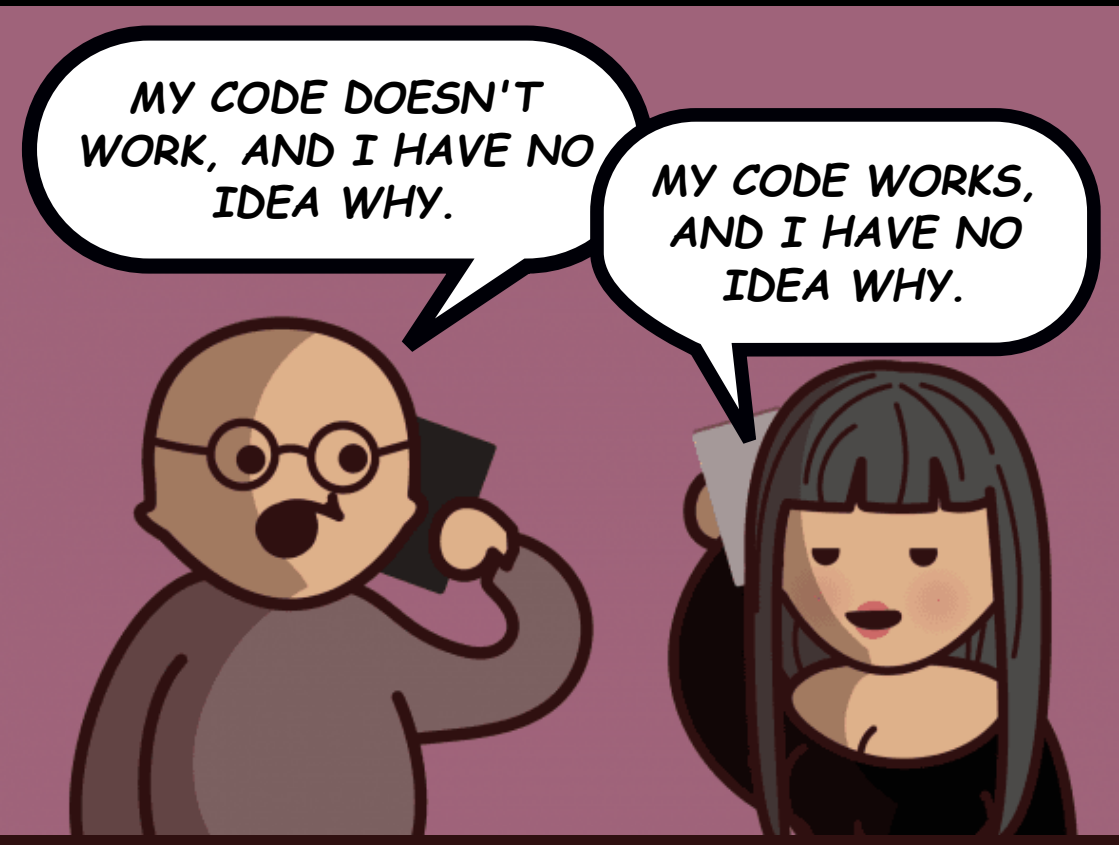
Vývoj aplikácií pre chytré zariadenia

Poznámky z prednášok 2020

Miroslav Biňas



kpi

Vývoj aplikácií pre chytré zariadenia

Vývoj aplikácií pre chytré zariadenia

Poznámky ku prednáškam 2020

Miroslav Biňas

Vývoj aplikácií pre chytré zariadenia
Poznámky ku prednáškam 2020
Miroslav Biňas

Vydavateľ:
Technická univerzita v Košiciach
Letná 9, 042 00 Košice, Slovensko
www.tuke.sk
Katedra počítačov a informatiky
www.kpi.fei.tuke.sk

prvé vydanie, Košice 2021
ISBN 978-80-553-3956-6

Obálku ilustrovala © Janka Slebodníková, www.maranveart.com. Ilustrácia bola prvýkrát publikovaná v komixoch autorky, ktoré vychádzali v rokoch 2014 až 2018 na stránkach portálu www.root.cz.

Obsah

[1] https://www.genymotion.com
[2] https://github.com/Genymobile/scrcpy

Course Introduction

úvod do problematiky vývoja aplikácií pre chytré zariadenia, tri typy aplikácií, predstavenie hybridného rámca *React Native*, architektúra, prvá aplikácia

> **Upozornenie**
> Materiál, ktorý sa Vám dostáva do rúk, predstavuje v prvom rade poznámky ku prednáškam pre (mňa ako) prednášajúceho. Preto v ňom nehľadajte detailné opisy preberanej problematiky, ako by ste ich očakávali od učebnice. Niekde sú poznámky strohé, niekde sú zasa naopak komplexnejšie.
> Aj napriek tomu však tento materiál vie výborne poslúžiť ako pomôcka pre štúdium tvorby aplikácií pre chytré zariadenia. Pri jeho príprave (teda hlavne pri príprave samotných prednášok) som si dával záležať na tom, aby som menej hovoril a viac ukazoval, ako veci fungujú. Ak teda hľadáte materiál, ktorý vám viac ukáže ako opíše, našli ste ho.

Welcome

Ahojte a vitajte v kurze *Vývoj aplikácií pre chytré zariadenia*. Volám sa *Miroslav Biňas* a budem vaším lektorom.

V tomto kurze sa naučíte, ako vyvíjať natívne aplikácie pre chytré zariadenia pomocou rámca *React Native* primárne pre platformu *Android*. A tým chytrým zariadením bude v tomto prípade váš mobilný telefón alebo tablet.

Nebudeme sa však venovať veciam ako používateľské rozhranie (UI) alebo používateľská skúsenosť (UX). Na to máte v tomto semestri samostatný predmet.

My sa budeme venovať viac práci so senzormi, ktorými sú chytré zariadenia vybavené. Budeme teda vytvárať aplikácie, ktoré budú vedieť pracovať s polohou zariadenia, s jeho kamerou, Bluetooth-om, NFC čítačkou a pod. Rovnako nás bude zaujímať, ako údaje z týchto zariadení uchovávať a ako ich zdieľať v prostredí internetu alebo s inými chytrými zariadeniami.

Na zvládnutie tohto kurzu nepotrebujete mať žiadne predchádzajúce znalosti s vývojom aplikácií pre niektorú z platforiem, ktorá sa používa na beh aplikácií na chytrých zariadeniach. Všetko potrebné, čo potrebujete vedieť, ste sa už naučili v predchádzajúcich kurzoch. To najdôležitejšie, na čom budeme stavať, je programovanie v jazyku *JavaScript*, pretože to je jazyk, ktorým sa hovorí v rámci *React Native*.

Takže - poďme na to!

Prednáška 1

Introduction to Smart App Development

úvod do problematiky vývoja aplikácií pre chytré zariadenia, tri typy aplikácií, predstavenie hybridného rámca *React Native*, architektúra, prvá aplikácia

1.1 What is a Smart Device?

Tento kurz bude o vývoji aplikácií pre chytré (smart) zariadenia. Takže čo je to vlastne chytré zariadenie?

Chytré zariadenia by sme mohli definovať ako **elektronické zariadenie, ktoré je schopné pracovať samostatne, interagovať s jeho používateľom, ako aj s inými zariadeniami pomocou rozličných komunikačných technológií**. Častokrát sa môže jednať o zariadenie umelej inteligencie.

Častokrát majú chytré zariadenia malú veľkosť, ale napriek tomu môžu poskytovať obrovský výkon, ktorý môže byť porovnateľný aj so súčasnými počítačmi.

Príkladom takýchto zariadení môžu byť chytré autá, hodinky, telefóny, fitness náramky, zámky, termostaty, zariadenia bielej techniky alebo obecne zariadenia inteligentnej domácnosti, reproduktory a pod.

V tomto kurze sa budeme venovať konkrétne vývoju aplikácií pre chytré telefóny, pretože ich zastúpenie na trhu je obrovské.

1.2 The 3 Types of Mobile Experiences

Keď sa na trhu objavili operačné systémy pre chytré zariadenia *Android* a *iOS* a chceli ste pre ne vyvíjať aplikácie, nemali ste veľa možností. V podstate to znamenalo, že pre vývoj aplikácií pre *iOS* ste používali *XCode* a aplikácie ste písali v jazyku *Objective-C*, a keď ste chceli písať aplikácie pre *Android*, písali ste ich v jazyku *Java* (tí náročnejší v *C++*) a používali ste s najväčšou pravdepodobnosťou IDE *Eclipse* s *Android SDK*.

Odvtedy sa však situácia diametrálne zmenila. Ako na úrovni jazykov, tak aj na úrovni nástrojov, ktoré môžete pre vývoj používať. Dnes je na výber obrovské množstvo rámcov, nástrojov, platforiem a ekosystémov pre komplexný vývoj aplikácií pre chytré zariadenia.

Aj napriek tomu však vieme to množstvo možností zoskupiť do niekoľkých kategórií. Ešte skôr, ako sa začneme venovať samotným technológiám, si predstavíme 3 rozličné spôsoby vývoja aplikácií, resp. 3 rozličné typy aplikácií pre chytré zariadenia, a to (*Research Guide* portálu DZone[1]):

1. **natívne aplikácie**
2. **webové aplikácie**
3. **hybridné aplikácie**

Na jednotlivé typy aplikácií sa pozrieme podrobnejšie.

1.2.1 Native Applications

Prvú skupinu tvoria **natívne aplikácie**. Tieto aplikácie sú vyvíjané pre konkrétnu platformu pomocou nástrojov, ktoré nám poskytol jej výrobca. Príkladom môže byť jazyk *Kotlin* (predtým jazyk *Java*) v prostredí *Android Studio* pre *Android* alebo jazyk *Swift* (predtým jazyk *Objective-C*) v prostredí *XCode* pre *iOS*.

Nasledujúci obrázok ilustruje architektúru natívnej aplikácie. Najspodnejšia vrstva označená ako *Platform* je operačný systém (platforma), ktorá komunikuje s hardvérom daného zariadenia. Natívna aplikácia s ňou komunikuje pomocou *SDK* danej platformy. Tu sa nachádza programové rozhranie, pomocou ktorého je možné pristupovať ku jednotlivým komponentom platformy, ako napr. posielať notifikácie, prehrávať mediálne súbory, komunikovať s ostatnými zariadeniami alebo so senzormi a akčnými členmi samotného zariadenia. V prípade komunikácie s externými službami pre potreby získania

[1] https://dzone.com/

údajov môže využiť potrebné komunikačné protokoly, ktorých podpora môže byť rovnako zabalená ako súčasť *SDK* platformy.

V porovnaní s ďalšími typmi aplikácií je dôležité spomenúť, že binárka aplikácie sa inštaluje z obchodu platformy a ukladá priamo do súborovom systéme daného zariadenia. Je spúšťaná priamo operačným systémom a pre svoju činnosť nepotrebuje žiadny špeciálne špeciálne behové prostredie.

V prípade aktualizácie aplikácie sa táto znova sťahuje.

výhody:

- Natívne aplikácie dosahujú v porovnaní s ostatnými typmi aplikácií **najvyšší výkon**. Preto sa často používajú napr. na vývoj hier.
- Ponúkajú **veľmi dobrý používateľský zážitok**, nakoľko vývojári používajú natívne UI zariadenia, resp. platformy.
- Ponúkajú **priamy prístup k natívnemu API** a teda aj ku všetkým vlastnostiam platformy a ku všetkým senzorom a akčným členom zariadenia.
- Pri vývoji natívnych aplikácií sú k dispozícii aj **veľmi dobré ladiace nástroje**.
- Sú **distribuované pomocou obchodu danej platformy**.

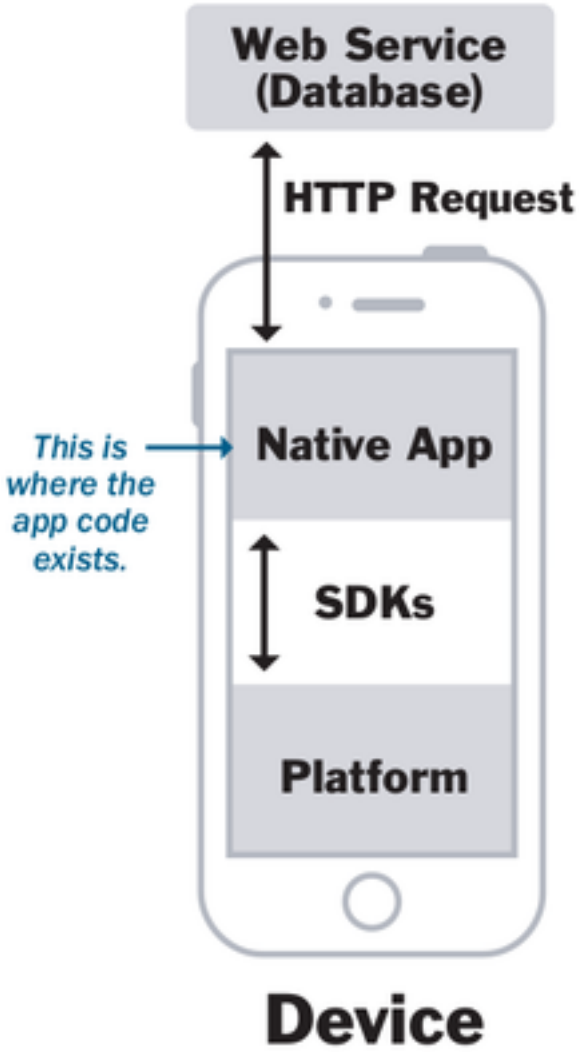
nevýhody:

- Za najväčšiu nevýhodu je možné považovať to, že **aplikácie nie sú platformovo nezávislé**. To znamená, že v prípade distribúcie pre viaceré platformy je potrebné aplikáciu vytvoriť pre každú zvlášť. To častokrát znamená mať samostatných vývojárov pre každú platformu, nakoľko vývoj pre každú z nich je odlišný (odlišný jazyk, odlišné SDK, odlišné paradigmy, odlišné rámce, …).
- Obyčajne je ich **vývoj náročnejší** a tým pádom aj **drahší**. To sa rovnako týka aj údržby už vytvorených aplikácií.

1.2.2 Web Applications

Druhým typom aplikácií sú *webové aplikácie* známe tiež pod názvom *Mobile Websites* alebo *Mobile Web Applications*.

Jednou zo základných aplikácií, ktorú nájdete na každom mobilnom zariadení, je určite *webový prehliadač*. Webové prehliadače už dávno neslúžia len na prehliadanie statických webových stránok, ale dnes fungujú ako isté kontajnery, v ktorých je možné spúšťať rozličné aplikácie. V prehliadači teda viete



Obrázok 1.1: Architecture of Native Applications [13]

dnes spustiť rozličné kancelárske aplikácie (Google Docs, MS Office 365), ale rovnako sa v nich viete hrať aj hry a častokrát aj náročné FPS.

Na sile webového prehliadača stavajú aj webové aplikácie. Tie sú napísané pomocou webových technológií HTML5 a spúšťa ich prehliadač a nie operačný systém. Architektúru takejto aplikácie ilustruje nasledujúci obrázok.

Spôsob spustenia takejto aplikácie však môže byť rôzny:

- môže sa jednať o URL adresu, ktorú treba do prehliadača napísať
- môže sa jednať o odkaz, na ktorý je treba kliknúť a až tak sa spustí prehliadač s aplikáciou
- môže byť zosnímaný QR kód s adresou aplikácie
- aplikácia môže byť stiahnutá z obchodu platformy, pričom sa bude tváriť ako normálna aplikácia, ktorá nebude zobrazovať adresný riadok, ale na pozadí sa bude jednať len o `WebView`, v ktorom sa otvorí linka s adresou aplikácie

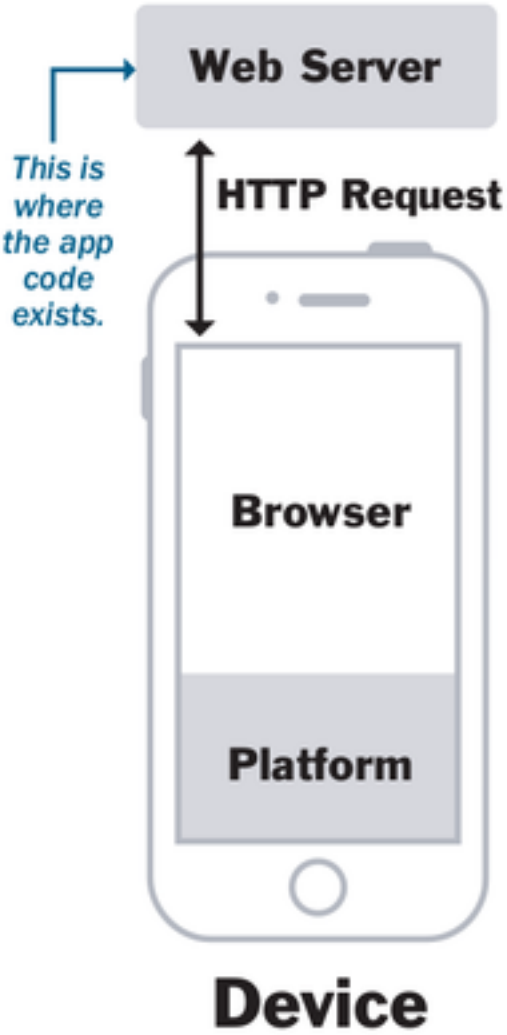
Aplikácie sú teda umiestnené na vzdialenom serveri a do prehliadača sa musia stiahnuť zakaždým (alebo takmer zakaždým), keď sa aplikácia spustí. Obyčajne sú to naozaj len webové stránky optimalizované pre zobrazovanie na mobilnom zariadení, napr. http://m.ebay.com, http://m.facebook.com a pod. Ale môže sa jednať aj o seriózne *Single Page Applications* (SPA), ktoré vedia v prípade potreby pracovať aj offline.

výhody:

- Aplikácie sa **neinštalujú na zariadenie**, pretože sú umiestnené na vzdialenom serveri. Spustia sa len vo webovom prehliadači.
- **Jednoduchá správa a údržba** aplikácií, nakoľko nevyžadujú od používateľa žiadne špecifické oprávnenia pre inštalovanie/aktualizáciu. Nová verzia sa distribuuje na webovom serveri a používatelia sa k nej dostanú automaticky, keď aplikáciu spustia.
- **Aplikácie sú prenositeľné** aj na iné systémy, nakoľko pre ich spustenie postačujúci je len webový prehliadač.

nevýhody:

- Pre ich spustenie je **potrebné pripojenie do internetu**.
- Webové aplikácie **nemajú prístup k natívnemu API a platforme**. To znamená, že aplikácia nebude mať prístup napr. ku kontaktom, notifikáciám alebo ku lokálne uloženým údajom.



