Samsung, Delta, Overleaf

Paypal

A Paypal napjainkig az egyik legnagyobb és leggyakrabban használt elektronikus fizetéseket bonyolító szolgáltató világszerte. 2021-ben több, mint 392 millió aktív felhasználóval rendelkezett. A tech óriás az Angular segítségével építette ki pénztárrendszerét, amely olyan kulcsfontosságú funkciókat tartalmaz, mint a fizetési áttekintési oldal és a hitelkártya oldalak kezelése. E szolgáltatásokkal szemben támasztott elsődleges követelmények között szerepelt a magas szintű biztonság és a zavartalan funkcionalitás.

Microsoft Office

A Microsoft teljes mértékben felhasználta az Angular keretrendszert az Office 365 API formájában, melyben szolgáltatásait egyesíti. Az Office 365 lehetővé teszi vállalkozások és a felhasználók számára, hogy csapatokban együttműködve dokumentumokat hozzanak létre.

Upwork

Az Upwork a világ egyik legnagyobb szabadúszókat foglalkoztató piaci platformja, több mint ötmillió ügyféllel, százmillió dollár nagyságrendű bevétellel és több milliárd dollár bruttó szolgáltatási volumennel*. Az Upwork platform is Angular keretrendszert használ.
*2020 körüli adat

A verziók életciklusa (Release Cycle)

A Google csapata 6 havonta kiad egy új Angular verziót, amelyet a kiadástól számított 6 hónapig továbbra is aktívan fejleszt, majd további 12 hónapig támogat, vagyis frissítésekkel lát el. Így az összes major verzió 18 hónapig támogatott. Ez 6 hónap aktív támogatást (frissítések és javítások) és 12 hónap long-term support (a továbbiakban LTS) támogatást tartalmaz (kritikus javítások és biztonsági javítások). Az újonnan azonosított biztonsági rések javításai belekerülnek az LTS-verziókba minor frissítésként, így fenntartva a stabil, és biztonságos működést.

Elavult technológiák kivezetése

A verziók előre ütemezett életciklusa által a Google ad magának lehetőséget az Angular-ba beépített, illetve az Angular által támogatott technológiák felülvizsgálatára 6 hónapos ciklusokban. Ha egy korábban hasznosnak bizonyult technológia felett eljárt az idő (megszűnt, vagy megszűnni látszik a támogatottsága, népszerűsége), esetleg utólag rossz döntésnek bizonyult, akkor azt általában első lépésként deprecated [elavult] státuszba teszik. Ilyenkor már nem ajánlják a további

használatát, majd egy későbbi major verzió megjelenésével teljes egészében kivezetik.

Ígéretes technológiák bevezetése

A Google csapatának az elavult technológiák kivezetése mellett a verziók kezelése által lehetősége van ígéretes technológiák bevezetésére 6 havonta. A technológiák bevezetésének tervéről és várható időpontjáról a hivatalos oldalukon, valamint a roadmap-jükben adnak egyfajta előre tekintést.

A verziókról bővebben olvasható a hivatalos dokumentáció „Verziók és Kiadások” fejezete: https://angular.io/guide/releases#angular-versioning-and-releases

Licenc

Az Angular fejlesztése az MIT Licenc égisze alatt zajlik, ami lehetőséget biztosít a keretrendszer kódjának korlátozások nélküli, díjmentes felhasználására és értékesítésére.

Forrás: https://angular.io/license

Performancia és SEO

Technológiai evolúció

A 2010-es években számos keretrendszer próbált választ adni az erőforrás-költséges DOM manipulációra, valamint a komplex adatstruktúrák konzisztens, átlátható összekapcsolására kliens és szerver között.

Az Angular az elmúlt időszakban sikerrel adott választ a biztonságos, gyors és böngésző-barát webes alkalmazások hatékony fejlesztésére.

DOM manipuláció

A REST hívások mellett a performanciát érintő szűk keresztmetszet a DOM manipuláció. Megvalósítása szintén „költséges” lenne modern webes keretrendszer használata nélkül.

Angular alkalmazás esetén az oldal dinamikus működéséhez az Angular egy absztrakciós réteget, egy ún. „View” objektumot használ. A „View” strukturális és adattag manipulációit a keretrendszer által biztosított különböző interfészek valósítják meg.

SSR (Szerver oldali renderelés)

A platform által biztosított Angular Universal lehetőséget ad az Angular-al fejlesztett weboldalak SEO és performancia optimalizációjához. A technológiáról és a különböző renderelési módokról részletesebben az alábbi címeken lehet olvasni:

Server-side rendering (SSR) with Angular Universal: https://angular.io/guide/universal
Rendering on the Web: https://web.dev/rendering-on-the-web

CLI

Az Angular Command Line Interface (Angular CLI) használható többek között a projekt build folyamatának kezelésére. Az Angular CLI további funkcionalitásokkal is bír, rajta keresztül a keretrendszer teljes vezérlése megvalósítható: új projekt létrehozása, komponensek vagy service-ek gyártása, build folyamat kezelése, tesztek futtatása stb.

Tiszta kód (clean code)

A fejlesztők implementációs megoldásait egységesítendő ajánlásokat az Angular „coding style guide tartalmazza”, mely a https://angular.io/guide/styleguide címen nyilvánosan elérhető. Az elvek követése segíti a kód áttekinthetőségét, olvashatóságát, karbantarthatóságát, valamint könnyíti a hibák javítását.

Az irányelvek kiterjednek többek között:

- kódszervezésre (pl. egy komponens - egy fájl)
  https://angular.io/guide/styleguide#small-functions
- a komponensekhez tartozó fájlszerkezeteket illető konvenciókra
- függvények hosszára
  https://angular.io/guide/styleguide#small-functions
- fájlok következetes elnevezésére
  https://angular.io/guide/styleguide#small-functions
- a kódon belül a komponens azonosítására használt „ *selector”*-ok elnevezésére
  https://angular.io/guide/styleguide#component-selectors
- Direktívák, Pipe-ok, Unit tesztek elnevezésére
- Modulok elnevezésére
  https://angular.io/guide/styleguide#component-selectors
- Applikáció szerkezetére
- LIFT (Locate, Identify, Flat, T-DRY) elvre
  https://angular.io/guide/styleguide#lift

Példa a megfelelő fájl- és mappastruktúrára

Forrás: https://angular.io/guide/styleguide#overall-structural-guidelines

React

Bevezetés

A React egy olyan könnyen elsajátítható JavaScript könyvtár, amelyet a META (korábban Facebook) fejlesztett ki a web- és mobil alkalmazásokat készítő fejlesztők számára.

Az egyik legnépszerűbb front-end fejlesztési könyvtár, amelyet sok nagyvállalat és startup használ (Facebook/Meta, Netflix, Uber, Yahoo!, Airbnb, Atlassian, Dropbox, The New York Times, Instagram, Discord, Walmart, Skype, Pinterest).

A React segítségével könnyen létrehozhatóak bonyolult SPA, illetve dinamikus és interaktív felhasználói felületek, valamint a virtuális DOM technológiának köszönhetően optimalizálható az alkalmazások teljesítménye.

Forrás
https://reactjs.org
https://reactjs.org/community/support.html

Alapelvei és architektúrája

A komponens-alapú fejlesztésnél a felhasználói felületet kisebb, önálló egységekre, azaz komponensekre bontjuk, amelyek önállóan működnek, és együtt dolgoznak a többi komponenssel egy egységes alkalmazás megvalósításához.

A virtuális DOM lehetővé teszi a gyors frissítéseket és optimalizálja a renderelést, mivel csökkenti a DOM-mal történő közvetlen műveletek számát.

Alapelvei között szerepel az immutabilitás és az egyirányú adatfolyam, ami azt jelenti, hogy az alkalmazás állapota csak egy irányban változhat, és a változásokat kizárólag a React kezeli.

A React-ben két fő komponens-típus létezik: osztály komponensek (class) és funkcionális (function) komponensek. Az osztály komponensek ES6 osztályok, a funkcionális komponensek pedig függvények. A funkcionális komponensek

egyetlen megkötése, hogy argumentumként elfogadják a prop-okat, és érvényes JSX-et adnak vissza.

A `useState` és `useEffect` hook-ok lehetővé teszik, hogy kiváltsuk az osztály komponensekben használt `componentDidMount`, `compnentDidUpdate`, `componentWillUnmount` és `componentWillReceiveProps` metódusokat. Ezáltal kezelhetjük a komponensek állapotát és effektusait anélkül, hogy új komponens osztályokat kellene létrehozni, így leegyszerűsítik a kódot és javítják az áttekinthetőséget.

Funkcionális komponensek és Hook-ok

A React-et funkcionális komponensekkel ajánlott használni, mert könnyen olvashatók, jobb teljesítményt nyújtanak, kevesebb kódból állnak, könnyen tesztelhetők, debug-olhatók és javíthatók, újra felhasználhatóak és csökkenthetik a szoros egymás közötti függőségeket (coupling) és könnyen karbantarthatóak.

> Ajánlott funkcionális komponenseket használni a React beépített hook-jaival.

Mindenképpen ajánlott elkülöníteni a logikát a megjelenítéstől, ha szükséges akár saját egyéni hook-ok létrehozásával. A logika újra-felhasználásához szintén célszerű egyéni hook-okat létrehozni a magasabb rendű komponensek (Higher-Order Components, HOC) helyett - amelyek igazából ugyanazt a célt szolgálják.

> Ajánlott egyéni hook-ok létrehozása
> Higher-Order Component-ek helyett.

Csak akkor használjunk osztály komponenseket, ha erre jó okunk van (például a Error Boundaries csak osztály komponensként érhető el).

Forrás: https://reactjs.org/docs/error-boundaries.html

A verziók életciklusa (Release Cycle)

A React fejlesztőcsapata rendszeresen megjelenő új verziókkal biztosítja a hibák javítását, a teljesítmény fokozását és a friss funkciók megjelenését. A React szemantikus verziózási (semver) elveket követ.

A React csapata a korábbi verziókhoz long-term support-ot (LTS) biztosít, így a projektcsapatoknak van idejük frissíteni alkalmazásaikat a legújabb verzióra.

A React kiadási ciklusát úgy tervezték, hogy a hibajavítások, az új funkciók és fejlesztések rendszeres, kiszámítható megjelenését a meglévő alkalmazásokra gyakorolt negatív hatások minimalizálásával biztosítsa. Továbbá garantálja a zökkenőmentes frissítési folyamatot.

Forrás: https://reactjs.org/docs/faq-versioning.html

A React csapata a szemantikus verziókezelést követi, így a főverzió-váltásokat (major release) kiemelt alapossággal és körültekintéssel szükséges kezelni.

Breaking Changes

A Breaking Changes beépítése mindenki számára kényelmetlen, ezért a React csapata igyekszik minimalizálni a Major Release-ek számát. Az erre való törekvést jól mutatja, hogy a React 15 2016 áprilisában, a React 16 2017 szeptemberében, a React 17 2020 októberében, a React 18 pedig 2022 márciusában jelent meg.

Elköteleződés a stabilitás mellett

A React csapata elkötelezett a stabilitás mellett. Igyekeznek minimalizálni az új funkciók bevezetésével járó erőfeszítéseket a React használói számára. Például az elévült, használatra már nem javasolt kódrészeket is bent tartják a frissebb kiadásokban, hogy az azokra korábban épített legacy kódbázisok probléma nélkül fussanak egy verziófrissítés után is.

Forrás: https://reactjs.org/docs/faq-versioning.html#commitment-to-stability

Licenc

A React MIT licenccel elérhető, így az korlátozás nélkül, díjmentesen használható és tovább értékesíthető.

Forrás: https://reactjs.org/docs/how-to-contribute.html#license

Next.js

React webalkalmazások fejlesztéséhez többféle keretrendszer áll rendelkezésre. Ezek közül az egyik legismertebb, egyúttal a React fejlesztői csapatával legszorosabban együttműködő és legdinamikusabban fejlődő keretrendszer a Next.js.

> Ajánlott a Next.js keretrendszert választani a React mellé.

A Next.js legfontosabb funkciói, előnyei:

- Könnyű beállítás
- Automatikus kódoptimalizálás és fájlrendszerezés
- Szerveroldali renderelés (Server-side rendering - SSR) támogatás
- Könnyen használható, automatikus útvonalkezelő
- API végpontok biztonságos összekapcsolása
- Többnyelvűség támogatása
- Beépített teljesítmény-optimalizálás, kép-, font- és script-optimalizáció
- Fejlett UX és gyors betöltési idő
- Könnyen skálázható architektúra
- Könnyen használható fejlesztői eszközök és dokumentáció
- Integráció a különböző fejlesztői eszközökkel, mint például SWC, Webpack, Babel, stb.
- CSS támogatás (SASS/SCSS, PostCSS, CSS-in-JS)
- TypeScript támogatás
- SEO optimalizálás támogatása

A Next.js a legújabb React funkciók kiterjesztésével és hatékony, Rust-alapú JavaScript-eszközök integrálásával képes a leggyorsabb webalkalmazások létrehozására. Legújabb, 13-as verziója lefekteti az alapokat a határok nélküli dinamizmus felé is: app könyvtár támogatás, layout-ok, szerver-komponensek, streaming kezelése, Turbopack (Rust-alapú Webpack alternatíva), stb.

Forrás

https://nextjs.org - weboldal

https://nextjs.org/blog/next-13 - Next 13

https://github.com/vercel/next.js - forráskód

https://nextjs.org/docs/getting-started - dokumentáció

Böngésző támogatás

Modern böngészők

Az evergreen (örökzöld) kifejezés azokra a böngészőkre utal, amelyek induláskor automatikusan frissülnek a legfrissebb verzióra, és erről tájékoztatják a felhasználót, nem pedig a gyártó weboldalán közzétett aktuális verzió manuális letöltésével és telepítésével frissíthetőek, mint régebbi társaik. A modern böngészők így küszöbölik ki, hogy a felhasználók biztonsági réseket, és hibákat tartalmazó, korábbi verziót futtassanak.

Vagyis az evergreen kifejezés a kiadási stratégiára utal: gyakran frissülnek a háttérben, folyamatosan frissítve a webes szabványoknak való megfelelésüket.

A Chrome, Mozilla Firefox, Microsoft Edge, és az Opera evergreen webböngészők – indításkor automatikusan frissülnek.

A Safari nem evergreen böngésző, viszont a macOS gondoskodik a frissítéséről.

100%-os funkcionalitás biztosítása

Az alábbi böngészőkben az alkalmazás 100%-os funkcionalitását kell biztosítani:
Chrome, Mozilla Firefox, Microsoft Edge, Safari.

Vizuálisan - a böngészők különbözősége miatt – megengedett némi eltérés, de ezeket szükséges a lehető legkisebb mértékűre redukálni. Azonban az eltérő platformokon, az azonos böngészőkben egyformán szükséges a tartalomnak megjelennie.

Ha egy vizuális elem megjelenése szerves részét képezi a funkcionalitásnak, akkor annak is 100%-osan kell működnie a fentebb felsorolt böngészőkben.

Internet Explorer

Az Internet Explorer támogatása megszűnt.

Forrás: https://techcommunity.microsoft.com/t5/windows-it-pro-blog/internet-explorer-11-desktop-app-retirement-faq/ba-p/2366549

> Az Internet Explorer 9 - 10 - 11 támogatása nem követelmény.

A biztonsági kockázatok miatt az Internet Explorer használata nem ajánlott. Következésképpen a webalkalmazásoknak sem szükséges támogatniuk.

Polyfill

Polyfill-ek használatával elérhető, hogy a modern böngészőkre optimalizált kódot a régebbi böngészők is tudják értelmezni/futtatni.

A polyfill lényegében az egyes böngésző funkcionalitások, ill. későbbi JavaScript fejlesztések teljes vagy részleges JavaScriptben való implementációi, amik lehetővé teszik az ezekre épülő modern technológiák és kódok használatát feláldozva a natív implementációból fakadó sebességet, hiszen „emulálják" az adott funkciót/viselkedést.

Forrás: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill

További böngészők

Léteznek további, kevésbé elterjedt böngészők, amelyeket érdemes figyelembe venni az alkalmazás fejlesztése során.

> Az alábbi böngészőkben is ajánlott tesztelni az alkalmazást:
> *Opera, Brave, Dolphin, DuckDuckGo*

Reszponzivitás

Általános szabályok

Az alábbiak tartalmazzák azokat a kötelező és ajánlott irányvonalakat, melyeket figyelembe kell venni a reszponzivitás kialakítása során. A fejezet példáiban szereplő felbontások mindegyike Viewport Size-ban értendő.

„Mobile First" szemlélet

Az „Mobile First" szemlélet egy olyan tervezési megközelítés, amely a mobil felhasználói élményt helyezi előtérbe más eszközökkel, például asztali számítógépekkel vagy laptopokkal szemben. Ennek a megközelítésnek az az alapja, hogy az internetes forgalom nagy része (~54%) ma már mobileszközökről történik. Éppen ezért kulcsfontosságú, hogy az optimális mobil felhasználói élmény biztosítva legyen.

A felületeket „mobile first" szemlélettel szükséges megtervezni, és implementálni.

Forrás: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Responsive/Mobile_first

Media Query

A „Media Query" egy CSS-technika, amely lehetővé teszi különböző stílusok alkalmazását a különböző eszközökön és képernyőméreteken. Hatékony eszközt jelentenek rugalmas tervezések létrehozásához, amelyek nagyszerű felhasználói élményt biztosítanak számos eszközön.

Kötelező a Media Query-k használata az eltérő méretű, arányú eszközökre.

Ajánlott előre megírt CSS framework használata. Ajánlott továbbá Flexbox és CSS Grid használata a reszponzivitás kialakítására.

Forrás

https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Container_Queries

https://getbootstrap.com

https://www.w3schools.com/css/css3_flexbox.asp

https://www.w3schools.com/css/css_grid.asp

Megfelelő méretű képek használata

A megfelelő képméret egy fontos szempont, amelyet figyelembe kell venni a reszponzív webalkalmazások létrehozásakor. A nagyméretű képek lelassíthatják az oldal betöltési idejét, ami rossz felhasználói élményhez vezethet, különösen mobileszközökön. Másrészt a kis képek pixelesnek és elmosódottnak tűnhetnek a nagyobb képernyőkön.

A képeket megfelelő, a megjelenítés méretéhez igazított felbontásban kell betölteni!

Lazy Image Loading

A „Lazy Image Loading” egy olyan technika, amelyet a webalkalmazások teljesítményének optimalizálására használnak azáltal, hogy elhalasztják a képek betöltését addig, amíg valóban szükség nem lesz rájuk. Javítják az alkalmazás kezdeti betöltési idejét, hiszen csak azokat a képeket töltik be, amelyek azonnal láthatóak a felhasználó számára. További képek csak akkor kerülnek betöltésre, ha azok a görgetés hatására a látótérbe kerülnek.

A képeknél a lazy loading használata kötelező!

A reszponzivitás tesztelése

Különböző felbontású eszközökön kell tesztelni az alkalmazást!

Ajánlott az alkalmazást nem csak emulátoron, hanem valódi eszközökön is tesztelni.

Az átméretezés lekövetése

A böngésző esetleges átméretezését a megjelenített tartalom minden elemének le kell követnie.

A tartalomnak folyamatos méretezés esetén is mindvégig jól olvashatónak kell maradnia!

A minőség megtartása érdekében fontos egy olyan betűtípust választani, aminek az adott kijelző felbontásához képest megfelelő a betűtípus szélessége és az aktuális kijelző DPI-je.

Például:

- kijelző felbontása: 1920x1080
- font mérete: 16px
- font DPI: 96 DPI

Publikus oldalak

Publikus alkalmazások esetében a UX/UI tervezése során szem előtt kell tartani, hogy minden oldalnak reszponzívnak kell lennie!

A felhasználó nem veszíthet a funkcionalitásból azáltal, hogy mobil eszközre váltott. Minden olyan műveletet, melyet asztali számítógépén el tud végezni, a mobil

eszközén is el kell tudnia végezni, minden információhoz hozzá kell jutnia. Ennek biztosítása során az elrendezés, vagy a megjelenítés módja eltérhet asztali-, illetve mobil eszközön.

A legkisebb szélesség, ahol még hibátlanul kell megjelennie az oldalnak: 360px.

Belső, adminisztratív oldalak

Ajánlott a belső oldalaknál is törekedni a reszponzivitásra. Az ajánlott legkisebb támogatott szélesség: 992px.

Töréspontok

Amennyiben a projekt specifikációja másként nem rendelkezik, az alkalmazás felületének tervezésekor és implementálásakor a következő töréspontokat szükséges használni:

Elnevezés	Töréspont
Extra Small	< 576px
Small	≥576px
Medium	≥768px
Large	≥992px
Extra Large	≥1200px
Extra Extra Large	≥1400px

Containers

Az alkalmazás vagy az oldal 100%-ig kiterjedő elemekből áll, vagy container-ekbe van rendezve. Az utóbbi esetén a következő container szélességeket kell alkalmazni:

Elnevezés	Töréspont	Conainer mérete
Extra Extra Small	< 320 px	100%
Extra Small	≥ 320 px	100%
Small	≥576px	540px
Medium	≥768px	720px
Large	≥992px	960px

Extra Large	≥1200px	1140px
Extra Extra Large	≥1400px	1320px

Kódolási konvenciók

A töréspontok és a container-ek létrehozására ajánlott SCSS mixineket, és változókat használni. Ehhez ajánlott a Bootstrap könyvtár. Példa kód a mixinek kialakításához:

```
$grid-breakpoints: (
  sm: 576,
  md: 768,
  lg: 992,
  xl: 1200,
  xxl: 1400,
);

$container-max-widths: (
  xs: 0px,
  sm: 540px,
  md: 720px,
  lg: 960px,
  xl: 1140px,
  xxl: 1320px,
);

@mixin media-breakpoint-up($name, $breakpoints: $grid-breakpoints) {
  $min: breakpoint-min($name, $breakpoints);

  @if $min {
    @media (min-width: $min) {
      @content;
    }
  } @else {
    @content;
  }
}

@function container-min($name, $breakpoints: $container-max-widths) {
  $min: map-get($breakpoints, $name);
  @return if($min !=0, $min, null);
}

@mixin make-container($max-width: 100%, $padding-x: 20px) {
  width: 100%;
  max-width: $max-width;
  padding: 0 $padding-x;
  margin: 0 auto;
}

.layout-container {
  @include make-container();

    {
```

```
    @include make-container(container-min(sm));
  }

  @include media-breakpoint-up(md) {
    @include make-container(container-min(md));
  }

  @include media-breakpoint-up(lg) {
    @include make-container(container-min(lg));
  }

  @include media-breakpoint-up(xl) {
    @include make-container(container-min(xl));
  }

  @include media-breakpoint-up(xxl) {
    @include make-container(container-min(xxl));
  }
}
```

Forrás: https://sass-lang.com/documentation/at-rules/mixin

Third Party Libraries

Általános definíció

Third party library egy újra felhasználható komponens, kódrészlet, ami külső forrásból származik, másik cég vagy közösségi fejlesztés az eredete. Használatuk nagyban tudja csökkenteni a fejlesztési időt és erőforrást, amennyiben megfelelően alkalmazható a probléma megoldásához.

Általános szabályok

Harmadik féltől származó könyvtárakat használni csak akkor lehet amennyiben:

- A könyvtár még aktív, folyamatosak a frissítések, hibajavítások. Amint egy könyvtár eléri az EOL (end of life) vagy legacy státuszt, azt le kell cserélni, és tilos tovább használni.
- Forkolt verziót használni tilos
- A projektet legalább féléves rendszerességgel frissítik, vagy rendelkezik verzióval a legfrissebb Angular vagy React verzióhoz
- Amennyiben open source könyvtár, a megoldásra alkalmas könyvtárak közül a legmagasabb a letöltésszáma
- A könyvtár kifejezetten a választott framework-re lett fejlesztve
- Rendelkezik elérhető fejlesztői dokumentációval és támogatással
- Rendelkezésre áll a megfelelő licenc
- A könyvtár megbízható helyről lett letöltve

A könyvtáraknál legfeljebb 2 minor verzió eltérés engedélyezett. Amennyiben a könyvtár major verziót vált, frissíteni szükséges miután megbizonyosodtunk róla, hogy biztonságos, amennyiben kompatibilis marad a választott keretrendszer verziójával. Ehhez ajánlott a dependabot használata.

Forrás: https://github.com/dependabot

Általánosan javasolt könyvtárak

Közös komponenskezelés – Storybook

A komponensek kezelésére javasolt a Storybook könyvtár használata. A Storybook MIT licensszel rendelkező, ingyenesen használható könyvtár, mely elszigetelt iframe-et biztosít az összetevők megjelenítéséhez anélkül, hogy az alkalmazás üzleti logikája és kontextusa zavarná. A komponenseket ezután történetekbe lehet rendezni. A Storybook előnyei közé tartozik:

- A komponensek izolálhatóak, és működésük modellezhető
- Elősegíti a szélsőséges esetek előidézését, és tesztelését
- Széleskörű kiegészítőivel akár API kérések és készülék funkciók is modellezhetőek
- Beépített munkafolyamattal rendelkezik az akadálymentesség, interakció és vizuális teszteléshez
- A történetek alapján automatikusan lehet dokumentációt generálni
- Integrálható a CI folyamatok közé

A Storybook legnagyobb előnye, hogy a megírt komponensek verziózhatóak, és később a könyvtárból bárhol, többször felhasználhatóak kódduplikáció nélkül.

Forrás: https://storybook.js.org

WYSIWYG szerkesztő - CKEditor

A CKEditor 5 egy modern, JavaScript alapú rich text szerkesztő MVC architektúrával, egyéni adatmodellel és virtuális DOM-mal. Zöldmezős projektként ES6-ban íródott, és kiváló webpack támogatással rendelkezik. Minden elképzelhető WYSIWYG

szerkesztési megoldást kínál kiterjedt együttműködési támogatással. GPL 2+ licenc alatt használható.

Forrás: https://ckeditor.com

Táblázat kezelés – AG Grid

Az AG Grid egy magas funkcionalitással ellátott, személyre szabható JavaScript adat táblázat. Kiemelkedő teljesítménnyel rendelkezik, nem függ semmilyen harmadik féltől származó könyvártól, és Angular-ral, illetve React-tel is egyszerűen integrálható. Elérhető ingyenes (MIT License), és fizetős változata is.

Egy táblázat alap funkcióin felül az alábbi funkciókkal rendelkezik:

- Csoportosítás / Aggregálás
- Akadálymentesség támogatása
- Egyedi szűrés
- Cellaszerkesztés
- Adatok „lusta” betöltése
- Szerveroldali adatműveletek
- Élő frissítés
- Hierarchikus adattámogatás és fanézet
- Testre szabható megjelenés
- Testre szabható cellatartalom
- Állapot perzisztálás
- Navigáció billentyűzettel
- Adatexportálás CSV-be
- Adatexportálás Excelbe
- Excel-szerű pivot tábla
- Sorok átrendezése
- Másolás és beillesztés
- Cellák egyesítése
- Rögzített sorok
- Teljes szélességű sorok
- Integrált diagram

Forrás: https://www.ag-grid.com

Dátumkezelés – Date FNS

A Date FNS egy JavaScript alapú dátumkezelő könyvtár, TypeScript támogatottsággal. Több mint 200 függvénnyel, és több tucat nyelvvel rendelkezik, moduláris, és natív dátumot használ. Pure és immutable, minden esetben egy új dátumpéldányt ad vissza. MIT licenc alatt elérhető.

Forrás: https://date-fns.org

CSS Keretrendszer – Bootstrap

A Bootstrap az egyik legelterjedtebb CSS. Biztosítja az alapvető szintaxis kollekciót és 12 oszlopos rács rendszert, mellyel gyorsan lehet reszponzív, és mobile first szemléletű alkalmazásokat fejleszteni. MIT licenc alatt elérhető.

Forrás: https://getbootstrap.com

UI Könyvtár – Telerik Kendo

A Kendo UI egy több mint 100 előre készített komponenst felsorakoztató könyvtár. Out of the box megoldásokat biztosít általános, és komplexebb komponensekre. Könnyen személyre szabható, WCAG 2.1-es akadálymentesítéssel, és nyelvesítéssel van ellátva. Natív komponenseket használnak, így elérhető az összes platform specifikus funkció Angular és React esetén is. Jól dokumentált, és üzleti licenc keretén belül teljes támogatottsággal elérhető.

Forrás: https://getbootstrap.com

Kód formázó – Prettier

A Prettier egy olyan kódformázó, mely egységes kódolási stílust biztosít a projekten belül. Minden egyéb stílust eltávolít, és egy előre definiált szabályrendszert felhasználva elemzi a kódot, majd újra kiírja azt, figyelembe véve a maximum sorhosszúságot, és szükség esetén új sorba töri a kódot. Támogatott nyelvek:

- JavaScript
- JSX
- Angular
- Vue
- Flow
- TypeScript
- CSS, Less, and SCSS
- HTML
- Ember/Handlebars
- JSON
- GraphQL
- Markdown, including GFM and MDX
- YAML

Továbbá valamennyi gyakran használt IDE-vel kompatibilis, vagy eleve beépítve érkezik. MTI licenc alatt elérhető.

Forrás: https://prettier.io

Clean Code

Bevezetés

A Clean code (tiszta kód) egy ajánlás, mellyel biztosítjuk a program hosszú élettartamát, garantáljuk a hosszú távú karbantarthatóságát. Ezeket az ajánlásokat követve a kód alapvetően könnyebben tesztelhető lesz, és nagyobb százalékban kiküszöbölhetjük a hibákat már a fejlesztés során, emellett segíti a fejlesztőket egymás kódjának könnyebb megértéséhez, segíti a későbbi újabb fejlesztők bevonását a már futó projektekbe.

SOLID elvek

A SOLID egy mozaikszó, amely az öt tervezési alapelv angol elnevezésének kezdőbetűiből áll össze:

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Célja a kódolás még érthetőbbé és karbantarthatóbbá tétele.

Forrás: https://hu.wikipedia.org/wiki/SOLID

Single Responsibility Principle

Egyetlen felelősség elve. Egy osztály vagy modul egy, és csak egy felelősséggel rendelkezzen (azaz: egy oka legyen a változásra). Ez a gyakorlatban annyit jelent, hogy az osztályunk lehetőség szerint csak egyetlen egy funkciót, feladatot lásson el. Az osztály változtathatóságát redukáljuk a legkisebbre. Legjobb, ha annak állapota csak egyetlen egy helyről (egy oka van) módosítható.

Legegyszerűbb módja a helyes használat ellenőrzésének, ha elmeséljük, hogy mit csinál az osztályunk, és a mesélés közben az ÉS-ek száma 0.

Open/Closed Principle

Nyílt/zárt elv. Egy osztály vagy modul legyen nyílt a kiterjesztésre, de zárt a módosításra. Ez a gyakorlatban azt jelenti, hogy az osztályokat, interfészeket úgy írjuk meg, hogy azok származtatása elegendő legyen a további fejlesztésekre, és a már kész, letesztelt programkódokat ne kelljen újból tesztelni, illetve a hozzá kapcsolódó implementációkat se kelljen átírni (hiszen az eredeti osztályhoz hozzá sem nyúltunk).

Liskov substitution principle

Liskov helyettesítési elv. Minden osztály legyen helyettesíthető a származtatott osztályával anélkül, hogy a program helyes működése megváltozna. Ez a gyakorlatban azt jelenti, hogy az ősosztályban definiált működéstől „nagy mértékben” ne térjünk el. A bemenő paraméterek, visszatérési értékek egyezzenek meg az ősosztályban lévővel/meghatározottal, illetve ne csináljunk egy adott metóduson belül olyan dolgot, amelyet az eredeti osztály nem csinált. Például az ősosztályban egy get metódus helyesen visszaadott egy értéket, akkor a származtatott osztályban - a helyes visszaadás mellet - már mást ne csináljunk, például ne módosítsuk az osztály egy másik változóját, hiszen akkor az eredeti osztályt a mi osztályunkkal helyettesítve már egy teljesen más működést adunk a programunknak.

Interface segregation principle

Interfész elválasztási elv. Az interfészek (kapcsolódási felületek) szétválasztásának elve: egyetlen kliens se legyen rákényszerítve arra, hogy olyan eljárásoktól függjön, amelyeket nem is használ.

Szorosan kapcsolódik az első alapelvhez, hiszen, ha azt követjük, akkor egy interfész meghatározás esetén sem fordul elő, hogy az a kelleténél több feladatot határoz

meg. Gyakorlatban ez azt jelenti, hogy igyekezzünk olyan interfészeket meghatározni, melyek adott feladatok implementálásakor valóban csak a feladat ellátásához szükséges metódusokat tartalmazza.

Dependency inversion principle

Függőség megfordítási elv. A magas szintű modulok ne függjenek az alacsony szintű moduloktól. Mindkettő absztrakcióktól függjön. A gyakorlatban ez azt jelenti, hogy ha az osztályunknak szüksége van egy másik osztályra, akkor ne a konkrét osztálytípust várja függőségként, hanem egy interfészt.

További ajánlások

SOLID elvek mellett rengeteg más ajánlás is létezik, amelyek követhetőek a megfelelő kódolás és karbantarthatóság érdekében.

KISS – Keep It Short, & Simple

Legyen (a kód) rövid, és egyszerű. Szoktak még rá többféleképpen hivatkozni, mint: „Keep It Simple, Stupid", „Keep It Simple and Straightforward". Az osztályok, metódosuk legyenek rövidek, egyszerűek, érthetőek.

YAGNI - You Ain't Gonna Need It

Nem lesz rá szükséged. Ne egy univerzális kódot, osztályt készítsünk, ami mindent tud, amire csak szükség lehet a jövőben, mert általánosságban elmondható, hogy sosem lesz rá szükség. Törekedjünk inkább arra, hogy ha valóban szükség lesz valamilyen funkcionalitásra a jövőben, akkor az beépíthető legyen a kódba annak radikális átírása nélkül.

DRY – Don't Repeat Yourself

Kerüljük a kód másolást. Kifejezetten indokolt esetet kivéve ne szerepeljen többször ugyanaz a kódrészlet a programkódban. Nehezíti a karbantarthatóságot, és

a későbbiekben valószínűsíthetően inkonzisztenciához vezetnek az ilyesfajta megoldások.

Javaslatok, megkötések

Referencia alapú változók

JavaScript-ben minden nem primitív típus objektum, ahol a változó csak a struktúra referenciáját tartalmazza. Ezek módosítását lehetőség szerint immutable módon kell elvégezni. Az eredeti változó értékét ne írjuk felül, hanem egy úgynevezett „deep copy / deep clone" létrehozása után az új változó értékét módosítsuk. Ezzel a módszerrel elkerülhető, hogy inkonzisztens állapotokat hozzunk létre az alkalmazásunkban.

> Lehetőség szerint kerüljük el a referencia alapú változók módosításait.

Kód struktúra

Egy nagy alkalmazásnál elengedhetetlen, hogy a kódunk jól strukturált, szeparált legyen. Kövessük a SOLID elveket, bontsuk az alkalmazást kisebb modulokra a karbantarthatóság érdekében. Építsük úgy a kódunkat, hogy az jól olvasható, átlátható, értelmezhető legyen.

> Törekedjünk jól strukturált kód kialakítására.

Interfészek

Az interfész egy olyan váz, amely meghatározza, hogy az őt implementáló osztálynak milyen elemeket kell kötelezően megvalósítania, konkrét implementációt nem tartalmaz. Ez azt jelenti, hogy az általunk használni kívánt, interfészben leírt osztály/metódus meghívásakor nem releváns az interfész számára, hogy azt az elemet miként valósítja meg az osztály, csak az, hogy megvalósítja azt. Interfészek

használatával a kód szeparálhatóbb, érthetőbb, átláthatóbb, és könnyíti a tesztelést is.

Használjunk interfészeket!

Feliratkozások

Minden feliratkozás (subscription) memóriát használ. Ha ezek a feliratkozások a használatukat követően nem kerülnek lezárásra (leiratkozás), akkor a memóriában ott maradnak, fölöslegesen foglalják a helyet, így idővel a memória túlcsordulásához vezet.

Minden (automatikusan nem záródó) feliratkozást le kell zárni a használatát követően!

Feliratkozáson belüli feliratkozások (nested subscription) szintén memória túlcsorduláshoz vezetnek és emellett nehezen olvashatóak, tesztelhetőek.

Kerüljük a feliratkozások egymásba ágyazását, helyette használjunk RxJs kompozíciót, pl. switchMap!

Template fájlok

A lehető legkevesebb összehasonlítást, logikai műveletet végezzük el ott, ahol a sablont, kinézetet készítjük A template fájlokban lévő üzleti logika műveleteket lehetőség szerint kerüljük el. Nehezítik a UI tesztelését, emellett rengeteg erőforrást elhasznál ezen műveletek állandó kiértékelése.

A template fájlokba ne kerüljön üzleti logika.

„Lazy loading"

Törekedjünk arra, hogy az alkalmazás indításakor csak azok a modulok töltődjenek be, amelyekre abban a pillanatban valóban szükség van! A többi modul a használat során szükség szerint töltődjön be. Ezzel a módszerrel gyorsítjuk az alkalmazás működését, betöltését, és nem terheljük feleslegesen sem a szervert, sem a klienst.

> Törekedjük a „module lazy loading" használatára.

Angular specifikus ajánlások

A konstruktor és az `OnInit`

A konstruktor az osztály példányosításakor lefutó metódus. Az Angular esetében ebben a pillanatban még nem feltétlen áll rendelkezésre az általunk ellátni kívánt feladathoz szükséges összes információ.

Amennyiben szükség van arra, hogy a DOM már rendelkezésre álljon, az input binding-ok meg legyenek, akkor a megfelelő hely az üzleti logikánk kifejtésére a legtöbb esetben az `OnInit` vagy `AfterViewInit` life cycle hook/metódus.

Feliratkozások

Használjuk a `take` és `takeUntil` operátorokat a megfelelő „automatikus" leiratkozások elérésének érdekében. A `take(1)` esetében az adatfolyamnak legalább egyszer kell adatot szolgáltatnia ahhoz, hogy valóban megtörténjen a leiratkozás. Ha nem jön adat, a feliratkozás „örökre" ott marad. Erre figyeljünk!

A template fájlok, és az üzleti logika

Kerüljük a template fájlokban lévő üzleti logikát! Az Angular nagyon sűrűn hasonlítja össze a DOM jelenlegi állapotát a kirajzolni kívánt állapottal. Ez egy nagy erőforrást

igénylő feladat, így a template fájlban kerüljük a metódus hívásokat, összehasonlításokat.

> Használjunk „pure pipe"-okat!

HTTP hívások

Helytelen, ha a komponensben direktben találhatóak HTTP hívások a back-end-es API-ok felé. Mind az újrafelhasználhatóság, mind a nagyobb üzleti logikával kapcsolatos műveletek érdekében a legjobb, ha ezeket egy külön szolgáltatásba (service-be) szervezzük ki.

> Használjunk service-eket a HTTP hívások kezeléséhez!

Shared modulok, standalone komponensek

Ahogy az alkalmazás nő, idővel rengeteg modul található majd benne. A gyakran, több helyről használt funkciókat szervezzük ki Shared modulokba, így az egész alkalmazáson keresztül tudjuk azokat (újra)használni kód duplikálás nélkül.

Amennyiben nem modulban szeretnénk kezelni ezeket az egész alkalmazásra kiterjedő közös funkciókat, akkor van lehetőség standalone komponenseket használni helyette. Így a komponenseket már nem feltétlen szükséges modulba helyezni (elhagyható az `NgModule` deklaráció).

> Lehetőség szerint használjunk shared modulokat, standalone komponenseket.

Strict mód

Az Angular CLI-vel (Command Line Interface) generált alkalmazások alapból strict mód beállítással jönnek létre. Ez javítja a karbantarthatóságot, és segíti a hibák korai

észlelését. Az ilyen alkalmazásokon könyebb elvégezni statikus elemzéseket, emellett a jövőbeni `ng updated`-ek alkalmával is megkönnyíti a munkánkat.

Forrás: https://angular.io/guide/strict-mode

Strict mód használata javasolt!

OnPush change detection

A change detection az Angular-ba épített rutin amely automatikusan szinkronban tartja az adatokat és a felhasználó által láttot vizuális (template) tartalmat. Ezt a műveletet alap beállítás mellett úgynevezett `ChangeDetectionStrategy::Default` beállítással másodpercenként rengetegszer elvégzi, hiszen minden felhasználó által kiváltott eseményre megvizsgálja, hogy szükséges-e az adat, és a kirajzolt tartalom között szinkronizálni - tehát, hogy van–e eltérés a között amit látunk, mint amit látnunk kellene?

OnPush change detection strategy használatakor már nem minden eseményre fog lefutni ez a nagy erőforrást igénylő feladat. Ezzel tudjuk javítani az alkalmazásunk használhatóságát, teljesítményét.

Ilyenkor figyeljünk oda, hogy a komponenst vagy manuálisan kell megjelölnünk, hogy változtatni kell (hiszen az automatizmus egy részét megszüntettük), vagy csak azokat a kiváltó eseményeket használhatjuk amelyekre ez a stratégia figyel.

Ilyen kíváltó esemény lehet:

- olyan esemény történik, amelyre a komponensünk figyel,
- egy async pipe-on keresztül új adat érkezik,
- egy `@Input` dekorátoron keresztül új adat érkezik.

Ne felejtsük, hogy az `@Input` dekorátoron keresztül érkező adat(ok) változását csak új referencia érkezésekor fogja detektálni az Angular, tehát ezért is érdemes immutable objektumokat használni.

Az OnPush change detection használata javasolt.

Reactive Formok

Angularban két lehetőségünk van az űrlapok használatára: Reactive From, vagy Template-Driven Form.

Habár csábító lehet a Template-Driven Formok használata kisebb űrlapok létrehozása esetén, az egyszerűsége miatt mégsem ajánlatos használni. Sérti a programozási alapelveket, hogy a nézet réteg szorosan együtt van az üzleti logikával. Emellett egy bonyolultabb űrlap ellenőrzése már igen nehézkessé válik egy Template-Driven Formban.

A Reactive Formok használata kötelező!

React specifikus ajánlások

Funkcionális komponensek

A funkcionális komponensek az osztály komponensekhez képest könnyeben olvashatóak, jobb teljesítményt nyújtanak, kevesebb kódból állnak. Könnyen tesztelhetők, debug-olhatók, és javíthatók. Újra felhasználhatóak, csökkenthetik a szoros egymás közötti függőségeket (coupling) és könnyen karbantarthatóak.

Csak akkor használjunk osztály komponenseket, ha erre jó okunk van (például, ha bizonyos komponensek csak osztály komponensként érhetőek el).

Ahol csak lehet, használjunk funkcionális komponenseket.

Használjunk Hookokat

A hook-ok lehetővé teszik, hogy a React komponensek életciklus metódusai (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`, `componentWillReceiveProps`) helyett a `useState` és `useEffect` hook-okat használjuk.

A hook-ok segítségével kezelhetjük a funkcionális összetevők állapot- és egyéb jellemzőinek használatát. Megkönnyítik a kód írását és megértését anélkül, hogy új komponens osztályokat kellene létrehozni, így leegyszerűsítik a kódot, és javítják az áttekinthetőséget.

Typescript

A TypeScript a JavaScript statikus típusait használja, amelyek lehetővé teszik a fordító számára, hogy az jelezze a hibákat a fejlesztés során, mielőtt az alkalmazás futna, illetve éles környezetbe kerülne. Ezáltal elkerülhetőek a futásidejű hibák, illetve javul a kódminőség.

> React webalkalmazás készítésekor a TypeScript használata kötelező!

Logika és megjelenítés elkülönítése

Különítsük el a logikát a megjelenítéstől! Hozzunk létre és használjunk saját, egyéni hook-okat!

A logika újra-felhasználásához szintén célszerű egyéni hook-okat létrehozni és lehetőség szerint kerülni kell a magasabb rendű komponensek (Higher-Order Components, HOC) használatát.

> Ajánlott a hook-ok használata a Higher-Order Component-ekkel szemben.

CSS-in-JS

A stílusok és téma létrehozása komoly kihívást jelenthet a nagy projektek fejlesztése során.

A nagyméretű CSS/SCSS-fájlok karbantartása sem egyszerű feladat. Ez az oka annak, hogy megjelent a CSS-in-JS megoldások koncepciója. Ilyen könyvtárak például az: EmotionJS, Styled Components, Linaria, Glamorous.

> A projekt követelményeitől függően ajánlott az EmotionJS, Styled Components, Linaria, Glamorous könyvtárak valamelyikének használata.

Ajánlások

- Minden stílust különítsünk el a komponens kódjától.
- A stílusokat használjuk újra más komponensekben.
- A stílusokat dinamikusan módosítsuk a React komponens state-jének vagy prop-jának függvényében.
- A stílusos komponensek neveit tegyük könnyen olvashatóvá, hogy jelezzék, milyen funkciót látnak el, használjunk PascalCase-t
- A stílusokat dokumentáljuk, hogy könnyen áttekinthetőek legyenek.

Egyéb konvenciók

Minden komponens neve legyen érthető és definiálja megfelelően a komponenst.

A komponens nevének nagy kezdőbetűvel kell kezdődnie és PascalCase formátumot kell használni. Erre azért van szükség, hogy a JSX az alapértelmezett HTML-tagektől eltérően azonosítani és kezelni tudja őket.

Használjunk camelCase-t a komponensek függvényeinek elnevezésére. Például: `handleSubmit()`.

Az egy komponenshez kapcsolódó fájlokat tároljuk egy helyen/könyvtárban. Ez vonatkozik a stílus fájlokra is.

A props objektum átadása helyett használjuk az objektum destructuring-et a komponens prop-ok átadásához. Így nem kell minden alkalmommal a prop objektumra hivatkozni, amikor az adott prop-ot használni szeretnénk:

```
export const Button = ({text}) => {
    return <button>{text}</button>;
};
```

JSX-en belül kerüljük a bonyolult renderelés-logikai szerkezeteket, feltételes osztályneveket, switch-eket és lehetőség szerint csak egyszerű, egy szintű, kompakt logikai szerkezeteket írjunk a JSX kódba (`if/else`, `map`, stb.). Az ennél bonyolultabb szerkezeteket szervezzük ki függvénybe.

Használjuk a `map` függvényt a JSX-en belüli HTML blokkok generálására.

Eszközök

Linter

A linter egy olyan eszköz, amely beolvassa az általunk írt program kódot, hogy olyan problémákat találjon, amelyek hibákhoz vagy inkonzisztenciákhoz vezethetnek. Nyomon követhető vele a TODO, FIXME és egyéb jelölések, ellenőrizhető az egységes kódstílus tartása. Emellett még rengeteg dologra használható.

Forrás: https://eslint.org

Prettier

A prettier eszköz használatával garantálható, hogy az általunk írt kód megfelelően lesz formázva. Legtöbb ma használt IDE (Integrated Development Environment) alapvetően rendelkezik ilyen funkcióval.

Forrás: https://prettier.io

Husky

A Husky Git-hook-ok használatával segít a fejlesztőknek hatékonyabban dolgozni. A beállítása szerinti szakaszokban elindítja a kódellenőrzést, a linter-t, vagy épp unit tesztelést.

Forrás: https://typicode.github.io/husky

A kódellenőrzés intézménye (code review)

A kódellenőrzés a kód módszeres (ki)értékelése. Célja a hibák azonosítása, a kód minőségének javítása és a fejlesztők forráskód megtanulásának, megismerésének segítése.

Miután a fejlesztő befejezte a kódolást egy adott feladaton, a kód áttekintése fontos lépés a folyamatban. Így a munkája véleményezhető, azzal kapcsolatosan megfogalmazhatóak kérések, javaslatok. A talált hibák, logikai problémák, feltárt szélsőséges esetek vagy egyéb problémák azonosítása után a fejlesztő lehetőséget kap azok javítására még azelőtt, hogy munkája bekerülne a fejlesztés fő ágába, vagy a tesztelői csoport elé.

A kódellenőrzési folyamat kidolgozása megalapozza a folyamatos fejlesztést, és megakadályozza, hogy instabil kód kerüljön az ügyfelekhez. A kódellenőrzést a fejlesztőcsapat munkafolyamatának részévé kell tenni, hogy javítsa a kód minőségét, és biztosítsa, hogy minden kódrészletet egy másik csapattag is megvizsgáljon. A kódellenőrzés segíti a fejlesztőket a tapasztaltabb társaiktól való tanulásban.

A fejlesztők más-más háttérrel és tudással rendelkeznek, amelyek befolyásolják kódolási stílusukat. Ha a csapatok szabványos kódolási stílust szeretnének, a kódellenőrzések segítenek mindenkinek betartani ugyanazokat a szabványokat.

A kódellenőrzések magas szintű biztonságot hoznak létre különösen akkor, ha a biztonsági szakemberek célzott felülvizsgálatot végeznek. Az alkalmazások biztonsága a szoftverfejlesztés szerves részét képezi, a kódellenőrzések pedig segítenek a biztonsági problémák észlelésében, javításában.

Code review minden projekten kötelező!

Performancia

Code Splitting és Module Lazy Loading

Annak érdekében, hogy egy adott oldal megnyitásakor minél kevesebb adatot kelljen letölteni, már eleve érdemes a kódból is csak azt küldeni a kliens számára, amely az adott oldal megjelenítéséhez szükséges. Erre egyik megoldás a lazy-load-olt modulok használata, amelynek során a modulból külön JavaScript fájl képződik, amely csak az adott oldalhoz tartozó kódokat tartalmazza. Így az inicializáló JavaScript fájlba (pl.: main.js) csak azok a shared elemek kerülnek be, amelyek magának a teljes portálnak az összeállításához és az aloldalak közös elemeihez szükségesek. Így minimalizálni lehet a kliensnek feleslegesen elküldött kódok mennyiségét.

> Érdemes minél több útvonalhoz lazy-load-olt route-okat felhasználni.

Ezeket általában a különálló funkcionalitás mentén szét lehet választani.

Tree shaking

Külső library-k kiválasztásakor érdemes arra törekedni, hogy azok támogassák a tree shaking-et és az alkalmazás moduljai is ezeket úgy importálják, hogy a tree shaking meg is tudjon valósulni.

Érdemes a Lighthouse ilyen irányú metrikáját kritikával kezelni, ugyanis a ténylegesen futtatott kódokat nézi az adott oldalbetöltésnél, de azokat, amelyek esetleg interakcióhoz, vagy olyan részhez kapcsolódnak, amelyek shared elemek és nem kerültek az oldalon felhasználásra, azt is külön kijelzi.

Érdemes egyeztetni és alaposan utánajárni, hogy tényleg van-e létjogosultsága az adott library projektbe importálásának. Ebben nagy segítség lehet a Bundlephobia | Size of npm dependencies oldal, ahol npm package-ekre tudunk rákeresni és

ezekről különféle információkat kapni. Lehetőség van akár a package.json feltöltésére is, így az egész projektünk dependenciáit is meg tudja vizsgálni és kiértékelni.

Amennyiben egy, már futó projektet szeretnénk alaposan megvizsgálni behúzott package-ek tekintetében, arra a webpack-bundle-analyzer használható. Ahhoz, hogy ezt használni tudjuk, a buildelés során egy stats.json fájlt kell a webpack-kel készíttetnünk, amelyet ezzel az analyzerrel tudunk megvizsgálni.

Kép, és tartalom Lazy Loading-ja

Képek esetében a legegyszerűbb módszer a böngészők által kínált natív lazy loading használata az img tag-re elhelyezett `loading="lazy"` attribútum használatával. Ilyenkor a viewport-on kívül eső képeket a böngésző nem kezdi el letölteni mindaddig, míg azokhoz le nem görgetünk. Ezeket a képeket ilyenkor csak preload-olja a böngésző.

> A legtöbb modern böngésző ma már támogatja ezt a megoldást, így használata javasolt.

Erre, illetve bármely egyéb DOM elem lazy loadolására lehet alternatíva az `IntersectionObserver` használata. Ugyanilyen módon tudunk eljárni bármilyen más DOM elem esetében, például, ha egy div-et és annak tartalmát csak akkor szeretnénk megjeleníteni, ha azok a viewport-ba kerülnének.

Ennek a módszernek két hátránya lehet, attól függően, hogy hogyan használjuk:

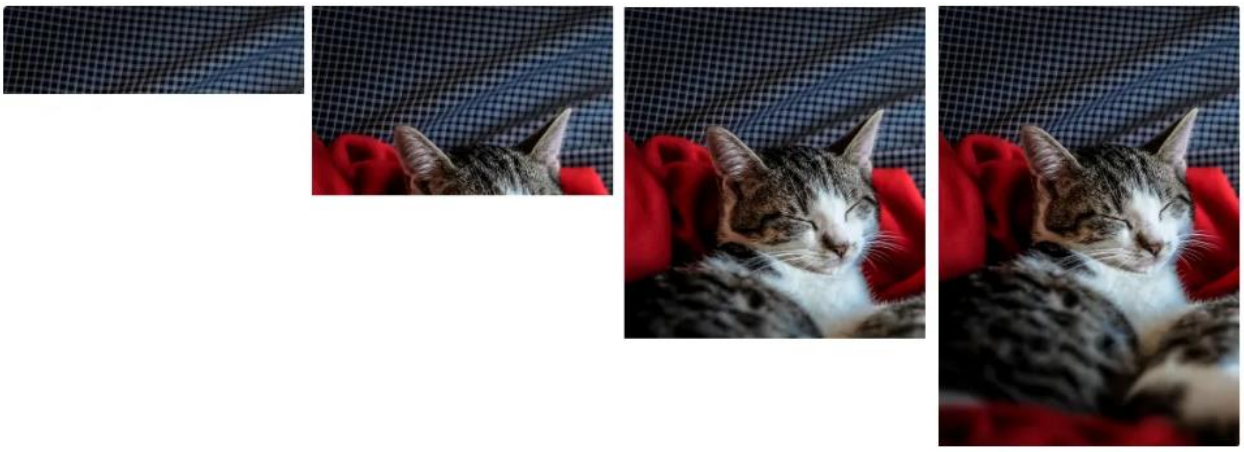
Amennyiben nagyon sok `IntersectionObserver`-t helyezünk el és azok szinte ugyanazokon a pozíciókon trigger-elnek, adott esetben lassíthatják az oldal betöltését, vagy a görgetés lereagálását. Így ez az optimalizációs mód egy bizonyos pont után fordított hatást válthat ki.

Amennyiben a tartalom nagy részét ilyen módon rejtjük el, akár egyfajta infinite scroll hatást is el lehet érni, de könnyen megzavarhatja a felhasználót az, hogy a scrollbar alapján kis terjedelmű az oldal, így lehet, hogy le sem görget az aljára. Az

ilyenkor történő betöltést is érdemes jól lekezelni, ugyanis alapvető viselkedése, hogy CLS-t (Cumulative Layout Shift) okoz, vagyis ugrálni fog a tartalom betöltésnél. Ez leginkább akkor lehet problémás, ha egyszerre több elem is úgy tölt be, hogy akár másodperceként többször dobják meg az oldal méretét, ami nagyon zavaró tud lenni.

Progressive Image Loading

A jpg, gif és png képek esetében van lehetőség azokat progresszív módon elmenteni és betölteni. Ennek lényege, hogy az alapból baseline módon mentődő és töltődő képek felülről lefelé töltődnek be, míg a progresszív verzió esetében egy sokkal gyengébb felbontású verzióból tudunk kiindulni, amelyet folyamatosan töltünk egyre nagyobb méretben. Ezáltal a kép eleve akkora helyet fog elfoglalni, mint a teljes betöltése után, illetve a felhasználó számára is egy jobb élményt ad, mert már eleve láthatja az egész képet, még ha nem is a legjobb minőségben.



baseline

progressive

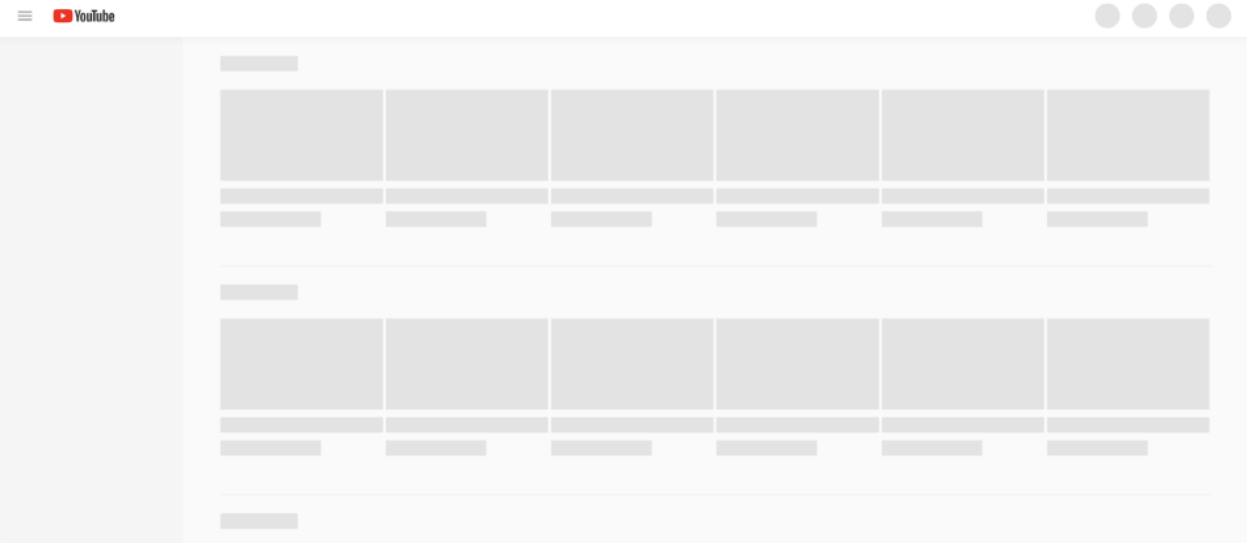
A progressive töltés használatához csupán a megfelelő beállításokkal kell elmentenünk a képet, így a használata elég egyszerű. Visszamenőleg bevezetni már sokkal nehezebb, ugyanis ilyenkor a már meglévő képeket is újra kell konvertálni. Ez függ attól is, hogy az adott oldalon a képek lokálisan vannak-e kiszolgálva, vagy valamilyen CDN-en keresztül, ugyanis ilyenkor akár automatizálni is lehet a folyamatot.

Skeleton Loading Screen

Amennyiben az adott oldalhoz aszinkron töltünk be további adatokat, ezek betöltődését különféle módon jelezhetjük a felhasználó felé. Erre bevett módszer különféle spinner-ek és progress bar-ok megjelenítése, azonban a betöltött adatok megjelenítése által elfoglalt hely általában nem azonos ezeknek a méretével, így

CLS-t (Cumulative Layout Shift) okoznak, tehát az oldal függőleges mérete hirtelen megnő, vagy lecsökken. Ez pedig a felhasználói élményt is drasztikusan rontja. A skeleton lényege, hogy próbáljuk vele előre jelezni a felhasználó felé, hogy ott milyen tartalom fog betöltődni és annak megfelelően veszi fel a méreteit. Ennek köszönhetően a betöltés végét követően a megjelenített tartalom egyáltalán nem, vagy csak minimálisan fog CLS-t okozni.

A skeleton-ok általában az adott portál design rendszeréhez alkalmazkodva, de a szürke valamilyen árnyalatával jelenítik meg a skeleton elemeit. Ezek általában szöveges elemeket, képeket, gombokat jeleznek előre. Nagyon fontos, hogy már az oldal tervezésekor érdemes a skeleton-okkal számolni, ugyanis úgy tudnak hatékonyan működni, hogyha az így lefoglalt hely meg fog egyezni a tényleges elemek méreteivel.



Skeleton-t érdemes akkor használni, hogyha egyszerre több elemet töltünk be, ugyanis így a felhasználó számára is egyértelműbbé válik, hogy mi történik. Például, ha csak egy képet töltünk be, egy szürke téglalapból nem igazán lesz egyértelmű a felhasználó számára, hogy ide egy kép fog betöltődni. Nagyon fontos tehát, hogy ezek az elemek kapjanak egyfajta kontextust, ami a felhasználó számára is értelmezhető.

Elvárások a betöltési sebességgel kapcsolatban

Általánosságban elmondható, hogy az oldal betöltési sebességét érdemes az adott portál által kínált feature set-hez viszonyítani. Egy kevés funkciót tartalmazó weboldalnál alapvető elvárás, hogy akkor gyorsan is töltsön be. Egy összetett, nagyon sok funkcióval rendelkező weboldalnál már jóval elfogadhatóbb, hogyha a betöltése valamivel több időt vesz igénybe. SPA-k esetében ezt az oldalak közötti navigálás gyors reakciójával architekturálisan ellensúlyozzuk.

Ehhez segítséget tud adni frontenden a Lighthouse által mért metrikák skálabeosztása, ugyanis ez a Google által begyűjtött weboldalak adatainak statisztikáján alapszik.

> Érdemes tehát arra törekedni, hogy az átlag fölött helyezkedjen el a weboldalunk a különféle metrikák tekintetében.

A böngészőkben lévő DevTools segíthet alapvető performancia hibák felderítésében. Itt a Performance fül és Chrome böngészők esetében a Lighthouse fül is a segítségünkre lehet.

A Performance fülön mérhető egy weboldal betöltése a nulláról, így látható, hogy a portál egyes kódrészei milyen futási idővel rendelkeznek. A felhasználói élmény miatt, amennyiben egy task 50ms-nél tovább blokkolja a szálat, azt long task-nak tekintjük, ezeket piros színnel ki is fogja nekünk emelni a DevTools. Ez azért lehet problémás, mert a felhasználói interakciók is ugyanezen a szálon történnek, így annak 50ms-nél tovább történő blokkolása esetén a UI nem tud reagálni a felhasználó által elvégzett interakciókra.

Mivel az, hogy 50ms alatt mennyi feladatot tudunk elvégezni, erősen függ a böngészőt futtató hardver számítási kapacitásától is, így ez erősen hardverfüggő.