Obrázok 1.2: Architecture of Web Applications [13]

- Aplikácie **nemusia mať prístup ku hardvéru zariadenia**, ako napr. ku kamere alebo GPS. Vzhľadom na to, že aj HTML5 technológie a prehliadače sa vyvíjajú, mnohé aj hardvérové komponenty zariadenia môžu byť dostupné cez vhodné volanie v prehliadači.
- **Obmedzené možnosti používateľského rozhrania**, ktoré sú dané možnosťami technológií HTML5. To sa môže prejaviť napr. tak, že používateľské rozhranie webových aplikácií bude odlišné od natívnych aplikácií pre danú platformu. Existujú však CSS rámce, ktoré vedia práve tento dopad minimalizovať a vzhľad webových aplikácií dokáže byť takmer identický so vzhľadom natívnych aplikácií.
- Pre spustenie je potrebné **napísať adresu do prehliadača** alebo si vytvoriť vlastný spúšťač.
- **(Nie) sú distribuované prostredníctvom obchodu** danej platformy.

1.2.3 Progressive Web Applications (PWA)

PWA sú špeciálnym prípadom webových aplikácií.

Sú pomerne novou technológiou (2015) od Google.

1.2.4 Hybrid Applications

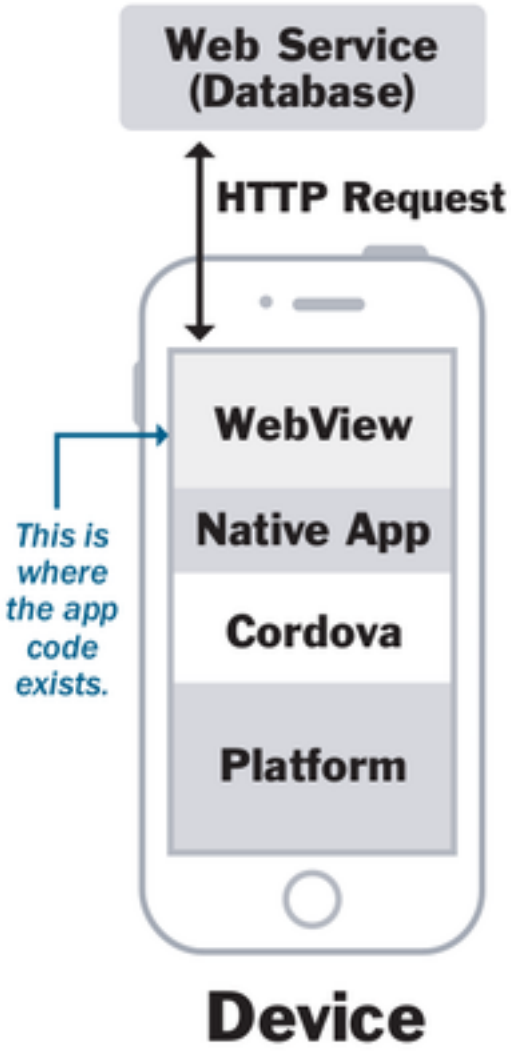
Hybridné aplikácie predstavujú kombináciu oboch predchádzajúcich prístupov. Sú to vlastne webové aplikácie, ktoré vyzerajú a pracujú ako natívne aplikácie.

Hybridné aplikácie sa vytvárajú pomocou HTML5 technológií a majú prístup k natívnemu API. Tým pádom sa vedia dostať ku špecifickým funkcionalitám platformy, ako aj ku hardvéru zariadenia.

Architektúra hybridnej aplikácie je zobrazená na nasledujúcom obrázku. Aplikácia obsahuje tzv. *Web View* (izolovanú inštanciu prehliadača), ktorý umožňuje spustiť webovú aplikáciu vo vnútri natívnej aplikácie. Tento *Web View* využíva *wrapper*, pomocou ktorého dokáže webová aplikácia komunikovať s natívnou platformou a mať tak prístup ku zdrojom zariadenia (napr. GPS, databáza, súbory atď.). Tento *wrapper* nie je súčasťou natívnej platformy, ale sú to produkty tretích strán (napr. [Apache Cordova).

V prípade inštalácie, spúšťania a aj aktualizácie sa aplikácia správa ako natívna aplikácia. Inštaluje sa z obchodu danej platformy, sťahuje sa do zariadenia a spúšťa ju operačný systém. Pri aktualizácii sa aplikácia znova sťahuje.

výhody:



Obrázok 1.3: Architecture of Hybrid Applications [13]

- **Aplikácie sú cross-platformové**, čo znamená, že rovnakú aplikáciu je možné spustiť na viacerých platformách. V špeciálnych prípadoch je samozrejme možné vytvoriť aj platformovo závislý kód, ktorý sa použije len v príslušnej platforme.
- Je potrebné **udržiavať len jeden kód**.
- Aplikácie majú **prístup ku samotnému zariadeniu** a jeho hardvéru.
- **Vývoj je relatívne rýchly, jednoduchý a lacný**. Na tvorbu hybridných aplikácií stačia znalosti a nástroje potrebné pre tvorbu webových stránok/aplikácií.
- Sú **distribuované pomocou obchodu** danej platformy.
- **Pracujú aj offline**.
- Hybridné aplikácie môžu slúžiť ako *Minimal Viable Product* na overenie nápadu pred jeho napísaním ako natívnej aplikácie.

nevýhody:

- Hybridné aplikácie **neposkytujú rovnaký výkon ako natívne aplikácie**. Sú teda typy aplikácií, pre ktoré sa hybridné aplikácie nehodia.
- *Wrapper* **nemusí mať implementovanú podporu všetkých možností natívneho API**. Obyčajne však má možnosť vlastnej implementácia formou rozširujúcich modulov (plugin-ov).
- Hybridné aplikácie podobne ako webové **nemusia vyzerať rovnako ako natívne aplikácie**. Je to však záležitosť dostupných CSS rámcov. Dokonca sa môže stať, že tá istá aplikácia môže vyzerať ináč na dvoch rozličných platformách.
- obmedzenie vzhľadom na možnosti *Web View-u*

1.2.5 Native? Web? Hybrid? Which one to Choose?

Takže chcete vytvoriť mobilnú aplikáciu a tá má byť… natívna? Alebo stačí webová? Alebo to bude hybrid?

Na otázku *"Ktorý prístup je najlepší?"* neexistuje univerzálna odpoveď. Toto rozhodnutie záleží od viacerých faktorov a v prípade, že ste len obyčajný vývojár, nezáleží ani od vás.

Často však rozhodnutie môže závisieť od mnohých faktorov:

- beh na konkrétnej platforme

Obrázok 1.4: Native vs. Web vs. Hybrid (zdroj[2])

- výkon aplikácie
- cena vývoja aplikácie
- multiplatformnosť
- práca s konkrétnym hardvérom zariadenia
- prístup ku konkrétnej súčasti platformy

1.3 Stack Overflow Developer Survey

Pozrime sa však na to, ako vyzerajú aktuálne trendy v oblasti vývoja aplikácií pre chytré zriadenia. V rokoch 2015 a 2016 portál DZone urobil na túto tému prieskum, ktorého sa zúčastnilo vyše 400 (v roku 2016) a 500 IT profesionálov, ktorí sú nejakým spôsobom zahrnutí v oblasti mobilných technológií. Ich výsledky zverejnili vo svojich DZone's Guide to Mobile Development 2015[3] a DZone's Guide to Mobile Development 2016[4]. Odvtedy však podobný prieskum neurobili :-/

Každý rok však portál Stack Overflow[5] robí svoj vlastný prieskum. Nie je síce zameraný výlučne na vývojárov mobilných aplikácií, ale z mnohých otázok sa dajú aktuálne trendy vyčítať. Posledný je z roku 2020[6] a zúčastnilo sa ho

[3] https://dzone.com/guides/the-dzone-guide-to-mobile-development-2015-edition
[4] https://dzone.com/guides/mobile-application-development-11
[5] https://stackoverflow.com
[6] https://insights.stackoverflow.com/survey/2020#technology-platforms

takmer 65000 vývojárov z celého sveta.

Obrázok 1.5: Stack Overflow Developer Survery 2020 (zdroj[7])

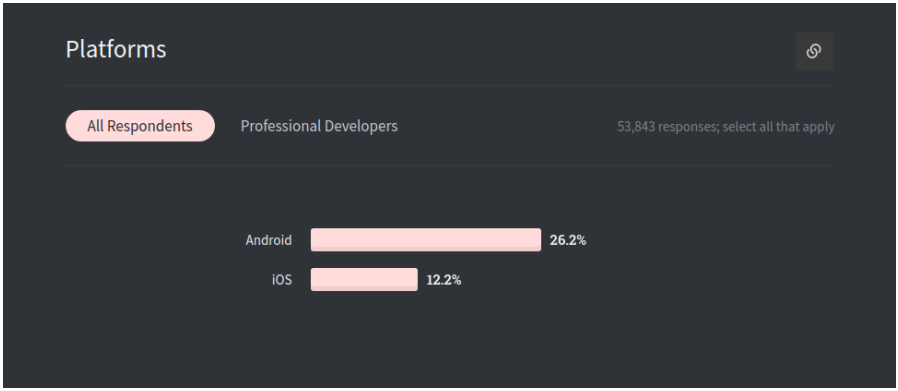
> **Poznámka**
> Keďže položiek vo výsledkoch bolo pomerne veľa, aby sa mi tie zaujímavé veci zmestili do prezentácie, zostrihol som nadbytočné položky ;)

Prvá otázka sa týkala platforiem, pre ktorú vývojári vyvíjajú svoje aplikácie. Spomedzi všetkých platforiem v porovnaní *Android* a *iOS* sa aplikácie viac vyvíjajú pre *Android*. To samozrejme nemusia byť len mobilné aplikácie, nakoľko *Android* má dnes už zastúpenie aj v telkách, hodinkách, autách, atď.

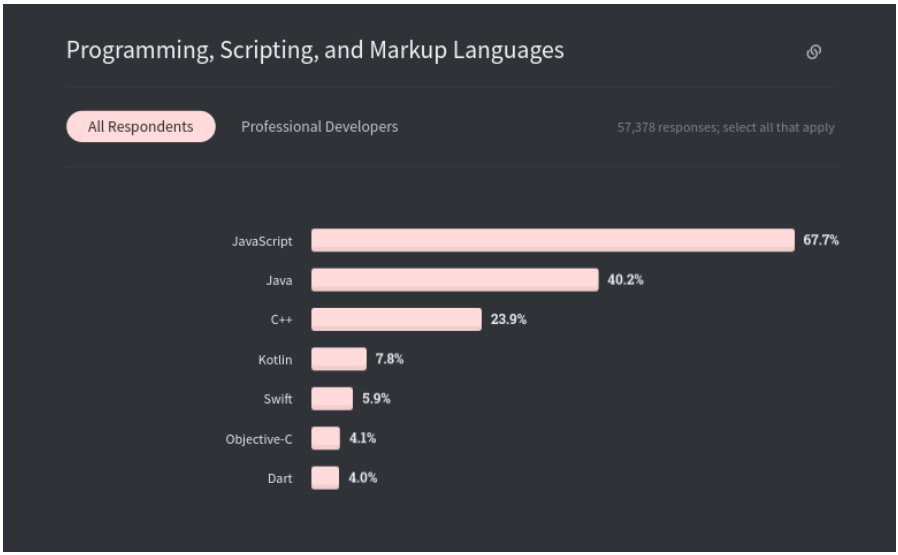
Ďalšia otázka sa týkala popularity programovacích jazykov. Už niekoľko rokov sa na prvom mieste drží jazyk *JavaScript*. Samozrejme - prieskum sa týka technológií celkovo nehľadiac na typ projektov. Ale v prieskume sa nachádzajú aj jazyky, ktoré sú takmer výlučne určené pre tvorbu mobilných aplikácií, ako napr. *Swift*, *Kotlin* a *Dart*.

V ďalších otázkach sa pýtali vývojárov na používané rámce vo vývoji webových aplikácií a ostatných aplikáciách. Nás budú zaujímať tie ostatné, kde sa ocitli aj rámce pre vývoj mobilných aplikácií, kde dominoval React Native[8].

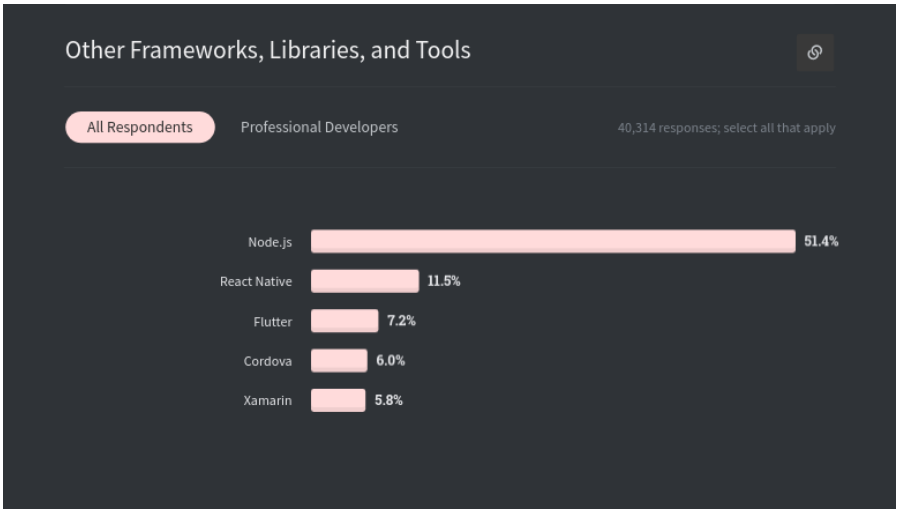
[8] https://reactnative.dev/



Obrázok 1.6: Stack Overflow Developer Survey 2020 - Platforms



Obrázok 1.7: Stack Overflow Developer Survey 2020 - Languages



Obrázok 1.8: Stack Overflow Developer Survey 2020 - Frameworks

Ak by sme mali z tohto prieskumu vyvodiť záver, tak by sme mohli povedať, že:

- popularita u vývojárov je silnejšia pre platformu *Android* ako pre *iOS*; to môže odrážať aj stav, kedy je zastúpenie zariadení s *Android*-om výrazne väčšie ako s *iOS* (a ďalšie systémy pre mobilné zariadenia nestoja za zmienku)
- vzhľadom na popularitu jazykov a rámcov sa dá povedať, je veľmi pravdepodobné, že sa určite stretnete s jazykom *JavaScript*, o čom svedčí aj popularita rámca React Native[9] priamo v prieskume
- v tomto predmete sa budeme venovať hlavne hybridnému vývoju aplikácií pomocou rámca React Native[10] primárne pre *OS Android*

1.4 Welcome to React Native

React Native je rámec pre tvorbu natívnych aplikácií pomocou *JavaScript*-u. Pri písaní aplikácií nepotrebujete poznať detaily cieľovej platformy ako je Android alebo iOS a aj napriek tomu je možné vytvárať komplexné aplikácie.

[9] https://reactnative.dev/
[10] https://reactnative.dev/

Stačí len poznať JavaScript a jeden kód je možné spúšťať na oboch cieľových platformách. Aj vďaka tomu je dnes React Native veľmi populárny, o čom svedčia aj spomínané výsledky prieskumu zo Stack Overflow.

Jedná sa o pomerne mladý projekt, ktorý vytvorila firma *Facebook* v roku 2015.

1.5 Torch Application

Miesto štandardnej *Killer App* s názvom *Hello world* sa pustíme rovno do niečoho použiteľnejšieho. Vytvoríme si aplikáciu, ktorá nemôže chýbať v žiadnej aplikácii typu *Swiss knife* - *flashlight* (baterku). My ju však nazveme poetickejšie - *Torch*

Aby sme si ukázali, ako táto aplikácia bude vyzerať a ako sa bude správať, jednoduché demo sa nachádza na nasledujúcom slajde.

1.6 WarmUp

Ešte predtým, ako začneme so samotným vývojom, si potrebujeme pripraviť prostredie. Potrebujeme zabezpečiť tieto veci:

- nainštalovať vývojové prostredie *Visual Studio Code*
- nainštalovať *node.js*
- nainštalovať rámec *React Native*
- nainštalovať klienta *Expo*
- nainštalovať *Android SDK*
- vytvoriť projekt

> **Upozornenie**
> Poprosím vás, aby ste si potrebný softvér nainštalovali do budúceho cvičenia. Bez neho si toho na cvičeniach veľa neodskúšate!

1.6.1 Installing React Native

Rámec *React Native* nainštalujeme pomocou správcu balíčkov `npm` pre `node.js`. Nainštalujeme ho do systému *globálne* inštaláciou klienta *Expo*, ktorý má rámec *React Native* uvedený v závislostiach. To vykonáme príkazom príkazom:

```
$ npm install -g expo-cli
```

Táto operácia bude samozrejme chvíľu trvať a vo výstupe sa môže objaviť niekoľko upozornení. Tie budeme ignorovať.

1.6.2 Create a project

Nový projekt pomocou nástroja `expo` vytvorím použitím voľby `init`:

```
$ expo init torch
```

Následne môžem pre vytvorenie projektu použiť niektorú z predpripravených šablón:

```
? Choose a template: (Use arrow keys)
  ----- Managed workflow -----
> blank                 a minimal app as clean as an empty
                        canvas
  blank (TypeScript)    same as blank but with TypeScript
                        configuration
  tabs (TypeScript)     several example screens and tabs using
                        react-navigation and TypeScript
  ----- Bare workflow -----
  minimal               bare and minimal, just the essentials
                        to get you started

  minimal (TypeScript)  same as minimal but with TypeScript
                        configuration
```

Ja zostanem kvôli jednoduchosti pri šablóne `blank`.

Po vytvorení projektu sa zobrazí krátky pomocník, ktorý hovorí o tom, ako spustiť projekt:

```
To run your project, navigate to the directory and run one of
the following npm commands.

- cd torch
- npm start # you can open iOS, Android, or web from here, or
  run them directly with the commands below.
- npm run android
- npm run ios # requires an iOS device or macOS for access to
  an iOS simulator
- npm run web
```

Najprv vojdem do priečinku s projektom a spustím v ňom *Visual Studio Code*:

```
$ cd torch
$ code .
```

1.7 Project Structure

Pozrime sa však najprv na to, ako vyzerá štruktúra vytvoreného projektu:

```
$ tree -L 1
.
  App.js
  app.json
  assets/
  babel.config.js
  node_modules/
  package.json
  package-lock.json

2 directories, 5 files
```

Význam jednotlivých súborov a priečinkov je nasledovný:

- `node_modules`, `package.json` a `package-lock.json` - s
- `assets/` - obsahuje všetky (najmä multimediálne) súbory, ktoré budú súčasťou vytváranej aplikácie
- `App.js` - predstavuje základný komponent
- `app.json` - konfiguračný súbor aplikácie

1.8 Metro Bundler

Ak sa pozrieme naspäť na výpis, ktorý sa zobrazil po vytvorení projektu, spustiť projekt máme príkazom `npm start`. Buď teda v samostatnom okne terminálu alebo v termináli vývojového prostredia napíšte príkaz

```
$ npm start
```

> **Upozornenie**
> Ak sa vám po spustení zobrazí chyba

```
Error: ENOSPC: System limit for number of file watchers
reached, watch '/path/to/file'
```

potrebujete vo vašom systéme navýšiť príslušný limit. To docielite príkazom:

```
$ echo fs.inotify.max_user_watches=524288 | \
sudo tee -a /etc/sysctl.conf && sudo sysctl -p
```

(Viac info[a])

[a] https://code.visualstudio.com/docs/setup/linux#_visual-studio-code-is-unable-to-watch-for-file-changes-in-this-large-workspace-error-enospc

Miesto aplikácie sa však spustí Metro Bundler[11]. *Metro Bundler* je *JavaScript*-ový balič (z angl. bundler) pre *React Native*. Je zodpovedný za zabalenie súborov celej aplikácie (kód, závislosti, asset-y) do jedného súboru.