A Performance fülön további méréseket tudunk elvégezni az oldal betöltésétől függetlenül is.

Érdemes vizsgálni nem csak az oldal betöltését, hanem olyan interakciókat, amelyek mögött komolyabb számítások futhatnak le, így képet kaphatunk azok futási idejéről is. Bármilyen long task-ot érdemes a lehetőségekhez mérten minimalizálni. A JavaScript egy szálon képes futni a böngészőkben, így párhuzamos programozásra nincs lehetőség.

Lighthouse referencia értékek

A Lighthouse a Google metrikai eszköze és egyfajta, de facto módszer frontend oldali performancia és egyéb webes metrikák, SEO, accessibility mérésére. A Google saját eszközén kívül sok más eszköz is létezik alternatívaként, azonban a legtöbb esetben a Lighthouse egy célszerű választás. Azonban a Lighthouse-nak is vannak komoly problémái, amely a mérési módszeréből is adódik. A Lighthouse pedig nem csak a talált problémákat tárja elénk, hanem próbál rájuk javítási javaslatot, hibafeltáráshoz egyszerűbb segítséget adni, hogy könnyebben meg tudjuk érteni a feltárt problémát.

A Lighthouse, Chrome böngészőkben a DevTools-ban alapból be van építve (F12 és Lighthouse tab), de létezik hozzá külön kiegészítő, valamint külön webes változat.

A vizsgált kategóriákat külön-külön kiértékeli a Lighthouse és mindegyiket pontozza. Az egyes pontok tudják reprezentálni a weboldal adott kategóriáihoz tartozó „teljesítményét”. Így elszeparálható egymástól például az oldalbetöltés és a SEO.

A Performance mérésére egy viszonylag jó módszert választottak, miszerint ténylegesen megnézik az oldal betöltődését és a betöltés bizonyos részeinek idejét használják fel a pontszám kiszámítására. Ezen felül külön vizsgálnak többféle részt, mint magának az oldal első megjelenítésének a betöltését, mire betölt az összes része az oldalnak, vagy például, hogy nem fut-e túl sokáig JavaScript, ami blokkolást eredményezne.

Mi számít jó pontszámnak?

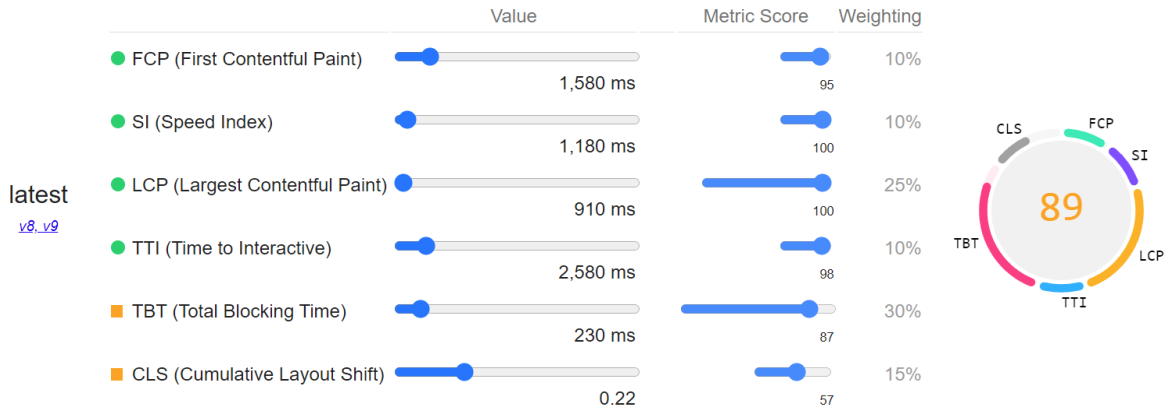
Amennyiben az adott projekt esetében előre ki szeretnénk kötni elvárt referenciaszámokat a Lighthouse mérésekből, akkor ahhoz szükséges kikötni a

tesztelendő hardvert is, vagy azt a platformot, amelyen megfelelő körülmények között azonos környezetben megismételhetőek és validálhatóak a mérések. Mivel a Lighthouse performancia pontszáma nagyban függ a futtató eszköztől, ebben az esetben ezt előre szükséges meghatározni. Ugyanez érvényes mind Desktop és Mobile mérésre is.

A Lighthouse a Performance mérések során 6 tényezőt vizsgál, ezeknek az értékeit pedig súlyozva számolja. Az egyes pontszámok 3 tartománnyal rendelkeznek, zöld, sárga és piros, így ránézésre van viszonyítási alap a kiírt érték mellé. Az egyes mérések tartományait, illetve a súlyozási arányokat időről időre a Google-nél újraszámolják.

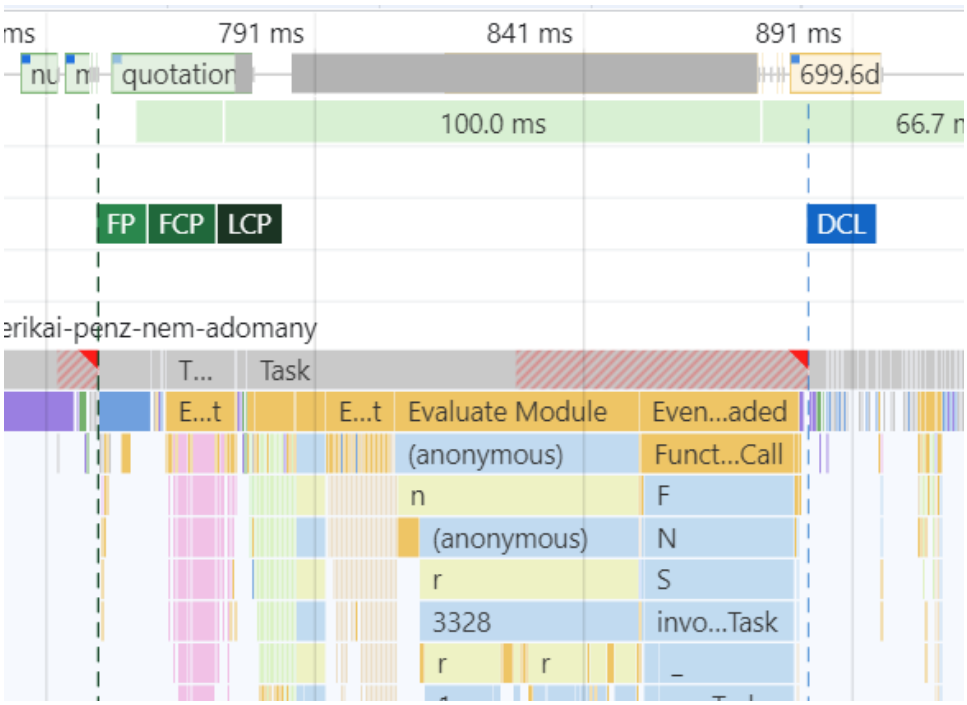
Ahhoz, hogy könnyebben érthető legyen a pontszám alakulása, minden Lighthouse verzióhoz készül egy Scoring Calculator, amelyben csúszkákkal tudjuk nézni, hogy az adott metrikában történő javulás vagy romlás milyen kihatással van végső pontszámunkra.

Kalkulátor - [Lighthouse Scoring calculator]



FCP – First Contentful Paint

A First Contentful Paint – magyarul hívható Első Tartalmi Elemnek - nevéből adódóan az az idő, ami az oldal megnyitásakor szükséges ahhoz, hogy az első látható szöveg, kép, tehát a tényleges oldal elemei meg tudjanak jelenni. DevTools Performance fülön egy mérést futtatva látható is külön, hogy ez mikor történik meg.



Fontos, hogy ez nem azt adja meg önmagában, hogy a szerver milyen gyorsan válaszol. Van köze a pontszám alakulásának hozzá, tehát például a TTFB (Time to First Byte) is ide számítható, azonban az FCP nagyobb szám lesz ennél. Nem elég, hogy a szervernek küldenie kell az oldalt, a böngészőnek is fel kell dolgoznia. Ez azt jelenti, hogy minden oldalon lévő CSS, olyan JS, ami nincs defer/async-hez kötve, vagy nem valamilyen esemény váltja ki a futását.

Optimális érték: < 1.8 sec

Elfogadható érték: < 3 sec

Túl lassú: 3 sec fölött

SI – Speed Index

Ez egy viszonylag nehezen megfogható pontszám, szerencsére nem igazán fordul elő olyan, hogy csak ezzel a pontszámmal lenne gond. Ha az FCP és az LCP jó, általában ezzel sem szokott gond lenni. Értéke erősen függ a viewport méretétől is, ugyanis azt méri, hogy a látható elemek mennyire gyorsan tudnak megjelenni az oldalon.

Optimális érték: <4 sec

LCP – Largest Contentful Paint

Megadja, hogy a teljes oldal elkészülése mennyi időt vesz igénybe. Ide beleszámít például a megjelenő képek betöltése is. A lényeg, hogy itt már minden letöltött asset, script és egyéb resource már bent van a DOM-ban és használatban van, az ezek által behúzott további resource-ok szintén tudják rontani ezt a számot.

Amennyiben például van egy oldalunk elég sok képpel és a viewport-on kívül eső képek nincsenek valamilyen formában lazy load-olva, akkor az LCP addig vár amíg az összes kép le nem fog töltődni.

Ezért is nagyon fontos a képeken lazy loadot használni, hogy egy nagyobb terjedelmű oldal esetében, ahol a képek jórésze a viewporton kívülre eshet, ne kelljen azokat letölteni.

Szintén ide tartozik minden egyéb, más olyan renderelést blokkoló JS és CSS, amely megakadályozza az oldal teljes kirajzolását.

TTI – Time To Interactive

A TTI nagyon hasonló a TBT-hez, viszont ez egy kalkulált szám az FCP-t követően arról, hogy mennyi időbe telik, hogy az oldal interaktív lehessen, tehát nincsen blokkoló kód. Tehát a TTI az oldalnak azon betöltési ideje, ami alatt az ténylegesen tud már felhasználó eseményeket fogadni.

Szoros kapcsolatban áll a TBT-vel, ami sokkal súlyosabb a pontszámban, így annak javítása ezen is javít.

Optimális érték: <3.5 sec

TBT – Total Blocking Time

A Total Blocking Time lényege az, hogy ha az oldalon bárhol fut olyan JS kód, amelynek a futási ideje meghaladja az 50 ms-t, akkor az long tasknak számít,

blokkolja a main thread-et. Mivel a JS single threaded és ugyanazon a szálon fut a weboldallal, ezért, ha nem tud elég gyorsan visszatérni az event loop-ba, a felhasználó számára nem fog interakció történni az oldallal, ugyanis annak feldolgozása nem tud megtörténni időben. Fontos kiemelni, hogy az 50ms alatti futási idő is okozhat frame drop-okat, azonban ez nem feltétlen lesz érzékelhető a felhasználó számára.

A gyakorlatban azonban sokszor lesz 50ms-nél hosszabb futási idő. Eleve az eszközönként eltérő számítási kapacitás miatt az a feladat, ami az egyik eszközön lefut 50ms alatt, lehet, hogy egy gyengébb eszköz esetében már egyáltalán nem tud megtörténni ennyi idő alatt.

Tipikus példa erre az Angular bootstrappelése, ugyanis ez sok esetben túlnyúlik ezen az időablakon, ami blokkoláshoz vezethet.

Hogyan lehet rajta javítani:

- Nem kritikus feladatokat késleltetni, pl.: `setTimeout`
- WebWorker-be csomagolni számításigényes feladatokat
- Hosszabb feladatokat feldarabolni, hogy közöttük reagálni lehessen a felhasználói interakciókra
- Kevesebb JS használata
- Bizonyos JS-ek felhasználói interakcióhoz kötése
- (akár) JS elhagyása ott, ahol az SSR-ből küldött tartalom nem interaktív, vagy partial hydration használata
- `runOutsideAngular` használata időigényes feladatokhoz, hogy azok ne tudjanak felesleges CD cycle-öket kiváltani

Optimális érték: <300 ms

CLS – Cumulative Layout Shift

A CLS az oldal vizuális stabilitását méri, azt követően, hogy az első paint elkészült. Ez abban nyilvánul meg, hogy az oldalon megjelennek-e olyan elemek, amelyek el

tudják tolni a körülöttük lévő többi elemet úgy, hogy az a felhasználó számára zavaró lehet.

Amennyiben olyan elemek jelennek meg, amelyek eltolják a layout-ot, ezeknek a mértékből egy pontszám képződik. Erősen beleszámít a végleges pontba, mert ez egyértelműen rontja a felhasználói élményt.

SSR-nél figyelnünk kell arra, hogy amennyiben egy adott elemet nem tud az SSR kirenderelni, de a böngésző már igen, úgy érdemes annak helyet lefoglalni, hogy ne okozzunk layout shiftet.

Másik példa erre, amikor képeket jelenítünk meg úgy, hogy nincsen előre lefoglalt helyük. Ilyenkor ahogy betöltődik a kép, csak akkor foglal el akkora helyet amekkora a mérete alapján lehetséges, ez pedig szintén layout shiftet okoz. CSS-ből meg kell ezeknek adni a méretét, azonban, ha csak a max-width vagy max-height-et adjuk meg, az nem fog javítani ezen.

Problémásak még a hirdetések is, ugyanis itt nem tudhatjuk előre, hogy érkezik-e hirdetés és ha igen, mekkora méretben. Megoldás lehet erre placeholder-ek használata, amennyiben biztosan tudjuk, hogy az adott pozícióba mindig érkezik hirdetés és csak a megadott mérettel.

A Lighthouse kijelzi külön, hogy milyen elemeket talált, amelyek Layout Shiftet okozhattak.

Hogyan tudunk rajta javítani:

- Statikus képeknek width és height megadása, ez képes reszponzívan is működni
- Az oldal betöltése után dinamikusan betöltődő elemeknek adjunk meg előre definiált méretet
- Ha lehetséges, foglaljuk le előre a helyet azon hirdetéseknek és külső scripteknek, amelyek elhelyeznek valamit az oldalon
- FOUC elkerülése

Optimális érték: < 0.1

Keresőoptimalizálás (SEO)

A keresőoptimalizálás alapköve, hogy a keresőmotorok (Google, Bing, Yahoo) beindexeljék weboldalunk page-eit és meg is jelenítsék azokat a találati listájukban. Egy SPA (Single Page Application) esetében a keresőmotor nem fogja tudni indexelni az oldalakat, mert az általunk megírt HTML kód is belefordul a main.js-be, amit a robotok nem tudnak kiolvasni onnan, ezért alakult az SSR (Server-Side Rendering), amely lehetővé teszi, hogy egy, a robotok által is olvasható általános DOM-ot kapjunk.

Publikus oldalakon a SEO használata kötelező!

Robots.txt

A robots.txt fájl (link) kötelező minden olyan végfelhasználói weboldalon, ami nem adminisztrációs webalkalmazás (private app). Mindent érdemes engedélyezni feltérképezésre, kivéve a keresés oldalt, és az API URL-eket. A keresés oldalt nem ajánlott indexelni, mivel önálló tartalmat nem képvisel, a API URL-eket pedig nem szeretnénk, ha látnák a robotok a security issue-k elkerülése végett.

```
User-agent: *
Disallow: /kereses
Allow: /publicapi/hu/rss/
Disallow: /publicapi/
Sitemap: https://pelda.hu/sitemapindex.xml
```

Sitemap.xml

A sitemap egy olyan fájl, amely a weboldalon fellelhető linkeket tartalmazza, akár kategorizáltan összegyűjtve. A különféle keresőrobotok, crawler-ek általában csak bizonyos szintig mennek el egy adott oldallátogatás során, így könnyen előfordulhat,

hogy lesznek olyan oldalak, amelyek csak nagyon sokára, vagy egyáltalán nem jelennek meg a keresőoldalakon. A sitemap segítségével minden oldalt át tudunk adni a keresőmotorok számára. A sitemap az egyes oldalakhoz kapcsolódóan tartalmazhat további információkat, például az adott oldal utolsó módosítását, vagy dinamikusan frissülő oldal esetében azt, hogy az milyen gyakorisággal szokott változni. Így a keresőmotor ez alapján képes ütemezni, vagy azonnal újra indexelni a weboldalt, így a változtatásaink hamarabb fognak megjelenni a találati oldalakon. Egy sitemap legfeljebb 50 Mb méretű lehet vagy 50000 URL-t tartalmazhat, így ha ennél több létezik az adott webes alkalmazásban, mindenképp több sitemap fájlra van szükség.

Példa a sitemap.xml struktúrájára:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://example.com/</loc>
    <lastmod>2022-01-01T23:12:10Z</lastmod>
    <changefreq>weekly</changefreq>
  </url>
</urlset>
```

A sitemap használata a robots.txt-hez hasonlóan kötelező!

De itt is csak abban az esetben, ha nem adminisztrációs webalkalmazás-t (private app) fejlesztünk.

Dinamikusan frissülő oldal esetében ajánlott a frissülés gyakoriságát és az utolsó módosítás dátumát is feltüntetni a sitemap-ben.

HTML meta tags

Például: title, description, Open Graph tags, Twitter tags.

Arra használjuk őket, hogy plusz információt szolgáltassunk az oldalunkról a webböngészők, keresőmotorok és egyéb webes szolgáltatások számára. A `<head>` elemben a következő meta tag-ek felvétele ajánlott:

A description meta az oldal tartalmának pár mondatos (160 karakterszám meghaladása nem javasolt) leírása, általában ez jelenik meg a keresők találati oldalán leírásként.

Az Open Graph meta tagek határozzák meg, hogy az URL-ek miként jelenjenek meg, mikor megosztják őket a közösségi médiában.

> Az Open Graph meta tagek használata a közösségi média fontossága miatt erősen ajánlott.

Ahhoz, hogy ez megfelelően működjön, a következő tageket kell mindenképpen megadnunk: `og:title`, `og:type` , `og:image`, `og:url`.

Forrás: https://ogp.me

Az Open Graph meta tagekhez hasonlóan ajánlott a Twitter Card Tags. Hasonló az Open Graph-hoz, de ez Twitter specifikus. Ennek segítségével létre tudunk hozni olyan kártyákat, amelyek jobb felhasználói élményt biztosítanak a Twitteren való megosztáskor. Több kártya közül is lehet választani.

Forrás: https://developer.twitter.com/en/docs/twitter-for-websites/cards/overview/abouts-cards

A `<title>` tag bár nem kifejezetten meta tag, mégis elengedhetetlen SEO szempontból. Fontos, hogy minden oldalnak olyan címet adjunk, ami röviden, és pontosan utal annak tartalmára.

A robots meta tag-ekkel az adott oldalhoz szolgáltathatunk plusz adatokat a keresőmotor számára, ilyen módon akár jelezhetjük az adott oldal indexelésére vonatkozó kéréseinket is.

Semantic elements

Például: article, header, footer, aside.

A szemantikus HTML elemek már nevükben is utalást tesznek a bennük található tartalomra. Ezzel jobban elkülöníthetővé válnak az oldal különböző tartalmi elemei, emellett felolvasó szoftvert használóknak is segít a tájékozódásban. Az akadálymentesítés fontos részeként ajánlott használatuk.

Headings

A heading-ek egy hierarchikus struktúrát határoznak meg, így fontos, hogy mindig tartsuk be a sorrendet.

Oldalanként csak egy <h1> tag legyen definiálva!

Érdemes a <h1>-nek olyan szövegezést adni, ami a legjobban leírja az oldal tartalmát, ez gyakran megegyezik a <title> tag-gel.

A heading-ek sorrendjének betartása azért is fontos, mert a felolvasó szoftvert használó felhasználók gyakran heading-ről heading-re navigálva térképezik fel az oldal tartalmát. Ha nem tartjuk be ezt a sorrendet, azzal összezavarhatjuk a felolvasó szoftvert, és ezáltal a felhasználót is.

A fenti megkötés alól kivételt képeznek olyan oldalak, amelyek kontextusában megengedik több <h1> tag létezését is egy oldalon. A SEO pontra egy megfelelő header hierarchiában használt oldal több <h1> tag-el nincs jelentősen negatív hatással. Mindennek tükrében is törekedni kell, hogy oldalanként csak egyszer használjunk <h1> tag-et

Image alt attribute

Az „alt” attribútummal leírást adhatunk meg egy képről. Két fontos szerepe van:

- ez fog megjelenni, ha a kép valamilyen okból nem jeleníthető meg
- a keresőmotorok számára kontextust szolgáltat a képről

Az akadálymentesítés miatt és SEO szempontból is erősen ajánlott az alt tag kitöltése.

A keresőmotorok előrébb sorolják a találati listájukon a weboldalunkat, ha a képek tartalma releváns az oldal tartalmához.

Amennyiben a képünk csupán dekoratív célt szolgál, hiánya esetén sem veszik el fontos információ a felhasználótól, akkor érdemes üres „alt” attribútumot használnunk (`alt=””`). Ezzel explicit tudjuk jelezni, a képnek ezen funkcióját.

Canonical URLs (*)

A Canonical URL az URL duplikáció kezelésének egy eszköze. Ha egy adott tartalom több URL-ről is elérhető, akkor a másolatokban “Canonical”-ként kell hivatkoznunk az eredeti URL-re, hogy a keresőmotor el tudja dönteni, hogy melyik URL-t indexelje.

```
1 <link rel="canonical" href="https://www.minta.hu/eredeti" />
```

Canonical URL-ek használata akkor lehet fontos, ha a weboldalunk több URL-ről is elérhető, több nyelven elérhető, vagy az adott oldal tartalma egy másik weboldalról került átemelésre. Amennyiben a weboldalunk és a rajta lévő oldalak egyértelműen csak egy URL-ről érhetőek el (http → https redirect, non-www → www, stb), a Canonical URL-ek opcionálisak, hiszen ilyen esetekben az egyes oldalak egyébként sem lennének megtalálhatóak másik URL-en.

Schema markup (Schema.org)

Miért jó a strukturált adat, mire használható?

A keresőmotorok minél több információval rendelkeznek az oldalunkról, annál pontosabb képet tudnak kialakítani annak tartalmáról, és ez hozzájárul ahhoz, hogy jobb helyezést érjünk el a keresőoldalak találati listáján. A strukturált adatok emellett a találati oldalakon „rich result”-ként tudnak megjelenni.

A strukturált adatoknak több formátuma van. A Google Search a következő formátumokban fogadja el a strukturált adatokat:

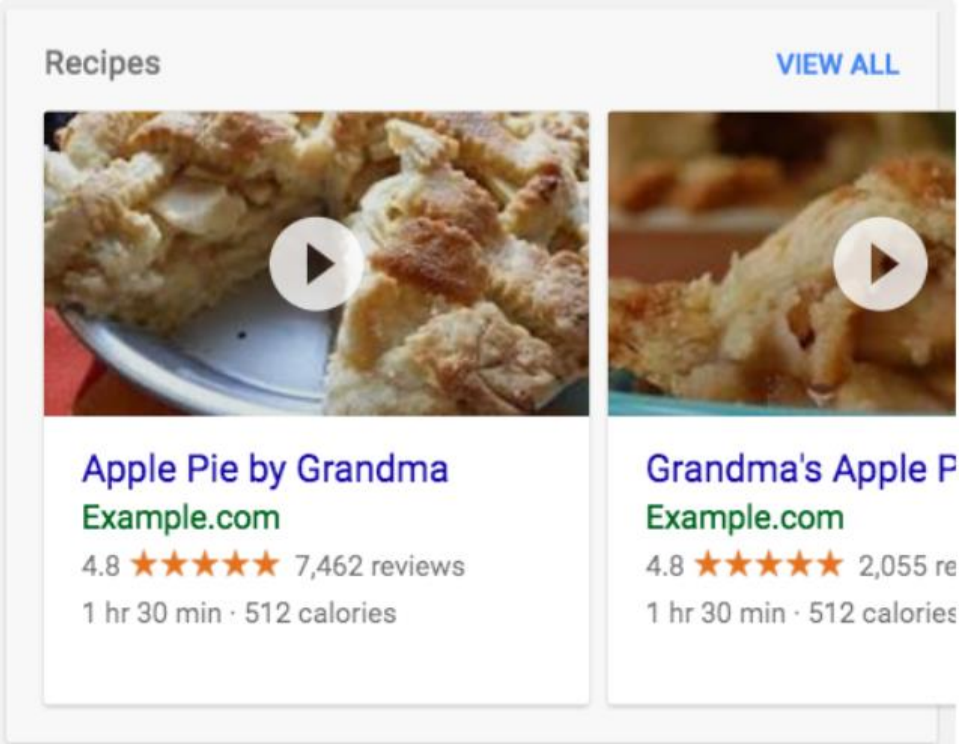
- JSON-LD
- Microdata
- RDFa

A Google a JSON-LD használatát ajánlja.

JSON-LD formátumú strukturált adatokra egy példa kód:

```
<html>
  <head>
    <title>Apple Pie by Grandma</title>
    <script type="application/ld+json">
    {
      "@context": "https://schema.org/ ",
      "@type": "Recipe",
      "name": "Apple Pie by Grandma",
      "author": "Elaine Smith",
      "image": "https://images.edge-generalmills.com/56459281-6fe6-4d9d-984f-385c9488d824.jpg"
      "description": "A classic apple pie.",
      "aggregateRating": {
        "@type": "AggregateRating",
        "ratingValue": "4.8",
        "reviewCount": "7462",
        "bestRating": "5",
        "worstRating": "1"
      },
      "prepTime": "PT30M",
      "totalTime": "PT1H30M",
      "recipeYield": "8",
      "nutrition": {
        "@type": "NutritionInformation",
        "calories": "512 calories"
      },
      "recipeIngredient": [
        "1 box refrigerated pie crusts, softened as directed on box",
        "6 cups thinly sliced, peeled apples (6 medium)"
      ]
    }
    </script>
  </head>
  <body>
  </body>
</html>
```

A fenti kód a következőképpen jelenik meg a Google találati oldalán:



Ezen a linken ellenőrizhetjük, hogy weboldalunk mennyire áll készen Rich result-ként megjelenni: Rich Results Test - https://search.google.com/test/rich-results

Itt pedig reportot kaphatunk a Google-től, hogy a már indexelt oldalunk miként teljesített: Rich result reports - https://support.google.com/webmasters/answer/7552505

A strukturált adatok „gondozásával" a Schema.org szervezet foglalkozik. Itt tájékozódhatunk az általunk használni kívánt modellekről: Schemas - schema.org

SEO friendly URLs

A SEO friendly URL a látogatók számára és a Google robotok számára is egyértelmű tájékoztatást ad az oldal tartalmáról.

- Egyszerű
- Könnyen megjegyezhető
- Mutatja a struktúrát
- Rövid és nem látszanak a fájlkiterjesztések
- Nem tartalmaz speciális karaktereket

Nem keresőbarát: https://www.pelda.com/index.php/2023/?id=436#main
Keresőbarát: https://pelda.hu/cimke/havazas

HTTP Response Status Codes

> A státuszkódok jelzésként szolgálnak a kereső robotok számára, használatuk ezért kötelező!

SPA esetén csak SSR-rel tudnunk HTTP státuszkódot küldeni.

301

A 301-es kód egy állandó átirányítás, melyet akkor használunk, amikor egy adott oldal más URL-en lesz már csak elérhető (Pl. https://pelda.hu/tag/havazas => https://pelda.hu/cimke/havazas). Ezzel jelezzük a robotoknak, hogy a korábbi URL már elavult és cseréljék azt az indexben.