Pomocou *Metro Bundler*-a následne môžeme:

- spustiť aplikáciu na Android zariadení/emulátore,
- spustiť aplikáciu na iOS simulátore,
- spustiť aplikáciu vo webovom prehliadači,
- odoslať odkaz na aplikáciu mailom, a
- publikovať aplikáciu verejne.

1.9 Running Your Application

1.9.1 Running in web browser

Ak v rozhraní nástroja *Metro Bundler* klikneme na položku *Run in web browser* alebo v príkazovom riadku stlačíme kláves `w`, spustí sa aplikácia v prehliadači.

1.9.2 Running on Android device/emulator

Ak v rozhraní nástroja *Metro Bundler* klikneme na položku *Run on Android device/emulator* alebo v príkazovom riadku stlačíme kláves `a`, spustí sa aplikácia v na platforme *Android* .

To, či sa spustí na zariadení alebo v emulátore, záleží od viacerých faktorov:

[11] https://facebook.github.io/metro/

- ak máte k počítaču pripojené zariadenie so systémom *Android* a toto zariadenie má zapnutý *Režim pre vývojárov*, spustí sa aplikácia na ňom
- ak zariadenie k počítaču pripojené nemáte, ale máte nainštalovaný v počítači emulátor so systémom *Android*, automaticky sa spustí tento emulátor

Prvé spustenie v každom prípade trvá dlhšie, nakoľko najprv sa nainštaluje klientska aplikácia *Expo*. Až potom dôjde k nahratiu vytvorenej aplikácie.

1.9.3 Running with QR Code

Pokiaľ je vaše mobilné zariadenie v rovnakej sieti, ako je váš počítač, môžete pomocou čítačky QR kódov naskenovať QR kód z aplikácie *Metro Bundler*. Tým sa automaticky spustí klient *Expo* na vašom zariadení a *Metro Bundler* do neho zabalí a pošle výslednú aplikáciu.

> **Poznámka**
> Samozrejme *Expo* klient musí byť na zariadení nainštalovaný. V opačnom prípade zariadenie nebude vedieť, čo má s adresou načítanou v QR kóde spraviť.

1.10 Fast Refresh

Jednou z výhod rámca *React Native* je "blesková" aktualizácia vykonaných zmien. Ak totiž vykonáte vo svojom kóde zmenu a uložíte ju, tá sa okamžite dostane na všetky výstupné zariadenia. Nepotrebujete teda čakať na preklad a prenos novej binárky na cieľové zariadenie, ako je to v prípade natívneho vývoja.

Túto vlastnosť si vieme vyskúšať v komponente `App`, ktorý sa nachádza v súbore `App.js`. Stačí, ak upravíte text v elemente `<Text>` a súbor uložíte. Zmena sa okamžite dostane na všetky zariadenia, kde ste vašu aplikáciu vypublikovali pomocou nástroja *Metro Bundler*.

1.11 Conclusion

Dnes sme si predstavili typy mobilných aplikácií, predstavili sme si rámec *React Native* a vytvorili sme kostru ukážkovej aplikácie.

Nabudúce budeme v tvorbe aplikácie *Torch* pokračovať.

Prednáška 2

Components and State

komponenty, stav a vlastnosti komponentov, základné komponenty

2.1 Project Torch Overview

Na minulej prednáške sme si predstavili rámec na vývoj natívnych mobilných aplikácií *React Native* a začali sme vyvíjať aplikáciu *Torch*. Dnes budeme vo vývoji tejto aplikácie pokračovať a popri tom sa budeme zoznamovať s vlastnosťami tohto rámca.

Táto aplikácia bude jednoduchou baterkou na svietenie, kde na svetlo využijeme blesk zadnej kamery/fotoaparátu.

2.1.1 Expo SDK 39 is Now Available

Keďže sme toho naposledy veľa nestihli a medzičasom bola vydaná nová verzia Expa s podporou SDK 39, ktorá podporuje React Native 0.63, začneme opäť rýchlym vytvorením aplikácie *Torch* pomocou nástroja `expo-cli`. Aktualizácia je potrebná, pretože v našom projekte budeme používať komponenty, ktoré v SDK 38 ešte nie sú podporované.

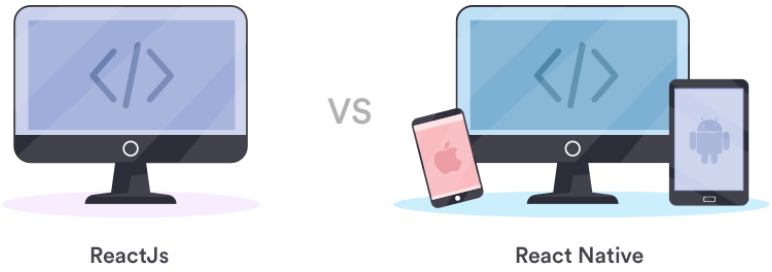
Projekt teda vytvoríme znova:

```
$ expo-cli init torch
# select blank
$ cd torch
$ code .
$ npm start
```

2.2 React Native vs React

Ak chceme zvládnuť prácu v rámci *React Native*, nemôžeme zabudnúť na rámec React[1], na ktorom je *React Native* postavený, a na ktorom je závislý. Obecne môžeme povedať, že rámec React Native sa používa na tvorbu (nie len) mobilných aplikácií pomocou React-u a schopností/možností natívnej platformy. Túto závislosť si môžete všimnúť aj v súbore `App.js`, kde sa *React* importuje riadkom:

```
import React from 'react';
```



Obrázok 2.1: ReactJS vs React Native (zdroj[2])

Pre zvládnutie rámca *React Native* je teda dobré poznať aspoň základy rámca *React*.

Takže - čo je *React*? *React* je knižnica pre *JavaScript* na tvorbu používateľských rozhraní. Tie sa vytvárajú pomocou malých izolovaných častí, ktoré sa nazývajú **komponenty**.

Komponenty je možné navzájom prepájať, čím je možné vytvárať komplexné používateľské rozhrania. Toto prepojenie je možné vizualizovať pomocou stromu, v ktorom existuje jeden koreňový komponent a každý ďalší komponent sa stáva jeho samostatnou vetvou. Každá ďalšia vetva môže mať samozrejme ďalšie podvetvy. Príklad takéhoto stromu komponentov (z angl. *component tree*) sa nachádza na nasledujúcom obrázku:

[1] https://reactjs.org

2.3 Components

Komponenty sú základných stavebným prvkom týchto rámcov. Pozrime sa teda zblízka na to, čo je vlastne komponent.

V našej aplikácii máme zatiaľ jediný komponent, ktorý je súčasne aj koreňovým komponentom aplikácie. Tento komponent sa nachádza v súbore `App.js` a vyzerá takto:

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text>Open up App.js to start working on your app!</Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

Komponent je teda definovaný ako funkcia. Čokoľvek, čo táto funkcia vráti, je vykreslené ako *React element*, ktorý definuje, čo bude viditeľné na obrazovke zariadenia. Definujeme vlastne **pohľad** (**view**) komponentu.

Obecne je každý komponent definovaný svojim **stavom** a **vlastnosťami**. O nich si však porozprávame trochu neskôr.

2.3.1 Views

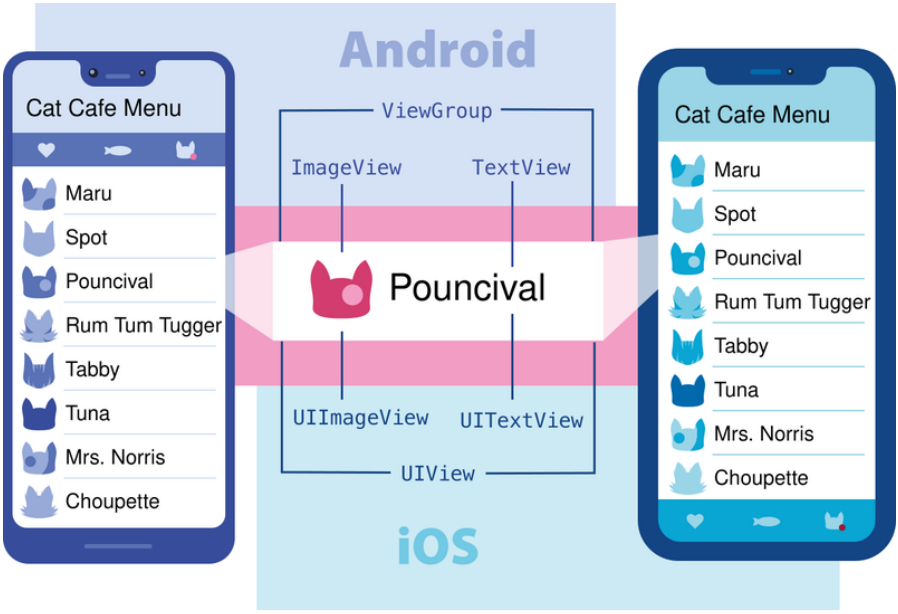
Pri vývoji aplikácií pre Android alebo iOS sú pohľady (views) záladným stavebným blokom pre tvorbu používateľského rozhrania. Obecne sa jedná o elementy obdĺžnikového tvaru, pomocou ktorých vieme zobrazovať text, obrázky, vedia reagovať na vstup od používateľa.

Pohľad (view) je vlastne všetko, čo je možné zobraziť na obrazovke zariadenia. Niektoré pohľady môžu obsahovať aj iné pohľady.

Rámec React Native obsahuje **generické (core) komponenty** (resp. pohľady), pomocou ktorých je možné vyskladať ľubovoľné používateľské rozhranie pre všetky podporované platformy. Následne pri spustení rovnakého kódu na konkrétnej platforme sú tieto pohľady nahradené **natívnymi komponentami** (resp. pohľadmi). Túto situáciu ilustruje nasledujúci obrázok.

React Native má mnoho **Core komponentov**. Ich kompletný zoznam nájdete v dokumentácii API[3]. Najčastejšie však budete pracovať s komponentami,

[3] https://reactnative.dev/docs/components-and-apis



Obrázok 2.2: Different Views in Android and iOS, but similar application [12]

ktoré sú uvedené v nasledujúcej tabuľke.

Tabuľka 2.1: Prehľad najčastejších komponentov React Native [12]

React Native UI Component	Android View	iOS View	Web Analog	Description
`<View>`	`<ViewGroup>`	`<UIView>`	A non-scrollling `<div>`	A container that supports layout with flexbox, style, some touch handling, and accessibility controls
`<Text>`	`<TextView>`	`<UITextView>`	`<p>`	Displays, styles, and nests strings of text and even handles touch events
`<Image>`	`<ImageView>`	`<UIImageView>`	`<img>`	Displays different types of images
`<ScrollView>`	`<ScrollView>`	`<UIScrollView>`	`<div>`	A generic scrolling container that can contain multiple components and views
`<TextInput>`	`<EditText>`	`<UITextField>`	`<input type="text">`	Allows the user to enter text

2.3.2 JSX

Keď sa však pozrieme bližšie na to, čo funkcia v skutočnosti vracia, nie je to ani reťazec a nie je to ani HTML. *React* totiž používa syntax JSX[4] (*JavaScript XML*), čo je XML/HTML rozšírenie pre kód jazyka JavaScript. React nie je na technológii JSX závislý, ale jeho použitím je tvorba komponentov jednoduchšia, prehľadnejšia a hlavne prirodzenejšia. Za syntaxou JSX stojí rovnako spoločnosť *Facebook*.

Príklad jednoduchého `Hello, world!` komponentu môže vyzerať takto:

```
function HelloWorld() {
    return <Text>Hello, world!</Text>;
}
```

Aby React (a vlastne JavaScript) rozumel JSX, je potrebné použiť transpiler Babel, ktorý dokáže preložiť tento značkovací jazyk do jazyka JavaScript. Ten okrem toho pozná aj všetky novinky, ktoré priniesol ES6 (ECMAScript 2015).

2.4 Button Component

Do UI našej aplikácie potrebujeme pridať tlačidlo, pomocou ktorého budeme *baterku* ovládať. Na to použijeme komponent `Button`, ktorý sa podľa potreby vyrenderuje na každej platforme.

Komponent bude vyzerať nasledovne:

```
<Button
  onPress={onPressButton}
  title="Turn On"
/>
```

Ako je možné vidieť, komponent tlačidla má dva povinné atribúty:

- `title` - text, ktorý sa na tlačidle zobrazí
- `onPress` - funkcia, ktorá bude zavolaná po stlačení tlačidla.

Z komponentu `App` môžeme vypustiť komponent `<StatusBar>` a pridáme do neho komponent `<Button>`:

```
export default function App() {
    return (
        <View style={styles.container}>
            <Text>
```

[4] https://facebook.github.io/jsx/

```
                Open up App.js to start working on your app!
            </Text>
            <Button
                onPress={function () {
                    console.log("Button was pressed");
                }}
                title="Turn On"
            />
        </View>
    );
}
```

Ak tlačidlo stlačíme, v konzole *Metro Bundler*-a alebo v nástroji *Visual Studio Code* uvidíme vypísaný text.

2.5 Component State

Stlačením tlačidla chceme, aby sa baterka

- rozsvietila, ak je zhasnutá, alebo
- zhasla, ak je rozsvietená.

Našu aplikáciu by sme teda mohli nazvať aj stavovým strojom, pretože v ktoromkoľvek momente vieme povedať, že sa bude nachádzať v jednom dvoch stavov:

- **zhasnutá**, a
- **rozsvietená**.

V rámci *React Native* môže mať každý komponent definovaný aj svoj vlastný **stav**, ktorý si bude následne pamätať. Stav sa môže v čase **meniť** napr. na základe interakcie používateľa (stlačením tlačidla na baterke).

2.5.1 Definig State in React Native

Pridať stav do komponentu je možné pomocou volani hook-u `useState()` nasledovne:

```
const [<getter>, <setter>] = useState(<initialValue>);
```

kde:

- `getter` reprezentuje premennú, ktorá bude stav komponentu reprezentovať,

- `setter` reprezentuje názov funkcie, pomocou ktorej bude možné zmeniť stav, a
- `initialValue` je hodnota, ktorá reprezentuje počiatočný stav komponentu.

2.5.2 Refactoring

V našom prípade môžeme nazvať stavovú premennú `isOn`, ktorá bude inicializovaná na hodnotu `false` (nesvieti). Funkcia na zmenu stavu sa bude volať `setIsOn`:

```
const [isOn, setIsOn] = useState(false);
```

Kód komponentu bude teda vyzerať nasledovne:

```
export default function App() {
    const [isOn, setIsOn] = useState(false);

    return (
        <View style={styles.container}>
            <Text>
                Open up App.js to start working on your app!
            </Text>
            <Button
                onPress={function () {
                    console.log("Button was pressed");
                }}
                title="Turn On"
            />
        </View>
    );
}
```

Stav sa bude meniť pri stlačení tlačidla. Upravíme teda anonymnú funkciu v komponente `Button`, ktorá sa zavolá pri stlačení tlačidla:

```
return (
    <View style={styles.container}>
        <Text>
            Open up App.js to start working on your app!
        </Text>
        <Button
            onPress={function () {
```

```
                setIsOn(!isOn);
                console.log(isOn);
            }}
            title="Turn On"
        />
    </View>
);
```

Zmenu stavu môžeme teraz vidieť vypísanú v konzole, kde sa bude stlačením tlačidla striedať hodnota `false` a `true`.

Na zmenu stavu môžeme zareagovať aj priamo v aplikácií zmenou textu v komponente `Text` a zmenou názvu tlačidla v komponente `Button`:

```
export default function App() {
    const [isOn, setIsOn] = useState(false);

    return (
        <View style={styles.container}>
            <Text>{isOn === true ? "On" : "Off"}</Text>
            <Button
                onPress={function () {
                    setIsOn(!isOn);
                    console.log(isOn);
                }}
                title={isOn === true ? "Turn Off" : "Turn On"}
            />
        </View>
    );
}
```

Poznámka

Ak budete pozorne sledovať, všimnete si, že pri ošetrení stlačenia tlačidla nedôjde k aktualizovaní hodnoty stavu okamžite. To je spôsobené tým, že funkcia `setIsOn()` pridá zmenu stavu ako úlohu do fronty, ktorá sa vyprázdňuje postupne. Ak chcete ale zareagovať okamžite, funkcia `setIsOn()` môže ako parameter dostať funkciu, ktorá vráti nový stav. Aktualizovaný komponent `Button` môže vyzerať nasledovne:

```
<Button
    onPress={function () {
        console.log(`>> before: ${isOn}`);
        setIsOn(function(state){
            state = !state;
            console.log(`>> after: ${state}`);
            return state;
        });
    }}
    title={isOn === true ? "Turn Off" : "Turn On"}
/>
```

2.6 Image Component

Jediné, čo nám aktuálne chýba do používateľského rozhrania baterky, je obrázok reprezentujúci stav baterky. Ten vytvoríme pomocou komponentu `Image`.

Komponent `Image` je možné použiť na zobrazenie obrázkov z rôznych zdrojov, ako

- obrázky umiestnené v internete/sieti,
- statické obrázky,
- dočasné lokálne obrázky, a
- obrázky z lokálneho disku.

Pre rozličné typy obrázkov je možné použiť rozličné atribúty. Preto komponent `Image` nemá povinné props, ako tomu bolo v prípade komponentu `Button`.

Pre naše potreby budeme pracovať s obrázkami, ktoré budú statickou súčasťou aplikácie. Tieto obrázky, ktoré reprezentujú zasvietenú a zhasnutú žiarovku, uložíme do priečinku `assets/`

Ak ich budeme chcieť použiť v komponente `Image`, použijeme property `source`, kde sa na príslušný obrázok odkážeme pomocou volania `require()`. Upravíme teda pohľad komponentu `App` pridaním komponentu `Image` medzi text a tlačidlo s obrázkom zhasnutej žiarovky:

```
<Image source={require("./assets/bulb_off.png")}></Image>
```

Tu je však potrebné si uvedomiť niekoľko vecí:

- výsledkom volania funkcie `require()` je len číslo zdroja, ktoré bundler zabalil do projektu
- funkcia `require()` sa volá pri balení aplikácie a teda nemôže obsahovať dynamicky konštruované reťazce

Upravíme teda kód komponentu tak, aby sa potrebný obrázok vybral mimo komponentu:

```
export default function App() {
    const [isOn, setIsOn] = useState(false);

    var image = isOn
        ? require("./assets/bulb_on.png")
        : require("./assets/bulb_off.png");

    console.log(image);

    return (
        <View style={styles.container}>
            <Text>{isOn === true ? "On" : "Off"}</Text>
```

```
            <Image source={image}></Image>
            <Button
                onPress={function () {
                    console.log(`>> before: ${isOn}`);
                    setIsOn(function (state) {
                        state = !state;
                        console.log(`>> after: ${state}`);
                        return state;
                    });
                }}
                title={isOn === true ? "Turn Off" : "Turn On"}
            />
        </View>
    );
}
```

2.7 Pressable Image

Jediné, čo nám aktuálne chýba v našej aplikácii, je možnosť zmeniť jej stav stlačením obrázku. Ak sa však pozrieme na vlastnosti komponentu `Image`, nenájdeme v zozname nič, čo by pripomínalo možnosť ošetriť stlačenie podobne, ako tomu bolo v prípade obrázka. V tomto prípade treba postupovať ináč.