```
// client side (basic redirection)
if (this.utilsService.isBrowser()) {
  window.location.href = redirectUrl;
  return;
}

// server side (SSR - 301 redirect with express js response injector)
this.response.status(301);
this.response.setHeader('location', redirectUrl);
```

404

A 404-es kód azt adja meg, hogy az adott URL-en nem létezik tartalom, így a kereső robot ezt nem fogja indexelni.

Mobile-friendliness

A mobil eszközök elterjedtsége és dominanciája miatt az oldal mobilbaráttá tétele kötelező!

A site mobilebarát megjelenését legegyszerűbben a responsive design-nal tudjuk produkálni.

Webanalitika

A webanalitika az az eljárás, amivel a weboldalak felhasználóinak tevékenysége monitorozható. Ide tartozik a weboldalon belüli navigáció, képek, videók és interaktív elemek használatának a nyomon követése. A felhasználói adatok összegyűjtése mellett feladat még az adatok felhasználhatóvá tétele, jelentések létrehozása, személyre szabott mérések előállítása is. Fontos megjegyezni, hogy a felhasználói tevékenységről való adatgyűjtés a felhasználói oldalról beleegyezést igényel, ennek jogi hátterét minden esetben érdemes szakemberrel egyezetni.

A felhasználói adatok közé tartozik:

- felhasználói forgalom forrása
- referáló oldalak adata (honnan jött a felhasználó)
- az oldal megtekintések
- a felhasználó által megtett navigációs út a weblapon belül
- látogatás átlagos időtartama
- új és visszatérő látogató megkülönböztetése

A Google által fejlesztett ingyenes, korszerű digitális analitikai eszköz a Google Analytics 4 (GA4).

Webanalitika kialakításában a Google Analytics 4 eszköz az ajánlott!

Kétféleképpen telepíthető a Google Analytics 4, tehetjük ezt a Global Site Tag (gtag.js) ami a natív kód vagy Google Tag Manager (GTM) segítségével.

Gtag egy egységes követő kód, amit egyes Google termékek használnak (pl.: Google Analytics, Google Ads stb.). A gtag.js-t direkt a weboldal kódjába kell beilleszteni.

Forrás: https://developers.google.com/tag-platform/gtagjs

Régebben minden Google terméknek saját követő kódja volt (Ads = conversion.js, Google Analytics = analytics.js). Ezek egységesítésére lett bevezetve a Global Site Tag. A gtag.js használata esetén a weblodalon más követések bevezetése esetén (pl.: Facebook, Twitter) újabb követőkódokat kell a kódban elhelyezni. Ezen felül minden egyéb konfiguráció (pl.: speciális event-ek követése) mind a weboldal kódjában történik, vagyis időigényes fejlesztői implementációt von maga után.

A Google Tag Manager telepítése után képesek vagyunk különböző követő kódokat egyetlen egy platformról menedzselni, legtöbbször fejlesztői beavatkozás nélkül. GTM-mel tudjuk a Google-höz és nem Google-höz tartozó követő kódokat szerkeszteni, telepíteni és törölni is.

GTM-mel rendelkezésünkre áll egy felhasználóbarát kezelő felület, amivel az általunk használt követőkódokat rendezhetjük. Míg a gtag.js használata esetén legtöbbször magát a weboldalon elhelyezett kódot kell változtatni. Ezért gtag.js használata csak abban az esetben ajánlott, ha a fejlesztők felelősek az analitikai beállításokért is. Egyedi igényektől és analitikai stratégiától függően kell a két eszköz közül választani. Ha az igények változnak, a két eszköz közötti váltás lehetséges, optimális esetben az adatok folytonossága sértetlen.

Kifutó Google analitikai termékek

Az analytics.js a jelenleg legelterjedtebb Universal Analitycs (GA3 vagy UA) Google Analitycs verzió natív kódja amely 2023. július 1-től leáll az adatgyűjtéssel. Az addig

gyűjtött adatokat még hat hónapig lehet elérni a rendszeren belül, azt követően a szolgáltatás teljesen leáll. A GA4-re történő átállás meglépése minél előbb javasolt az analitikai igények figyelembe vételével gtag.js-sel vagy GTM-mel.

Az Google Analytics és a Google Tag Manager szoros kapcsolatban van a cookie (süti) kezeléssel, ugyanis, amíg a felhasználó a beleegyező nyilatkozatot (consent) nem fogadja vagy utasítja el, az analitikai mérőkódoknak nem szabad lefutnia semmilyen formában!

Az analitikai mérőkódok csak a felhasználó beleegyezése után léphetnek működésbe!

SPA esetében az oldal nem töltődik újra (a pageview nem fut le) navigáláskor. Ilyen esetben, az adott front-end library/framework-nek kell a saját router-én keresztül kiváltani az analitikához a pageview küldését. Kiemelt figyelmet kell fordítani arra, hogy a pageview minden oldal betöltődéskor lefusson.

A helyes működés ellenőrzéséhez érdemes használni az adott analitikai eszközhöz tartozó valamilyen debugger eszközt. Google Analytics 4 esetében beépített debugger-rel, míg GTM esetében Tag Assistant Companion kiegészítővel a GTM Preview módjában ellenőrizhető a beállított mérés.

Forrás: https://chrome.google.com/webstore/detail/tag-assistant-companion/jmekfmbnaedfebfnmakmokmlfpblbfdm

A felsoroltokon kívül használhatók más bővítmények is, a fenti lista csak egy ajánlás. Amennyiben más analitikai eszközt is használnak, érdemes a hozzá tartozó dokumentációban ellenőrizni, hogy a támasztott igényeknek és elvárásoknak megfelel, valamint, hogy milyen debuggolási lehetőségek tartoznak hozzá.

State Management

Bevezetés

Ahogy egy alkalmazás komplexitása nő, egyre nehezebb a konzisztens állapotkezelés. Ezen probléma kiküszöbölésére jött létre számos architektúrális megoldás, mely segítségével egyszerűsödik ez a feladat.

Az állapotkezelés a front-end fejlesztésben általában az alkalmazás adatainak kezelését jelenti. Az értékek változásának nyomon követését és az erre adott UI frissítését foglalja magában, hogy az alkalmazás folyamatosan konzisztens és reagáló maradjon a felhasználói interakciók alatt.

A state management megoldások talán legfőbb ismérve, hogy az alkalmazás egy globális, immutable állapottal rendelkezik, amely bármilyen módosítás során teljes mértékben cserélődik. Ezen központi állapotot manipulálják a reducer-ek, és ezt az értéket kérdezik le az alkalmazás komponensei.

> Mára már minden népszerűbb front-end framework-nek, library-nek van saját állapotkezelő megoldása, melyek használata erősen ajánlott.

Az állapotkezelés példák közé tartoznak a következők:

- **felhasználói hitelesítés:** a felhasználó bejelentkezési munkamenetének állapotkezelése
- **bevásárlókosár:** a felhasználó által a kosárba helyezett tételek követése
- **űrlap adatai:** az űrlapra beírt adatok mentése és frissítése
- **dinamikus UI frissítések:** az UI elemek megjelenésének változtatása a felhasználói műveletek alapján
- **szűrés és rendezés:** a felhasználó szűrési és rendezési választásainak megőrzése.

A fenti példákon kívül minden olyan esetben is alkalmazható az állapotkezelés, ahol az adatok dinamikusan változnak egy webalkalmazásban.

Sokszor nehéz meghúzni a határt, hogy melyek azok a funkciók melyek már megkövetelhetik ezen állapotkezelő megoldások használatát, hiszen ha egy alkalmazás csak megjeleníti a szervertől kapott adatokat akkor nem érdemes ezen technikákat használni.

> Ha a fenti listában található, vagy hasonló komplexitással rendelkező funkció létrehozása a cél akkor ajánlott egy globális store használata.

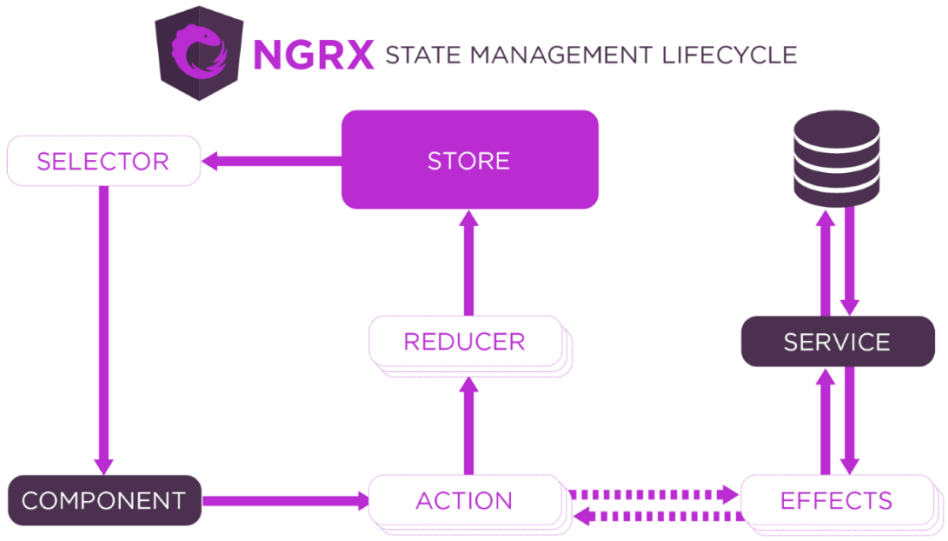
Angular - State Management

> Ajánlott megoldás az adatok kezelésre (főleg nagyobb projektek esetén) az erre készített külső könyvtárak használata.

Jelenleg a legnépszerűbb az NgRx, de lehet más bevált store-t is alkalmazni (NGXS, Akita, RxAngular).

Az adatok kezelése a különböző store könyvtáraknál ugyanazzal az alapmetodikával történik. A komponens egy elkülönített store-al kommunikál: érkezik egy kérés és arra egy aszinkron válasz. A service nem közvetlenül a komponenssel kommunikál, hanem a store-al (persze a komponens közvetett parancsára). Ezt jól szemlélteti az NgRx store felépítése, amit a következő ábra mutat.

NgRx



A különböző store könyvtárak működésére és használatára a saját dokumentációjuk mindig naprakész leírást ad.

> Az adat manipulálásnak „immutable” módon kell történnie!

Ennek ellenőrzésére ajánlott használni az erre vonatkozó saját beállításokat.

NgRx store-nál ebben segíthetnek a `strictStateImmutability`, vagy a `strictActionImmutability` paraméterek, amiket a Module importálásánál lehet beállítani.

```
StoreModule.forRoot(reducers, {
  runtimeChecks: {
    strictStateImmutability: true,
    strictActionImmutability: true
  }
})
```

React – State Management

A React alkalmazások olyan komponensekből épülnek fel, amelyek saját állapotukat (state) kezelik. Ez kis alkalmazásoknál jól működik, de ahogy az alkalmazás

összetettsége növekszik, a komponensek közötti megosztott state-ek kezelése egyre bonyolultabbá és problémásabbá válik.

Ezért lényeges a state kezelés egy skálázható React alkalmazás fejlesztése során. Hosszú távon, ha a state-et nem megfelelően kezeljük, az alkalmazás működése során problémák adódnak. Egy ilyen alkalmazás folyamatos hibaelhárítása és újraépítése körülményessé, ezáltal gazdaságtalanná válhat.

A state kezelése a React alkalmazásokban nem merül ki a `useState` vagy a `useReducer` hookok használatában. Nemcsak sokféle state létezik, hanem gyakran tucatnyi módja van az egyes state fajták kezelésének.

State típusok

Négy fő állapot (state) típus létezik, amelyeket megfelelően kell kezelni a React alkalmazásokban.

Lokális state

Olyan adatok, amelyeket egy vagy más komponensben kezelünk.

> Ezeket a useState és useReducer hook-okkal ajánlott kezelni.
> Külső könyvtár használata nem ajánlott.

Forrás
https://reactjs.org/docs/hooks-state.html
https://beta.reactjs.org/learn/state-a-components-memory
https://beta.reactjs.org/learn/choosing-the-state-structure
https://reactjs.org/docs/hooks-reference.html#usereducer
https://beta.reactjs.org/learn/extracting-state-logic-into-a-reducer

Form state

Formok esetében sokféle állapotot kell lekezelnünk, többek között az input-ok state-jeit, a validációt, a hibaüzenetek megjelenítését, submit-álást stb. Ez mind időigényes és komplex feladat, ha az alapoktól akarjuk felépíteni.

Példa form state-ekre: `disabled, validating, validated, onBlur-validation, onSubmit-validation, loading, submitting, stb.`

Egyszerű form state követést lehet készíteni a beépített `useState` hook-kal.

> Ha nem szükséges egyedi megoldást készítenünk, akkor ajánlott egy bevált külső könyvtár használata.

A legnépszerűbbek és leggyakrabban használt külső könyvtárak a Formik és a React Hook Form.

> Könnyű használata és komplexitása miatt, valamint a népszerű Yup nevű validációs könyvtárral való egyszerű integrációja miatt azonban a Formik használata ajánlott elsősorban.

Forrás
https://formik.org
https://react-hook-form.com
https://github.com/jquense/yup

Globális state

Olyan adatok/állapotok, amelyeket több komponensből, vagy az alkalmazás bármely pontjáról szeretnénk elérni, frissíteni. Parent-child relációban legkönnyebben pár komponens mélységen keresztül (a *lift the state up principle*-t követve) a `useState` vagy a `useReducer` hook-okkal ajánlott kezelni.

Forrás
https://reactjs.org/docs/lifting-state-up.html
https://beta.reactjs.org/learn/sharing-state-between-components
https://beta.reactjs.org/learn/preserving-and-resetting-state

Nem parent-child reláció esetében legkönnyebben a beépített React Context API-al lehet kezelni, azonban fontos megjegyezni, hogy a Context API nem state kezelő megoldás, hanem egy módja az olyan problémák elkerülésének, mint a prop drilling. Gyakran frissítendő state-ek kezelésére nem alkalmas, mert a Context

alapértelmezett viselkedése az, hogy az összes child komponens újra fog renderelődni, ha a prop-ként megadott érték megváltozik.

Ha a state nem frissül gyakran, inkább csak olvassuk, akkor ez a megoldás hasznos lehet, például authentikáció esetében.

Forrás
https://reactjs.org/docs/context.html#when-to-use-context
https://reactjs.org/docs/context.html#before-you-use-context
https://beta.reactjs.org/learn/passing-data-deeply-with-context

A globális state kezeléséhez olyan bevált és tesztelt külső könyvtárakhoz nyúljunk, mint a Zustand, a MobX, vagy a Recoil.

A Redux, illetve Redux Toolkit is megoldás lehet, azonban a feleslegesen nagy overhead-jük, a komplexebb mentális modelljük megértésének nehézsége, illetve az immutabilitásra való nagyfokú támaszkodás nehézkessé teheti a reducerek megírását, emiatt nem kifejezetten ajánlott a használatuk.

A Recoil könyvtár még viszonylag fiatal, így a közösségi erőforrások és a legjobb gyakorlatok még nem olyan robosztusak, mint más könyvtárak esetében.

Egyszerűsége, könnyű olvashatósága miatt a Zustand az ajánlott.

Szerver state

A szerver state külső szerverről érkező adat, amelyet integrálni kell a felhasználói felületek state-jével. Minden alkalommal kezelni kell, amikor szerverről hívunk le vagy frissítünk adatokat, beleértve a betöltési és hibaállapotot is (loading és error state).

A külső könyvtárak nagyban megkönnyítik a szerver state kezelését, ezért használatuk ajánlott.

A lehetséges és népszerű szerver-state-kezelő könyvtárak közül (SWR, React Query) a könnyű használata és nagyobb komplexitása miatt a React Query használata ajánlott, de az SWR is jó választás lehet egyszerűbb igényekre, illetve különösen Next.js használatakor. Apollo Client-et GraphQL esetén használhatjuk.

URL state

Ezek az URL-jeinkben létező adatok, beleértve az elérési utat és a lekérdezési paramétereket. Nagyban megkönnyítik az URL state kezelését, ezért ajánlott olyan külső könyvtárak és framework-ok használata, mint például a Next.js, vagy a React Router.

Ha a React Router-t használjuk, a `useHistory` vagy a `useLocation` segítségével minden szükséges információt megkaphatunk.

Továbbá, ha van olyan útvonalparaméterünk, amelyet használnunk kell, például az adatok lekérdezéséhez, akkor használhatjuk a `useParams` hook-ot.

Forrás: https://reactrouter.com

Next.js-t használatával szinte mindent elérhetünk közvetlenül a useRouter hook meghívásával, ezért ajánlott a Next.js framework használata. A routing mellett még rengeteg más optimalizációval is fel van szerelve.

Forrás: https://nextjs.org

Statikus fájlkiszolgálás (resource, asset)

A cél a gyors betöltés

Törekednünk kell arra, hogy a resource-okat (képek, fontok, ikonok, stb.) a jó felhasználói élmény biztosítása érdekében minél hamarabb tudja renderelni a böngésző. Ügyeljünk a helyes fájlformátumok alkalmazására, és a fölösleges asset-ek törlésére!

Képek

A pixelalapú (bitmap) és vektorgrafikus formátumú képek egyik legszembetűnőbb tulajdonsága a méretbeli skálázhatóság: míg a bitmap a 100%-os nagyításon a legélesebb, ennél tovább nagyítva a kép torzzá válik, mivel hiányoznak azok a pixelek, amik nagyobb részletességhez lennének szükségesek addig a vektorgrafikus gyakorlatilag bármeddig nagyítható, mivel geometriai információkat tartalmaz konkrét pixelek helyett.

> Bitmap alapú kép esetén WebP formátum használata javasolt.

A többi bitmap alapú kiterjesztéssel összehasonlítva, a WebP tudja nyújtani a legjobb tömörítést, így a képek mérete nagymértékben redukálható.

A WebP kiterjesztés támogatja az alpha channel-t csakúgy, mint a PNG. Két tömörítési típusát különböztetjük meg, a veszteségmentes és veszteséges tömörítésűt. A veszteségmentes eljárás akár ~25%-kal kisebb tárhelyet foglal, mint a PNG. A veszteséges eljárás tárhelyigénye pedig ~25-34%-kal kevesebb, mint JPG esetében.

Videók

Ha videós tartalmat használunk weboldalunkon, érdemesebb azt feltölteni egy külön videószolgáltató platformra, mint YouTube, vagy Videa. Ha mégis self-hosted videót használunk, akkor MP4-ben, vagy WebM-ben érdemes tárolni őket. Míg a WebM kisebb fájlmérettel dolgozik, addig az MP4 jobb minőséget biztosít, és nagyobb kompatibilitást élvez a böngészőkkel szemben.

Betűtípusok

A betűtípusokat illetően érdemes WOFF, vagy WOFF2 formátumot használni, ezek a kifejezetten webes környezetre szánt, legjobban tömörített típusok.

Ikonok

Az ikonok esetében ajánlott az SVG kiterjesztés használata a minőségveszteség elkerülése végett (pl. nagy pixelsűrűségű mobil eszközök esetén). Ha 3rd party library-t használunk, érdemes olyat választani, ami támogatja a tree-shakinget.

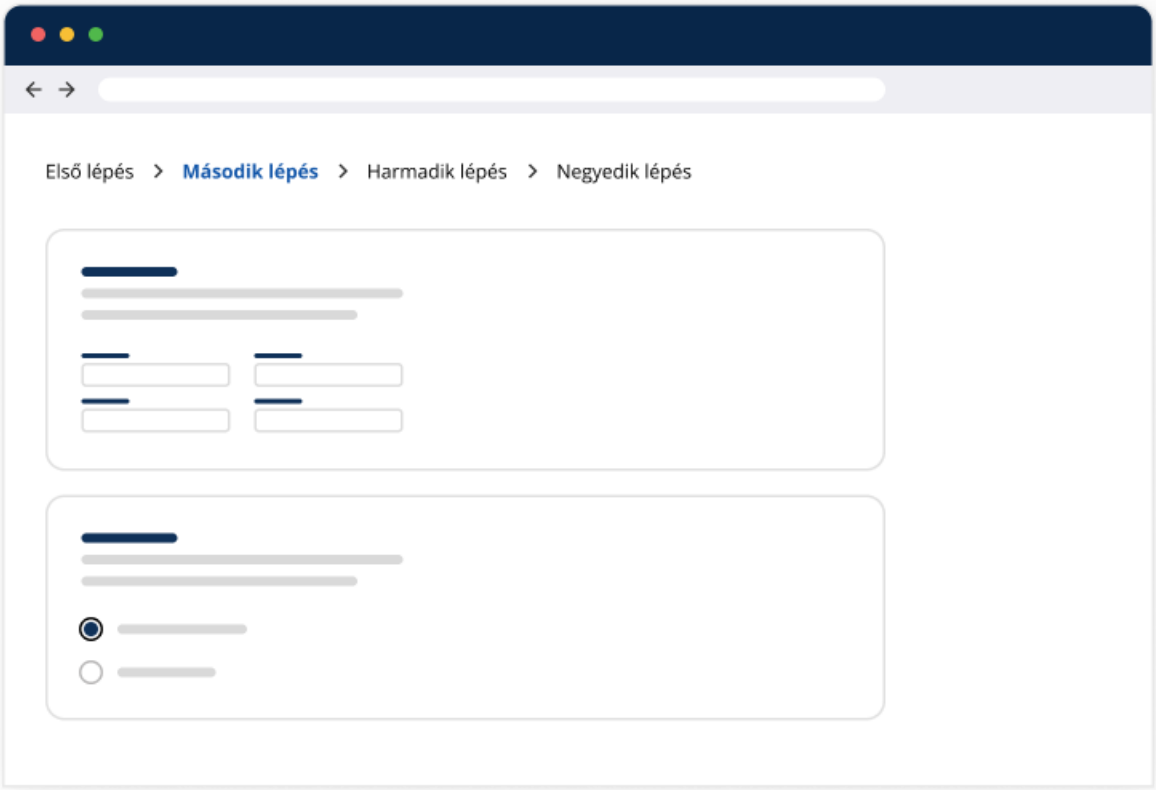
Forrás
https://en.wikipedia.org/wiki/JPEG
https://en.wikipedia.org/wiki/PNG
https://en.wikipedia.org/wiki/Scalable_Vector_Graphics
https://developers.google.com/speed/webp/gallery2

UX/UI design alapelvek és követelmények

A törekvés az alábbi alapelvek betartására nagy mértékben növelheti a weboldalak felhasználói élményét. Indokolt esetekben az ezektől való eltérés elképzelhető, de ez nem tanácsos, hogy ha jól működő végeredményt szeretnénk kapni.

A rendszer állapota látható

Mindig érthető és észlelhető legyen, mi történik az oldalon, a rendszer időben és egyértelműen adjon visszajelzést arról, hogy hol tartózkodik a felhasználó, hogy hova tart és arról is, hogy az interakciók, amelyekbe az oldallal bocsátkozott, sikeresen lezajlottak-e vagy még épp folyamatban vannak. Pl. breadcrumb, progress bar stb.



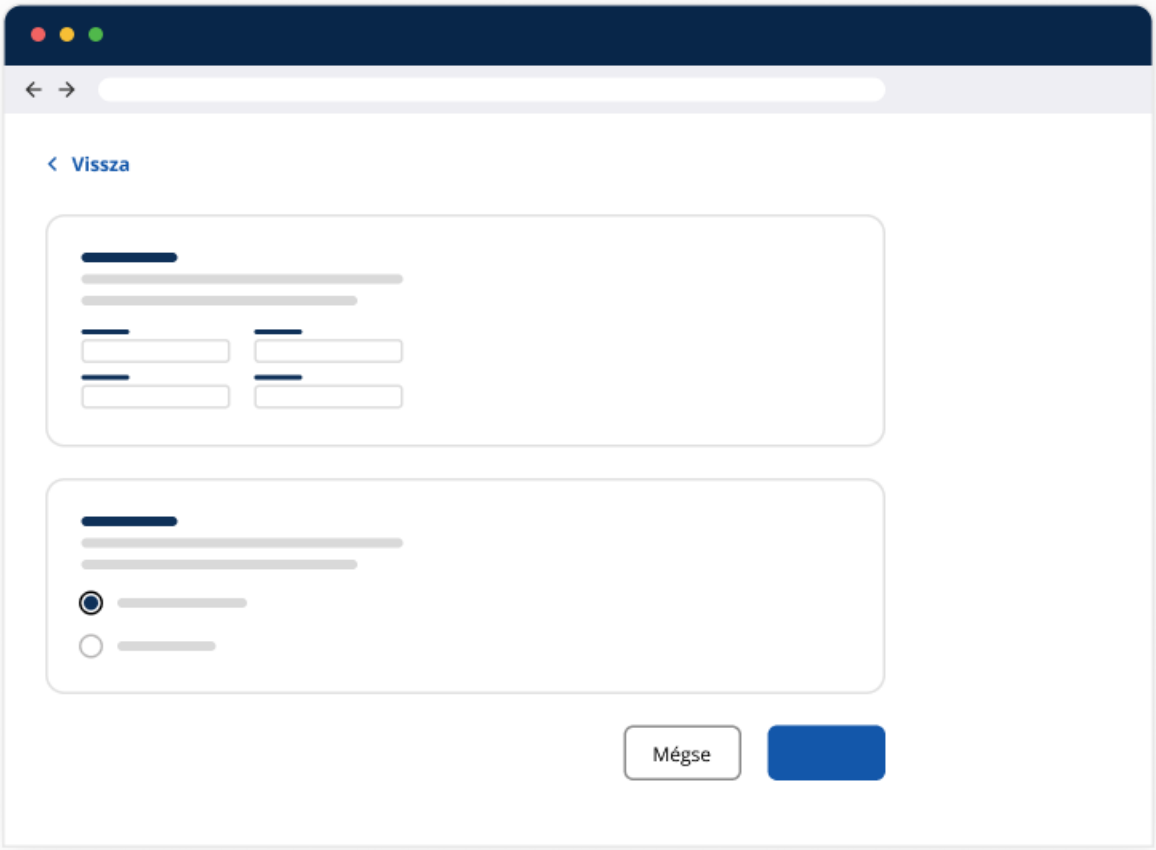
Egyezés a rendszer és a felhasználó való világa között

A felhasználók által ismert megfogalmazással, szóhasználattal, közérthetően kommunikáljon a rendszer az emberekkel, minél kevésbé szaknyelven (technológiai vagy egyéb specifikus zsargont használva), amennyire a konkrét ügyintézési szakterület nyelvezete csak megengedi. Valamint a való világból szerzett tapasztalatok mentén kialakult mentális modellek figyelembevételével szükséges a felületek megtervezése az elrendezés és vizualitás terén is.

Felhasználói szabadság és irányítás

Ha a felhasználó valamilyen hibát vét, rossz helyre navigál, mindig legyen visszavonási vagy visszalépési lehetősége, mellyel a hibáját korrigálni tudja.

- Pl. „Mégse” gomb, „Vissza az előző oldalra” link, „Bezárás”,”“Visszavonás” használata és jól látható megjelenítése szükséges.
- Az egyes linkek, gombok ne új ablakban vagy tabon nyissák meg az egyes aloldalakat, hogy a böngészőprogramok „Vissza” gombja is használható legyen erre a célra.
- Ha a visszalépés megszakít valamilyen folyamatot vagy interakciót, vagy semmisé teszi az addig megtörténteket, az legyen egyértelműen kommunikálva. Pl. figyelmeztető felugró ablak.
- Az átfedést (overlay interaction) létrehozó elemek használatát próbáljuk meg kerülni, ha alkalmazásuk mégis szükséges, akkor azok soha ne töltsék ki a teljes képernyőt. Ha lehetséges, a böngésző Vissza gombja az átfedés létrejötte előtti állapotot állítsa vissza.



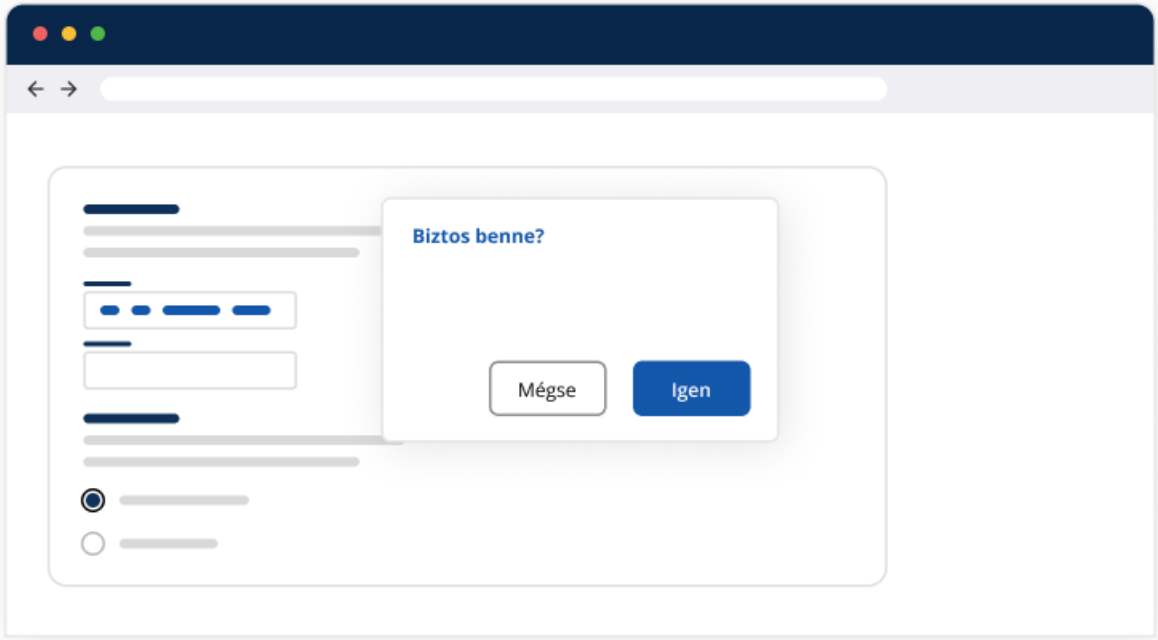
Következetesség és konvenciók

Platformra jellemző minták követése, következetes működés és fogalmak használata, a felhasználók által már ismert konvenciók megtartása elengedhetetlen mind vizuálisan mind tartalmilag (pl. kommunikációs stílus). Az effajta következetesség és „ismerősség” nagy mértékben csökkentheti azt az időt, mely a felület használatának elsajátításához szükséges a felhasználók számára.

Hibamegelőzés

A felület segítsen elkerülni a hibákat, az elvárások vagy tudnivalók egyértelmű kommunikálásával (pl. milyen jelszó vagy egyéb adat megadása szükséges, mintaadatok és alapértelmezések, tooltip, megerősítő pop-up nagyobb döntések előtt stb.), valamint akadályozza meg az értelmetlen adatbeviteleket (pl. az elvárt karakterszámnál több karakter beírásának lehetőségének korlátozása lásd adószám stb.).

A felület ajánljon fel lehetőségeket pl. Legördülő menük, prediktív keresések. A bevitt adatokat jól felismerhetően adja vissza pl. telefonszámok tagolása. Különböző segítségnyújtások csökkenthetik a hibázás lehetőségét pl. irányítószám alapján a település kitöltése automatikusan.



Hibakezelés

Hiba esetén támogassa a rendszer a felhasználót jól és tapintatosan megfogalmazott, egyértelmű hibaüzenetekkel, a teendők érthető kommunikálásával, akár tanító jelleggel és céllal is, hogy a felhasználó legközelebb már ne kövesse el az adott hibát. Maga a hiba jelzése legyen megfelelően feltűnő, észlelhető (lásd még: Inkluzivitás és akadálymentesség). A felület törekedjen arra, hogy hiba esetén minél többet megőrizzen a felhasználó által elvégzett munkafolyamatból.

Felidézés helyett felismerésre késztetés

A felhasználók emlékezőképességét ne terhelje a rendszer, támogassa őket abban, hogy a fontos információk, elemek jelen vannak a folyamatok során és nem a felhasználóknak kell azt felidézniük.

Rugalmas és hatékony használat

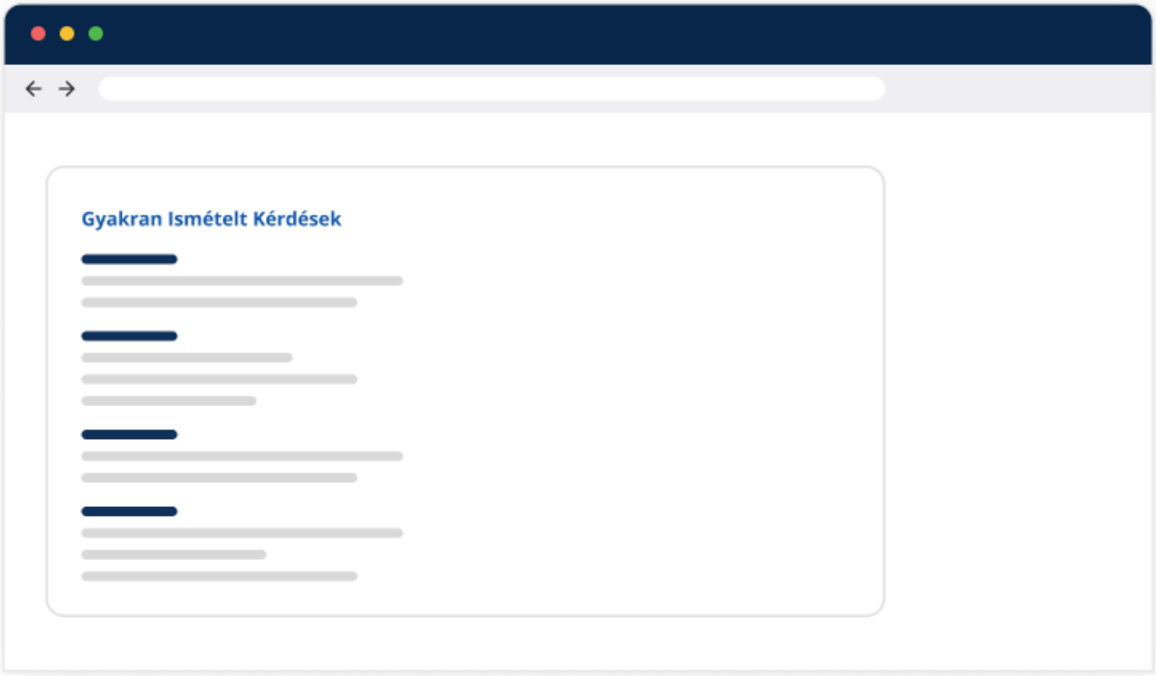
A rendszer mind a kezdő, mind a tapasztalattal rendelkező felhasználók számára is használható legyen, olyan módon, hogy utóbbiak idejét ne rabolja feleslegesen, eljuthassanak gyorsan és egyszerűen a céljukhoz a felületen. A felhasználónak legyen lehetősége eldönteni, hogy több információra van szüksége ahhoz, hogy elvégezze az adott feladatot vagy birtokában van a tudásnak, amely ehhez kell és tovább tud lépni. Alapértelmezetten ne terheljük a felhasználókat felesleges információkkal, csak ha azt ők kifejezetten igénylik, pl. Kinyitható-összecsukható információs kártyák.

Esztétikus és minimalista design

A rendszer vizuális észlelése ne jelentsen kognitív terhelést a felhasználók számára, a felület legyen letisztult, átlátható, felhasználóbarát (UI). A megjelenített információk mennyisége és tálalása, tagolása álljon összehangban a felhasználó céljaival.

Intuitív rendszer

A rendszer legyen intuitív, melyet a felhasználók hosszabb tanulási folyamat, kísérletezgetés, felfedezés nélkül rögtön képesek használni, de elakadás esetén proaktív és reaktív segítséget kaphassanak (jól megtervezett onboarding, súgó rendszer, tutorial funkciók kísérjék útjukat), valamint rendelkezzen az oldal „Gyakran Ismételt Kérdések” menüponttal vagy ehhez hasonló megoldásokkal, ahol a felhasználók választ kaphatnak kérdéseikre).



Design átadás a fejlesztés szolgálatában

A felületek végső UI tervezése kapcsán különös gondot kell fordítani a fejlesztéssel való együttműködésre, a fejlesztésre való átadás sikerességéhez ajánlatos lehet az alábbi szolgáltatások használata: Storybook, Figma Token Studio, Zeplin.

Az alább felsorolt gyakorlatok biztosíthatják a fejlesztés hatékonyságát és a létrejövő weboldalak szakszerű működését:

- **Bootstrap**-re könnyedén adaptálható grid használata a screen tervezés során
- Minél több **vektoros elem** (.svg asset-ek) használata indokolt esetekben (ikonok, rajzok, illusztrációk, ábrák), hogy az elemek ne jelenjenek meg pixelesen a különböző képernyőméreteken, fotók esetében megfelelnek a raszteres fájlformátumok
- **Standard ikonok használata** (pl. 24x24 px, fekete színnel #000000), melyek méret és színvariációit a CSS állítja elő, a felületi tervek követelményeinek megfelelően

- **Jól felépített komponenstár,** a variánsok megjelenítésével, asset típusonként, atomic design szemlélettel létrehozva
- A végső design system tartalmazza ne csak a komponenseket, hanem a **használati szabályokat** és új elemek rendszerbe illesztési követelményeit is

Inkluzivitás és akadálymentesség

Minthogy a publikus portálok széles közönséghez szólnak, így már a tervezés első fázisaiban szem előtt tartandó a leendő felhasználók diverzitása. A vonatkozó törvény által előírt **WCAG 2.2 AA** szabvány elvárásainak való megfelelés szükségessége mellett ajánlott gondot fordítani rá, hogy az ugyan komolyabb korlátozottsággal és/vagy fogyatékossággal nem rendelkező, de nagyon különböző hátterű, életkorú, iskolai végzettségű, kultúrájú, képességű felhasználók inkluzivitása biztosított legyen. Ezen alapelvek és a **WCAG 2.2 AA** szabvány mentén haladó tervezés fogja biztosítani a célcsoportok számára a legmagasabb mértékű használhatóságot és ügyfélélményt.

Az oldalakkal interakcióba lépő felhasználók kognitív terhelését mind tartalmi, szövegezési oldalról, mind a vizuális megvalósítások (felhasználói felület) viszonylatában minél inkább alacsonyan ajánlatos tartani, funkcionálisan támogató, bevonó szándékkal szükséges a felületeket és azon való navigálást megtervezni.

Ezen területen a teljesség igénye nélkül kiemelendő legfontosabb gyakorlatok:

- Felületen megjelenő színek **kontrasztarányának** meghatározása az elvárásoknak megfelelően
- Színtévesztők figyelembevétele: az információk kijelzése, megjelenítése **nem alapulhat** csak a színeken. Pl. Hibaüzenetek jelzése csupán piros színnel.
- Jól értelmezhető **betűméret hierarchia**
- Szellős eltartások (white space), **a zsúfoltság kerülése,** a kognitív terhelés csökkentésére
- Animációk, mikrointerakciók **kikapcsolásának lehetősége,** vagy eleve elhagyása

- **Jól észlelhető és azonosítható** gombok, linkek, navigációs elemek, **inkább szöveges,** mint grafikus megoldások (ábra, ikon, rajz stb.)
- **Egyértelmű üzenetek,** pontos információ átadás vagy annak kommunikálása, hogy a felület mit vár a felhasználótól, pl. Űrlapok beviteli mezőinek ellátása label-lel, dátum mezők kitölthetősége valamilyen dátumválasztó funkcióval stb.

A kötelezően betartandó követelmények részletes kifejtése megtalálható WCAG 2.2 AA szabvány dokumentációjában: https://www.w3.org/TR/WCAG22/

Forrás: https://www.nngroup.com/articles/ten-usability-heuristics

GDPR megfelelés

Átlalános Adatvédelmi Rendelet

A GDPR (**G**eneral **D**ata **P**rotection **R**egulation), vagyis Általános Adatvédelmi Rendelet az Unió területén tartózkodó természetes személyek személyes adatainak védelme érdekében jött létre.

Adatvédelmi nyilatkozat

Ha a weboldalon lehetősége van a felhasználónak bármilyen személyes adat megadására (pl. kapcsolati form), akkor a weboldal már adatkezelőnek minősül. Ilyen esetben kötelezően lehetőséget kell biztosítani a felhasználó számára, hogy megismerhesse a róla tárolt adatokat, és elhatározása szerint bármikor törölhesse azokat. Ha van lehetőség regisztrációra, akkor a felhasználói fiók törlését is biztosítani kell.

Minden esetben, ahol a weboldal adatokat kezel, lennie kell egy adatvédelmi nyilatkozatnak, amit a felhasználó elolvashat.

Az adatvédelmi nyilatkozatnak tartalmaznia kell, hogy az alkalmazás milyen adatokat gyűjt a felhasználóról, és mire használja ezeket az adatokat.

A nyilatkozatban fel kell tüntetni a tárhelyszolgáltató adatait is.

Minden kapcsolat form-nál kell lennie egy checkbox-nak, amivel a felhasználó hozzájárulását adja adatainak kezeléséhez.

Cookie consent

A „sütik” (cookie) által küldött információk segítségével a felhasználók releváns és „személyre szabott” tartalmat kaphatnak. Ezek által a weboldalak üzemeltetői névtelen (anonim) statisztikákat is készíthetnek az oldallátogatók szokásairól. Az

információk birtokában még jobban személyre szabható az oldal kinézete, és a tartalma.

A felhasználónak explicit nyilatkoznia kell arról, hogy mely cookie-kat fogad el és melyeket nem.

Erre ajánlott használni a Google Funding Choices API-t, ami GDPR biztos és mindig naprakész.

Forrás: https://developers.google.com/funding-choices/fc-api-docs

Tesztelhetőség

Tesztelhetőség definíciója

A tesztelhetőség általános szoftverfejlesztési életciklusban annak foka, ami megmutatja, hogy a szállítandó termék (szoftverrendszer, szoftvermodul, követelmény vagy tervdokumentum), milyen mértékben támogatja a tesztelést egy adott környezetben. Amennyiben a szállítandó termék tesztelhetősége magas, úgy a rendszer esetleges hibáinak felkutatása és meghatározása könnyebb, azaz egyszerűbb a tesztelése.

A tesztelhetőség fokának meghatározása, lehetővé teszi a szoftver tesztelési tevékenységeinek elvégzéséhez szükséges erőforrások egyszerű értékelését és meghatározását, beleértve a teszt időtartamát, a tesztesetek számát és a forgatókönyveket. Tehát ez egy alapvető jellemző, amelyet figyelembe kell venni a tesztelési stratégia meghatározásánál és a szervezési szakaszban.

Automata tesztelés támogatása

Automata tesztek esetében, a front-end oldali implementációt kötelező úgy szervezni, hogy támogassák a tesztelési folyamatok karbantarthatóságát.

Az ilyen regressziós céllal futtatásra kerülő automata tesztek az új kód bevonásával gyakran válnak elavulttá, frissítésük erőforrás igényes. Amennyiben általános lokalizációs stratégia helyett egy előre meghatározott tesztelési attribútum kerül bevezetésre a felhasználói felületen, az csökkenti a teszt karbantartási igényét. Azoknál a felületi elemeknél (gombok, formok, és beviteli mezők stb..), ahol felhasználói interakció történik, a tesztvezérlés már ezeket az attribútumokat felhasználva kerüljön futtatásra. Például:

```
<button id="login" automation-id="login-btn">Login</button>
```

Funkcionális tesztelés

Számos kritérium létezik, amelyek alapján a funkcionális tesztelést lehet értékelni az elvárt működés alapján. Funkcionális szempontból vannak pozitív és negatív paraméterek. A funkcionális teszteket kézi és automatizált eszközökkel hajtják végre. A tesztelők érvényes és érvénytelen bemeneteket adnak az alkalmazásnak, és ellenőrzik, hogy a generált kimenetek megfelelnek-e a vártnak. A front-end oldali kódszervezés támogatja azt a működést, ami a bemenő paramétereket feldolgozza, és kezeli az esetleges negatív eseteket.

A rendszeres, manuális funkcionális tesztelés kötelező egy alkalmazás fejlesztésének életútja során!

Az üzleti folyamatok lefutásáról keletkezett logokat és kliens oldali üzeneteket, a fejlesztett alkalmazás egyértelműen és perzisztens módon szolgáltatja. Az így keletkezett információkból lehet következtetni, az esetleges hibákra és a rendszer stabil működésére.

Komponensek tesztelése elkülönített módon

Az elkülönített tesztelés során az egész rendszert kisebb alrendszerekre vagy modulokra bontják. Ezután ezeket az alrendszereket vagy modulokat egymástól függetlenül tesztelik az eredmény ellenőrzése érdekében. A fejlesztési életciklusban belül ez a technika felhasználható az implementálás és tesztelés során, és ezzel a megközelítéssel pontosabban azonosíthatóak a hibák.

Ajánlott olyan megoldást használni, amit a piacon széles körben alkalmaznak és a dokumentálási folyamatot is támogatják.

Ilyen eszköz például a Storybook.

Forrás: https://storybook.js.org

Performancia teszt

A teljesítménytesztelés egy szoftvertesztelési folyamat, amelyet egy szoftveralkalmazás sebességének, a válaszidőnek és erőforrás-felhasználásának tesztelésére használnak adott munkaterhelés mellett. Célja olyan teljesítménybeli problémák azonosítása és megoldása, amelyek negatívan befolyásolhatják a felhasználói élményt.

> Ajánlott olyan eszközök bevezetése, amik támogatják a performancia tesztek kivitelezését.

Ilyen a Lighthouse vagy a PageSpeed Insights. Ezeket a teszteket automatizált módon, minden új verziónál érdemes futtatni, vagy monitoring rendszerbe kötve folyamatosan riportokat készíteni.

> Ajánlott olyan eszközök bevezetése, ami valós idejű metrikákat biztosíthatnak a teljesítménymutatókról.

A valós idejű metrikák lehetővé teszik a fejlesztők számára a teljesítményproblémák azonosítását és megoldását.

Kódlefedettség

A kódlefedettség a front-end fejlesztésben arra vonatkozik, hogy az alkalmazás kódja milyen mértékben fut le a tesztelés során. Segít a fejlesztőknek meghatározni, hogy a kód mely részeit fedik le a tesztek, és melyeket nem, lehetővé téve számukra a további tesztelésre szoruló területek azonosítását. Egy magas tesztlefedettségű programnak több forráskódja fut le a tesztelés során, kisebb az esélye annak, hogy fel nem fedezett szoftverhibákat tartalmaz, mint egy alacsony tesztlefedettségű programnál.

Számos eszköz áll rendelkezésre a kódlefedettség mérésére a front-end fejlesztés során, amely konfigurálható és beállíthatóak vele a megengedett mérési értékek.

```
coverageReporter: {
  dir: require('path').join(__dirname, './coverage/<project-name>'),
  subdir: '.',
  reporters: [
    { type: 'html' },
    { type: 'text-summary' }
  ],
  check: {
    global: {
      statements: 80,
      branches: 80,
      functions: 80,
      lines: 80
    }
  }
}
```

Forrás: https://angular.io/guide/testing-code-coverage

Elfogadási teszt (Acceptance test)

Az elfogadási tesztelés megmutatja, hogy a szoftver megfelel-e a megrendelő követelményeinek és elvárásainak. A végrehajtott tesztesetek és tesztforgatókönyv a valós felhasználói használat alapján kerülnek futtatásra. A kimenet ellenőrzésével történik, a specifikációban rögzített elfogadási kritériumok alapján. A tesztet általában a szoftver fejlesztése után, egy elfogadási környezetben történik és kiadás előtt hajtják végre. Az elfogadási környezetnek az átadás utáni, üzembehelyezési környezetnek tulajdonságaival kell, hogy megegyezzen.

A tesztadatokat érdemes úgy szervezni, hogy minél inkább a valós adatok állapotát tükrözzék.

Amennyiben modern fejlesztői keretrendszert vagy csomag gyűjteményt használunk a tesztadatokat fejlesztés során, az adatréteggel való kommunikáció könnyen konfigurálható, igy a teszteléshez szükséges adatok előállíthatóak.

A terméket szükséges tesztelni a specifikációban foglaltak szerint, minden olyan böngészőben és operációs rendszeren, amit a végső felhasználói használat érint.

Regressziós teszt

A regressziós tesztelés a szoftvertesztelés egy fajtája annak igazolására, hogy a szoftverfejlesztési ciklus során, bevonásra kerülő új kódrészlet nem befolyásolja hátrányosan a meglévő funkciókat. A futtatása minden új funkció hozzáadásával és meglévő funkciók változásával történik. Regressziós tesztek alkalmazása front-end oldali fejlesztés során, kiszűri a nem várt változásokat a felhasználói felületen. Ezek a változások gyakran elemi komponenseket, formokat és a megjelenést leíró CSS-t érintik.

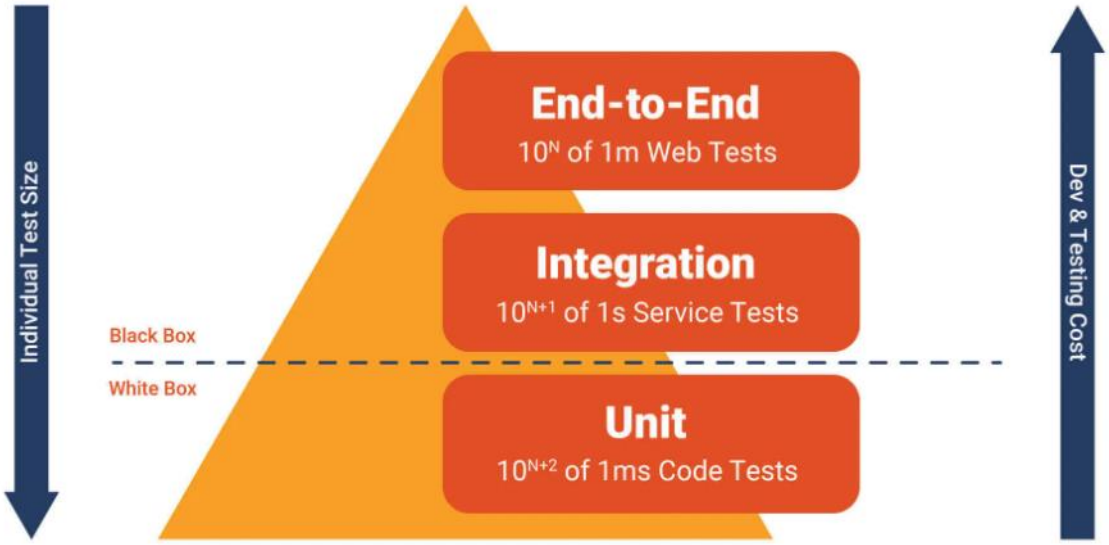
Regressziós teszt alkalmazása esetén a tesztelést ajánlott kiterjeszteni a nemfunkcionális tesztelésre is.

Annak érdekében, hogy webes interfészeket és elemeket automatikus módon lehessen validálni, érdemes vizuális regressziót alkalmazni. Az alkalmazása során szükséges meghatározni egy olyan alap mintát (Base Line), ami viszonyítási alapként szolgál a későbbi tesztlefutásoknak, módosított felhasználói felület esetében, ezt az alap mintát szükséges frissíteni.

Forrás: https://storybook.js.org/tutorials/intro-to-storybook/angular/en/test

Tesztelési piramis

A tesztelési piramis egy olyan szoftvertesztelési koncepció, amely a különböző típusú tesztek közötti relációt szemlélteti az alkalmazások átfogó lefedettségének biztosítása érdekében. A tesztelési piramis a következőképpen épül fel:



A tesztelési piramis mögött az az elgondolás áll, hogy a tesztek többsége unit teszt legyen, amelyet kisebb számú integrációs teszt követ, és még kevesebb E2E teszt. Ez a megközelítés lehetővé teszi a fejlesztők számára, hogy a fejlesztési folyamat korai szakaszában észleljék a problémákat, amikor azok javítása olcsóbb, ugyanakkor átfogó lefedettséget biztosít az alkalmazásra.

Unit tesztelés

Az alkalmazás egységeinek, atomi részeinek tesztelése. Front-end oldali programozás során az egység gyakran egy teljes felület/interfész, például egy osztály, de lehet egyedi metódus is.

A unit teszt a szoftverfejlesztési folyamat korai szakaszában találja meg a problémákat, és feltárja a produktum specifikációjának esetleges hibáit, vagy annak hiányzó részeit is. A unit tesztelés egyszerűsíti a telepítési és fejlesztési folyamatokat. Automatizált tesztek, amit a szoftverfejlesztők írnak és futtatnak annak biztosítása érdekében, hogy a lehetséges bemeneti tesztadatokra, az alkalmazás az elvárt kimenetet produkálja, továbbá rögzítésre kerüljenek a ~~hibafeltételek~~ negatív esetek. A unit tesztelés szükséges része a fejlesztésnek. A modern teszt keretrendszerek egyszerűsítik ezeknek a teszteknek a kivitelezését, könnyen szimulálható

objektumokat eredményez. Egy ilyen keretrendszer több fejlesztési platformra is bevezethető, mint például a JEST, ami Angular és React környezetben is bevethető.

Integrációs teszt (Integration test)

Az integrációs tesztelés magába foglalja a különböző összetevők és szolgáltatások közötti interakciók és kommunikáció tesztelését egy webalkalmazásban. A cél annak ellenőrzése, hogy a rendszerek moduljai a várt módon működnek-e együtt, és az integrációs problémák a fejlesztési folyamat korai szakaszában feltárásra kerüljenek. Az integrációs tesztek, a unit tesztek mellett és azokat nem kiegészítve kerülnek implementálásra.

Ajánlott az alkalmazásfejlesztés során figyelembe venni a tesztvezérelt (TDD) fejlesztést, és már a projekt indulásakor bevezetni.

End-to-end teszt

Ez a fajta tesztelési megközelítés a teljes felhasználói munkafolyamatra összpontosít. Elfogadási környezetben, érdemes a tesztelést folyamatokon végrehajtani, ami lefedi azokat a használati eseteket, amelyek az alkalmazás felhasználását illetően kritikusak.

A különböző technikai rétegek kommunikációja során új típusú hibák jelenhetnek meg. Ezek a hibák gyakran az események szinkronizációjával és végrehajtás sorrendjével kapcsolatosak, ezért hibás működés léphet fel, például hálózati késleltetés miatt és konkurencia kezelésből adódóan is.

Tekintettel ezekre az eseményekre, ajánlott mindig az adott front-end környezethez megfelelő, és karbantartott keretrendszer használatával lefedni ezeket a típusú teszteket. Amennyiben az alkalmazás tesztelésére használt csomag elavulttá válna, szükséges ennek a kiváltása. Egy ilyen példa a Protractor, ami Angular alkalmazásoknál használt end-to-end teszt keretrendszer, de ma már nincs nyomós ok arra, hogy a Protractort válasszuk versenytársaival szemben. Alternatívaként

használjunk olyan eszközt, ami támogatja az font-end oldali implementációt, akár komponens szinten is, továbbá van karbantartott verziója.

Forrás
https://docs.cypress.io/guides/component-testing/angular/quickstart
https://github.com/angular/protractor/issues/5502

Használhatósági (usability) teszt

> Már az alkalmazás tervezési fázisaiban is több ízben megvizsgálandó a felületek használhatósága.

Egy ilyen teszt szituációban valós időben figyeli meg a UX/UI designer (vagy több designer), hogy a felhasználók hogyan használják az alkalmazás egyes, már elkészült részeit vagy azok prototípusát. A cél, hogy feltárják azokat a pontokat, ahol a felhasználók nem értik teljesen az alkalmazás kommunikációját, megtorpannak az útjukon, a kívánt feladatok végrehajtásában akadályba ütköznek, illetve ahol az interakciókba bocsátkozás során nehézségeik támadnak. A megfigyelések és a felhasználók visszajelzései alapján ezeket a problémákat beazonosítják, rögzítik és a felület átalakításával küszöbölik ki őket. Majd újabb tesztekkel vizsgálják meg, hogy a módosított, javított felületeken már képesek-e a felhasználók gond nélkül végrehajtani az elvégzendő feladatokat.

Forrás
https://angular.io/guide/testing
https://reactjs.org/docs/testing.html
https://reactjs.org/docs/testing-recipes.html#mocking-modules
https://en.wikipedia.org/wiki/Testability
https://en.wikipedia.org/wiki/Code_coverage
https://www.browserstack.com/guide/code-coverage-vs-test-coverage
https://www.browserstack.com/guide/functional-testing
https://www.getxray.app/blog/testability-in-the-software-development-lifecycle
https://hu.education-wiki.com/6851380-functional-testing-vs-non-functional-testing
https://www.360logica.com/blog/testability-role-software-testing
https://testfort.com/blog/what-is-testability-in-software-testing
https://testing.googleblog.com/2020/08/code-coverage-best-practices.html

Biztonság

OWASP Top Ten

Az OWASP top 10 egy lista a tíz leggyakoribb biztonsági sebezhetőséggel, amelyek a webalkalmazások fejlesztése és karbantartása során felmerülhetnek. Az OWASP (Open Web Application Security Project) egy nonprofit szervezet, amelynek fő fókusza a webalkalmazások biztonságának javítása és emellett a webalkalmazások fejlesztőinek oktatása.

Forrás: https://owasp.org/www-project-top-ten

A jelenlegi OWASP Top Ten listája az alábbi sebezhetőségekből áll.

Hibás hozzáférés-ellenőrzés

Broken Access Control. Ennek a sebezhetőségnek a forrása a felhasználók jogosultságainak helytelen kezelése vagy ellenőrzése. Ekkor egy potenciális támadó olyan oldalakhoz vagy objektumokhoz fér hozzá (például a nem megfelelő autentikáció, autorizáció vagy session kezelés miatt), amelyekhez alapesetben nem szabadna.

A legegyszerűbben ez ellen védekezni megfelelő felhasználói validációval lehet, Angular esetében pl. Http interceptor-ok alkalmazásával, amellyel minden http kérésnél ellenőrzi az alkalmazás, hogy van-e megfelelő token-je a felhasználónak. Router guard-ok használatával pedig minden route-nál egyedileg megszabhatjuk, hogy milyen jogosultságú felhasználó léphet csak be az adott route-ra, így biztosítva, hogy nem megfelelő jogosultságú vagy nem autentikált felhasználó olyan tartalomhoz jusson, amelyhez nem kellene. A megfelelő autentikációról legbiztonságosabban egy http hívással a szerver felé bizonyosodhatunk meg, ami ellenőrzi és visszadja a kliensnek, hogy az adott felhasználónak érvényes-e még a token-je.

Hibás titkosítás

Cryptographic Failures. Az iparban többféle titkosítási algoritmust használnak különféle adatok titkosítására. Ilyenek a jelszavak, a hitelkártya adatok vagy egyéb személyes, szenzitív információk. Ha ez a fajta titkosítás nem megfelelően van implementálva, akkor egy potenciális támadó visszafejthet szenzitív adatokat. Például, ha gyenge vagy elavult titkosítási algoritmus van használatban, vagy esetleg a titkosítási kulcsok tárolása nem biztonságos.

Frontend fejlesztés esetében arra kell figyelni, hogy szenzitív adatot ne jelenítsünk meg, ne tároljunk pl. Store-ban, sütikben vagy local storage-ban, illetve ne legyen kódszinten beégetve API kulcs, amelyet valamely csomaghoz/könyvtárhoz használunk, erre használjunk pl. Dotenv-et.

Egyéb titkosítási eljárások, mint pl. jelszavak hash-elése vagy egyszer használatos jelszavak (One Time Password) generálása minden esetben szerveroldali feladat kell, hogy legyen.

Injektálási támadások

Injection. Az adatbázisokból, fájlokból, parancssori argumentumokból érkező, a bemenetek nem megfelelő kezeléséből származó sebezhetőségek. Ekkor egy potenciális támadó képes arra, hogy a fent említett sebezhetőséget kihasználva lekérdezés(eket), adatbázis utasításokat, szélsőséges esetben programkódot vagy parancsot futtasson az applikáción belül, melynek nem kívánt következményei lehetnek (pl. információ szivárgás, destruktív tevékenység, kártékony kód bejuttatása, irányítás átvétel).

Injection vagy Injektálási támadáshoz a támadónak kell először egy input mező, amibe beleírhat valamilyen kártevő scriptet vagy parancsot, amit aztán le szeretne futtatni a szerverrel.

Hogy ezt kivédjük, fontos a megfelelő form validáció. Tegyük fel, hogy bámiféle igényből adódóan szükségünk van egy text inputra, amibe bármit beírhat a

felhasználó, az input értékét pedig egy az egyben elküldjük egy http hívással a szervernek, ami lefuttatja a kapott értéket.

Ekkor egy támadó, aki hozzáfér ehhez az input mezőhöz, bármilyen kártékony parancsot beírhat, és a kliens ezt el is fogja küldeni. Az első, amit ilyenkor tehetünk validáció szinten, hogy bizonyos speciális karaktereket kiszűrünk, pl:

- &
- |
- ;
- `

Ezeket akár Regex-el kiszűrhetjük az inputból, így megakadályozva azt, hogy a támadó egymás után több parancsot adjon a szervernek, a “;” pontosvessző kiszűrésével.

Egy másik fajta injektálási sérülékenységi felület a template-injection. Angular-ban pl. Lehetőség van a HTML template-be dinamikus adatot megadni:

```
@Component({
  selector: 'app-header',
  template: '<h1>' + (window.location.hash || 'Home') + '</h1>'
})
export class HeaderComponent {}
```

Kerülendő a template ily módon megadása, mivel ilyenkor kikerüljük az Angular compiler-e által nyújtott template behelyező metódusát, ami kiszűri ezeket a sérülékenységeket:

```
@Component({
  selector: 'app-header',
  template: '<h1>{{ title }}</h1>'
})
export class HeaderComponent {
  title = ''

  ngOnInit() {
    this.title = window.location.hash || 'Home';
  }
}
```

Nem biztonságos architektúra

Insecure Design. Az olyan sebezhetőségek gyűjtőfogalma, amelyek a szoftver architektúra hiányosságai miatt jelentkeznek. Például gyenge vagy hiányos autentikáció vagy nem megfelelő és hiányos biztonsági naplózás.

Fontos tehát figyelni a megfelelő route guard-ok alkalmazására, a http hívásoknál a megfelelő header-ök és a megfelelő és érvényes JWT token ellenőrzésére, valamint a megfelelő form validációkra.

Hibás biztonsági konfigurációk

Security Misconfiguration. Egy szoftver biztonsági rendszerének a hibás konfigurációja is vezethet sebezhetőségekhez. Például, ha az alapértelmezett jelszavak nincsenek átállítva, vagy ha bizonyos szolgáltatások (service-ek) feleslegesen futnak a háttérben, így plusz támadási felületet kínálva a potenciális támadóknak.

Sebezhető és elavult komponensek

Vulnerable and Outdated Components. A legtöbb webapplikáció használ harmadik fél által fejlesztett és üzemeltetett könyvtárakat. Ezeknek a könyvtáraknak is megvannak a saját sebezhetőségeik, ezért nagyon fontos, hogy csak olyan könyvtárakat használjunk, amelyeket rendszeresen karbantartanak, és a különböző biztonsági auditoknak megfelelnek.

Frontend oldalon ezért fontos kiemelni, hogy mindig a legújabb (lásd. Verziókövetés támogatott projekteknél fejezet) verziót használjuk Angular-ból vagy React-ból, illetve az általunk használt harmadik fél által fejlesztett csomagokat és könyvtárakat is rendszeresen frissítsük (lásd Harmadik fél által fejlesztett könyvtárak fejezet).

Hibás identifikáció és autentikáció

Identification and Authentication Failures. Ennél a sebezhetőségnél egy potenciális támadó meg tudja kerülni az alapvető identifikációs és autentikációs réteget (pl. jelszó feltöréssel).

Frontend oldalon a megfelelő router guard-ok és http interceptor-ok alkalmazásával ezek kivédhetőek, valamint fontos arra figyelni, hogy semmilyen, a felhasználóhoz kapcsolódó szenzitív adatot (jelszó, session id, stb.) ne jelenítsünk meg. Kiemelendő továbbá, hogy annak érdekében, hogy a felhasználót minél jobban védjük az adatai kiszivárgásától, érdemes egy bizonyos idő eltelte után kijelentkeztetni, pl. Feliratkozni a token lejárati idejére vagy egy bizonyos idő után küldeni egy logout requestet a szerver felé.

Szoftver- és adatintegritási hibák

Software and Data Integrity Failures. Jogosulatlan szereplő megváltoztathatja a szoftver kódját vagy az általa tárolt adatokat. Például man-in-the-middle támadással, ahol két szolgáltatás közötti kommunikáció közé beékelődve egy támadó elkapja a küldött adatokat, azt megváltoztatja és úgy küldi tovább a fogadó félnek.

Frontend oldalról a megfelelően ellenőrzött és karbantartott csomagok használatával, valamint megfelelően ellenőrzött és biztonságos DevOps háttérrel tudjuk kivédeni ezt a sérülékenységet.

Hiányos biztonsági naplózás

Security Logging and Monitoring Failures. Ha nem megfelelő a biztonsági naplózás egy szoftverben, sokkal nehezebb észlelni a potenciális támadásokat vagy a lehetséges sebezhetőségeket.

Frontend oldalon is fontos ezért, hogy a megfelelő mértékben naplózzuk az esetleges hibákat. Fontos megkülönböztetni a várt (expected) és nem várt (unexpected) hibákat. Várt hiba lehet például, hogy egy kérés hibával tér vissza (pl. egy file-t akartunk feltölteni AWS-re, de a file túl nagy volt). Az ilyen hibákat le kell

kezelni frontend oldalon, esetleg a felhasználónak is egy érthető hibaüzenettel jelezni. Nem várt hibának azok a hibák számítanak, amelyeknek az eredete nem tisztázott, például ha az említett AWS nem egy custom hibaüzenettel tért vissza, hanem egy általános (general) hibaüzenetet kaptunk. Az ilyen hibákat fontos logolni, akár a browser-t használva (console.error), akár népszerű logging eszközöket használva, pl. Datadog vagy Sentry. Ezeknek a használatával átfogó képet kapunk az előforduló nem várt hibákról, valamint segít beazonosítani azokat a várt hibákat is, amelyeket még nem kezel az alkalmazásunk.

Szerveroldali kéréshamisítás

Server-Side Request Forgery (SSRF). A támadó képes kéréseket küldeni egy sebezhető szerverről más webalkalmazások vagy belső rendszerek felé, melyeket használva jogosulatlan hozzáférést nyerhetnek szenzitív adatokhoz vagy szolgáltatásokhoz.

Az OWASP Top Ten alkalmazása

Minden webalkalmazás tervezésénél és fejlesztésénél kiemelt fontosságú a biztonság kérdésköre. Az OWASP Top Ten listáját, illetve megelőzési ajánlásait érdemes egy új projekt indításánál vagy egy meglévő projekt karbantartásánál átnézni. Egy alkalmazás tervezésénél és üzemeltetésénél szükséges a sebezhetőségeket megelőzni az esetleges biztonsági hiányosságokat felderíteni, és elhárítani.

> Ajánlott az OWASP Top Ten listáját alapul véve rendszeresen (évente) átnézni az alkalmazást biztonsági szempontból.

CAPTCHA

A CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) egyfajta biztonsági intézkedés, amelyet kihívás-válasz hitelesítésként ismernek. A CAPTCHA segít megvédeni a felhasználót a kéretlen levelekkel és a

jelszavas visszafejtéssel szemben azáltal, hogy egy egyszerű teszt elvégzésére kéri, amely bizonyítja, hogy ember, és nem számítógép, amely egy jelszóval védett fiókba próbál betörni.

Miért használjuk a CAPTCHA-t?

A CAPTCHA-k alapvetően visszatartják a hackereket az online szolgáltatásokkal való visszaéléstől, mivel megakadályozzák, hogy a robotszoftver hamis vagy aljas online kéréseket küldjön be.

A CAPTCHA tesztek segítségével:

- Megvédhető az online szavazások integritása azáltal, hogy megakadályozza, hogy a hackerek robotokat használva ismételten hamis válaszokat küldjenek.
- Megállíthatóak az online fiókok elleni brute force támadások, amelyek során a hackerek ismételten több száz különböző jelszó használatával próbálnak bejelentkezni.
- Megakadályozható, hogy a hackerek több e-mail fiókot regisztráljanak, amelyeket aztán aljas célokra használnak fel.
- Megakadályozható, hogy a kiberbűnözők rosszindulatú megjegyzésekkel és más webhelyekre mutató hivatkozásokkal spameljenek blogokat vagy híroldalakat.
- Megakadályozható, hogy előadások és koncertek jegyeinek tömeges vásárlására robotokat használjanak.
- Az online vásárlások bizonságosabbá tehetőek.

Mi az a reCAPTCHA?

A reCAPTCHA a Google egyik szolgáltatása ami felismeri és elemzi a felhasználó viselkedését, hogy megállapítsa, mennyire emberszerű. Ha a reCAPTCHA szolgáltatás úgy ítéli meg, hogy a viselkedés meglehetősen emberi, akkor nem fog teljes captcha tesztet végezni. Csak arra kéri a felhasználót, hogy jelöljön be egy négyzetet, hogy megerősítse, hogy „Nem vagyok robot”. Ha azonban van valami

robotikus abban, ahogyan a felhasználó interakcióba lép az oldallal, akkor egy bonyolultabb captcha-teszt megoldására kell kérni.

> Ajánlott a Google reCAPTCHA v2 checkbox-ot használni mert az a legbiztonságosabb.

A reCAPTCHA v2 megjelenítése

Ha különböző front-end keretrendszerek segítségével fejlesztjük az applikációnkat akkor a reCAPTCHA megjelenítését érdemes explicit módon végrehajtani az automatikus helyett. Ezt a betöltési visszahívási függvény megadásával és paraméterek hozzáadásával lehet elérni a következő módon:

1. Adja meg a betöltési visszahívási függvényt. Ez a függvény akkor kerül meghívásra, ha az összes függőséget betölti.