Na použitie obrázku ako tlačidla použijeme komponent `Pressable`. Ten funguje ako istý kontajner, do ktorého zabalíme komponent, ktorý má byť stlačiteľným.

> **Poznámka**
> Komponent `Pressable` je novinkou v rámci *React Native*. Pred jeho zavedením sa používali komponenty s prefixom `Touchable*`, ako napr. `TouchableHighlight`, `TouchableOpacity` a `TouchableWithoutFeedback`. Komponent `Pressable` ich má postupne nahradiť, takže sa dá očakávať, že uvedené komponenty zmiznú z budúcich verzií rámca *React Native*.

Do komponentu `Pressable` teda zabalíme komponent `Image`:

```
<Pressable
    onPress={function () {
```

```
        console.log("Image was pressed.");
    }}
>
    <Image source={image} />
</Pressable>
```

Keďže po stlačení obrázka očakávame rovnaké správanie ako po stlačení tlačidla, vytvoríme samostatnú funkciu, ktorú budeme volať v prípade stlačenia oboch komponentov. Upravený komponent `App` bude vyzerať takto:

```
export default function App() {
    const [isOn, setIsOn] = useState(false);

    var image = isOn
        ? require("./assets/bulb_on.png")
        : require("./assets/bulb_off.png");

    const onPressed = function () {
        console.log(`>> before: ${isOn}`);
        setIsOn(function (state) {
            state = !state;
            console.log(`>> after: ${state}`);
            return state;
        });
    };

    return (
        <View style={styles.container}>
            <Text>{isOn === true ? "On" : "Off"}</Text>
            <Pressable onPress={onPressed}>
                <Image source={image} />
            </Pressable>
            <Button
                onPress={onPressed}
                title={isOn === true ? "Turn Off" : "Turn On"}
            />
        </View>
    );
}
```

2.8 Complete Solution

Po drobnom refaktorovaní bude výsledné riešenie vyzerať nasledovne:

```
import React, { useState } from "react";
import {
  StyleSheet, View, Button, Image, Pressable
} from "react-native";

export default function App() {
  const [isOn, setIsOn] = useState(false);

  var image = isOn
      ? require("./assets/bulb_on.png")
      : require("./assets/bulb_off.png");

  const toggleState = function () {
      setIsOn(!isOn);
  };

  return (
      <View style={styles.container}>
          <Pressable onPress={toggleState}>
              <Image source={image} />
          </Pressable>
          <Button
              onPress={toggleState}
              title={isOn === true ? "Turn Off" : "Turn On"}
          />
      </View>
  );
}

const styles = StyleSheet.create({
  container: {
      flex: 1,
      backgroundColor: "#fff",
      alignItems: "center",
      justifyContent: "center",
  },
});
```

2.9 Conclusion

Dnes sme teda vytvorili kostru používateľského rozhrania aplikácie baterka. Popri tom sme sa zoznámili s niektorými komponentmi rámca *React Native* a predstavili sme si ich vlastnosti. Nabudúce budeme pokračovať a pokúsime sa rozsvietiť blesk fotoaparátu vo vašom zariadení. Samozrejme len v prípade, že ho zariadenie obsahuje vo svojej výbave.

Prednáška 3

App Features

ikona aplikácie, class based komponent, inštalácia modulov tretích strán

3.1 Project Torch Overview

Pokračujeme vo výrobe baterky, ktorá bude na svetlo využívať blesk zadnej kamery/fotoaparátu.

3.2 Application Icon

nastaviť ikonu aplikácie

licencia: tango icons (https://commons.wikimedia.org/wiki/Tango_icons)

nahrat do `assets/`

aktualizovat `app.json`

reštartnúť/znovu načítať aplikáciu v Expo kliente na zariadení/v emulátore

3.3 Class Component

ešte kým máme toho kódu málo, pozrieme sa na to, ako bude vyzerať ako trieda

takto vyzerali komponenty v React-e aj React Native donedávna

```
import React, { Component } from "react";
import {
```

Obrázok 3.1: Application Icon (zdroj[1])

```
  StyleSheet, View, Button, Image, Pressable
} from "react-native";

export default class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
            isOn: false,
        };
    }

    toggleState() {
        const { isOn } = this.state;
        console.log(">> state change...");
        this.setState({ isOn: !isOn });
    }

    render() {
        const image = this.state.isOn
            ? require("./assets/bulb_on.png")
            : require("./assets/bulb_off.png");
```

```
        return (
            <View style={styles.container}>
                <Pressable onPress={this.toggleState.bind(this)}>
                    <Image source={image} />
                </Pressable>
                <Button
                    onPress={this.toggleState.bind(this)}
                    title={this.state.isOn === true ?
                        "Turn Off" : "Turn On"}
                />
            </View>
        );
    }
}
```

3.4 Houston, We Have a Problem!

Expo nemá možnosť pracovať s bleskom foťáku

- `expo-camera`, ale robí náhľad z foťáku + ovláda blesk, ale nejde len samostatne ovládať blesk

Kedy sa teda oplatí používať Expo? Aké výhody ponúka?

- Fastest way to build React Native Apps
- You don't need to know Native Mobile coding
- No Xcode, No Android Studio
- Publish Over The Air (OTA) Updates Instantly
- In-built access to Native APIs
- It is FREE and Open Source

A kedy sa naopak neoplatí Expo používať? Aké sú nevýhody Expa?

- Not all iOS and Android APIs are available yet

Ak to zhrnieme, tak Expo je výborné, ak ste noví vo vývoji aplikácií pre chytré zariadenia a nemáte (alebo nechcete mať) skúsenosti s natívnym vývojom aplikácií. Ak naopak skúsenosti s natívnym vývojom máte alebo potrebujete vo svojej aplikácii pracovať so súčasťami, ktoré nie sú zatiaľ podporované v Expo SDK, Expu sa vyhnite.

3.4.1 Ejecting from Expo

Nemusíme začínať projekt odznova, ale môžeme spraviť tzv. *"eject"* z Expa. To vykonáme spustením nasledovného príkazu z koreňového priečinku projektu:

```
$ expo eject
```

Následne prejdeme krátkym dialógom, kde sa nás systém opýta na balík, v ktorom sa bude novovytváraná aplikácia pre Android aj iOS nachádzať. Ako balík uvedieme `sk.tuke.smart.torch`.

3.4.2 Running the Project

Podobne, ako v prípade Expo, je aj tu odporúčanie nespúšťať príkazy priamo cez `npm`, ale pomocou nástroja `npx`.

Najprv spustíme *Metro Bundler* príkazom:

```
$ npx react-native start
```

Jeho podoba síce aktuálne nebude taká sexi ako v prípade *Expo*-a, ale rovnako bude Metro spustené na pozadí a bude sledovať zmeny v projekte.

V novom termináli spustíme príkaz:

```
$ npx react-native run-android
```

Tým dôjde:

- k spusteniu emulátora, ak nie je pripojené Android zariadenie
- k zostaveniu aplikácie pre Android
- nainštalovaniu a spusteniu aplikácie na Android zariadení.

3.5 Understanding the Android Configuration

na pozadí sa však stalo niečo, čo hneď nevidieť:

- stiahlo sa Android API 29
- aplikácia sa zostavila na základe Android API 29

pozrime sa teda na konfiguráciu Android projektu

3.5.1 `compileSdkVersion`, `minSdkVersion` and `targetSdkVersion`

Prvý súbor, ktorý nás bude zaujímať je konfiguračný súbor `build.gradle` v priečinku `android/`. Zaujímať nás bude časť označená ako `buildscript`:

```
buildscript {
    ext {
        buildToolsVersion = "30.0.2"
        minSdkVersion = 23
        compileSdkVersion = 29
        targetSdkVersion = 29
    }
}
```

Význam jednotlivých volieb je nasledovný:

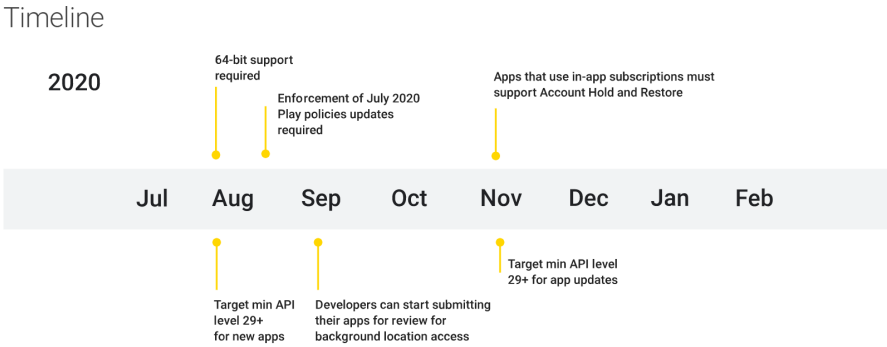
- `compileSdkVersion` - Táto voľba hovorí, akú verziu *Android SDK* má *Gradle* použiť pri preklade vašej aplikácie. Odporúča sa používať vždy najnovšiu verziu SDK.
- `minSdkVersion` - Označuje minimálnu verziu *Android SDK*, ktorú má vaša aplikácia podporovať. Rámec *React Native* podporuje všetky verzie *Android*-u od 4.1 (od API 16).
- `targetSdkVersion` - Označuje verziu *Android*-u, na ktorú svoju aplikáciu cielite.

To teda znamená, že:

- hodnota `compileSdkVersion` by mala vždy byť najnovšia dostupná (alebo aspoň nie veľmi pozadu)
- hodnota `minSdkVersion` by mala reprezentovať verziu najstaršieho Android-u, pre ktorý chceme, aby bola aplikácia spätne kompatibilná, v našom prípade teda 23
- hodnota `targetSdkVersion` by mala reprezentovať verziu systému, na ktorú je aplikácia cielená. A keďže spoločnosť Google stanovila, že do novembra 2020 musia mať aplikácie, ktoré sú publikované na *Play Store*, nastavenú cieľovú verziu SDK na hodnotu 29 a vyššie, nemáme o čom špekulovať ;)

3.5.2 `AndroidManifest.xml`

Hodnota `targetSdkVersion` a `minSdkVersion` sa budú nachádzať aj vo finálnej aplikácii v súbore `AndroidManifest.xml`. Ak ich tam pridáte ručne,



Obrázok 3.2: Apps must target API level 29 (Zdroj[2])

budú ignorované, nakoľko ich hodnoty budú prepísané nástrojom Gradle na základe konfiguračného súboru `build.gradle`.

3.6 Application Icon Again

Aplikácia sa teraz spustí bez Expo klienta. Jej funkcionalita zostala síce zachovaná, ale ak sa pozrieme na to, ako tentokrát vyzerá ikona aplikácie, tak v závislosti od rozlíšenia zariadenia môže vyzerať rozlične. To je spôsobené tým, že v prípade Android-a už záleží na tom, na akom zariadení je aplikácia spustená.

V priečinku `android/` sa okrem konfigurácie a zdrojových kódov nachádzajú aj zdroje (resources) aplikácie. Medzi zdroje patrí aj ikona aplikácie, tzv. spúšťača aplikácie. Tie sa nachádzajú v priečinku `android/app/src/main/res/`.

V tomto priečinku sa nachádzajú priečinky s rozličnými postfixmi na základe rozlíšení[3] cieľových zariadení:

- **xxxhdpi** (extra-extra-extra-high) ~ 640dpi
- **xxhdpi** (extra-extra-high) ~ 480dpi
- **xhdpi** (extra-high) ~ 320dpi
- **hdpi** (high) ~ 240dpi
- **mdpi** (medium) ~ 160dpi
- **ldpi** (low) ~ 120dpi

Aby sme sa veľmi nenamakali a neupravovali ikonu pre každý jeden typ zvlášť,

[3] http://developer.android.com/guide/practices/screens_support.html#range

môžeme na to použiť on-line nástroj Android Asset Studio[4]. Ten obsahuje niekoľko nástrojov, pričom jedným z nich je práve nástroj na tvorbu ikon spúšťačov. Zrejme po vzore tohto online nástroja vzniklo *Asset Studio*, ktoré je súčasťou aj *Android Studia*.

Prejdeme teda do *Launcher Icon Generator*-a, kde nahráme vlastnú ikonu:

Obrázok 3.3: Launcher Icon (zdroj[5])

Keď budeme spokojní so všetkými úpravami, klikneme na tlačidlo *Download ZIP*. Stiahneme k sebe balík s upravenými ikonami pre jednotlivé rozlíšenia. Vo vnútri balíka sa nachádza priečinok `res/`, ktorý rozbalíme cez `res/` priečinok v našom projekte.

> **Poznámka**
> Miesto, kde definujeme, ktorá ikona sa v skutočnosti použije ako ikona aplikácie, sa nachádza v *manifeste* aplikácie (v súbore `AndroidManifest.xml`). Konkrétne sa jedná o atribút elementu `<application>` v tvare:

[4] https://romannurik.github.io/AndroidAssetStudio

```
android:icon="@mipmap/ic_launcher"
```

3.7 Let There be Light!

React Native nevie priamo ovládať baterku

skúsime vyhľadať vhodný modul cez https://www.npmjs.com/

- react native flashlight

pouzijeme modul react-native-torch[6]

inštalácia modulu

```
$ npm install --save react-native-torch
$ react-native link react-native-torch
```

môžeme skúsiť teraz aplikáciu preložiť a spustiť

Upozornenie
Môže sa stať, že proces stuhne na časti appDebug. V tom prípade vyskúšajte:

- vojdite do priečinku `android/` a spustite:

  ```
  $ ./gradlew clean
  ```

  vyjdite von a znova spustite preklad pomocou

  ```
  $ npx react-native run-android
  ```

- reštartnite adb server pomocou

  ```
  $ adb kill-server
  $ adb start-server
  ```

aktualizovanie kódu pre ošetrenie stlačenia tlačidla:

```
import Torch from 'react-native-torch';

...

const toggleState = function () {
    Torch.switchState(!isOn);
```

[6] https://www.npmjs.com/package/react-native-torch

```
    setIsOn(!isOn);
};
```

3.8 Complete Solution

Dnes zmien priamo v kóde veľa nebolo, pretože sme sa venovali viac projektu a jeho nastavení. Kód výsledného komponentu bude teda vyzerať nasledovne:

```
import React, { useState } from "react";
import {
    StyleSheet, View, Button, Image, Pressable
} from "react-native";
import Torch from "react-native-torch";

export default function App() {
    const [isOn, setIsOn] = useState(false);

    var image = isOn
        ? require("./assets/bulb_on.png")
        : require("./assets/bulb_off.png");

    const toggleState = function () {
        Torch.switchState(!isOn);
        setIsOn(!isOn);
    };

    return (
        <View style={styles.container}>
            <Pressable onPress={toggleState}>
                <Image source={image} />
            </Pressable>
            <Button
                onPress={toggleState}
                title={isOn === true ? "Turn Off" : "Turn On"}
            />
        </View>
    );
}

const styles = StyleSheet.create({
```

```
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
    },
});
```

Prednáška 4

Toasts and Alerts

komunikácia s používateľom pomocou `Toast`-u a `Alert` dialógu, ukončenie aplikácie, zistenie platformy

4.1 Introduction

stále pracujeme na baterke

- už nám síce pracuje (a čo je najdôležitejšie - svieti), ale to nie je všetko
- musíme ošetriť napr. prípady, kedy bude aplikácia spustená na zariadení, ktoré nie je vybavené bleskom a teda svietiť nebude
- a taktiež musíme vyriešiť situáciu, keď aplikácia bude spustená na *Android*-e, ale nebude mať práva na prístup ku blesku
- a popri tom by bolo dobré komunikovať s používateľom a informovať ho o všetkom, k čomu došlo

4.2 Device doesn't have a Torch

Prípad, kedy zariadenie nie je vybavené bleskom, bude zastupovať emulátor. Pozrime sa teda na spustenie aplikácie na ňom.

Ak aplikáciu spustíme, tak sa spustí normálne. Ak sa však pokúsime baterku zasvietiť, skončíme s hláškou `Possible Unhandled Promise Rejection` a so správou `Error: Bad argument passed to camera service`.

Táto chyba priamo súvisí s kamerou, kedy dochádza k požiadavke o jej ovládanie na zariadení, ktoré ale nemá blesk.

Fragment kódu, ktorý slúži na ovládanie blesku vo funkcii `toggleState()`

teda obaľme do volania `try-catch` tak, ako je to uvedené v dokumentácii modulu `react-native-torch`[1]. To znamená, že z funkcie musíme spraviť asynchrónnu funkciu a počkáme si na výsledok volania `Torch.switchState()` pomocou kľúčového slova `await`:

```
const toggleState = async function () {
    try {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    } catch (e) {
        console.log(e);
    }
};
```

4.3 Notifying the User

V kóde sme síce situáciu o neexistencii blesku vyriešili bez toho, aby beh aplikácie zlyhal, musíme však o vzniknutej situácii informovať aj používateľa. Na to nám totiž rozhodne nebude stačiť vypisovanie do konzoly cez `console.log()`.

Na to samozrejme môžeme použiť niekoľko spôsobov.

4.3.1 Toast

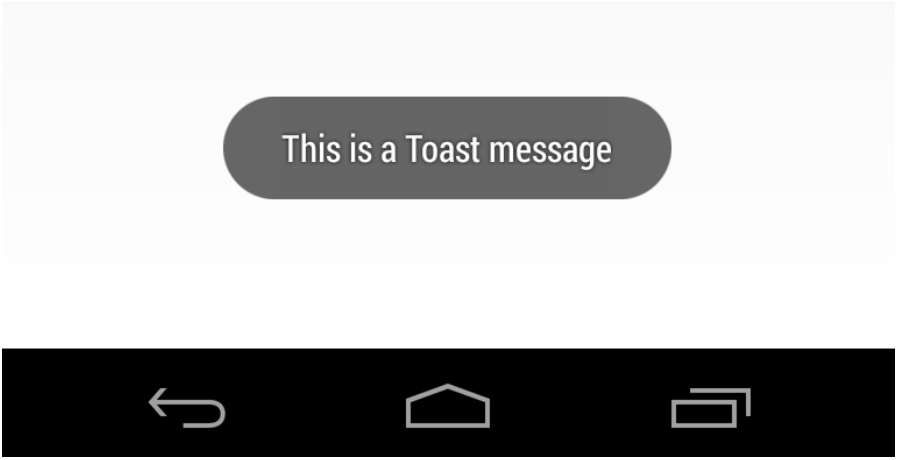
React Native's ToastAndroid API exposes the Android platform's ToastAndroid module as a JS module. It provides the method `show(message, duration)` which takes the following parameters:

- *message* A string with the text to toast
- *duration* The duration of the toast-–either `ToastAndroid.SHORT` or `ToastAndroid.LONG`

Upravíme kód:

```
// import ToastAndroid first
import { ToastAndroid } from "react-native";

const toggleState = async function () {
    try {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
```

[1] https://www.npmjs.com/package/react-native-torch



Obrázok 4.1: Android Toast

```
    } catch (e) {
        ToastAndroid.show(
            "No camera available. Go and buy a device \
            with some and come back later.",
            ToastAndroid.SHORT
        );
    }
};
```

Toto však funguje len na *Android*-e. Aby sme to použili správne, môžeme si overiť platformu pomocou modulu `Platform`:

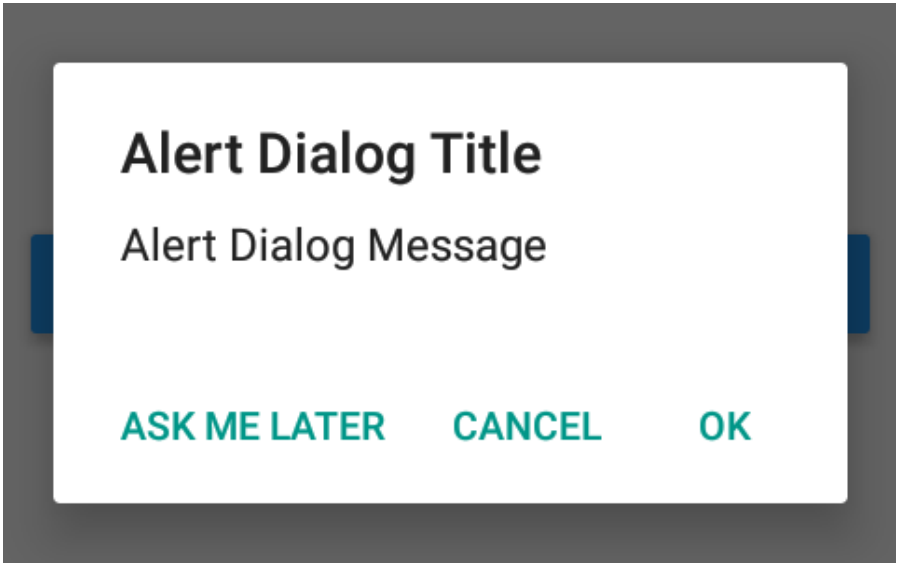
```
// import Platform first
import { Platform } from "react-native";

const toggleState = async function () {
    try {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    } catch (e) {
        if(Platform.OS === 'android'){
            ToastAndroid.show(
                "No camera available. Go and buy a device \
                with some and come back later.",
```

```
                ToastAndroid.SHORT
            );
        }else{
            console.log('No camera available. Go and buy \
            a device with some and come back later.')
        }
    }
};
```

Ak chceme, aby aplikácia fungovala multiplatformne, tak `Toast` asi nie je najlepší nápad, aj keď vyzerá dobre ;) Existujú však rozšírenia, pomocou ktorých je možné správanie Toast-u zabezpečiť aj na iných platformách. Napríklad react-native-simple-toast[2] alebo react-native-toast-message[3].

4.3.2 Alert

Launches an alert dialog with the specified title and message.



Obrázok 4.2: Alert Dialog

Poznáme zo štandardného JavaScript-u v prehliadači, ale v React Native re-

[2] https://www.npmjs.com/package/react-native-simple-toast
[3] https://www.npmjs.com/package/react-native-toast-message

prezentuje API, ktoré funguje na platformách *Android* aj *iOS*. Existuje však rozdiel v jeho používaní na jednotlivých platformách. O tom sa dočítate v dokumentácii[4].

Obecne sa `Alert` skladá z:

- **titulku** / nadpisu,
- **správy**, a
- voliteľne zo **zoznamu tlačidiel** s možnosťou vlastného ošetrenia ich stlačenia

Jednoduchý `Alert` pre náš prípad môže vyzerať nasledovne:

```
// update the import for Alert
import { Alert } from "react-native";

const toggleState = async function () {
    try {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    } catch (e) {
        Alert.alert(
            'Error',
            'No camera available. Go and buy a device \
            with some and come back later.'
        );
    }
};
```

4.4 Exit App

Používateľa sme teda síce upozornili, takže teraz môže klikať a šťukať a bude mu zobrazovať len `Alert` dialóg. To je však celkom trápne, nakoľko vlastne na existencii blesku stojí celá naša aplikácia. Vhodnejšie by bolo napríklad aplikáciu rovno vypnúť, keď príde na to, že blesk nemá.

Priama podpora pre vypnutie aplikácie v *React Native* však nie je. Túto funkcionalitu nám teda zabezpečí externý modul s názvom react-native-exit-app[5]. Nainštalujeme ho príkazom:

[4] https://reactnative.dev/docs/alert
[5] https://www.npmjs.com/package/react-native-exit-app

```
$ npm install react-native-exit-app --save
```

a zlinkujeme s projektom príkazom:

```
$ react-native link react-native-exit-app
```

Upozornenie
Je pravdepodobné, že bude potrebné znovu zostaviť celý projekt. Preto najprv vojdite do priečinku `android/` a spustite príkaz:

```
$ ./gradlew clean
```

Následne sa vráťte do koreňového priečinku a spustite aplikáciu príkazom:

```
$ npx react-native run-android
```

Jej použitie je následovne veľmi jednoduché - na vhodnom mieste stačí zavolať:

```
import RNExitApp from 'react-native-exit-app';
...
RNExitApp.exitApp();
...
```

Upravíme teda kód našej aplikácie tak, že po kliknutí na tlačidlo v `Alert` dialógu dôjde k ukončeniu aplikácie pomocou tohto volania:

```
const toggleState = async function () {
    try {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    } catch (e) {
        Alert.alert(
            "Error",
            "No camera available. Go and buy a device \
            with some and come back later.",
            [
                {
                    text: "Quit",
                    onPress: function () {
                        RNExitApp.exitApp();
                    },
```

```
                },
            ]
        );
    }
};
```

4.5 Test Before Run

Aktuálne teda funguje všetko ako má. Ak sa však zamyslíme nad fungovaním, tak nie je veľmi praktické, aby používateľ aplikáciu musel spustiť a až po kliknutí na obrázok alebo tlačidlo došlo k vyhodnoteniu, či je alebo nie je blesk k dispozícii. Toto naozaj nie je UX, ktorý by sme chceli mať. Ak sa zamyslíme, tak vieme, že kód sa vykonáva v rámci funkcie zhora nadol, pričom funkcia musí vrátiť *view* komponentu. Ak teda na začiatku kódu vložíme fragment, ktorým overíme, či máme alebo nemáme blesk k dispozícii, vyriešili by sme tento problém.

4.6 Conclusion

Na to, aby sme problém so spustením špecifického kódu po spustení kontroléru vyriešili, potrebujeme poznať a porozumieť **životnému cyklu kontajnera**. A ten si predstavíme nabudúce.

Prednáška 5

Component Life Cycle

životný cyklus komponentov reprezentovaných triedami a funkciami, povolenia aplikácie v OS Android

5.1 Adverts and Annoucements

5.1.1 Tool Genymotion[1]

- emulátor postavený na projekte Android-x86[2]
- spúšťaný pomocou VirtualBox-u
- ľahší manažment, nie je až taký žrút ako oficiálny emulátor
- nemá však všetky vlastnosti pôvodného emulátoru
  - napr. NFC
  - zrejme len v neplatenej verzii

5.1.2 Tool scrcpy[3]

- This application provides display and control of Android devices connected on USB (or over TCP/IP[4]). It does not require any *root* access. It works on *GNU/Linux*, *Windows* and *macOS*.
- How to Mirror & Control Your Android Phone from the Ubuntu Desktop[5]

[1] https://www.genymotion.com
[2] https://www.android-x86.org
[3] https://github.com/Genymobile/scrcpy
[4] https://www.genymotion.com/blog/open-source-project-scrcpy-now-works-wirelessly/
[5] https://www.omgubuntu.co.uk/2019/07/scrcpy-mirror-android-to-ubuntu-linux

5.2 Introduction

stále pracujeme na baterke

- baterka svieti
- Dokonca vieme aj v prípade, že zariadenie baterkou nedisponuje, zobraziť správu a aplikáciu vypnúť. To však robíme neskoro - až vtedy, keď príde požiadavka na zasvietenie/zhasnutie.
- Dnes vykonáme túto kontrolu rovno pri spustení aplikácie.

5.3 Component Lifecycle

Keďže je rámec *React Native* založený na rámci *React*, jeho komponenty sa riadia **životným cyklom komponentov** *React*-u. Tento životný cyklus je reprezentovaný niekoľkými metódami, ktoré je možné v našej implementácii prepísať (override).

Životný cyklus je možné ilustrovať nasledovným diagramom[6], ktorý nám poslúži ako istý ťahák:

Aby sme mu lepšie rozumeli, potrebujeme porozumieť nasledovným termínom:

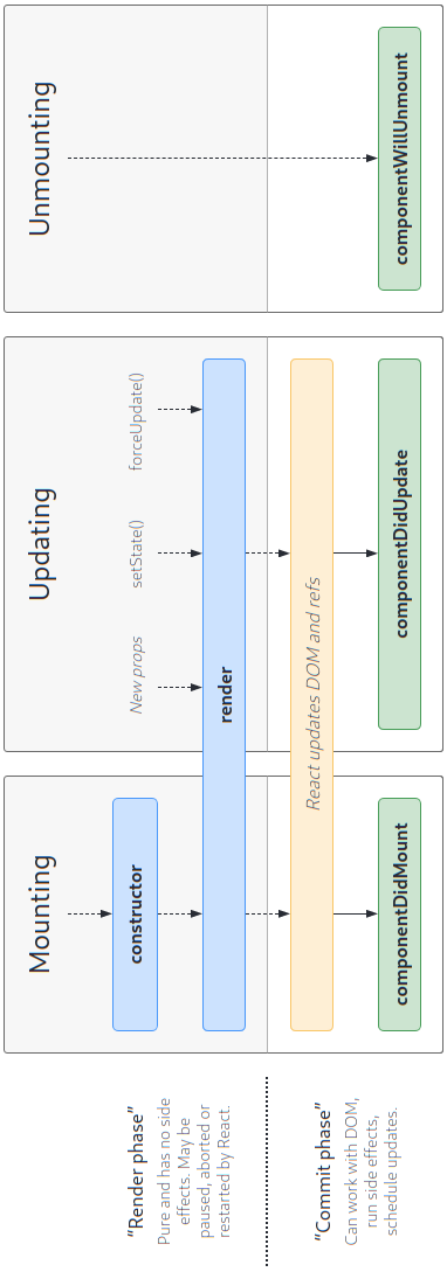
- **mounting** - proces pripojenia komponentu do DOM-u
- **updating** - proces aktualizácie vlastností komponentu
- **unmounting** - proces odpojenia komponentu z DOM-u

5.3.1 Component Lifecycle in Class Components

V prípade reprezentácie komponentov pomocou tried vieme do tohto procesu vstúpiť prepísaním metód

- `constructor()`[7] - Konštruktor komponentu je volaný ešte predtým, ako je komponent pripojený. Obyčajne sa konštruktor používa z dvoch dôvodov:
  1. na inicializáciu lokálneho stavu priradením objektu do `this.state`.
  2. na prihlásenie metód v prípade vzniku udalosti
- `componentDidMount()`[8] - Metóda je volaná okamžite po pripojení do DOM-u. Používa sa na inicializáciu, pri ktorej sa vyžaduje existencia

[6] https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/
[7] https://reactjs.org/docs/react-component.html#constructor
[8] https://reactjs.org/docs/react-component.html#componentdidmount



Obrázok 5.1: React Component Lifecycle [28]

uzlov v DOM-e. Metóda je rovnako dobrým miestom, ak potrebujete získať údaje zo vzdialenej služby. Môžete sa v nej tiež prihlásiť na odber udalostí. V tom prípade sa z nich nezabudnite odhlásiť v metóde `componentWillUnmount()`.

- `componentDidUpdate()`[9] - Metóda je zavolaná okamžite po aktualizácii komponentu. Nie je však zavolaná po prvom renderovaní (volaní `render()`). Táto metóda sa používa v prípadoch, ktoré súvisia s aktualizovaním komponentu.
- `componentWillUnmount()`[10] - Je zavolaná tesne predtým, ako je komponent odpojený a odstránený. Funkcia sa používa na činnosti súvisiace s *cleanup*-om pri odstraňovaní komponentu, ako rušenie časovačov, ukončenie sieťových operácií alebo odhlásenie sa z odberu udalostí, ku ktorým sa komponent prihlásil v metóde `componentDidMount()`.

Metód, ktoré sa používajú v procese životného cyklu komponentu je síce viac, ale tie ostatné sa používajú v špeciálnych prípadoch. Tieto uvedené sú najčastejšie používanými.

5.3.1.1 Clock as Class Component Example

Aby sme lepšie porozumeli tomu, ako životný cyklus komponentu funguje, ukážeme si ho na jednoduchom príklade komponentu reprezentujúcom hodiny. Jeho kód bude vyzerať nasledovne:

```
import React, { Component } from "react";
import { StyleSheet, View, Text } from "react-native";

export default class App extends Component {
    constructor(props) {
        super(props);

        console.log();
        console.log(">> constructor");
        this.state = {
            now: new Date(),
        };
    }

    render() {
```

[9] https://reactjs.org/docs/react-component.html#componentdidupdate
[10] https://reactjs.org/docs/react-component.html#componentwillunmount

```
        console.log(">> render()");

        return (
            <View style={styles.container}>
                <Text style={styles.clock}>
                    {this.state.now.toLocaleTimeString()}
                </Text>
            </View>
        );
    }

    componentDidMount() {
       console.log(">> componentDidMount()");

       this.timerId = setInterval(() => this.tick(), 1000 * 1);
    }

    tick() {
        console.log(">> tick");
        this.setState({
            now: new Date(),
        });
    }

    componentDidUpdate() {
        console.log(">> componentDidUpdate()");
    }

    componentWillUnmount() {
        console.log(">> componentWillUnmount()");
        clearInterval(this.timerId);
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
    },
```

```
    clock: {
        fontSize: 40,
    },
});
```

5.3.2 Component Lifecycle in Functional Components

Situácia v prípade reprezentácie komponentu pomocou funkcie je však iná. Tu totiž všetky uvedené funkcie nahradíme pomocou jedného hook-u s názvom `useEffect()`

```
import { useEffect } from "react";

useEffect(function() {
    // code to run
});
```

V závislosti od zápisu tohto hook-u dôjde k použitiu, ako sme videli v prípade komponentov reprezentovaných ako triedy. Na jednotlivé možnosti sa pozrieme bližšie.

5.3.2.1 Run Once

Tento spôsob použitia je podobný použitiu metódy `componentDidMount()`

Funkcia dostane v tomto prípade prázdny zoznam ako druhý parameter:

```
useEffect(function(){
  // code to run
}, []);
```

5.3.2.2 Run on Props Change

Tento spôsob použitia je podobný použitiu metódy `componentDidUpdate()`

Komponent dostane v tomto prípade ako parameter `props`. Tie sa stanú parametrom funkcie `useEffect()`. Samozrejme - nemusí sa jednať len o jeden z nich, ale je možné ich vymenovať viac.

Použitie hook-u je nasledovné:

```
function Component({someProp}){
    useEffect(function(){
        // code to run
```

```
    }, [someProp]);
}
```

5.3.2.3 Run on State Change

Tento spôsob použitia je podobný použitiu metódy `componentDidUpdate()`

Parametrom hook-u je v tomto prípade premenná reprezentujúca stav. Pri jej zmene je zavolaný kód funkcie. Samozrejme - nemusí sa jednať len o jeden z nich, ale je možné ich vymenovať viac.

Použitie hook-u je nasledovné:

```
function Component(){
    const [state, setState] = useState();
    useEffect(function(){
        // code to run
    }, [state]);
}
```

5.3.2.4 Run After Every Render

Tento spôsob použitia je podobný použitiu metódy `componentDidUpdate()`

V tomto prípade sa hook spustí po každom aktualizovaní, resp. vykreslení komponentu.

Použitie hook-u je nasledovné:

```
useEffect(function(){
    // code to run
});
```

5.3.2.5 Run on Unmount

Tento spôsob použitia je podobný použitiu metódy `componentWillUnmount()`

V tomto prípade hook vráti funkciu. Jej kód sa spustí vtedy, keď dôjde k zrušeniu komponentu.

Použitie hook-u je nasledovné:

```
useEffect(function(){
    return function(){
        // code to run
```

```
    };
});
```

5.3.2.6 Clock as Functional Component Example

Výsledná implementácia komponentu hodín, ktoré budú reprezentované pomocou funkcie, je nasledovná:

```
import React, { useState, useEffect } from "react";
import { StyleSheet, View, Text } from "react-native";

export default function App() {
    const [now, setNow] = useState(new Date());

    useEffect(function () {
        console.log(">> componentDidMount()");
        const intervalId = setInterval(function () {
            console.log(">> tick");
            setNow(new Date());
        }, 1000 * 1);

        return function () {
            console.log(">> componentWillUnmount()");
            clearInterval(intervalId);
        };
    }, []);

    useEffect(
        function () {
            console.log(">> componentDidUpdate()");
        },
        [date]
    );

    return (
        <View style={styles.container}>
            <Text style={styles.clock}>
                {now.toLocaleTimeString()}
            </Text>
        </View>
    );
```

```
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
    },
    clock: {
        fontSize: 40,
    },
});
```

5.4 Checking the Presence of Flashlight

V našom prípade budeme teda potrebovať zabezpečiť, aby sa prítomnosť blesku overila pri spustení aplikácie - pri zavedení (mount) komponentu. Takže potrebujeme, aby sa konkrétny kód vykonal len raz po spustení. Použijeme teda hook `useEffect()` v roli metódy `componentDidMount()`:

```
useEffect(function () {
    Torch.switchState(isOn).catch(function (e) {
        Alert.alert(
            "Missing Flashlight",
            "No camera available. Go and buy a device " +
            "with some (or two) and come back later.",
            [
                {
                    text: "Quit",
                    onPress: function () {
                        RNExitApp.exitApp();
                    },
                },
            ]
        );
    });
}, []);
```

Ostatný kód z funkcie `toggle()`, ktorý bol doteraz zodpovedný za prepínanie stavu baterky, môžeme vyhodiť a funkcia môže zostať v pôvodnom tvare:

```
const toggleState = async function () {
    await Torch.switchState(!isOn);
    setIsOn(!isOn);
};
```

Po spustení aplikácie v emulátore sa nám hneď zobrazí dialógové okno, kde po kliknutí na tlačidlo `Quit` sa aplikácia vypne. Ak však aplikáciu spustíme na reálnom zariadení, ktoré je bleskom vybavené, bude aplikácia pracovať správne.

5.5 Android Permissions

Je tu však ešte jeden problém, ktorý je potrebné vyriešiť na platforme *Android*. S príchodom verzie *7* sa totiž zmenili možnosti používania práv/povolení aplikáciami. Do tejto verzie *6* sa povolenia pre aplikáciu povoľovali len raz a to pri inštalácii. Aby ste aplikáciu mohli nainštalovať, museli ste povoliť všetko. Ináč ste si aplikáciu nainštalovať nemohli.