```
<script type="text/javascript">
  var onloadCallback = function() {
    alert("grecaptcha is ready!");
  };
</script>
```

2. Illessze be a szkriptet, állítsa be az onload paramétert a betöltési visszahívási függvény nevére, a renderelési paramétert pedig explicitre.

```
<script src="https://www.google.com/recaptcha/api.js?onload=onloadCallback&render=explicit"
    async defer>
</script>
```

Amikor a visszahívás végrehajtódik, meghívhatja a grecaptcha.render metódust a JavaScript API-ból.

A betöltési visszahívási függvényt a reCAPTCHA API betöltése előtt kell meghatározni!

Új projekt indítása

Egy új projekt indulása előtt biztonsági szempontokból is meg kell vizsgálni a választott keretrendszert, annak kiválasztott verzióját, illetve az esetlegesen használt harmadik fél által fejlesztett könyvtárakat.

Angular

Új projektet a mindenkori aktuális, vagy legfeljebb egyel korábbi Angular verzióval szükséges indítani!

React

A React könyvtár fejlesztői az Angular-al szemben nem rendszeres időközönként adnak ki frissítéseket. A React csapat minden major verzióhoz tartozó utolsó stabil patch verzióhoz ad ki folyamatosan biztonsági frissítéseket.

Forrás: https://endoflife.date/react

Új projekt indulásánál azonban a legújabb React verziót kell választani, hiszen a React-hoz íródott és karbantartott könyvtárak is a legújabb verziót fogják majd biztosan támogatni.

Új projektet mindig az aktuális React major verzióval kell indítani!

Node.js

A major Node.js verziók megjelenésükkor Current státuszba kerülnek 6 hónapra. Ezt követően a páratlan kiadások (15, 17, stb.) nem kapnak további támogatást, a páros kiadások (16, 18, stb.) Active LTS státuszba kerülnek, és ekkor válnak késszé az általános használatra. Az LTS verziókban menet közben felefedezett kritikus hibák a

verzió megjelenését követő 30 hónapon keresztül javításra kerülnek (Maintenance LTS).

Forrás: https://nodejs.dev/en/about/releases

Új projektet a mindenkori Active LTS vagy az ahhoz legközelebb álló Maintenance LTS státuszban lévő Node.js verzióval kell indítani!

Harmadik fél által fejlesztett könyvtárak

Legtöbb esetben több, harmadik fél által fejlesztett könyvtárat is használ egy webalkalmazás. Biztonsági (és egyéb más) szempontból kiemelt fontosságú, hogy lehetőleg ismert, megbízható kiadótól származó könyvtárakat válasszunk, amelyek rendszeresen kapnak frissítéseket. Ezzel csökkentve a projektünkre ható biztonsági kockázatot.

A könyvtár típusától függ, hogy mi az a heti letöltési szám és frissítési frekvencia, amellyel már nyugodtan használhatunk egy-egy könyvtárat. Ökölszabályként elmondható, hogy a legnépszerűbb könyvtárakat hetente több százezren vagy millióan töltenek le, valamint legalább 1-2 havonta kapnak új verziót. Ezeket az információkat például az NPM honlapján lehet megnézni.

Forrás: https://www.npmjs.com

Ajánlott olyan könyvtárakat használni, melyeket legalább több százezren töltenek le heti szinten, illetve rendszeresen, 1-2 havonta kap frissítéseket!

Verziókövetés támogatott projekteknél

Egy hosszú távon támogatott projekt esetében fontos odafigyelni a keretrendszer és a használt könyvtárak karbantartására.

Angular verzió léptetése

Az Angular minden major verziójához 18 hónapig ad biztonsági frissítéseket a fejlesztői csapat. Éppen ezért biztonsági szempontból szükséges rendszeres időközönként verziót léptetni egy hosszú távú szoftvertámogatás esetén.

> Éles Angular alkalmazásokat a mindenkori LTS verziók valamelyikén kell üzemeltetni.

Az Angular verzióval együtt a TypeScript és a Node.js verziót is frissítenünk kell, ebben segít a következő táblázat:
https://gist.github.com/LayZeeDK/c822cc812f75bb07b7c55d07ba2719b3

Az Angular verzió léptetésében pedig segít az Angular hivatalos verzióléptetési útmutatója: https://update.angular.io/

React verzió léptetése

A React fejlesztői csapata a React 15-ig visszamenőleg minden major verzióhoz ad biztonsági frissítéseket. Ennek ellenére érdemes biztonsági szempontból is a React verzióját léptetni időről-időre, a különböző, React-hez írt könyvtárak és csomagok biztonsági kockázatának csökkentésére. A React csapata minden új major verzióhoz kiad egy verzióléptetési útmutatót.
Például: https://reactjs.org/blog/2022/03/08/react-18-upgrade-guide.html

Amennyiben a Next.js keretrendszert is használja a fejlesztett vagy támogatott webalkalmazás, szintén kapunk egy útmutatót a verziók léptetésére:
https://nextjs.org/docs/upgrading

> Ajánlott mindig a legújabb major verziót használni mind React, mind Next.js esetében!

NodeJS verzió léptetése

Éles alkalmazásokat kizárólag Active LTS vagy Maintenance LTS státuszban lévő verziókon kell üzemeltetni.

Forrás: https://nodejs.dev/en/about/releases

Harmadik fél által fejlesztett könyvtárak

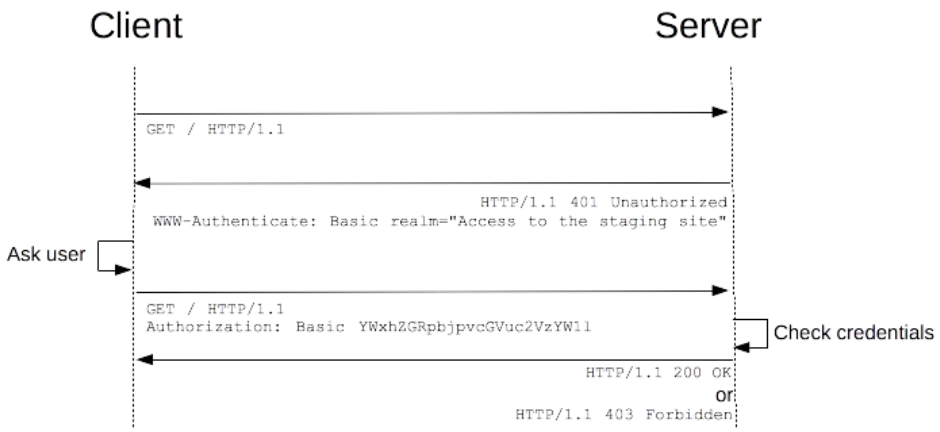
Harmadik fél által fejlesztett könyvtárak és csomagok esetében a biztonsági sérülékenységeket biztonsági audit segítségével tudjuk felderíteni. A Node.js csomagkezelőjének, az npm-nek van beépített audit funkciója, amelyet lefuttatva kilistázza nekünk a veszélyesnek ítélt könyvtárakat, valamint csoportosítja is azokat veszélyesség szerint. Hosszú távú fenntartás esetében rendszeresen, például havonta le kell futtatni egy audit ellenőrzést. A talált sérülékenységeket meg kell szüntetni az adott könyvtár verzióléptetésével, vagy lecserélésével.

Hosszú távú projekttámogatás esetében havi rendszerességgel le kell futtatni audit ellenőrzést! Egy audit ellenőrzés után nem maradhat kritikus sebezhetőség!

Autorizáció, autentikáció

Egy webalkalmazás működésében kiemelt fontosságú a megfelelő autorizáció és autentikáció. A legtöbb alkalmazásnak vannak olyan részei, melyek elérése felhasználókhoz vagy azok csoportjához rendelt különböző szabályok függvénye. Az autorizáció és az ehhez szükséges autentikáció lehetővé teszi a védett tartalmak jogosultságokhoz kötött kezelését.

Az autentikáció felhasználó azonosításának a folyamata.



A webes autorizáció általános folyamata a következő lépésekből áll:

1. **Felhasználó azonosítása:** A felhasználók jelentkeznek be az alkalmazásba egy felhasználói név és jelszó segítségével, vagy más azonosítási módokkal, például a közösségi média fiókokkal vagy OAuth protokollal.
2. **Hitelesítés:** Az alkalmazás ellenőrzi, hogy a felhasználó által megadott azonosító és jelszó helyesek-e. Ha igen, akkor a felhasználó azonosítva van és hozzáférést kap az alkalmazáshoz.
3. **Jogosultság ellenőrzése**: Az alkalmazás ellenőrzi a felhasználói fiók jogosultságait, hogy eldöntse, milyen tartalmakat és funkciókat láthatnak vagy használhatnak a felhasználók.

4. **Engedélykérés:** Ha a felhasználó megpróbál hozzáférni egy védett tartalomhoz vagy funkcióhoz, amelyhez nincs jogosultsága, az alkalmazás kérheti a felhasználó engedélyét, vagy automatikusan visszautasíthatja a hozzáférést.
5. **Engedélyezés:** Ha a felhasználónak megfelelő jogosultsága van, az alkalmazás engedélyezi a hozzáférést a tartalomhoz vagy a funkcióhoz.

JSON Web Token (JWT)

Webalkalmazások esetén az egyik leggyakrabban használt megoldás az autorizáció és autentikáció folyamatára a JSON Web Token-ek, vagy JWT-k.

- **Kompaktabb**: A JSON kevésbé bőbeszédű, mint az XML, így kódolva a JWT kisebb, mint egy SAML-token. Ez jó választássá teszi a JWT-t HTML és HTTP környezetben való átadáshoz.
  Forrás: https://auth0.com/docs/secure/tokens/json-web-tokens#benefits
- **Biztonságosabb**: A JWT az aláíráshoz használhat publikus és privát kulcspárt X.509 tanúsítvány formájában. A JWT szimmetrikusan aláírható egy megosztott kulcs segítségével a HMAC algoritmus használatával. És bár a SAML-tokenek használhatnak publikus és privát kulcspárt, például a JWT-t, az XML aláírása XML Digitális Aláírás megvalósítása biztonsági rések nélkül nehézkes a JSON aláírásának egyszerűségéhez képest.
- **Elterjedtebb**: A JSON felolvasók a legtöbb programozási nyelvben elterjedtek, mivel közvetlenül az objektumként vannak leképezve. Ezzel szemben az XML-nek nincs megfelelő dokumentum-objektum leképezése. Ez megkönnyíti a JWT-vel való munkát, szemben az SAML definíciókkal.
- **Könnyebben feldolgozható**: A JWT használata elterjedt. Ebből következik, hogy egyszerűbb a feldolgozása a felhasználó eszközein, különös tekintettel a mobil telefonokra.

JWT használata

- **Autentikáció:** Amikor egy felhasználó sikeresen bejelentkezik az azonosító adataival, egy azonosító token kerül visszaadásra. Az OpenID Connect (OIDC)

specifikációk szerint az azonosító token mindig JWT formátumban kerül elkészítésre.

- **Autorizáció**: Sikeres bejelentkezést követően az alkalmazás engedélyt kérhet arra, hogy az adott felhasználó nevében hozzáférjen útvonalakhoz, szolgáltatásokhoz vagy erőforrásokhoz (pl. API-k). Ehhez minden kérésben át kell adni egy hozzáférési tokent, amely lehet JWT. Az Egyszeri Bejelentkezés (SSO) széles körben használ JWT-t, mert a formátum kis mérete miatt költséghatékony, és könnyen használható különböző tartományok között.
- **Információcsere**: A JWT egy biztonságos módja az információk átvitelének a felek között, mivel aláírhatók, ezzel garantálva a küldő fél identitását. Ezen felül a JWT struktúrája lehetővé teszi a tartalom eredetiségének biztosítását, kizárva a manipuláció lehetőségét.

A JWT használata ajánlott.

Kulcsok

A privát és nyilvános kulcsokat a szerverhez közeli tárolóban kell tárolni, és csak a szükséges alkalmazásoknak kell hozzájuk férni.

Jelszóval való védelem: a privát kulcsot jelszóval kell védeni, hogy csak az engedélyezett felhasználók tudják használni.

A privát kulcsot lehet titkosítani, hogy megakadályozza annak visszaállítását, ha szükséges.

Jogosultságokkal való védelem: a privát kulcsot csak azok a felhasználók és alkalmazások használhatják, akik számára a szerveren engedélyezett a hozzáférés.

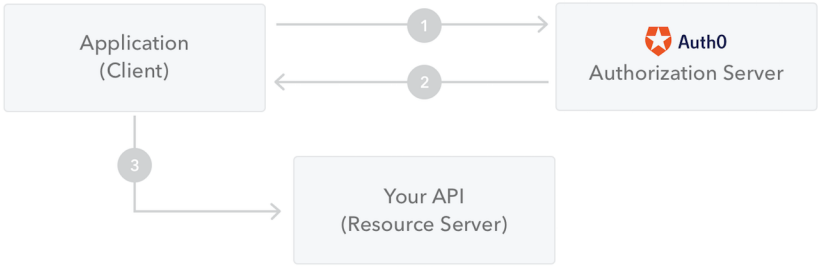
Másodlagos hitelesítés: az alkalmazásoknak és felhasználóknak további hitelesítési folyamatot kell végrehajtaniuk mielőtt hozzáférhetnének a privát kulcshoz, például hitelesítő kódokat, hitelesítő kártyákat, vagy biometrikus hitelesítést.

A privát kulcsot tároló szervert lehet tűzfallal védeni, hogy csak az engedélyezett IP címekről lehessen hozzáférni.

A privát kulcsról érdemes biztonsági mentést készíteni, hogy biztosítva legyen a helyreállítás lehetősége.

A JWT payload-jában nem szabad szenzitív adatokat tárolni.

Auth token



Példa:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

```
Header: {"alg": "HS256", "typ": "JWT"}
Payload: {"sub": "1234567890", "name": "John Doe", "iat": 1516239022}
Signature: SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

A header tartalmazza az információkat arról, hogy melyik titkosítási algoritmus van használatban a jel aláírásához, míg a payload tartalmazza az információkat a felhasználóról, mint például a felhasználó azonosítóját és nevét. A signature pedig a header és a payload hash értékét tartalmazza, amely privát kulcsal van titkosítva, így az alkalmazás a publikus kulcsával ellenőrizheti az érvényességét.

A JWT biztonságos tárolása

- Be kell állítani a „HttpOnly" és „Secure" opciókat a cookie-ban. A „HttpOnly" lehetővé teszi, hogy a cookie-t csak HTTP kérésből olvashassa, míg a „Secure" csak HTTPS kapcsolatokon keresztül engedélyezi a cookie elérését.
- Meg kell adni az access tokenek érvényességi idejét a cookieban.
- Óvatosnak kell lenni a privát kulcsok tárolásával. Csak a szükséges személyek férjenek hozzá, és biztonságos helyen kell őket tárolni.

Tiltott a jelszavak szabad szövegként való tárolása!

Application Programming Interface (API)

Bevezetés

Az API egy olyan protokoll-, rutin- és eszközkészlet, amely lehetővé teszi alkalmazások felépítését és a különböző rendszerek közötti kommunikációt.

A legismertebbek: REST, SOAP, GraphQL, WebSockets.

Az API egyfajta kommunikációs hídként működik a front-end és a back-end között. Az egyik legismertebb API típus a REST. A következőkben erről lesz szó.

REST API

REST (Representational State Transfer) használata esetén néhány dologra oda kell figyelni a fejlesztés hatékonysága érdekében.

Állapotmentesség

> A REST szolgáltatásoknak állapotmentesnek kell lenniük.

Tehát minden kérésnek tartalmaznia kell azokat az információkat, amire a szervernek szüksége van a döntéshozatalhoz.

Cache-elhetőség

> A REST szolgáltatások kérésének eredményeinek cache-elhetőnek kell lenniük.

Ezáltal a kliens gördülékenyebben tudja használni a szolgáltatást.

Idempotencia

A REST metódusoknak a POST metóduson kívül idempotensnek kell lenniük.

Ez azt jelenti, hogy ugyanazt a kérést többször elküldve ugyanazt az eredményt kell adniuk.

HTTP címek

A REST szolgáltatásoknak törekedniük kell a különböző HTTP metódusok használatára, hogy a szolgáltatások érthetőek és kezelhetőek legyenek.

A kommunikáció megvalósításakor a HTTPS használata biztonsági okok miatt erősen ajánlott.

URL konvenció

A RESTful URL erőforrásokra (főnevek) mutat, kerülni kell a cselekvést kifejező szavak (igék) használatát.

Túlzott állapot kódok használata

Kerülni kell a túlzott állapot kódok használatát, csak akkor használhatóak amikor ténylegesen szükségesek.

Biztonság

Gondolni kell az API biztonságára, védelemre van szükség a közös támadások ellen, mint például az „injection” és a „spoofing” támadások.

Teljesítmény

Biztosítani kell, hogy az API képes legyen magas forgalom kiszolgálására.

Szabályok, és ajánlások

Az API-nak HTTP metódusokat kell használnia, mint például a GET, POST, PUT, és DELETE, amelyeket a HTTP protokoll meghatároz.

Az HTTP (Hypertext Transfer Protocol) metódusok meghatározzák, hogy milyen műveletet hajtsunk végre egy adott URI-n. A REST API-k esetén az alábbi metódusokat használják leggyakrabban:

1. **GET:** A GET metódus segítségével lekérdezhetjük az adott URI-n található információkat.
2. **POST:** A POST metódus segítségével új adatot adhatunk hozzá az adott URI-hoz.
3. **PUT:** A PUT metódus segítségével módosíthatjuk az adott URI-n található adatokat.
4. **PATCH:** A PATCH metódus segítségével módosíthatjuk az adott URI-n található adatok egy szűkebb részét.
5. **DELETE:** A DELETE metódus segítségével törölhetjük az adott URI-n található adatokat.

A GET metódusnak nem szabad változtatnia az adatokat, csak lekérdezni őket.

A HTTP metódusok használatának jól kell illeszkedni a rendszer működéséhez, és segíteni kell a rendszer általános érthetőségét.

Az API-nak az URI-k (Uniform Resource Identifier) használatával kell azonosítania a forrásokat.

Az API-nak a kérés és a válasz adatainak JSON formátumban kell lennie.

Az API-nak indokolt esetben biztosítania kell a kérés autentikációját és autorizációját, hogy megakadályozza a véletlen vagy rosszindulatú hozzáférést.

Az API-t dokumentálni kell!

Ajánlott az API struktúráját leíró technológia használata, mint Swagger vagy APIDoc.

A Swagger nagyon elterjedt, könnyen kezelhető és átlátható dokumentációt készít, ezért ennek a használata az ajánlott.

HTTP Cache

Alapvetően három féle Cache-elést tartunk számon.

- Kliens Oldali Cache
- Szerver Oldali Cache
- Proxy Cache

A HTTP metódusok közül kettőt cache-elhetünk.

- GET metódus - Ezeket a böngészők a válaszban beállított Cache-Control HEADER érték szerint alap esetben letárolja.
- POST metódus - Alapértelmezetten nem történik Kliens oldali mentés, de igény szerint, ha szükséges implementálható.

A többi, PUT, PATCH és DELETE metódusok nem cache-elhetőek.

Kliens Oldali Cache

A kliens oldali cache-elést a böngésző végzi. Amikor egy kérés elmegy a szerver felé a válasz Cache-Control header értékének megfelelően a böngésző tárolja a tartalmat. Továbbiakban egészen addig, amíg le nem jár a tárolt válasz cache-elés ideje az újonnan, ugyan arra az URL-re küldött kérés a böngésző cache tárából kap választ. Ebben az esetben a kiszolgáló szerverig el sem jut a kérés.

Ezeket a cache-eket kliens oldalról a kérés megfelelő HEADER adatainak beállításával bármikor újra validálhatjuk, bizonyos esetekben kikényszeríthetjük a cache-felülírását.

Egyes statikus fileok esetében (CSS, JS, Dokumentumok) érdemes a file-ok elérését kiegészíteni egy verzióval vagy időbélyeggel. Példa.: `main.js?v=123`. Ilyenkor egészen addig amíg nem változtatjuk meg a verziót az első kérést követő összes többi esetben ugyan azt a tartalmat kapjuk vissza, de a böngésző cache tárából.

Előnye a megfelelő kliens oldali cache használatnak, hogy a nagyobb forgalmú oldalak esetében felesleges kérések nem jutnak el a kiszolgáló szerver felé, ezzel is csökkentve a terhelést.

Szerver Oldali Cache

Működési elve hasonló a kliens oldali működéshez. Ezeket valamilyen Reverse Proxy végzi (pl.: NGINX). A megfelelő Szerveroldali cache beállítások esetében az alkalmazás egyéb komponensei felé nem irányul új kérés, hanem a Reverse Proxy a tárolt adatokból adja vissza a választ. Így ismételten csökkenthetjük a szerver terheltségét és az alkalmazás válaszidejét.

Érdemes Szerver oldalon beállítani az egyes végpontok válaszának megfelelő lejárattal cache-elési időket. Két féle típust különböztetünk meg a statikus és a dinamikus válaszokat.

Statikus válaszok esetében (pl.: Képek, PDF-ek és hasonlók) amik semennyire vagy ritkán változnak, a cache lejárata lehet hosszabb, akár napok vagy hetek.

Dinamikus tartalmak esetén, amik nagyon sűrűn változnak (például egy fórum esetében a kommentek) nehéz megfelelő lejáratot beállítani. Ezekben az esetekben nem ajánlott az ilyen tartalmakat cache-elni.

Viszont, ha egy tartalom változást tudunk explicit időhöz kötni, hogy mondjuk naponta egyszer frissül egy megadott időpontban, akkor érdemes a cache lejártát úgy időzíteni, hogy az a tartalmi frissítést követően járjon le. Ebben az esetben elkerüljük, azt, hogy a felhasználóknak ne az aktuális friss tartalom jelenjen meg.

Proxy Cache

A Kliens és a Szerver között meghúzódó rétek, ami szintén cache-elhet adatokat.

Privát és Publikus Cache

A válaszok esetében megkülönböztethetünk még Privát és Publikus cache-eket is. Publikus cache mindenki számára is elérhető cache. Privát cache pedig csak egy felhasználó/kliens felé szánt cache. Amennyiben olyan adatokat cache-elünk, ami szenzitív egy felhasználó felé, akkor minden esetben Privátként kell megjelölni, hogy más véletlenül se férjen hozzá.

Tömörítés

HTTP esetén lehetőség van a válaszok tömörítésére, ezt a kiszolgáló szerverek esetében érdemes beállítani. A ma ismert legtöbb böngésző támogatja ezeknek a feldolgozását.

Ismertebb tömörítési eljárások:

- Brotli - Speciálisan a HTTP tartalmak kódolására szolgáló algoritmus
- Gzip – Egyik jelenleg is legelterjedtebb tömörítési algoritmus, amit használnak.

Előnye, hogy a kérések és a válaszok tartalma a kliens és a szerver között (tartalomfüggő) jóval kisebb mérettel rendelkezik és ez által a gyorsabb betöltést tesz lehetővé.

Mobil alkalmazások

Mikor ajánlott mobil alkalmazás?

A webalkalmazás és az esetlegesen ahhoz tartozó mobil alkalmazás között a tervezés szempontjából vannak közös kapcsolódási pontok, amelyekre ügyelni szükséges.

Az alkalmazásfejlesztés során az egyik legfontosabb kérdés, hogy milyen típusú alkalmazást érdemes készíteni? Mobil alkalmazás vagy webalkalmazást?

Általában mobil alkalmazást akkor ajánlott fejleszteni, ha igény szerint szükség van az eszközök által nyújtott funkciókra, mint például:

- GPS
- kamera
- érintőképernyő
- mozgásérzékelők
- push-értesítések

Emellett a mobil alkalmazások általában gyorsabbak, mint a webalkalmazások, mivel az nagy részük a felhasználói eszközön kerülnek tárolásra, így nem kell - feltétlen - folyamatosan kapcsolódniuk az internethez.

A webalkalmazásokat általában akkor érdemes használni, ha az alkalmazásnak nincs szüksége az eszközök által nyújtott funkciókra.

Átjárhatóság a mobil alkalmazás és webalkalmazás között

A mobil alkalmazások és webalkalmazások csatlakozási pontjai lehetnek az adatbázisok, a szerverek és az API-ok (Application Programming Interface). Ezek a pontok lehetővé teszik a mobil alkalmazások és a weboldalak számára, hogy kapcsolódjanak a szükséges adatforrásokhoz és szolgáltatásokhoz.

Az API-ok lehetővé teszik a különböző szolgáltatások és adatforrások integrálását a mobil alkalmazások és a weboldalak frontendjébe. Az API-ok használatával a

különböző alkalmazások könnyen csatlakoztathatóak a harmadik féltől származó szolgáltatásokhoz és adatforrásokhoz, például fizetési rendszerekhez, térképekhez, közösségi média platformokhoz.

A mobil alkalmazások és a webalkalmazások csatlakozási pontjai jelentős biztonsági kockázatot jelenthetnek. További részletek a Biztonság című fejezetben találhatóak.

Mind a mobil-, mind a webalkalmazások tekintetében kiemelt figyelmet szükséges fordítani az adatvédelemre, a hitelesítésre, és az autentikációra.

Natív vagy Flutter (cross-platform)

Mobil alkalmazás fejlesztésnél az első lépés, hogy eldöntsük milyen technológiát használjunk. A paletta széles: léteznek natív-, hybrid-, cross platform technológiával készült alkalmazások, és PWA-k.

Ha a biztonság, a széles támogatottság, az időtállóság elsődleges szempont, akkor natív fejlesztést szükséges választani. Ha a gyors fejlesztési idő és a költséghatékonyság az elsődleges, akkor a Flutter fejlesztői keretrendszert szükséges használni.

Az aktuális iparági trendek alapján cross-platform fejlesztésre a Google Flutter keretrendszere a jövőbiztos megoldás.

Mobil alkalmazás fejlesztéséhez kizárólag natív (iOS, Android) megoldást vagy Flutter keretrendszert szükséges választani!

Natív alkalmazás

Előnyök

- **Teljesítmény:** A natív alkalmazásokat az adott operációs rendszerre tudjuk optimalizálni, valamint rendelkezésünkre állnak a platform által használt API-k

és SDK-k, ezért gyorsabb és reszponzívabb felhasználói élményt tudunk biztosítani.

- **Biztonság:** A natív SDK-k és API-k számára azonnal elérhetővé válik minden biztonsági frissítés és javítás, amelyek azonnal integrálhatók az alkalmazásokba. Lehetőség van a platform által biztosított összes biztonsági funkció használatára.
- **Felhasználói élmény:** A natív alkalmazások öröklik a platform által használt felületi elemeket, így konzisztens felhasználói élményt tudnak nyújtani.