Od verzie *7* sa však tento prístup zmenil a aplikáciu nainštalujete bez toho, aby ste čokoľvek povoľovali. *Android* totiž umožňuje zapínať a vypínať povolenia aplikácie selektívne počas jej behu. To pre nás znamená, že síce test na prítomnosť blesku nám zbehne v poriadku, ale nemôžeme si byť istý, či je prístup k blesku pre aplikáciu povolený na úrovni operačného systému.

Preto musíme aplikáciu aktualizovať a pri každom prístupe k blesku vo funkcii `toggle()` najprv overiť, či na platforme *Android* príslušné povolenia máme alebo nie.

Test platformy je jednoduchý - pomocou triedy `Platform` sa vieme opýtať na bežiaci operačný systém a v prípade *Android*-u urobíme overenie:

```
const toggleState = async function () {
    if(Platform.OS === 'android'){
        console.info('>> checking permissions first')
    }

    await Torch.switchState(!isOn);
    setIsOn(!isOn);
};
```

5.5.1 Checking Permissions with Module `react-native-torch`

Modul react-native-torch[11], ktorý používame na svietenie baterkou, túto kontrolu už obsahuje, takže sa inšpirujeme ukážkou kódu, ktorú má na stránke a jemne ju upravíme pre naše použitie:

```
const toggleState = async function () {
    let cameraAllowed = true;

    if (Platform.OS === "android") {
        cameraAllowed = await Torch.requestCameraPermission(
            "Camera Permissions",
            "We require camera permissions to use the " +
            "torch on the back of your phone."
        );
    }

    if (cameraAllowed) {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    }
};
```

Ak teraz aplikáciu spustíme prvýkrát a klikneme buď na obrázok alebo na tlačidlo, systém *Android* si od nás vypýta explicitne povolenie pre prístup ku kamere. Ak aplikácii povolenie nedáme, pri ďalšom kliknutí sa nás bude pýtať znova. Ak naopak povolenie aplikácii udelíme, bude aplikácia pekne svietiť.

Overiť, poprípade zmeniť povolenia aplikácie môžeme aj ručne v systéme *Android*. To môžeme zabezpečiť cez `Settings > Applications > Torch > Permissions`.

5.5.2 Checking Permissions Manualy

V tomto prípade sme mali k dispozícii rovno volanie, ktoré poskytovalo API daného modulu. Čo však v prípade, že takúto možnosť nemáme?

React Native vo svojom API obsahuje objekt `PermissionsAndroid`, pomocou ktorého je možné overiť ktorékoľvek povolenie systému *Android*. Používa na to funkciu `.request()`, ktorej povinným parametrom je je požadované povolenie. Toto povolenie je dostupné v tomto objekte ako konštanta cez

[11] https://www.npmjs.com/package/react-native-torch

`PermissionsAndroid.PERMISSION`. Napríklad povolenie pre prístup ku kamere je dostupné ako konštanta `PermissionsAndroid.PERMISSIONS.CAMERA`. Komplentý zoznam povolení je možné nájsť v dokumentácii[12].

Funkcia vráti výsledok, ktorý môže byť buď

- `PermissionsAndroid.RESULTS.GRANTED` - povolené
- `PermissionsAndroid.RESULTS.DENIED` - zakázané
- `PermissionsAndroid.RESULTS.NEVER_ASK_AGAIN` - zakázané a už sa viac nepýtať

Upravíme teda implementáciu funkcie `toggleState()` nasledovne:

```
const toggleState = async function () {
    let cameraAllowed = true;

    if (Platform.OS === "android") {
        const granted = await PermissionsAndroid.request(
            PermissionsAndroid.PERMISSIONS.CAMERA
        );

        cameraAllowed =
            (granted === PermissionsAndroid.RESULTS.GRANTED);
    }

    if (cameraAllowed) {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    }
};
```

Druhým nepovinným parametrom funkcie `.request()` je tzv. `rationale`. V prípade, že bude uvedený, tak predtým, ako sa zobrazí samotný systémový dialóg s povolením/zakázaním príslušného povolenia, sa zobrazí pomocný dialóg s dodatočnými informáciami napr. s vysvetlením použitia daného povolenia:

```
const toggleState = async function () {
    let cameraAllowed = true;

    if (Platform.OS === "android") {
        const granted = await PermissionsAndroid.request(
```

[12] https://reactnative.dev/docs/permissionsandroid

```
            PermissionsAndroid.PERMISSIONS.CAMERA,
            {
                title: "Torch Needs Camera Permission",
                message:
                    "Torch app uses camera flashlight as " +
                    "torch. To make Torch work, you need " +
                    "to allow Camera Permission.",
                buttonNeutral: "Ask Me Later",
                buttonNegative: "Cancel",
                buttonPositive: "OK",
            }
        );

        cameraAllowed =
            granted === PermissionsAndroid.RESULTS.GRANTED;
    }

    if (cameraAllowed) {
        await Torch.switchState(!isOn);
        setIsOn(!isOn);
    }
};
```

5.6 Conclusion

Dnes sme sa teda pozreli na to, ako vieme ovplyvniť správanie komponentu vzhľadom na jeho životný cyklus a ako na platforme *Android* zabezpečiť potrebné prístupové práva v prípade špeciálnej funkcionality.

Nabudúce sa pozrieme na to, ako aplikácie prekladať do iných jazykov a lepšie zorganizujeme projekt.

Prednáška 6

i18n and l10n

internacionalizácia (i18n) a lokalizácia (l10n) projektu (aplikácie)

6.1 Introduction

stále pracujeme na baterke

- baterka už svieti
- Dokonca vieme aj v prípade, že zariadenie baterkou nedisponuje, zobraziť správu a aplikáciu vypnúť. Dokonca to robíme už tesne po spustení aplikácie.
- Pred prístupom k baterke vieme dokonca overiť aj to, či máme povolený prístup ku kamere na *OS Android*.
- Dnes sa pozrieme na to, ako správne ovládnuť globálny trh s baterkami a na štruktúru nášho projektu.

6.2 I18n Stands for Internationalization

Dajme tomu, že máme s našou aplikáciou vyššie ciele, ako len urobiť baterku z telefónu - chceme ňou ovládnuť svetový trh bateriek ;) Aby sa nám to podarilo, rozhodne nesmieme podceniť otázku **lokalizácie** aplikácie pre použitie v iných krajinách. V našom prípade sa jedná o pomerne jednoduchý proces, pretože skončíme len pri lokalizovaní/preklade textov do jazyka cieľovej krajiny. Na čo všetko však nesmieme v prípade lokalizácie zabudnúť?

Pri prvom priblížení si môžeme myslieť, že otázka lokalizácie a internacionalizácie sa týka len prekladov. Nie je to však pravda. Lokalizácia a internacionalizácia so sebou totiž nesie ďalšie dôsledky, ako:

Obrázok 6.1: I18n (zdroj[1])

- iné jazyky môžu vyžadovať iné **množstvo miesta na obrazovke** (krátky reťazec v pôvodnom jazyku a dlhý reťazec v preklade)
- iný jazyk môže používať iný **smer písma** (RTL jazyky)
- v inej krajine sa môže používať **iná abeceda** a teda iné znaky
- formátovanie čísiel (oddeľovanie desatinných miest, tisícok), času a dátumu, meny
- používanie titulov pri menách
- formátovanie kontaktných informácií (telefónne čísla, adresy)

Na vyriešenie týchto problém samozrejme už existujú hotové riešenia. Napr. v linuxových/unixových riešeniach sa dlhodobo používa nástroj gettext[2], ktorý oddeľuje program od prekladu a elegantne rieši mnohé problémy spojené s lokalizáciou. Jeho podpora je v rozličných programovacích jazykoch, čo z neho robí univerzálny nástroj.

Iný prístup k tejto problematike má aj samotný systém Android. Je však jednotný pre všetky aplikácie napísané pre tento systém.

6.2.1 i18next Framewok

Odlišný prístup je však vo svete JavaScript-u. K dispozícii je veľké množstvo knižníc, pomocou ktorých je možné tento problém riešiť. Každý má samoz-

[2] https://en.wikipedia.org/wiki/Gettext

Obrázok 6.2: Different Aspects of i18n and l10n [20]

rejme svoje vlastné špecifiká. My sa pozrieme konkrétne na rámec i18next[3]. Jeho výhodou je, že sa dá použiť aj inde, ako len v prípade tvorby mobilných aplikácií pomocou rámca *React Native*. Jeho heslom je **learn once - translate everywhere** ;)

Obrázok 6.3: i18next [18]

6.2.2 Installation

Nainštalujeme niekoľko balíkov:

```
$ npm install react-i18next i18next --save
$ npm install react-native-localize --save
```

> **Poznámka**
> Pravdepodobne bude potrebné opäť projekt aj vyčistiť pomocou:
>
> ```
> $ cd android
> $ ./gradlew clean
> $ cd ..
> $ npx react-native run-android
> ```

[3] https://www.i18next.com

6.2.3 Configure i18next

Začneme podľa pokynov v Quick Start Guide[4]: vytvoríme súbor `i18n.js` a upravíme ho do nasledovnej podoby:

```
import i18n from "i18next";
import { initReactI18next } from "react-i18next";

// the translations
const resources = {
  sk: {
    translation: {
      "Turn On": "Zasvietiť",
      "Turn Off": "Zhasnúť",
    }
  }
};

i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    resources,
    lng: "en",

    // we do not use keys in form messages.welcome
    keySeparator: false,

    interpolation: {
      // react already safes from xss
      escapeValue: false
    }
  });

  export default i18n;
```

6.2.4 Translate your content

Rámec *i18next* poskytuje niekoľko spôsobov, ako je možné lokalizovať text. Pre naše riešenie použijeme hook `useTranslation()`, pomocou ktorého získame funkciu `t()` na prekladanie reťazcov:

[4] https://react.i18next.com/guides/quick-start

```
import { useTranslation } from 'react-i18next';

export default function App() {
    const { t, i18n } = useTranslation();
    // code ...
}
```

Preklad následne vykonáme použitím funkcie `t()` napr. na popisku tlačidla na rozsvietenie a zhasnutie baterky:

```
<Button
    onPress={toggleState}
    title={isOn === true ? t('Turn Off') : t('Turn On')}
/>
```

Tu je možné vidieť, ako lokalizácia pomocou rámca *i18next* funguje. Preklad je reprezentovaný pomocou JSON objektu, kde sa ku každému kľúču viaže nejaký preklad. Týmto spôsobom pracuje množstvo ďalších lokalizačných rámcov pre jazyk JavaScript.

Rámec i18next však ponúka viac, ako napr. používanie menných priestorov na zoskupovanie prekladov pre konkrétne moduly aplikácie. Predvoleným menným priestorom je **`translation`** a min. jeden musí existovať. My však ako kľúče budeme používať celé texty, ktoré je potrebné preložiť. To nám dá tú výhodu, že v prípade neexistujúceho prekladu sa na mieste použije priamo tento text. Tým pádom nemusíme vytvárať osobitne preklad predvoleného jazyka, ktorým bude angličtina.

Vytvoríme teda preklad všetkých textov do slovenčiny v podobe JSON objektu:

```
{
  "translation": {
    "Turn On": "Zasvietiť",
    "Turn Off": "Zhasnúť",
    "Missing Flashlight": "Chýba blesk",
    "No camera available. Go and buy a device with some " +
      "(or two) and come back later.": "Zariadeniu chýba " +
      "blesk. Kúp si najprv zariadenie s bleskom (alebo " +
      "dve) a potom to skús znova.",
    "Camera Permissions": "Povoliť kameru",
    "We require camera permissions to use the torch on the " +
      "back of your phone.": "Aby baterka svietila, je " +
```

```
      "potrebné povoliť kameru.",
    "Quit": "Ukončiť"
  }
}
```

Každý reťazec v aplikácii zabalíme do funkcie `t()`.

Ak teraz spustíme aplikáciu, nič sa nezmení. Ak však v konfigurácii modulu `i18next` zmeníme jazyk na `sk`, uvidíme jednotlivé texty preložené do slovenčiny. Ako však proces prepínania zautomatizovať tak, aby sa jazyk aplikácie zvolil na základe jazyka systému zariadenia?

6.2.5 Get the Current Locale

Na zistenie aktuálneho jazyka systému použijeme modul s názvom `react-native-localize`. V ňom pomocou funkcie `getLocales()` získame zoznam preferovaných jazykov, kde ten prvý bude aktuálne používaný. Kód bude vyzerať takto:

```
import * as RNLocalize from 'react-native-localize';

const deviceLanguage = RNLocalize.getLocales()[0].languageCode;
```

Poznámka
Pre získanie aktuálneho jazyka môžeme použiť aj `NativeModules` napr. takto (zdroj[a]):

```
import { NativeModules, Platform } from 'react-native';

const deviceLanguage =
   Platform.OS === 'ios'
      ? NativeModules.SettingsManager.settings.AppleLocale ||
      // iOS 13
      NativeModules.SettingsManager.settings.AppleLanguages[0]
      : NativeModules.I18nManager.localeIdentifier;

console.log(deviceLanguage); //en_US
```

[a] https://stackoverflow.com/a/56425150/1671256

Vďaka tomu môžeme upraviť inicializáciu objektu `i18n`, kde miesto jazyka

napevno môžeme práve použiť premennú `deviceLanguage` alebo priamo získanie kódu jazyka z výsledku volania `getLocales()`:

```
i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    resources: locales,
    fallbackLng: 'en',
    lng: RNLocalize.getLocales()[0].languageCode,

    // we do not use keys in form messages.welcome
    keySeparator: false,

    interpolation: {
      escapeValue: false, // react already safes from xss
    },
  });
```

Ak teraz aplikáciu spustíme, priamo po štarte bude použitý jazyk systému. Ak teda budeme experimentovať a pred spustením aplikácie jazyk zmeníme, uvidíme buď slovenčinu, ak vyberieme slovenský jazyk, alebo angličtinu, ak vyberieme ktorýkoľvek iný jazyk.

6.2.6 Language Change on the Fly

Ku zmene jazyka však môže dôjsť aj počas behu aplikácie. Ak sa pokúsime zmeniť jazyk systému pri spustenej aplikácii, k žiadnej zmene jazyka nedôjde. Potrebujeme ju totiž znova vypnúť a zapnúť, aby bol objekt `i18n` inicializovaný nanovo na základe aktuálneho nastavenia systému. Ako však zabezpečiť to, aby sa jazyk prepol automaticky, keď ho zmeníme v systéme?

Tu nám pomôže nastaviť listener z modulu `react-native-localize` na udalosť `change`[5], ktorá nastane práve vtedy, keď k zmene jazyka dôjde. Následne jazyk v aplikácii zmeníme volaním `i18n.changeLanguage()`:

```
RNLocalize.addEventListener('change', () => {
  const language = RNLocalize.getLocales()[0].languageCode;
  console.log(`>> language has been changed to ${language}`);
  i18n.changeLanguage(language);
});
```

[5] https://www.npmjs.com/package/react-native-localize#addeventlistener--removeeventlistener

Ak teraz aplikáciu vyskúšame, zmena jazyka sa prejaví okamžite po zmene jazyka v systéme.

6.2.7 Effective Resource Management

Pozrime sa však späť na to, ako vyzerá organizácia prekladu - nachádza sa vo vnútri súboru `i18n.js`. Aktuálne sme využili vlastnosť `i18next`, že ak neexistuje preklad, použije sa predvolený reťazec, ktorý je parametrom funkcie `t()`.

Preklad do slovenčiny má aktuálne 7 reťazcov. Aplikácie však majú bežne stovky až tisícky reťazcov, ktoré je potrebné preložiť. To znamená, že údržba všetkých jazykov v jednom súbore, ktorý súčasne obsahuje aj kód, je veľmi neefektívna a nepraktická. Rozumnejšie by bolo udržiavať každý jazyk v osobitnom súbore.

Vytvoríme preto osobitný priečinok s názvom `locales/`, ktorý bude pre každý jeden jazyk obsahovať samostatný súbor vo formáte JSON s prekladom všetkých reťazcov. V našom prípade to znamená, že v ňom vytvoríme len súbor `sk.json`, ktorého obsah bude prekladom všetkých reťazcov do slovenčiny:

```
{
  "translation": {
    "Turn On": "Zasvietiť",
    "Turn Off": "Zhasnúť",
    "Missing Flashlight": "Chýba blesk",
    "No camera available. Go and buy a device with some " +
      "(or two) and come back later.": "Zariadeniu chýba " +
      "blesk. Kúp si najprv zariadenie s bleskom (alebo dve) " +
      "a potom to skús znova.",
    "Camera Permissions": "Povoliť kameru",
    "We require camera permissions to use the torch on the " +
      "back of your phone.": "Aby baterka svietila, je " +
      "potrebné povoliť kameru.",
    "Quit": "Ukončiť"
  }
}
```

Aby sa nám manažment prekladov robil jednoduchšie, tak vytvoríme z tohto priečinku modul. Vytvoríme teda súbor `index.js`, v ktorom načítame všetky preklady a exportneme ich von ako jeden objekt `locales`:

```
export const locales = {
  sk: require('./sk.json'),
```

```
};
```

Následne už len upravíme pôvodný súbor `i18n.js`, v ktorom odstránime preklad do slovenčiny a zameníme ho za preklady získané z modulu `locales`:

```
import i18n from 'i18next';
import {initReactI18next} from 'react-i18next';
import * as RNLocalize from 'react-native-localize';
import {locales} from './locales';

RNLocalize.addEventListener('change', () => {
  const language = RNLocalize.getLocales()[0].languageCode;
  i18n.changeLanguage(language);
  console.log(`>> language has been changed to ${language}`);
});

i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    resources: locales,
    fallbackLng: 'en',
    lng: RNLocalize.getLocales()[0].languageCode,
    // we do not use keys in form messages.welcome
    keySeparator: false,
    interpolation: {
      escapeValue: false, // react already safes from xss
    },
  });

export default i18n;
```

Tým pádom pridať nový jazyk znamená vytvoriť nový súbor v module `locales` a do exportovaného objektu `locales` tohto modulu pridať riadok navyše, ktorý zabezpečí nahratie tohto jazyka.

6.3 Conclusion

Dnes sme sa pozreli na to, ako vieme zabezpečiť preklad UI do iných jazykov, kedy sa jazyk vyberie na základe jazyka systému. Tým sme aplikáciu pripravili pre nasadenie pre globálny trh ;)

Projekt nám začína rásť. Nabudúce sa pozrieme na to, ako projekt udržiavať

vo vhodnej štruktúre, aby sme sa v ňom nestratili, a aby pridávanie nových vlastností, resp. nových komponentov nebolo bolestivé.

Prednáška 7

Component Composition

štruktúra projektu, kompozícia komponentov, odovzdávanie údajov medzi komponentami, properties, `Context`

7.1 Tips

`.gitignore` pre *React Native* projekty

- aktuálne posielate do git-u množstvo vecí, ktoré tam nemajú čo robiť
- priamo od facebook-u na Github[1]-e
- Pri vytváraní projektu ale dôjde k vytvoreniu aj priečinku `.git` a aj súboru `.gitignore`. Ak zmažete priečinok `.git`, spustite ešte tento príkaz:

```
$ git rm --cached priecinok.s.projektorm -f
```

7.2 Introduction

je to neuveriteľné, ale stále pracujeme na baterke

- baterka svieti
- Dokonca vieme aj v prípade, že zariadenie baterkou nedisponuje, zobraziť správu a aplikáciu vypnúť. Dokonca to robíme už tesne po spustení aplikácie.