Hátrányok

- **Költség:** A natív alkalmazások fejlesztése költségesebb. Általában külön fejlesztői csapatok dolgoznak a különböző platformokon.
- **Fejlesztési idő:** Mivel minden funkciót külön kell implementálni a különböző platformokra, ezért a fejlesztési idő is lényegesen hosszabb. Hasonlóan a karbantartás is időigényesebb.
- **Kód újrahasználhatóság:** A különböző platformok között nincs egyértelmű átjárhatóság, így a funkciókat többször kell implementálni.

Architektúra

Az ismertetett architektúra a Google hivatalos ajánlása az Android alkalmazásokra tekintve. Viszont más platformokra fejlesztett mobil alkalmazások esetén is bevett szokás az ilyen jellegű tervezés.

UI layer

A UI réteg feladata az adatok megjelenítése, illetve ez a fő interakciós lehetőség a felhasználó számára. Két fő részből épül fel, a felhasználói felületből (elements) és a hozzá tartozó állapotból (state).

A tervezés során a megjelenítést teljesen függetlenné kell tennünk az üzleti logikától. Ennek elérésére a UI réteg a következő lépéseket hajtja végre:

1. Az adat átalakítása egy állapottá, amely már megjeleníthető a felhasználó számára.
2. Felületi elemek elkészítése az állapot alapján.
3. A felhasználó által végrehajtott műveletek feldolgozása, állapot frissítése.
4. 1-3. pontok ismétlése amennyiben szükséges.

Forrás: https://developer.android.com/topic/architecture/ui-layer

UI state

A UI state változásai azonnal megjelennek a felhasználó számára is. A felhasználói felület közvetlenül nem változtatja az állapotot, az adatforrás feladata egy új állapot létrehozása. Ezt biztosítandó, az állapot immutable.

```
data class NewsUiState(
    val isSignedIn: Boolean = false,
    val isPremium: Boolean = false,
    val newsItems: List<NewsItemUiState> = listOf(),
    val userMessages: List<Message> = listOf()
)

data class NewsItemUiState(
    val title: String,
    val body: String,
    val bookmarked: Boolean = false,
    ...
)
```

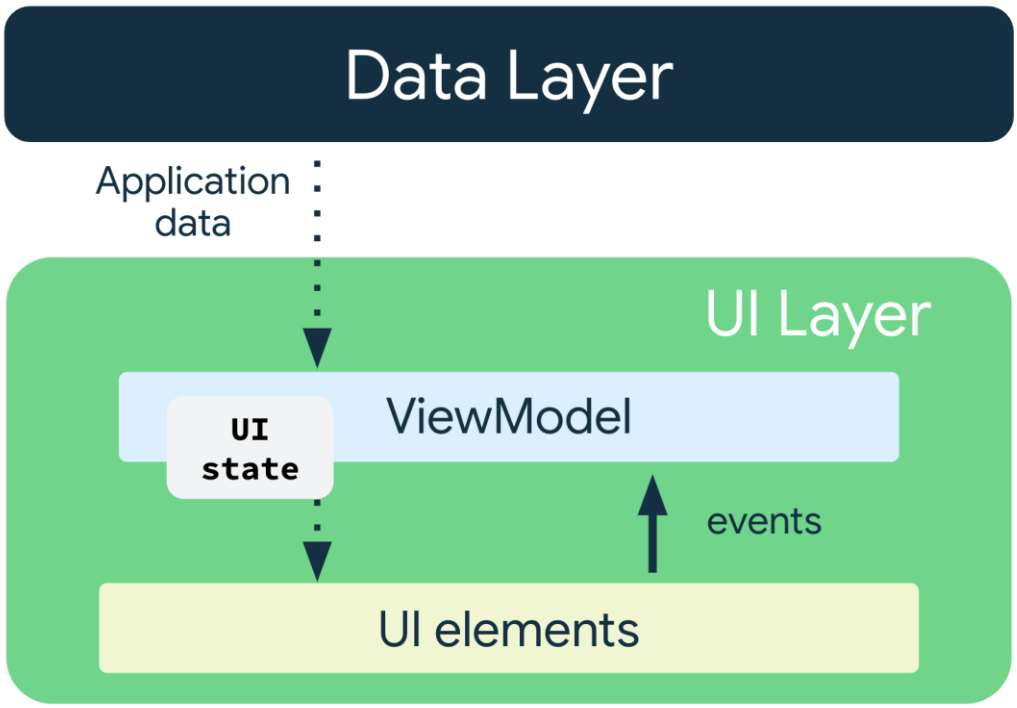
```
struct NewsUiState {
    let isSignedIn: Bool,
    let isPremium: Bool,
    let newsItems: [NewsItemUiState]
    ...
}

struct NewsItemUiState {
    let title: String,
    let body: String,
    let bookmarked: Bool
}
```

Forrás: https://developer.android.com/topic/architecture/ui-layer#define-ui-state

State Holder

Az állapotok létrehozásáért és tárolásáért az úgynevezett *state holder* felelős. Az alkalmazás képernyői esetében általában a ViewModel látja el a state holder feladatait.

A UI és a ViewModel között egy kétirányú kapcsolat áll fenn. A UI a felhasználói interakciók hatására eseményeket küld a ViewModel-nek, mely ezek hatására frissíti az állapotot.



Ezt egyirányú adatáramlásnak (unidirectional data flow - UDF) hívjuk.

Képernyű szintű állapotok esetében a ViewModel ajánlott mint state holder.

Csak a state holder változtathatja az állapotot, ezzel garantálva az adatok hitelességét.

Forrás: https://developer.android.com/topic/architecture/ui-layer/stateholders

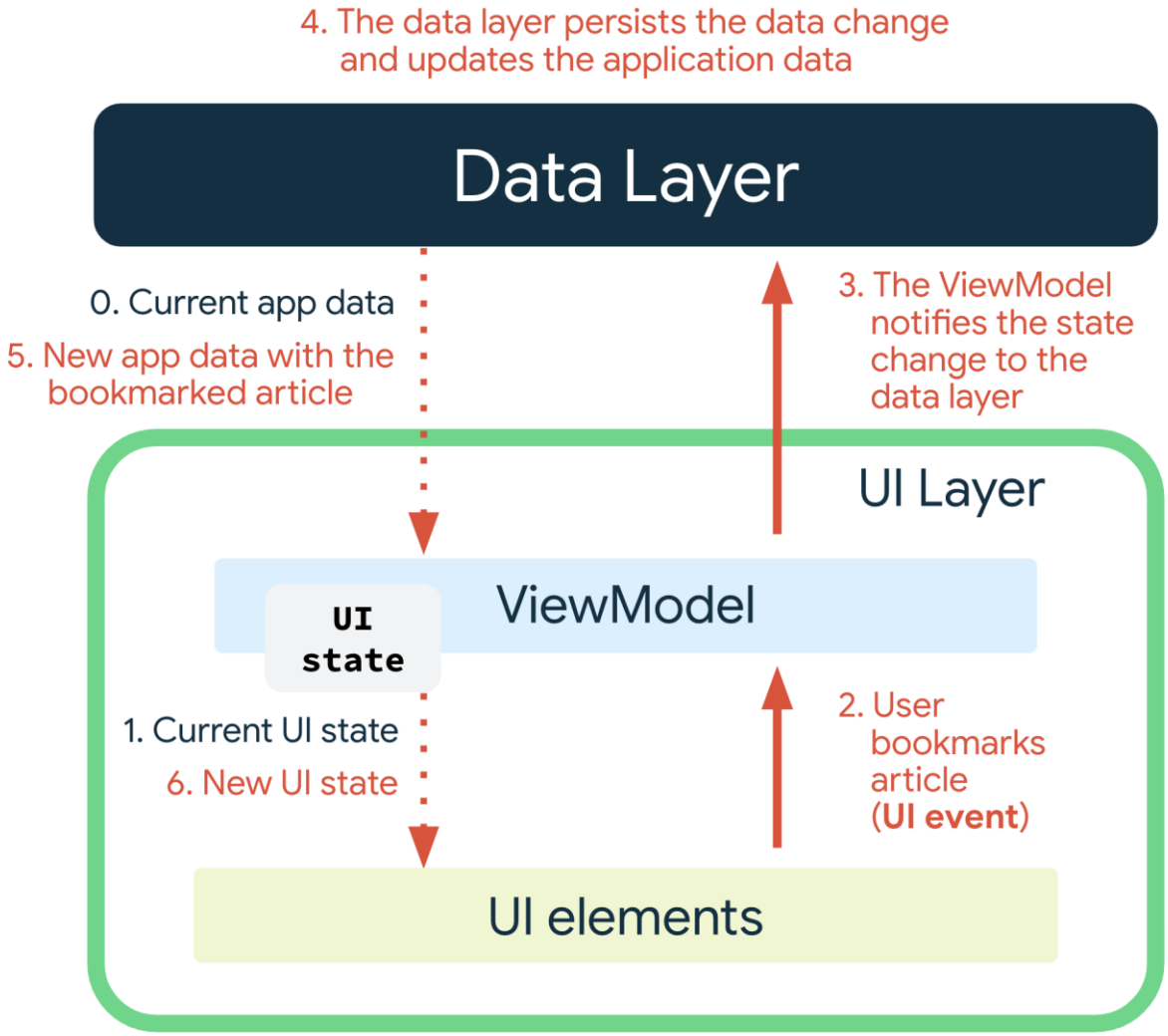
UDF

UDF esetén az események felfelé, míg az állapotok lefelé mozognak. Az alkalmazás architektúrában ez a következőt jelenti:

- A ViewModel tartalmazza és a UI számára elérhetővé teszi az állapotokat. Az alkalmazás adatok feldolgozása a ViewModel-ben történik.
- A UI eseményeket küldd a ViewModel számára a felhasználói interakciókról.

- A ViewModel kezeli az eseményeket és frissíti az állapotot.
- A frissített állapot visszakerül a UI-hoz, mely megjeleníti azt.

UDF segítségével biztosíthatjuk az adatok hitelességét, a ViewModel az egyetlen igazságforrás.



Az eseményeket mindig a UI kezdeményezi és küldi a ViewModel számára. Az állapotot a UI mindig a ViewModel-től kapja.

Forrás: https://developer.android.com/topic/architecture/ui-layer#udf

Állapot átadás

Az állapotból minden felhasználói esemény hatására egy új verzió fog készülni. Erre azért van szükség, hogy a UI egyértelműen tudjon reagálni az állapot változásaira. Az állapotot egy reaktív struktúrába célszerű csomagolni, tovább erősítve a kapcsolatot a state holder és a hozzá tartozó képernyő között.

```
class NewsViewModel(...) : ViewModel() {

    var uiState by mutableStateOf(NewsUiState())
        private set

    ...
}
```

```
class NewsViewModel: ObservableObject {
    @Published private var state = NewsUiState()
}
```

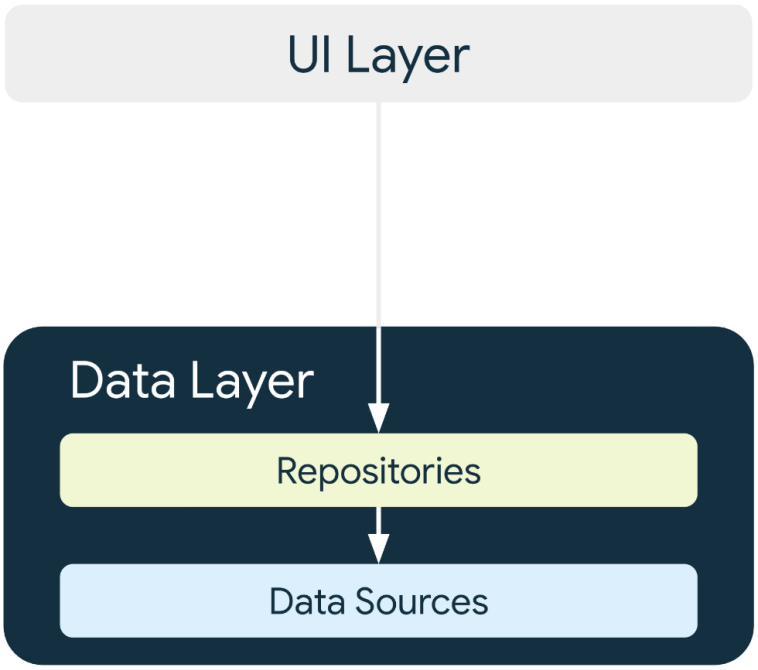
A ViewModel elérhetővé kell tegye a metódusokat, melyek módosíthatják az állapotot.

> Amennyiben egy képernyőn több, egymástól teljesen független állapotot is meg kell jeleníteni, célszerű ezeket külön folyamként kezelni.

Forrás: https://developer.android.com/topic/architecture/ui-layer/state-production

Data layer

A data layer-en belül a Repository tervezési minta használata ajánlott. Célszerű adattípusonként (API függőség miatt, amennyiben nem lehetséges képernyőnként) külön repository-t készíteni. Ezek tartalmazzák a szükséges adatforrásokat.



Forrás: https://developer.android.com/topic/architecture/data-layer

Repository

Az alkalmazás a repository osztályokon keresztül jut adathoz. Minden módosítás is itt történik, így mindig a legfrissebb adatból dolgozik az alkalmazás. A különböző források közötti eltéréseket a repository-ban található üzleti logikának kell feloldania.

Az alkalmazás adatot csak a Repository-n keresztül kap.

A szolgáltatott adat mindig immutable, csak a repository változtathatja.

Dependecy Injection esetén a repository a függőségeket konstruktor paraméterként kapja.

Repository elnevezése
adattípus + Repository

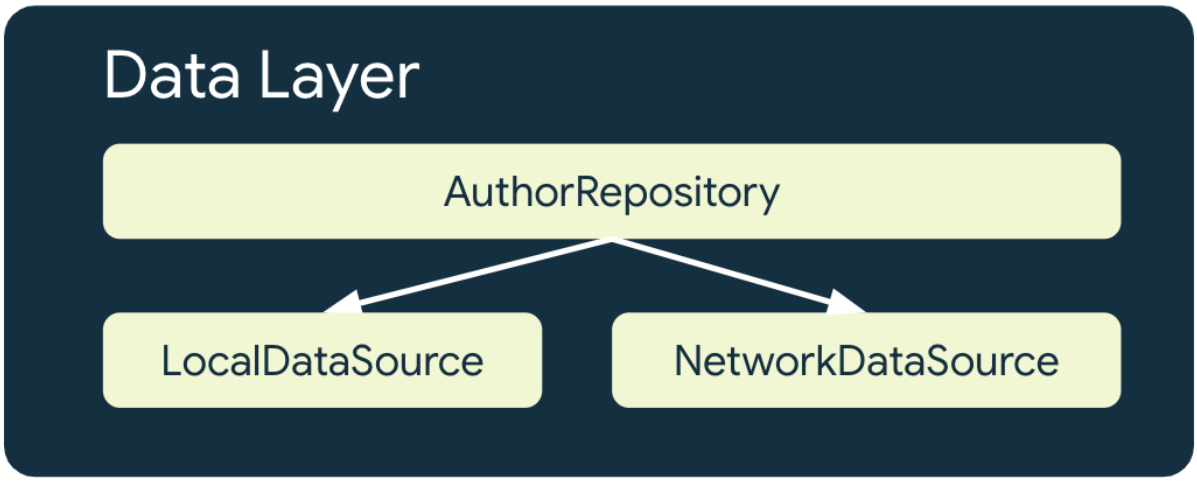
Adatforrás

Minden adatforrás csak egy helyről nyer adatot (pl. REST API, adatbázis), amennyiben több forrást is használunk, akkor a repository felelőssége ezeket összekapcsolni. A forrást csak a repository használhatja, az alkalmazás többi része nem. Adatforrások elnevezése az *adattípus + forrás típus + DataSource* minta alapján.

Adatforrások elnevezése
adattípus + forrás típus + DataSource

Offline first

Az offline first alkalmazás egy olyan alkalmazás, amely a funkcionalitása nagy részét vagy egészét internet kapcsolat nélkül is képes végrehajtani. Offline first szemlélet esetén a data layer-nél van lehetőségünk szétválasztani az adatok elérését.



Ebben az esetben a repository egy lokális adatbázisból, illetve az internetről is tud adatot gyűjteni. Ebben az esetben mindig a lokális adatbázis lesz az egyetlen igazság forrás, az alkalmazás először távolról lekéri az adatot, ezt eltárolja a lokális adatbázisban, majd onnan teszi elérhetővé az alkalmazás számára.

Az adatok feltöltése során különösen figyelni kell a verseny kondíciók feloldására.

Felhasználói élmény javítása érdekében az alkalmazás megjelenítheti a mentett állapotot, majd a lekérés befejezése után frissítheti azt.

Forrás: https://developer.android.com/topic/architecture/data-layer/offline-first

REST

Android

REST API kezelésére Androidon a Retrofit library használata a legelterjedtebb.

API deklarálás

Az API egy interface segítségével deklarálható. Az interfész metódusai felelnek meg egy API hívásnak. Annotációk segítségével adhatjuk meg az API viselkedését, minden REST metódushoz rendelkezésünkre áll egy annotáció.

```
interface NewsApi {

    @GET("news")
    suspend fun getNews(): List<News>

}
```

A Retrofit önmagában kezeli az API hívások háttér szálon futtatását, valamint együtt tud működni a Kotlin Coroutines-szal.

Az így készült API leírásból a Retrofit osztály kódgenerálással készít egy implementációt.

Az interfész metódusoknak az API relatív elérési útját kell megadni, az API címének átadása a Retrofit példányosításakor történik.

```
val retrofit = Retrofit.Builder()
    .baseUrl("myapi.com/")
    .build()

val newsApi = retrofit.create(NewsApi::class)
```

DataSource

A létrehozott API-t a DataSource konstruktor paraméterként kapja meg.

```
class NewsNetworkDataSource(newsApi: NewsApi) {
    suspend fun getNews(): List<News> {
        return newsApi.getNews()
    }
}
```

Header

Statikus header-t az interfész metódus annotálásával van lehetőségünk adni.

```
@Headers("Cache-Control: max-age=640000")
@GET("news")
suspend fun getNews(): List<News>
```

Lehetőségünk van dinamikusan is beállítani header-t a híváshoz, ekkor paraméterként tudjuk átadni.

```
suspend fun getNews(@Header("Authorization") String authorization): List<News>
```

Converter

A Retrofit alapértelmezetten csak OkHttp RequestBody típusra tudja feldolgozni a hívás eredményét. Más típus támogatásához Converter-t kell használni. A leggyakrabban használt Converter-ek rendelkezésünkre állnak, pl. JSON esetén:

```
val retrofit = Retrofit.Builder()
    .baseUrl("myapi.com/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Interceptor

Az Interceptor segítségével az API hívásokat lehet monitorozni, módosítani vagy akár újra próbálni. Például, ha minden API híváshoz szükséges az autentikáció, akkor célszerű ezt Interceptor-ral megvalósítani.

```
class AuthInterceptor : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request()

        val newRequest = request.newBuilder()
            .addHeader("Authorization", API_TOKEN)
            .build()

        return chain.proceed(newRequest)
    }
}
```

Forrás: https://square.github.io/retrofit/

iOS

iOS esetén a network kezelésre a beépített URLSession osztály áll rendelkezésünkre.

URLSession

UrlSessionTask-ok segítségével lehetőségünk van távoli szerverről adatot kérni. A URLSessionDataTask a kapott adatot memóriában tárolja el, míg az URLSessionDownloadTask az adatot közvetlenül a fájlrendszerre írja.

REST hívásokhoz az URLSessionDataTask használata javasolt.

```
struct NewsApi {
    let baseUrl = "myapi.com/"
    var urlSession = URLSession.shared

    func fetchNews(completionHandler: @escaping [News] -> Void) {
        let url = URL(string: baseUrl + "/news")

        var request = URLRequest(
            url: url,
            cachePolicy: .reloadIgnoringLocalCacheData
        )
        request.setValue(
            "authToken",
            forHTTPHeaderField: "Authorization"
        )

        request.httpMethod = "GET"

        let task = urlSession.dataTask(
            with: request,
            completionHandler: { data, response, error in
                ...
            }
        )
    }
}
```

Egyszerűbb hívásokhoz használható a beépített shared URLSession, komplex hívások (pl. autentikáció, redirect) esetén célszerű saját session létrehozása delegate segítségével.

Forrás
https://developer.apple.com/documentation/foundation/urlsession
https://developer.apple.com/documentation/foundation/urlsessiontaskdelegate

Perzisztencia

Android

Preferences DataStore

A Preferences DataStore kulcs-érték párok tárolására alkalmas. Használatához nincs szükség séma definiálására.

```
// At the top level of your kotlin file:
val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "settings")
```

Mivel Preference DataStore esetében nincs előre definiált séma, így a megfelelő típusos írás-olvasás funkciókat kell használni.

```
val EXAMPLE_COUNTER = intPreferencesKey("example_counter")
val exampleCounterFlow: Flow<Int> = context.dataStore.data
  .map { preferences ->
    // No type safety.
    preferences[EXAMPLE_COUNTER] ?: 0
}
```

A DateStore edit() metódusa tranzakciónként módosítja a tartalmát. A lambda tartalmát egy tranzakciónak tekintjük.

```
suspend fun incrementCounter() {
  context.dataStore.edit { settings ->
    val currentCounterValue = settings[EXAMPLE_COUNTER] ?: 0
    settings[EXAMPLE_COUNTER] = currentCounterValue + 1
  }
}
```

Forrás: https://developer.android.com/topic/libraries/architecture/datastore#preferences-datastore-dependencies

Proto DataStore

A Proto DataStore segítségével típusos adatok tárolására van lehetőség. Használatához séma definíció szükséges, melyet az app/src/main/proto mappában kell elhelyezni.

A sémát az app/src/main/proto mappába kell helyezni.

```
syntax = "proto3";

option java_package = "com.example.application";
option java_multiple_files = true;

message Settings {
  int32 example_counter = 1;
}
```

A definiált osztály fordítási időben generálódik, használat előtt újra kell fordítani az alkalmazást.

Használat előtt a típus séma megadásán túl további két lépést kell megtenni:

1. Le kell írnunk, hogy milyen formában szerializálható a létrehozott osztály. Ehhez implementálni kell egy Serializer-t.
2. Delegate segítségével példányosítjuk a DataStore-t.

```
object SettingsSerializer : Serializer<Settings> {
  override val defaultValue: Settings = Settings.getDefaultInstance()

  override suspend fun readFrom(input: InputStream): Settings {
    try {
      return Settings.parseFrom(input)
    } catch (exception: InvalidProtocolBufferException) {
      throw CorruptionException("Cannot read proto.", exception)
    }
  }

  override suspend fun writeTo(
    t: Settings,
    output: OutputStream) = t.writeTo(output)
}

val Context.settingsDataStore: DataStore<Settings> by dataStore(
  fileName = "settings.pb",
  serializer = SettingsSerializer
)
```

A DataStore mindig Singleton. Több hivatkozás esetén IllegalStateException dobódik.

A DataStore-t globálisan hozzuk létre.

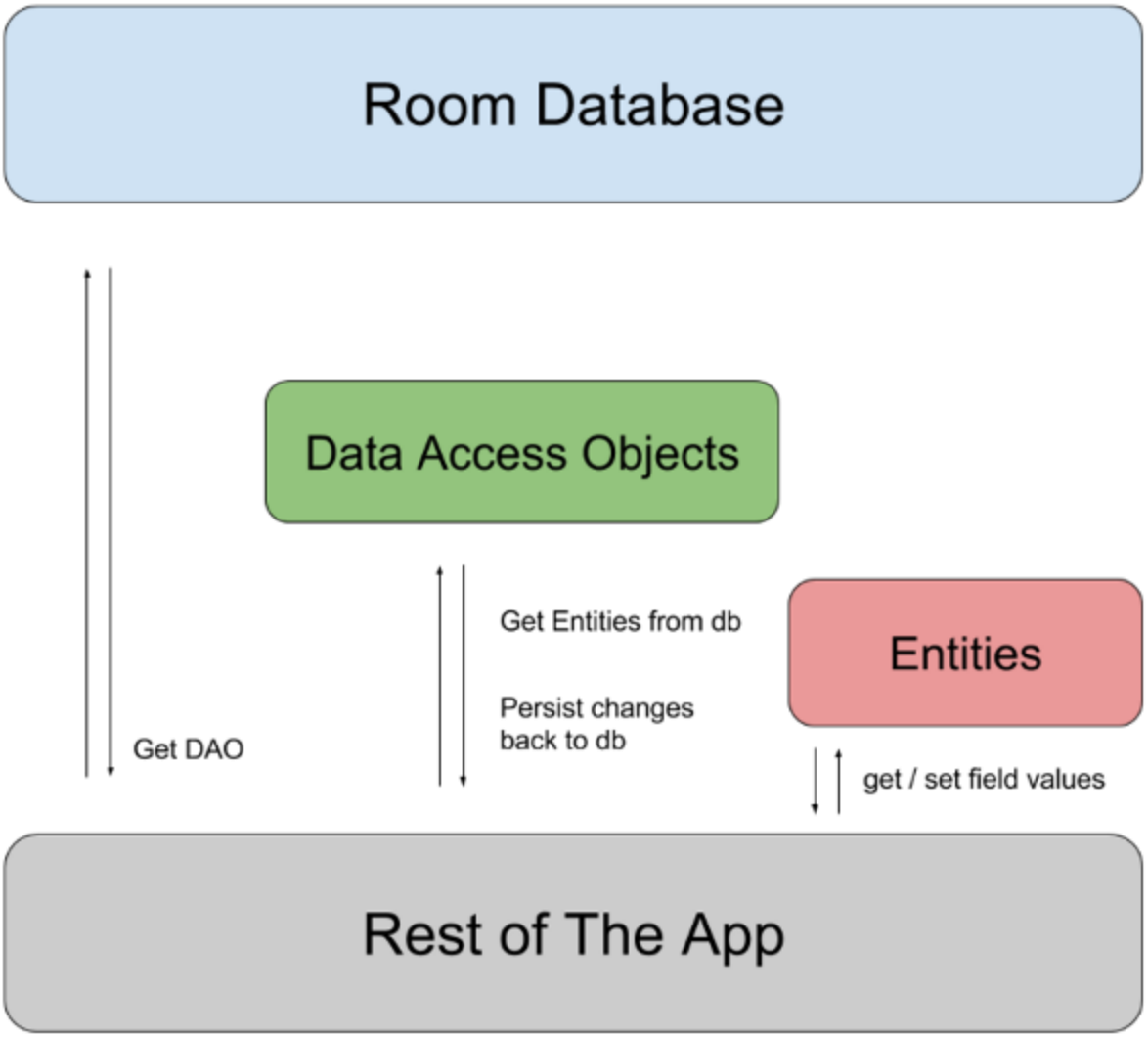
Forrás: https://developer.android.com/topic/libraries/architecture/datastore#proto-datastore-dependencies

Room

Azokban az alkalmazásokban, amelyekben nagy mennyiségű, strukturált adattal kell dolgozni, előnyt jelenthet az adatok lokális tárolása pl. network cache, offline működés. A Room könyvtár egy SQLite adatbázis absztrakció, amely megkönnyíti az adatbázis létrehozását, karbantartását.

Három fő komponensből áll:

- **Adatbázis osztály:** Tartalmazza az adatbázist és elérhetőséget biztosít a benne tárolt adatokhoz.
- **Entity:** Az adatbázisunkban lévő táblák reprezentációja.
- **DAO (Data Access Object):** Az adatbázison elvégezhető műveleteket tartalmazza.



Egy Entity-t létrehozhatunk, mint egy data class az Entity annotációval. Az osztály minden példánya az Entity-hez tartozó tábla egyik sora.

```
@Entity
data class User(
    @PrimaryKey val uid: Int,
    @ColumnInfo(name = "first_name") val firstName: String?,
    @ColumnInfo(name = "last_name") val lastName: String?
)
```

A DAO-t egy interfészként hozhatjuk létre. Metódus annotációk segítségével adhatjuk meg az adatbázis viselkedését. A főbb funkciókra elérhetőek beépített

annotációk, egyébként a Query annotáció segítségével SQLite query-k írására van lehetőség.

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<User>

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    fun findByName(first: String, last: String): User

    @Insert
    fun insertAll(vararg users: User)

    @Delete
    fun delete(user: User)
}
```

Általában minden Entity-hez tartozik egy DAO, mely az adott tábla viselkedését írja le.

Az adatbázis osztálynak meg kell felelnie a következő feltételeknek:

- Rendelkezik a Database annotációval, mely tartalmazza az Entity-k listáját.
- Abstract osztálynak kell lennie, amely a RoomDatabase osztályból származik.
- Minden DAO-hoz definiálnia kell egy paraméter nélküli abstract függvényt, amelynek a visszatérési értéke a DAO osztály.

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Az adatbázis osztály mindig Singleton.

Forrás: https://developer.android.com/training/data-storage/room

iOS

UserDefaults

A UserDefaults kulcs-érték párok tárolására alkalmas. Primitív típusok és objektumok tárolására is használható.

```
let defaults = UserDefaults.standard
defaults.set(25, forKey: "age")
defaults.set(true, forKey: "UseTouchID")
defaults.set(Date.now, forKey: "lastRun")
```

Primitív típusok esetén beépített metódusok állnak rendelkezésünkre. Objektumokat típuskényszerítéssel lehet kiolvasni. Amennyiben egy kulcshoz még nem tartozik érték, úgy primitív típusok esetén egy alapértelmezett érték kerül visszaadásra, objektumok esetén nil.

```
let age = default.integer(forKey: "age")
let useTouchId = default.bool(forKey: "UseTouchID")
let lastRun = defaults.object(forKey: "lastRun") as? Date ?? Date.now
```

> UserDefaults-ban sok adat tárolása lelassítja az alkalmazás indítását.

Forrás: https://developer.apple.com/documentation/foundation/userdefaults

Realm

A Realm egy kimondottan mobil alkalmazások számára fejlesztett objektum-tár, így nincs szükség az adatbázis működésének a definiálására. Azt kell megadnunk, hogy az osztályunk milyen attribútumai kerüljenek tárolásra.

```
class User: Object {
    @Persisted(primaryKey: true) var _id: String
    @Persisted var firstName: String
    @Persisted var lastName: String
}
```

Az adatok írásánál közvetlenül a Realm-en hajtjuk végre a tranzakciót.

```
try! Realm()
try! realm.write {
    realm.add(User())
}
```

Lehetőségünk van feliratkozni egy objektum vagy akár egy osztály változásaira is.

```
user.observe { change in
    switch (change) {
        case .change(let properties):
            for property in properties {
                print("Property '(property.name)' changed to '(property.newValue!)'");
            }
        case .error(let error):
            print("An error occurred: (error)")
        case .deleted:
            print("The object was deleted")
    }
}
```

A Realm alapértelmezetten támogatja a Query-k használatát.

```
val users = realm.objects(User.self)
let Johns = users.where {
    $0.firstName == "John"
}
```

> A Realm objektum csak arról a szálról érhető el, amelyen létrejött.