[1] https://github.com/facebook/react-native/blob/master/.gitignore

- Pred prístupom k baterke vieme dokonca overiť aj to, či máme povolený prístup ku kamere na *OS Android*.
- Máme ju dokonca pripravenú pre globálny trh, pretože sme vyriešili aj problematiku lokalizácie aplikácie do cudzieho jazyka.

Dnes sa pozrieme na to, ako sa nestratiť v projekte jeho vhodným štruktúrovaním a urobíme z baterky viackomponentovú aplikáciu.

7.3 Project Structure

React a *React Native* sú rámce, ktoré nám nijakým spôsobom nepredpisujú štruktúru projektu alebo aplikácie. Aktuálne je náš projekt jednoduchý, ale s novými funkciami, obrazovkami, komponentmi, prekladmi sa počet súborov, z ktorých sa skladá, rapídne zväčší. Ako teda organizovať súbory tak, aby sme sa dokázali v projekte jednoducho orientovať aj napriek veľkému počtu súborov?

Existuje niekoľko spôsobov, ako organizovať štruktúru projektu. Spoločné majú hlavne logické zoskupovanie súborov podľa ich typu alebo funkcionality (tzv. type vs feature). Ukážka jedného z týchto prístupov sa nachádza napr. v projekte React Native Easy Starter[2].

Nebudeme rozoberať a upravovať štruktúru nášho projektu pre veci, ktoré nemáme, takže urobíme niečo, čo majú všetky štruktúry spoločné: súbory, ktoré tvoria náš projekt a nie sú automaticky vygenerované nástrojmi rámca, umiestnime do osobitného priečinku s názvom `src/`.

> **Poznámka**
> V Expo projekte sa dá stretnúť s názvom tohto priečinku `app/`.

Vytvoríme teda priečinok `src/` a presunieme do neho všetko, čo sme vytvorili včetne priečinku `assets/`:

```
src/
  App.js
  assets/
     bulb_off.png
     bulb_on.png
     torch-icon.png
```

[2] https://github.com/HarishJangra/react-native-easy-starter

```
i18n.js
locales/
    index.js
    sk.json
```

Prostredie *Visual Studio Code* urobí potrebné úpravy v súboroch projektu za nás, takže projekt by mal fungovať bez prípadných zásahov. Môžeme však z priečinku `src/` vytvoriť rovno modul pridaním súboru `index.js`, v ktorom exportujeme komponent `App` zo súboru `App.js`:

```
import App from './App';

export default App;
```

Pre istotu môžeme overiť ešte súbor `index.js` v koreňovom priečinku projektu a v prípade potreby upraviť import:

```
import App from './src';
```

A okrem toho aj odkazy na ikony v súbore `app.json`.

> **Upozornenie**
> Pravdepodobne bude potrebné zmazať aj výsledné zostavenie *Android* projektu v priečinku `android/`:
>
> ```
> $ cd android
> $ ./gradlew clean
> $ cd ..
> ```

7.4 Multi Component Application

Vráťme sa teraz naspäť k definícii komponentu. V predchádzajúcich prednáškach sme komponenty v rámci *React* definovali ako

> malé izolované časti na tvorbu používateľských rozhraní

Používateľské rozhranie našej aplikácie sa aktuálne skladá z dvoch takýchto častí

- z obrázka
- z tlačidla

Miesto toho, aby sme sa na ne pozerali ako na dva samostatné komponenty, sú súčasťou jedného komponentu, v ktorom sa tento obrázok spolu s tlačidlom nachádzajú. Upravme teda projekt tak, že ho rozbijeme na tri komponenty:

- obrázok ako komponent `ImageSwitch`,
- tlačidlo ako komponent `ButtonSwitch`, a
- samotnú aplikáciu ako komponent `App`.

V štruktúre projektu v priečinku `src/` preto vytvoríme ďalší priečinok s názvom `components/`, do ktorého tieto komponenty umiestnime - každý do samostatného súboru.

> **Upozornenie**
> Z tohto priečinku nie je dobré vytvárať modul, v ktorom by boli všetky komponenty, a teda nie je dobré v ňom vytvárať súbor `index.js`, v ktorom budeme exportovať všetky komponenty. Veľmi ľahko totiž môže dôjsť k problému *Require Cycle Warnings*. Vtedy napr. modul `A` importuje module `B` a ten zasa (napr. prostredníctvom ďalších modulov) zasa vyžaduje modul `A`. Komponenty sú samozrejme súčasťou iných komponentov, takže porušiť tento princíp je veľmi jednoduché. Riziko rastie tým, ak sa na importovanie v každom module/komponent používa práve `index.js` súbor.

Zatiaľ vytvoríme len kostru týchto komponentov a postupne opätovne vytvoríme celú aplikáciu. Komponent `ImageSwitch` bude uložený v súbore `ImageSwitch.js` a bude vyzerať nasledovne:

```
import React from "react";
import { Image, Pressable } from "react-native";

export function ImageSwitch() {
    const image = require("../assets/bulb_off.png");

    return (
        <Pressable onPress={null}>
            <Image source={image} />
        </Pressable>
    );
}
```

Komponent `ButtonSwitch` bude uložený v súbore `ButtonSwitch.js` a bude

vyzerať nasledovne:

```
import { useTranslation } from "react-i18next";
import React from "react";
import { Button } from "react-native";

export function ButtonSwitch() {
    const { t } = useTranslation();

    return (
        <Button
            onPress={null}
            title={t("Turn On")}
        />
    );
}
```

Komponent `App` bude uložený v súbore `App.js` a bude vyzerať nasledovne:

```
import React, { useState } from "react";
import { StyleSheet, View } from "react-native";

import { ButtonSwitch } from "./ButtonSwitch";
import { ImageSwitch } from "./ImageSwitch";

import "../i18n";

export function App() {
    const [isOn, setIsOn] = useState(false);

    return (
        <View style={styles.container}>
            <ImageSwitch></ImageSwitch>
            <ButtonSwitch></ButtonSwitch>
        </View>
    );
}
```

Po spustení by sa aplikácia mala spustiť, ale tentokrát po ťapnutí na jeden alebo druhý komponent nedôjde k žiadnej zmene. To preto, že stav zostal izolovaný len v komponente `App`.

7.5 Share Data Between the Components

Ešte predtým, ako sa však vrhneme do samotnej implementácie, sa pozrieme na problém, ktorému sa vo viac komponentových aplikáciách rozhodne nevyhneme. Tým problémom je **zdieľanie údajov medzi komponentmi**.

A nevyhneme sa mu ani v našom prípade aj napriek tomu, že vytvoríme len tri komponenty. Identifikovali sme stav našej aplikácie, ktorý hovorí o tom, či je baterka zasvietená alebo zhasnutá. A tento stav môžeme zmeniť kliknutím buď na obrázok so žiarovkou alebo na tlačidlo pod ním. A zmenou stavu dôjde aj k zmene vzhľadu či už tlačidla a textu na ňom alebo obrázku reprezentujúcom stav baterky.

Komponent je malá izolovaná časť na tvorbu používateľských rozhraní. My sme vytvorili tri takéto izolované časti. Ako zabezpečíme to, že po kliknutí na tlačidlo sa o tom dozvie aj obrázok, keď aj jeden aj druhý komponent sú navzájom izolované? Čo znamená, že aj jeden aj druhý disponuje vlastným stavom komponentu?

(slides[3]) Existuje viacero mechanizmov, ktoré je možné použiť na vyriešenie tohto problému. Každý je však špecifický a záleží od toho, medzi ktorými komponentmi v hierarchickej štruktúre komponentov chceme údaje zdieľať. Preto je potrebné rozlišovať, ktorým smerom chceme údaje zdieľať:

- **od komponentu rodiča ku potomkovi** (z angl. *parent to child component*),
- **od komponentu potomka ku rodičovi** (z angl. *child to parent component*),
- **medzi súrodencami** (z angl. *between siblings*), alebo
- **medzi ľubovoľnými komponentmi** (z angl. *sharing data between not related components*)

Na jednotlivé mechanizmy pre jednotlivé typy zdieľania údajov sa pozrieme bližšie.

> **Poznámka**
> Jednotlivé mechanizmy sa môžu líšiť aj v závislosti od toho, či sú komponenty reprezentované triedami alebo funkciami. V našom prípade sa pozrieme na možnosti zdieľania údajov medzi komponentmi, ktoré sú reprezentované funkciami. V prípade záujmu

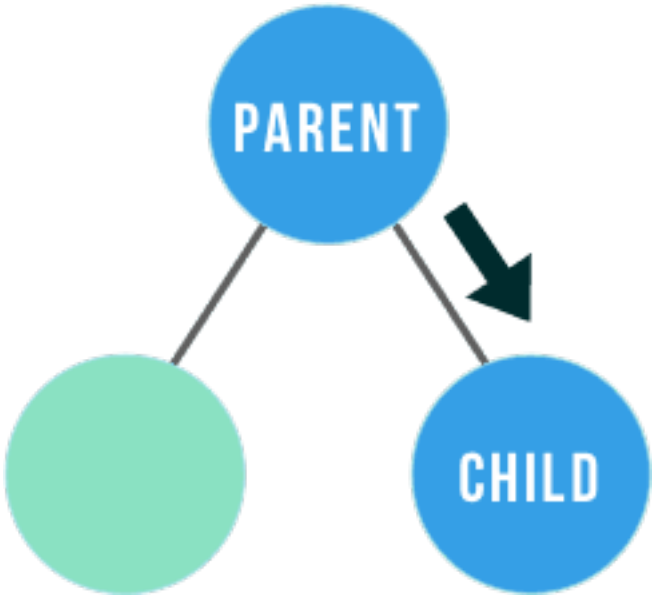
[3] slides.07.html#relationships

o existujúce mechanizmy, ktoré je možné použiť pre komponenty reprezentované ako triedy, sa pozrite napr. na článok 8 no-Flux strategies for React component communication[a].

[a] https://www.javascriptstuff.com/component-communication/#6-observer-pattern

7.5.1 Parent to Child Component

Toto je veľmi častý prípad zdieľania údajov medzi komponentmi. Každý (rodičovský) komponent sa totiž môže odkazovať na iný komponent (potomka) pri svojom renderovaní a v tomto momente im môže predať potrebné informácie.



Obrázok 7.1: Parent to Child [4]

7.5.1.1 Properties

Predať údaje od rodiča svojmu potomkovi je možné zabezpečiť pomocou **vlastností** (z angl. *properties* alebo skrátene *props*). Toto predanie sa deje vo forme argumentu funkcie `props`, ktorá reprezentuje komponent:

```
function Hello(props){
    return (
        <Text>Hello {props.name} {props.surname}!</Text>
    );
}
```

Rodičovský komponent odovzdá údaje do komponentu `Hello` pri jeho renderovaní napríklad takto:

```
return (
    <View style={styles.container}>
        <Hello name="Sherlock" surname="Holmes"></Hello>
    </View>
);
```

Vlastnosti `props` sú objektom, čo si môžeme jednoducho overiť tým, že ich vypíšeme do log-u:

```
console.log(props);
```

Vlastnosti sa obecne používajú na prispôsobenie, resp. úpravu komponentu v dobe jeho vytvárania pomocou rozličných parametrov (z angl. *properties*). Väčšina komponentov rámca *React Native* obsahuje vlastnosti, pomocou ktorých je možné nastaviť ich vlastnosti v čase vytvárania. Napr. komponent `Image` obsahuje vlastnosť `src`, pomocou ktorej máme možnosť špecifikovať obrázok, ktorý sa má zobraziť.

Vlastnosti sú určené **len na čítanie**! Komponent nesmie modifikovať hodnoty vlastností! **Vlastnosti sú nemenné** (z angl. *immutable*)!

7.5.1.2 Pure Function

Takéto funkcie, ktoré nemenia svoje vstupy a vždy vrátia rovnaký výstup pri rovnakom vstupe, sa nazývajú **"pure[4]" funkcie**. Dokonca samotný React, aj napriek tomu, že je pomerne flexibilný, má jedno veľmi striktné pravidlo:

All React components must act like pure functions with respect to their props.

7.5.1.3 Torch Update

V našom prípade sa jedná o vzťah rodiča, ktorým je komponent `App` a jeho potomkov, ktorými sú `ImageSwitch` a `ButtonSwitch`. Údaj, ktorý je potrebné,

[4] https://en.wikipedia.org/wiki/Pure_function

aby každý potomok dostal, je počiatočný stav komponentu:

- aby komponent `ImageSwitch` vedel, aký obrázok má zobraziť, a
- aby komponent `ButtonSwitch` vedel, aký popisok má na tlačidle zobraziť.

Upravíme teda renderovanie komponentu `App`, kde každému potomkovi pošleme stav komponentu pomocou vlastnosti `isOn`:

```
return (
    <View style={styles.container}>
        <ImageSwitch isOn={isOn}></ImageSwitch>
        <ButtonSwitch isOn={isOn}></ButtonSwitch>
    </View>
);
```

Na základe tejto vlastnosti upravíme komponent `ButtonSwitch`, kde na základe jej hodnoty aktualizujeme popisok tlačidla:

```
export function ButtonSwitch(props) {
    const { t } = useTranslation();

    return (
        <Button
            onPress={null}
            title={props.isOn === true ?
                t("Turn Off") : t("Turn On")}
        />
    );
}
```

Podobne na základe hodnoty tejto vlastnosti zvolíme správny obrázok v komponente `ImageSwitch`:

```
export function ImageSwitch(props) {
    const { t } = useTranslation();

    const image = props.isOn
        ? require("../assets/bulb_on.png")
        : require("../assets/bulb_off.png");

    return (
        <Pressable onPress={null}>
            <Image source={image} />
```

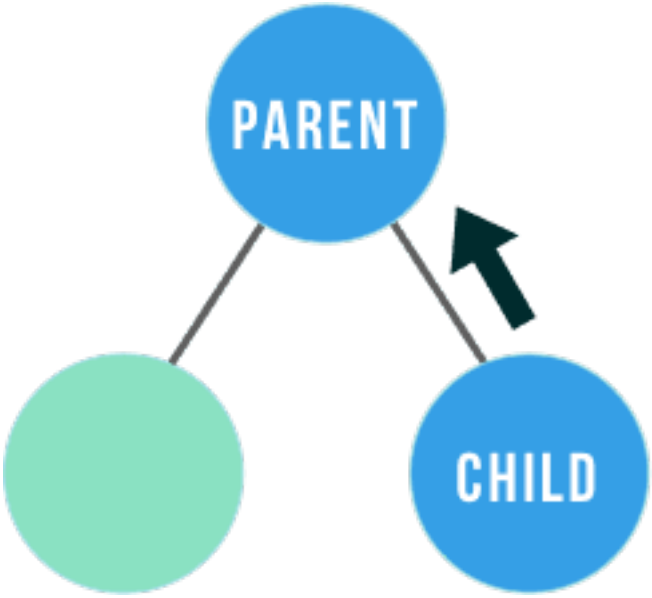
```
        </Pressable>
    );
}
```

Po spustení aplikácie síce nevidíme žiadnu mimoriadnu zmenu, ale môžeme si byť istý, že aktuálne sa jej vzhľad odvíja od vlastnosti `isOn`. Zmenou jej počiatočnej hodnoty sa o tom môžeme presvedčiť.

Takže problém odovzdávania údajov od rodiča smerom k potomkovi máme vyriešený. Ako však teraz zabezpečiť to, aby potomok informoval o zmene svojho rodiča? A to v našom prípade hlavne vtedy, keď dôjde ku stlačeniu tlačidla alebo ťapnutiu na obrázok?

7.5.2 Child to Parent Component

Pozrieme sa teda na druhý veľmi častý prípad odovzdávania údajov medzi komponentmi, keď potomok potrebuje informovať svojho rodiča.



Obrázok 7.2: Child to Parent [4]

Najjednoduchší spôsob, ako môže potomok kontaktovať rodiča, je pomocou tzv. **callback funkcie**, ktorú rodič odovzdá potomkovi ako vlastnosť (z angl. *property*). Táto funkcia sa pritom bude nachádzať u rodiča.

7.5.2.1 Torch Update

V našom prípade teda v komponente `App` vytvoríme funkciu `onStateChange()`, ktorá po zavolaní zmení jeho stav na opačný:

```
function onStateChange() {
    setIsOn(!isOn);
}
```

Referenciu na túto funkciu následne odovzdáme vo forme vlastnosti `onPress` ostatným komponentom pri renderovaní komponentu `App`:

```
return (
    <View style={styles.container}>
        <ImageSwitch
          isOn={isOn}
          onPress={onStateChange}>
      </ImageSwitch>
        <ButtonSwitch
          isOn={isOn}
          onPress={onStateChange}>
      </ButtonSwitch>
    </View>
);
```

Následne upravíme renderovanie tlačidla, kde hodnotu vlastnosti `props.onPress` priradíme ku vlastnosti `onPress` komponentu tlačidla `Button`:

```
return (
    <Button
        onPress={props.onPress}
        title={props.isOn === true ?
          t("Turn Off") : t("Turn On")}
    />
);
```

Podobne upravíme aj renderovanie obrázku, kde podobne nastavíme vlastnosť `onPress` komponentu `Pressable`:

```
return (
    <Pressable onPress={props.onPress}>
        <Image source={image} />
    </Pressable>
);
```

Ak spustíme aplikáciu teraz, "automagicky" bude všetko fungovať tak, ako má. To je dané vlastnosťami rámca *React*, kedy pri zmene stavu komponentu sa jeho nová hodnota automaticky propaguje všetkým ďalším potomkom, ktorí ho nejakým spôsobom zdieľajú. Napr. vo forme vlastnosti.