Forrás: https://www.mongodb.com/docs/realm/sdk/swift/

Erőforrás fájlok

Android

Az erőforrás fájlok a forráskódon kívüli fájlok, statikus adatok, melyeket futási időben használ az alkalmazás, pl. képek, animációk, felhasználói felület leírások stb. Ezek a fájlok a `res/` könyvtárban típus szerint csoportosítva találhatóak, a forráskódban pedig a generált `R` osztályból érhetőek el.

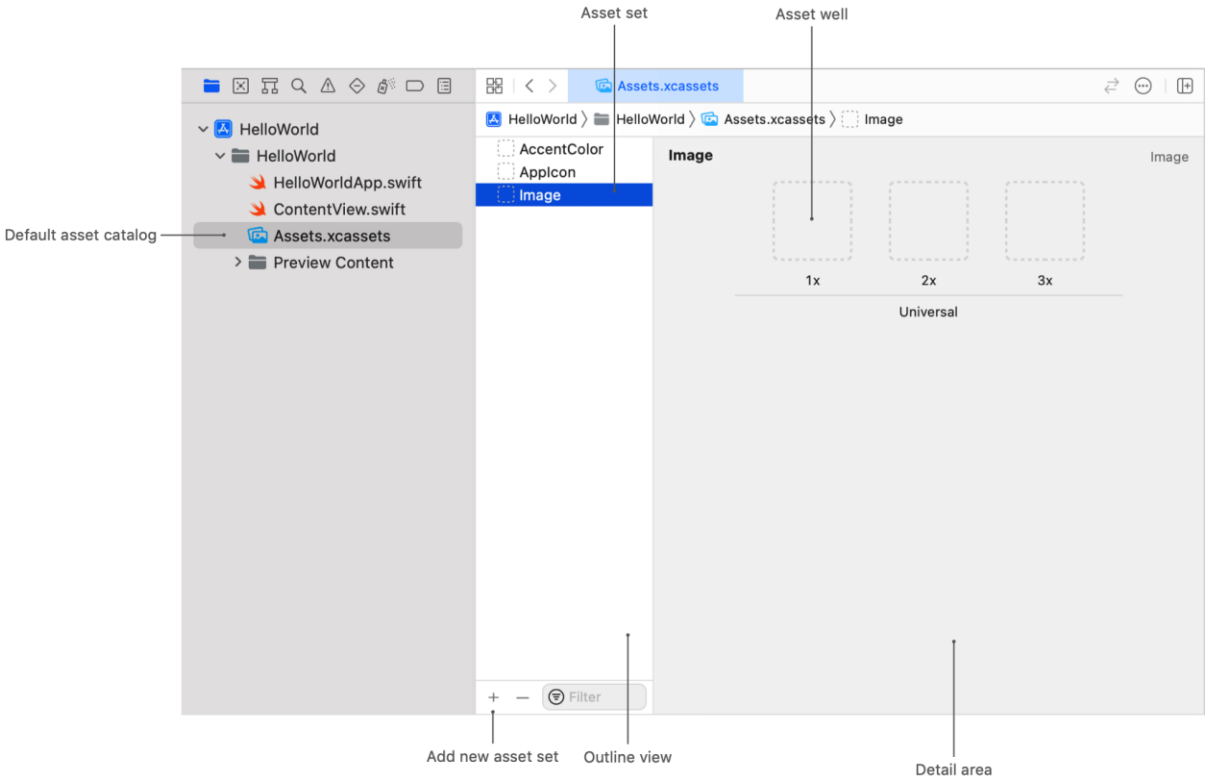
Mivel egy alkalmazással több konfigurációt is támogatni kell (képernyő méret, nyelvi beállítások, orientáció stb.), ezért minden konfigurációnak megfelelően létrehozhatunk specifikus erőforrás fájlokat. A különböző konfigurációkat egy új `<resources_name>-<qualifier>` formájú könyvtárba vegyük fel.

> Az erőforrás fájlokat mindig a típusának megfelelő könyvtárban hozzuk létre.

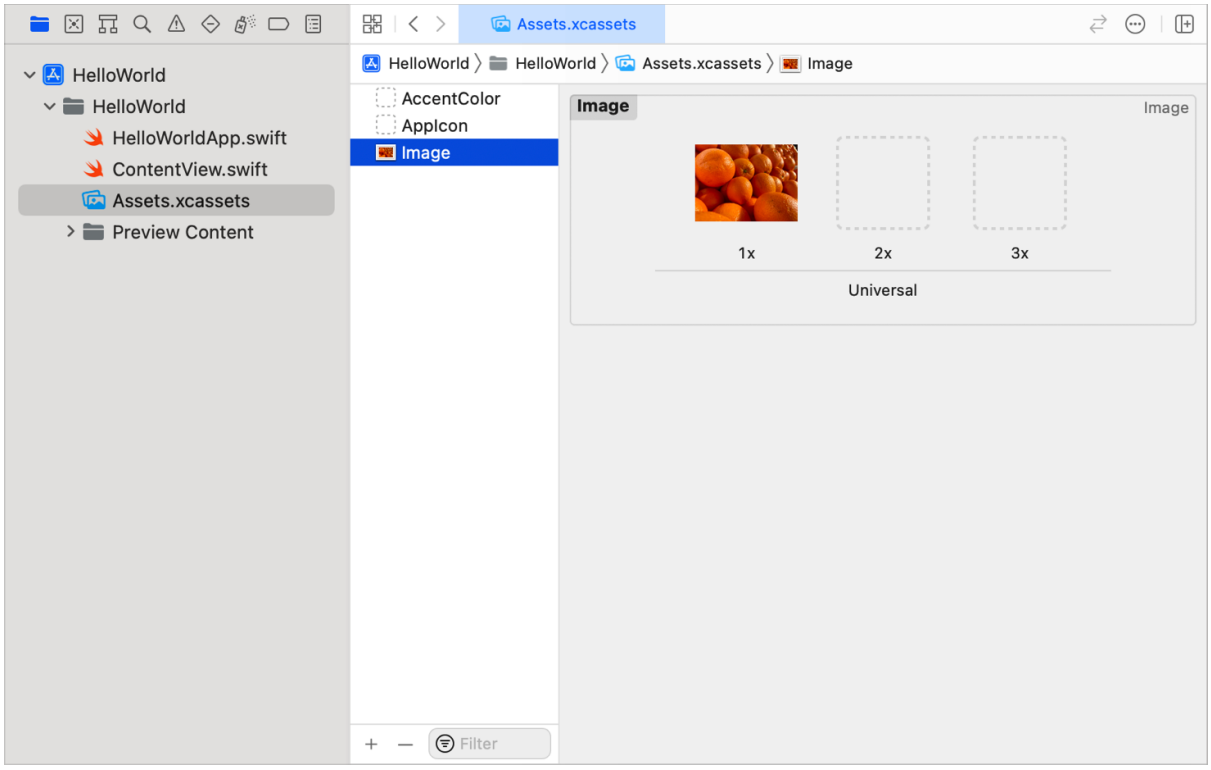
Forrás: https://developer.android.com/guide/topics/resources/providing-resources

iOS

Az erőforrás fájlokat asset catalog-ok segítségével tudjuk rendszerezni. Egy iOS projekt létrehozásánál automatikusan generálódik egy alapértelmezett asset catalog (Assets.xcassets).



Ezek a catalog-ok úgynevezett asset set-eket tárolnak. Egy asset set egy vagy több well-t tartalmaz, amelyek a különböző eszközökön használt variánst tartalmazzák az erőforrás fájlunkból.



A különböző erőforrás típusokat célszerű külön asset catalog-ban tárolni.

Forrás: https://developer.apple.com/documentation/xcode/managing-assets-with-asset-catalogs

Publikálás

Publikálás előtt meg kell adnunk, hogy milyen eszközöket és OS verziókat támogatunk az alkalmazással. iOS alkalmazás esetén általános, hogy a utolsó három OS verzió a támogatott. Az Android alkalmazások általában ettől megengedőbbek, ebben az esetben öt korábbi verzió támogatása megszokott. Abban az esetben, ha egy eszköz elveszti a támogatást, az alkalmazás továbbra is használható marad, de új frissítések már nem érkeznek. A minimum támogatott verzió növelése előtt célszerű megvizsgálni az alkalmazásunk felhasználói bázisának eloszlását.

Androidon lehetőségünk van az alkalmazás boltokat kikerülni és direkt telepíteni az alkalmazást, azonban iOS-en nincs erre lehetőségünk.

Android

Az alkalmazások telepítéséhez egy .apk kiterjesztésű fájlra van szükség. Ez az alkalmazás futtatásához szükséges minden forrásfájlt tartalmaz. Az .aab fájl azonban csak az alkalmazások megosztására szolgál, ezt önmagában nem lehet telepíteni az eszközökön. Az alkalmazásboltok az .aab fájlból tudnak generálni egy telepíthető .apk állományt, amely már csak az adott eszközön való futtatáshoz szükséges forrásokat tartalmazza.

Lehetőségünk van az alkalmazásunkat az .apk fájllal is publikálni, ennek azonban tartalmaznia kell az összes konfigurációhoz tartozó forrásokat, így a letöltött fájl mérete nagyobb lesz.

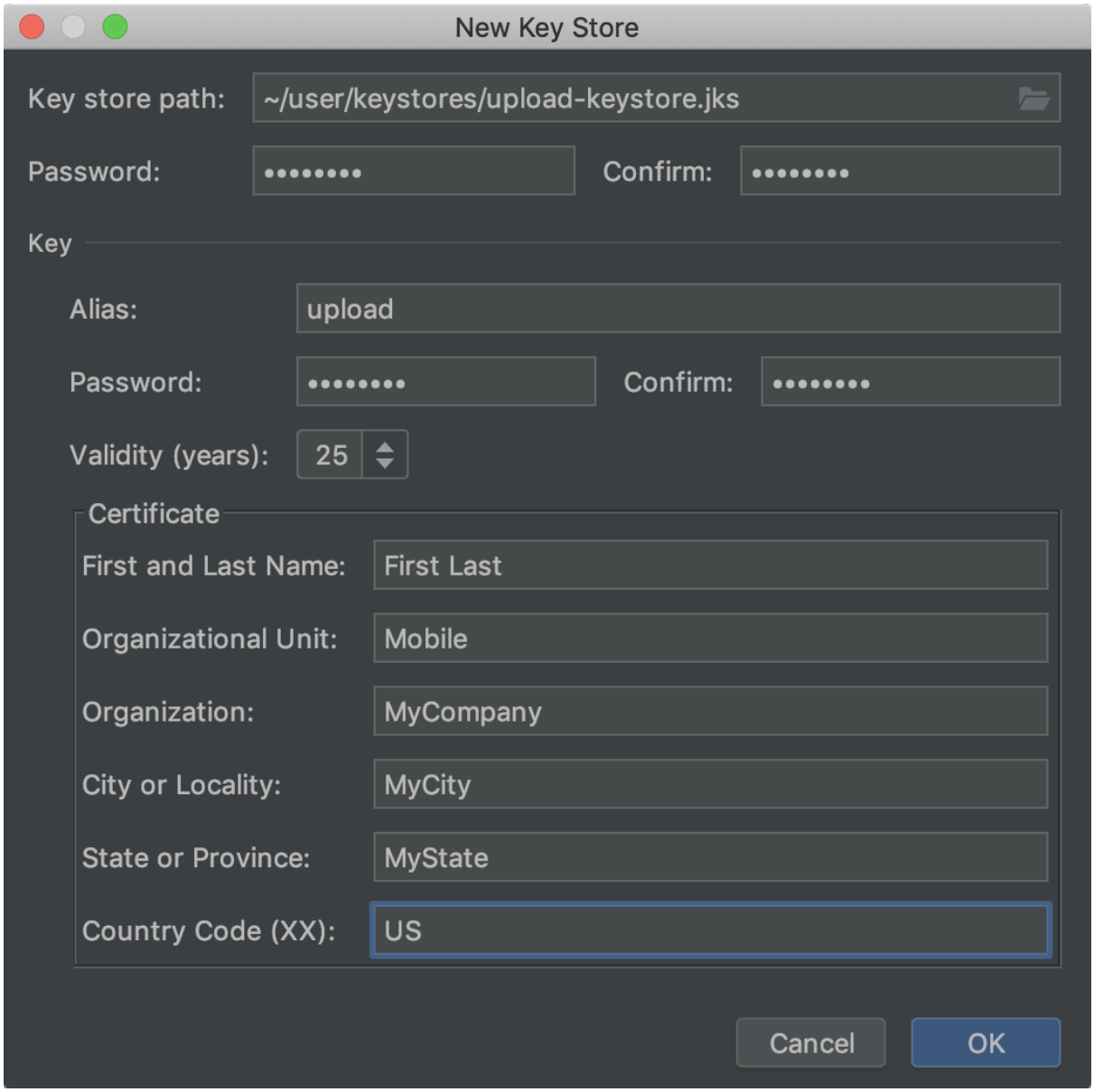
> Az alkamazásboltokba az .aab fájlt töltsük fel, így csökkenthetjük az alkalmazás méretét.

Forrás: https://developer.android.com/studio/publish

Aláírás

Ahhoz, hogy biztosítani tudjuk, hogy a feltöltött alkalmazás valóban hiteles forrásból származik, alá kell írnunk azt. Ehhez egy aláíró kulcsra van szükség, melyet legegyszerűbben az Android Studio-ban tudunk generálni.

1. Nyissuk meg a Build > Generate Signed Bundle/APK menüpontot.
2. A felugró ablakban válasszuk ki, hogy Bundle-t vagy APK-t szeretnénk generálni.
3. Válasszuk ki, hogy már meglévő keystore-t használunk, vagy egy újat szeretnénk generálni.
4. Az új keystore választása esetén adjuk meg a szükséges adatokat.



Az így létrehozott kulccsal már alá tudjuk írni az alkalmazásunkat. 2021. augusztusától a Google Play Store kötelezővé teszi a Play App Signing használatát. Ebben az esetben a fent létrehozott aláíró kulcs csak a Play Store-ba való feltöltés hitelesítéséhez használatos.

1. A release aláírása során kapcsoljuk be az Export encrypted key checkbox-ot és mentsük el a .pepk fájlt.
2. Lépjünk be a Play Console-ba.
3. Válasszuk a Release > Setup > App integrity menüpontot.
4. Kövessük az utasításokat, majd töltsük fel az első pontban mentett .pepk fájlt

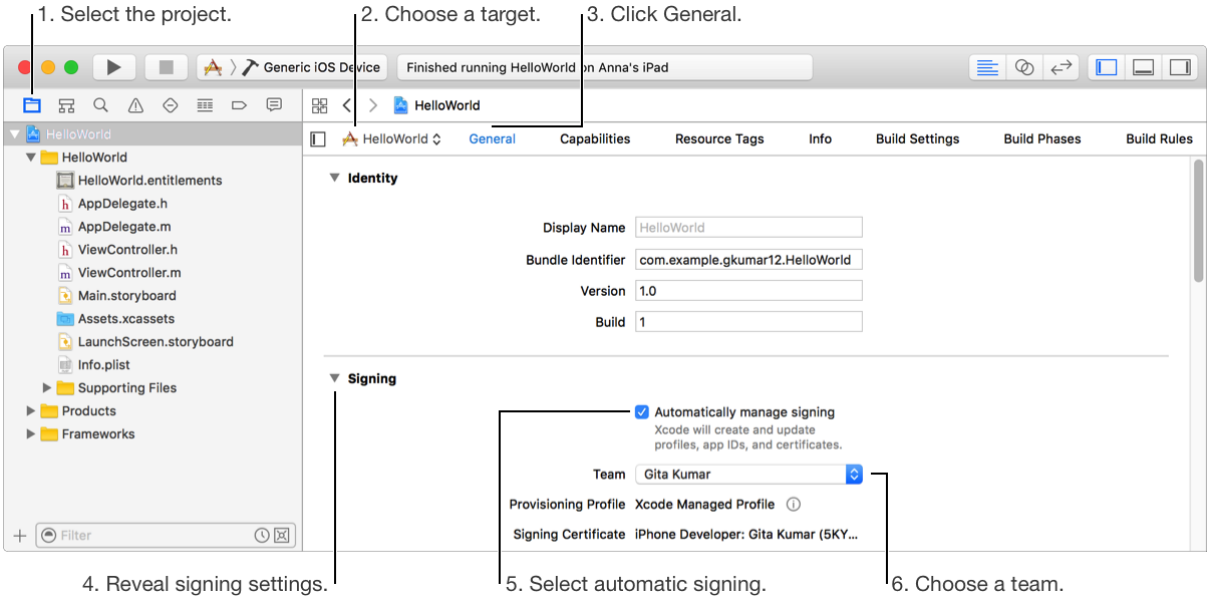
Forrás: https://developer.android.com/studio/publish/app-signing

iOS

iOS esetén az Xcode lehetőséget nyújt nekünk a különböző certificate-ek kezelésére és a publikálásra.

Először létre kell hoznunk egy Apple Developer Program felhasználót és ezt hozzáadni az Xcode-hoz. Ezt a Preferences > Accounts menüben tehetjük meg.

Ezután lehetőségünk van bekapcsolni az automatikus aláírást. A Project Editor > General > Signing szekció alatt kapcsoljuk be az Automatically manage signing opciót és válasszuk ki a fejlesztői csapatot. Amikor egy új eszközt csatlakoztatunk, az Xcode automatikusan hozzáadja a fejlesztői csapat provisioning profiljához.



Feltöltéshez szintén használhatjuk az Xcode-ot. Deployment Target-nek állítsuk be a Generic iOS Device-t, majd a Product > Archive menüpontot választva megjelenik az Xcode Organizer. Innen van lehetőségünk a kiválasztott verziót feltölteni az App Store-ba.

Forrás: https://developer.apple.com/help/account/get-started/about-your-developer-account

Tesztelés

A tesztelési piramist a mobil alkalmazások esetén is tudjuk alkalmazni. Sok esetben ez a piramis mobil esetén meg is fordul, hiszen a manuális tesztelés sokkal nagyobb hangsúlyt kap. A unit és integrációs tesztek karbantartása mobilon is fejlesztői feladatkör, a manuális tesztelés, illetve az end-to-end tesztek már QA feladatkörbe tartoznak.

Mobil környezetben az integrációs teszteket már fizikai eszközön vagy emulátoron/szimulátoron futtatjuk. Azonban ezek a tesztek lassúak, így valódi eszközön csak azokat az eseteket célszerű futtatni, melyeknél fontos az eszközzel való interakció.

Androidon a JUnit az ajánlott tesztelési keretrendszer, ezen felül a függőségek mockolásához a Mockk framework, integrációs és e2e tesztekhez az Espresso használata javasolt.

iOS-en a beépített XCtest és XCUItest keretrendszerek használata az ajánlott.

Forrás: https://developer.android.com/training/testing/local-tests
Forrás: https://developer.android.com/training/testing/instrumented-tests
Forrás: https://developer.apple.com/documentation/xctest/xctest
Forrás: https://developer.apple.com/documentation/xctest/user_interface_tests

Android

A Play Store-ban a Tesztelés menüpont alatt kezelhetjük a teszt csatornákat. Nyílt tesztelés esetén egy meghívó link segítségével bárki beléphet a tesztelői csoportba.

Zárt tesztelés esetén a fejlesztő hívhat meg tesztelőt a Google fiókjához tartozó e-mail cím alapján. Erről a tesztelő egy meghívó email-t fog kapni, elfogadás után pedig a Play Store-ból tudja telepíteni az alkalmazást.

A különböző teszt csatornákra külön kiadást kell készíteni a tesztelendő alkalmazáshoz.

Forrás: https://support.google.com/googleplay/android-developer/answer/9845334?hl=en

iOS

Béta tesztelést TestFlight-on keresztül tudunk folytatni. Az alkalmazás App Store Connect adatlapján a Test Flight tabon először új test group-ot kell létrehoznunk. Ehhez a csoporthoz e-mail címük alapján tudjuk a tesztelőket felvenni. A tesztelők ezután e-mail-ben fognak meghívást kapni az alkalmazásunkhoz, meghívó elfogadása után pedig a TestFlight alkalmazásból tudják telepíteni a tesztelendő alkalmazást.

Ahhoz, hogy a verziót elérhetővé tegyük a tesztelők számára, válasszuk ki a teszt csoportot, majd a Builds tab alatt adjuk hozzá a korábban már feltöltött verziót.

> A TestFlight Apple ID-hoz kötött. A tesztelőnek először Apple ID-t kell regisztrálnia a meghívó elfogadásához.

Forrás: https://testflight.apple.com/

Cross platform

Előnyök

- **Költségek:** A cross platform fejlesztés esetében a fejlesztési költségek alacsonyabbak, mint a natív fejlesztésnél. Egy fejlesztői csapat támogatni tudja az összes platformot.
- **Kód újrahasználhatóság:** A natív fejlesztéssel ellentétben a cross platform esetén egy közös kódbázisra épül az alkalmazás.
- **Fejlesztési idő:** A közös kódbázis miatt a fejlesztés ideje is lényegesen lerövidül.

Hátrányok

- **Nagyobb projektek:** Az optimalizálások, illetve a platformok közötti különbségek áthidalása miatt a cross platform projektek nagyobb méretűek.
- **Funkcionalitás hiánya:** Technológiától függően nem érhetőek el bizonyos platform specifikus funkciók, pl. szenzorok, GPS, kamera, stb.
- **Teljesítmény:** A cross platform technológiák esetében egy köztes szintet kell bevezetni az operációs rendszer és az alkalmazás között. Ezáltal az alkalmazás működése lassabb.

Flutter

A Flutter egy Google által készített cross platform framework. Segítségével egy közös kódbázisból lehet Android, iOS, macOS, Windows és Linux alkalmazásokat készíteni. A framework Dart-ra épül, így megörökli a Pub package manager-t is.

UI

Flutter-ben a felhasználói felület Widgetek-ből épül fel. A konfiguráció és a widget állapota alapján a widget-ek leírják hogyan épüljön fel a felhasználói felület.

A különböző platformokon már megszokott UI elemekre léteznek beépített widget-ek, iOS esetén a Cupertino, Androidon a Material widget-ek használata javasolt. Ehhez azonban el kell választani a két megjelenítési logikát.

Forrás: https://docs.flutter.dev/ui/widgets-intro

Stateless Widget

A Flutter két fajta widget-et különböztet meg, a stateless és stateful widgeteket. A stateless widget, ahogy a nevében is benne van, nem függ az alkalmazás állapotától. Kirajzolást követően ezek az elemek már nem módosíthatók, pl. ikonok, képek, stb.

```
class MyApp extends StatlessWidget {
    @override
    Widget build(BuildContext context) {
        return Text('My App')
    }
}
```

Stateful Widget

A stateless widget-ekkel ellentétben, a stateful widget függ az alkalmazás állapotától. Konfiguráció vagy állapot változás esetén a widget újraépül, a framework pedig a két állapotot összehasonlítva megadja a minimális változásokat, amelyekkel az új állapot megjeleníthető.

```
class MyApp extends StatefulWidget {

    @override
    _MyAppState createState() => _MyAppState()
}

class _MyAppState extends State<MyApp> {

    int tapCount = 0;

    void _handleTap() {
        setState(() {
            tapCount += 1;
        });
    }
    @override
    Widget build(BuildContext context) {
        return Column(
            children: [
              Text('Tapped $tapCount times'),
              FlatButton(
                child: Text('Increase'),
                onPressed: () {
                  _handleTap()
                }
              )
            ]
        );
    }
}
```

Architektúra

A natív alkalmazásokhoz hasonlóan Flutter-ben is az MVVM architektúra az ajánlott.

State Holder

A state holder feladatokat Flutter-ben is a ViewModel látja el.

```
class NewsViewModel(NewsRepository repository) extends ChangeNotifier {

  List<NewsUiState> news = List<NewsUiState>();

  Future<void> fetchNews() async {
    final results = await repository.fetchNews();
    this.news = results.map((item) => NewsUiState(news: item)).toList();
    notifyListeners();
  }
}
```

Állapot átadás

A widget számára az állapotot ChangeNotifierProvider segítségével adhatjuk át. A state holder az állapot változása esetén a notifyListeners() metódus hívással jelez az observereknek, hogy az állapot megváltozott.

```
ChangeNotifierProvider(
  create: (context) => NewsViewModel(),
  child: NewsListPage(),
)
```

REST

Flutter-ben a http könyvtár segítségével van lehetőségünk REST hívásokat készíteni.

```
Future<News> fetchNews() async {
  final response = await http.get(
    Uri.pasrse('myapi.com/news'),
    headers: {
      HttpHeaders.authorizationHeader: 'apiToken'
    }
  );

  if (response.statusCode == 200) {
      return News.fromJson(jsonDecode(response.body));
  } else {
    throw Exception('Failed to load')
  }
}
```

Forrás: https://docs.flutter.dev/cookbook/networking/fetch-data

Szerializálás

A JSON szerializáláshoz a json_serializable package használható. Segítségével a JSON adatot kódgenerálással tudjuk Dart modellekre fordítani. A szükséges kódot a `flutter pub run build_runner build --delete-conflicting-outputs` paranccsal tudjuk generálni.

```dart
@JsonSerializable()
class News {
  News(this.title, this.body);

  String title;
  String body;

  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);

  Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

Forrás: https://docs.flutter.dev/data-and-backend/json

PWA

A PWA-k (Progressive Web Application) webes technológiákkal készített alkalmazások. Ahhoz, hogy használni tudjuk nincs szükség telepítésre, böngészőben is futtathatóak. Azonban lehetőség van a telepítésre is, így kiterjedtebb funkcionalitás érhető el (pl. push notification, offline működés). A legtöbb alkalmazásbolt támogatja a PWA-k megosztását, azonban az Apple App Store nem tartozik ezek közé. Az App Store-ba egy WebKit-be csomagolt PWA-t van lehetőség feltölteni, azonban a funkciók nagy része nem lesz támogatott.

> iOS-en a legtöbb PWA funkcionalitás nem támogatott.

Forrás: https://web.dev/progressive-web-apps/

Hybrid

A hibrid alkalmazások használatával lehetőségünk van a natív és cross platform technológiákat együtt használni. Ebben az esetben natív alkalmazásokat készítünk, melyek becsomagolják a cross platform kódunkat. Így a felhasználói élmény - hasonlóan a cross platform alkalmazásokhoz - gyengébb, de lehetőségünk van a platform által nyújtott funkciókat is kihasználni.

> A Hybrid módszer alkalmazása nem indokolt. Így ezt a megoldást kerülni szükséges!

JavaScript

A kommunikáció a natív alkalmazás és a webes tartalom között egy JavaScript Interface segítségével lehetséges. Ezzel az interfésszel fel lehet iratkozni a webes tartalom által küldött eseményekre, illetve JavaScript kód futtatására.

Android

A WebView-n először engedélyezni kell a JavaScript kódok futtatását.

```
val myWebView: WebView = findViewById(R.id.webview)
myWebView.settings.javaScriptEnabled = true
```

Ezek után a natív alkalmazás implementálja a JavaScript interfészt, melyet aztán átad a WebView számára.

```
/** Instantiate the interface and set the context  */
class WebAppInterface(private val mContext: Context) {

    /** Show a toast from the web page  */
    @JavascriptInterface
    fun showToast(toast: String) {
        Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show()
    }
}
```

```
val webView: WebView = findViewById(R.id.webview)
webView.addJavascriptInterface(WebAppInterface(this), "Android")
```

Ezek után az interfész elérhető a web alkalmazás számára.

```
<input type="button" value="Say hello" onClick="showAndroidToast('Hello Android!')" />

<script type="text/javascript">
    function showAndroidToast(toast) {
        Android.showToast(toast);
    }
</script>
```

> Az interfészt nem kell külön inicializálni, a WebView hozzáadás után elérhetővé teszi.

Forrás: https://developer.android.com/develop/ui/views/layout/webapps/webview

iOS

Ahhoz, hogy kódot tudjunk injektálni a web alkalmazásunkba a WKUserContentController-t tudjuk használni.

```
let contentController = WKUserContentController()
let scriptSource = "document.body.style.backgroundColor = `red`;"
let script = WKUserScript(source: scriptSource, injectionTime: .atDocumentEnd, forMainFrameOnly: true)
contentController.addUserScript(script)

let config = WKWebViewConfiguration()
config.userContentController = contentController

let webView = WKWebView(frame: .zero, configuration: config)
```

A source maga a JavaScript kód, amit injektálni szeretnénk. Az injectionTime segítségével megadhatjuk, hogy mikor történjen a beillesztés, a document feldolgozása előtt vagy után.

WKScriptMessageHandler segítségével a webes alkalmazásból eseményeket tudunk küldeni a natív alkalmazás számára. Ehhez először a natív alkalmazásnak egy message handler-t kell hozzáadnia a WebView-hoz.

```
override func viewDidLoad() {
  super.viewDidLoad()

  let config = WKWebViewConfiguration()
  let userContentController = WKUserContentController()

  userContentController.add(self, name: "test")

  config.userContentController = userContentController

  ...

}
```

Majd ezekre az eseményekre lehet feliratkozni a UserContentController-en keresztül.

```
extension ViewController: WKScriptMessageHandler {
  func userContentController(_ userContentController: WKUserContentController, didReceive message: WKScriptMessage) {
      if message.name == "test", let messageBody = message.body as? String {
          print(messageBody)
      }
  }
}
```

Ezután a web alkalmazásban elérhetjük a hozzáadott message handler-t.

```
<script>

  function printHelloWorld() {
      window.webkit.messageHandlers.test.postMessage("Hello, world!")
  }

  window.onload = printHelloWorld;

</script>
```

Forrás
https://developer.apple.com/documentation/webkit/wkusercontentcontroller
https://developer.apple.com/documentation/webkit/wkscriptmessagehandler

Accessibility

A mobil alkalmazások esetén a WCAG 2.0 által lefektetett előírások követése az általános. Már az alkalmazás tervezési fázisában is gondolnunk kell a megfelelő accessibility támogatásra, pl. megfelelő kontraszt arányú színek használata, megfelelő szövegméret választás, nem szöveges vizuális elemekhez tartozó leírások megadása.

A megfelelő minőség biztosítása főleg manuális feladat, de léteznek beépített eszközök, melyekkel ellenőrizhetjük az alkalmazást.
Android Studio-ban az Accessibility Scanner, míg Xcode-ban az Accessibility Inspector segítségével van lehetőség az alkalmazás felhasználói felületén megtalálni azokat az elemeket, amelyek nem felelnek meg az előírásoknak. Ezekkel azonban csak a statikus felületi elemeket tudjuk vizsgálni.

Az accessibility tesztelés legfontosabb része, hogy manuálisan megvizsgáljuk az alkalmazásunkat. Android esetén TalkBack, iOS-en a VoiceOver segítségével navigáljunk az alkalmazásunkban és ellenőrizzük a működést.

Forrás
https://www.w3.org/TR/mobile-accessibility-mapping/#wcag-2.0-and-mobile-content-applications
https://developer.android.com/guide/topics/ui/accessibility/apps
https://developer.apple.com/documentation/accessibility/