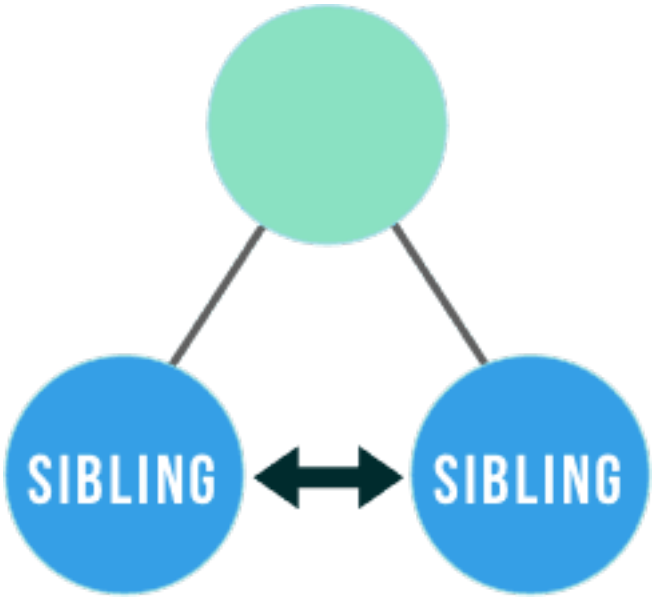
Tým sme vlastne náš problém vyriešili - z jedno komponentovej aplikácie sme vytvorili viac komponentovú aplikáciu, v ktorej sme zabezpečili zdieľanie údajov medzi jednotlivými komponentmi. Aktuálne stačí len do komponentu `App` vložiť kontrolu prístupových práv ku fotoaparátu a kontrolu existencie fotoaparátu ako takého.

```
export function App() {
    const [isOn, setIsOn] = useState(false);
    const { t } = useTranslation();

    useEffect(function () {
        Torch.switchState(isOn).catch(function (e) {
            Alert.alert(
                t("Missing Flashlight"),
                t(
                    "No camera available. Go and buy a " +
                    "device with some (or two) and come " +
                    "back later."
                ),
                [
                    {
                        text: t("Quit"),
                        onPress: function () {
                            RNExitApp.exitApp();
                        },
                    },
                ]
            );
        });
    }, []);

    async function onStateChange() {
        let cameraAllowed = true;

        if (Platform.OS === "android") {
            const granted = await PermissionsAndroid.request(
                PermissionsAndroid.PERMISSIONS.CAMERA,
```

```
                {
                    title: t("Torch Needs Camera Permission"),
                    message: t(
                        "Torch app uses camera flashlight " +
                        "as torch. To make Torch work, you " +
                        "need to allow Camera Permission."
                    ),
                    buttonNeutral: t("Ask Me Later"),
                    buttonNegative: t("Cancel"),
                    buttonPositive: t("OK"),
                }
            );

            cameraAllowed =
                granted === PermissionsAndroid.RESULTS.GRANTED;
        }

        if (cameraAllowed) {
            await Torch.switchState(!isOn);
            setIsOn(!isOn);
        }
    }

    return (
        <View style={styles.container}>
            <ImageSwitch
                isOn={isOn}
                onPress={onStateChange}>
            </ImageSwitch>
            <ButtonSwitch
                isOn={isOn}
                onPress={onStateChange}>
            </ButtonSwitch>
        </View>
    );
}
```

7.5.3 Sharing Data Between Siblings

Ďalší prípad zdieľania údajov je medzi súrodencami. Komponenty sú súrodenci, ak majú spoločného predka.



Obrázok 7.3: Sibling to Sibling [4]

Toto je vlastne aj náš prípad, kedy komponenty `ButtonSwitch` a `ImageSwitch` majú spoločného predka `App`.

Odporúčanie je v tomto prípade na komunikáciu medzi súrodencami použiť kombináciu predchádzajúcich stratégií, ktoré zahŕňajú spoločného predka. A ako sme mohli vidieť, pomáha v tomto prípade aj samotný rámec, ktorý zmeny stavu automaticky propaguje dotknutým komponentom.

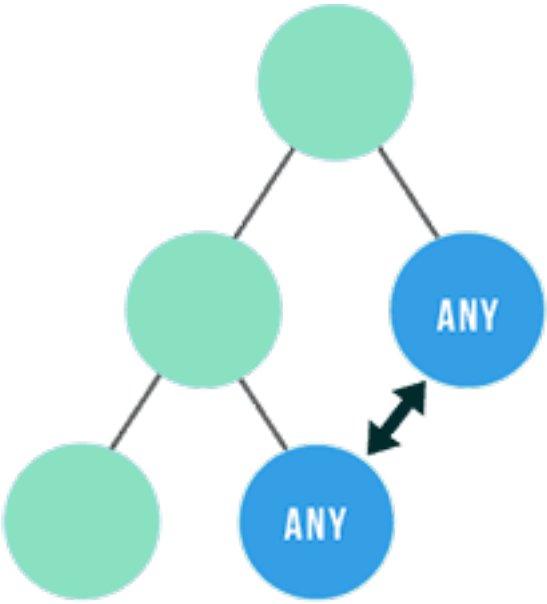
7.5.4 Sharing data between not related components

Posledný prípad zdieľania údajov je medzi ľubovoľnými dvoma komponentmi aplikácie.

Tento prípad už nie je taký triviálny, ako tomu bolo doteraz, pretože doteraz sme videli, že React údaje distribuuje prostredníctvom komponentov. Rýchlosť doručenia teda silne závisí od vzdialenosti medzi komunikujúcimi komponentmi.

(slides[5]) React totiž nemá vlastné riešenie, ktoré zabezpečuje takúto nepriamu

[5] slides.07.html#the-problem



Obrázok 7.4: Any to Any [4]

komunikáciu. React podporuje len **jednosmernú komunikáciu** (z angl. *unidirectional data flow*) v smere od rodiča ku potomkovi. Takže doručiť údaje medzi dvoma uzlami, ktoré nie sú rodičmi, je problém.

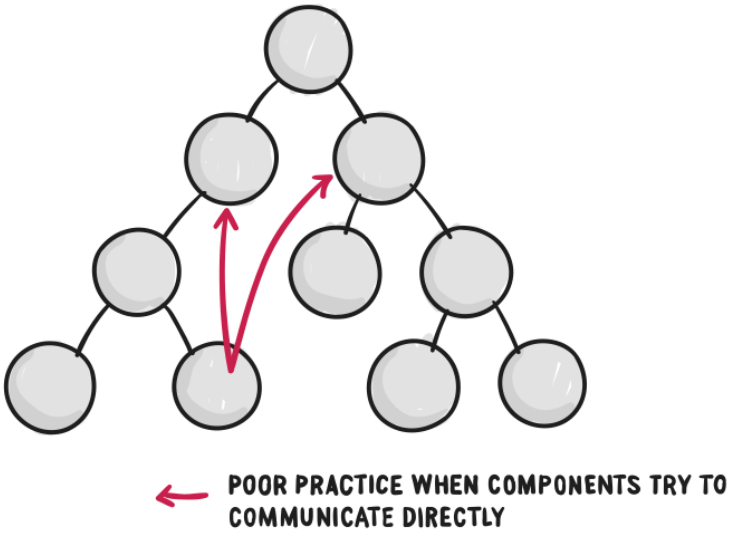
Aj napriek tomu existuje niekoľko prístupov, ktoré je možné použiť na tento typ komunikácie medzi komponentmi:

- **globálne premenné**,
- použiť `context`, alebo použiť
- návrhový vzor **pozorovateľ** (z angl. *observer*)

7.5.4.1 Global Variables

Globálne premenné nie je dobré používať, ale pre rýchle prototypovanie sa občas môžu hodiť.

V tomto prípade môžete na globálne premenné použiť objekt `window`.



Obrázok 7.5: The Problem [22]

7.5.4.2 Context

`Context` funguje podobne ako vlastnosti (props), ale s tým rozdielom, že neumožňuje poslať údaje len jednému potomkovi, ale naraz celej vetve stromu (resp. podstromu). Údaje je možné poslať len smerom od rodiča k potomkom v danej vetve.

`Context` sa používa v prípadoch, keď je potrebné zdieľať údaje spoločné pre komponenty danej vetvy stromu. To môžu byť údaje ako napr. prihlásený používateľ, téma prostredia, zvolený jazyk a pod.

V samotnej dokumentácii autori upozorňujú na opatrné používanie `Context`-u, pretože jeho nadmerným resp. nesprávnym používaním sa znižuje znovupoužiteľnosť komponentov.

7.5.4.3 Observer Pattern

Pomocou tohto návrhového vzoru je možné kontaktovať jedným komponentom ľubovoľný počet iných komponentov. Dôležité je, aby sa tieto komponenty prihlásili, že chcú byť v prípade potreby (napr. vzniku udalosti) kontaktované, resp. notifikované. Tieto komponenty sa na odber obyčajne prihlásia v čase *pripojenia* (z angl. *mounting*) komponentu a odhlásia sa pri *odpojení* (z angl.

unmounting), resp. odstránení komponentu. To je možné zabezpečiť pomocou hook-u `useEffect()`.

Implementáciu je možné vytvoriť vlastnú, ale samozrejme existujú aj hotové knižnice, ako napr.:

- EventEmitter[6] - EventEmitter is an implementation of the Event-based architecture in JavaScript.
- Flux[7] - Application architecture for building user interfaces
- MobX[8] - MobX is a battle tested library that makes state management simple and scalable by transparently applying functional reactive programming (TFRP)
- Redux[9] - A Predictable State Container for JS Apps

7.6 Conclusion

Dnes sme úspešne rozbili náš komponent a vytvorili sme viac komponentovú aplikáciu. Ukázali sme si problém, ktorý takýto prístup tvorby komponentov so sebou prináša. Tým problémom je zdieľanie údajov medzi komponentmi. Ukázali sme si štyri prípady zdieľania údajov medzi komponentmi a mechanizmy, ktorými je možné toto zdieľanie zabezpečiť.

Nabudúce sa pozrieme na Redux.

[6] https://www.npmjs.com/package/EventEmitter
[7] http://facebook.github.io/flux/
[8] https://mobx.js.org/README.html
[9] https://redux.js.org

Prednáška 8

Redux

vzor/architektúra *Flux*, knižnica *Redux*

8.1 Introduction

je to neuveriteľné, ale stále pracujeme na baterke

- baterka svieti
- Dokonca vieme aj v prípade, že zariadenie baterkou nedisponuje, zobraziť správu a aplikáciu vypnúť. Dokonca to robíme už tesne po spustení aplikácie.
- Pred prístupom k baterke vieme dokonca overiť aj to, či máme povolený prístup ku kamere na *OS Android*.
- Máme ju dokonca pripravenú pre globálny trh, pretože sme vyriešili aj problematiku lokalizácie aplikácie do cudzieho jazyka.
- Roztrhli sme ju na viac komponentov, takže miesto jedného, máme tri.

Dnes sa pozrieme na to, ako uchovávať stav aplikácie pomocou knižnice *Redux*.

8.2 Redux

Redux je knižnica, ktorá slúži ako stavový kontajner pre vašu aplikáciu.

Použitie knižnice *Redux* má veľký význam vo veľkých aplikáciách alebo v *Single Page Application*-s (SPA), kedy riadenie stavu môže byť v priebehu času zložité.

Obrázok 8.1: Redux Logo

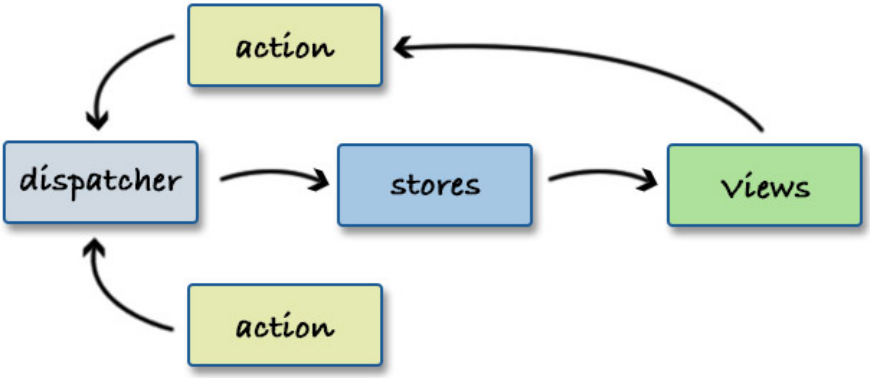
8.2.1 Redux and Flux

Je veľmi podobná *Flux*-u a má s ním veľa spoločného. O *Flux*-e môžeme povedať, že je to vzor, takže *Redux* je *"Flux-like"*, pretože je na tomto vzore založený. Skôr, ako sa teda budeme venovať Redux-u, tak si predstavíme architektúru, resp. vzor *Flux*.

Obrázok 8.2: Flux Logo

8.2.2 Flux Architecture and its Main Characteristics

Začnime teda predstavením architektúry *Flux*-u. Hlavným prvkom tohto vzoru (architektúry) je **dispečer**. Pracuje ako istý hub pre všetky udalosti, ktoré môžu v systéme nastať. Jeho hlavnou úlohou je prijímať notifikácie o vzniknutých udalostiach, ktoré nazývame **akcie** a poslať ich následne do všetkých **skladov**.



Obrázok 8.3: Flux Architecture [39]

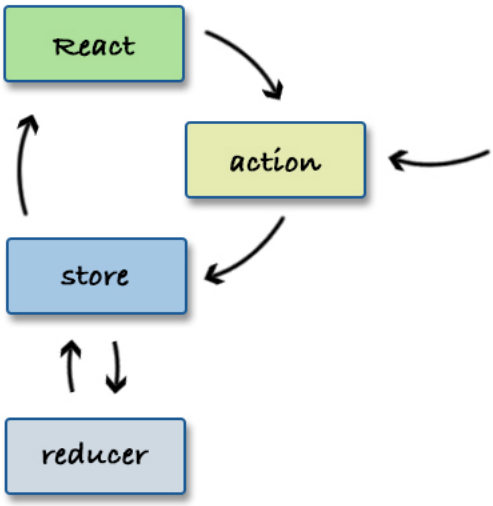
Sklad sa rozhodne, či ho prijatá akcia zaujíma alebo nie. V prípade záujmu sklad zmení svoj vnútorný stav (uložené údaje). Táto zmena sa následne stáva spúšťačom pre prekreslenie dotknutých **pohľadov**, ktoré sú v našom prípade komponenty *React*-u.

> **Poznámka**
> Ak by sme chceli *Flux* porovnať s *MVC*, tak by sme mohli povedať, že sklad reprezentuje model, pretože sklad udržiava údaje aktuálne.

Akcie môžu prichádzať do dispečera z pohľadov (*React* komponentov), ale rovnako tak z ľubovoľných iných komponentov systému. Napr. modul zodpovedný za spracovanie HTTP požiadavky vyšle akciu, keď takúto požiadavku úspešne spracuje.

8.2.3 Redux Architecture

Teraz sa pozrime na to, ako vyzerá architektúra *Redux*-u. Ako bolo uvedené, *Redux* je založený na *Flux*-e, takže architektúra bude veľmi podobná.



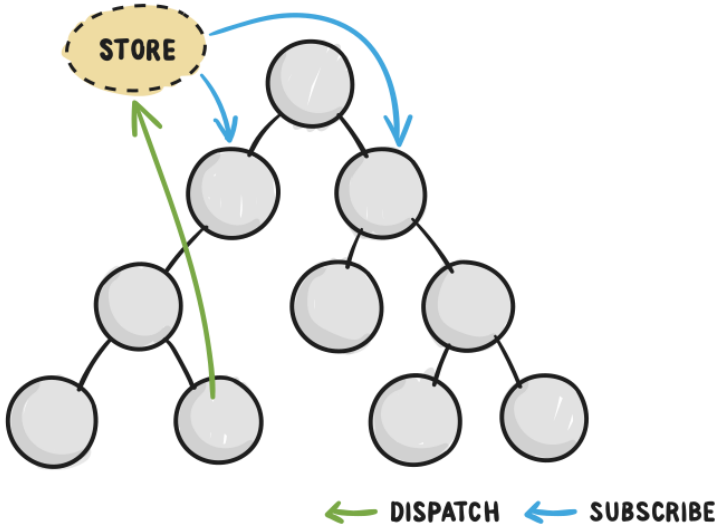
Obrázok 8.4: Redux Architecture [39]

Komponenty aplikácie (blok React) vysielajú (dispatch) **akciu**. Tá so sebou nesie info o tom, čo sa stalo, ako napr.: bolo stlačené tlačidlo, vypršal časovač, vypadla sieť a pod. Tú istú akciu môžu samozrejme vyslať aj iné časti systému. Akcia sa odosiela priamo do **skladu**, de sa rozhodne, čo sa má na základe vzniknutej akcie vykonať. To je uvedené v **reducer**-och. Reducer je "pure" funkcia, pomocou ktorej je možné zmeniť aktuálny stav aplikácie. Takže akonáhle sklad prijme akciu, požiada reducer-ov, aby na základe aktuálneho stavu a prijatej akcie vrátili nový stav. Následne je aktuálny stav doručený všetkým komponentom, ktoré oň majú záujem.

8.2.4 Differences Redux vs Flux

Tým hlavným rozdielom je to, že *Redux* používa len *jeden zdroj pravdy* (z angl. *single source of truth*) - tzv. **sklad** (z angl. *store*). V sklade sú uložené všetky údaje, ktoré vytvárajú stav vašej aplikácie. Jednotlivé komponenty v prípade zmeny stavu túto informáciu pošlú do *skladu* (operácia s názvom **dispatch**)

miesto toho, aby ju poslali zvlášť samostatným komponentom. Komponenty, ktoré majú záujem o dané údaje, sa prihlásia na ich odber zo *skladu* (operácia s názvom **subscribe**).



Obrázok 8.5: Redux Store [22]

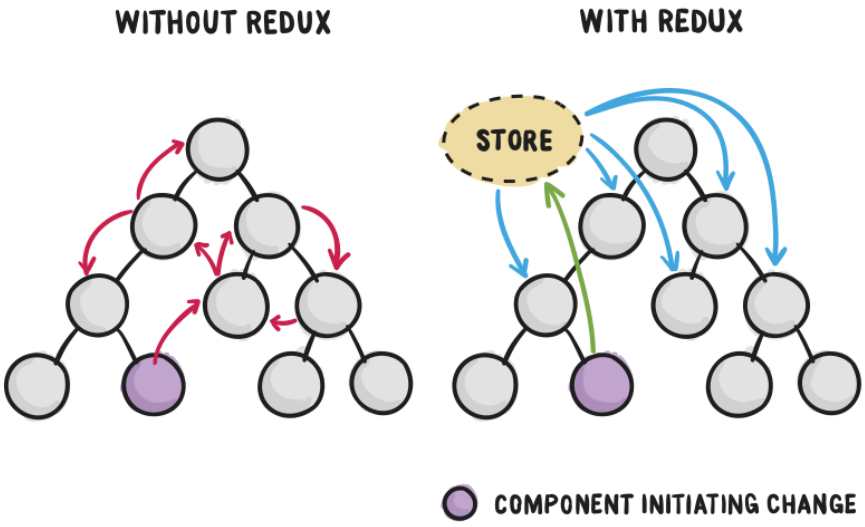
Sklad je v prípade *Redux*-u akýsi prostredník pre každú zmenu stavu v aplikácii. A vďaka *Redux*-u komponenty nekomunikujú medzi sebou priamo, ale cez sklad. Tým sa je možné vyhnúť aj prípadnému vzniku chýb, ktoré so sebou môžu niesť iné stratégie. V prípade *Flux*-u bol týmto prostredníkom dispečer.

8.3 Redux and Torch

8.3.1 Bootstrap

Ako bolo uvedené, význam Redux-u je značný v prípade veľkých aplikácií. V našom prípade jeho použitie až taký význam nemá, takže v tomto prípade to bude len ukážka jeho použitia.

Ak však budete pátrať po best practices, ako používať *Redux* v aplikáciách *React Native*, resp. obecne ako najlepšie používať *Redux*, dôjdete aj ku téme štruktúra projektu. V našom prípade nemá táto téma veľké opodstatnenie, ale pre oddelenie vecí, ktoré súvisia s *Redux*-om od zvyšku aplikácie vytvoríme



Obrázok 8.6: Error Prone and Confusing Strategies vs Redux [22]

samostatný priečinok s názvom `redux/`. Do neho uložíme všetky súbory, ktoré budú s *Redux*-om súvisieť.

Poďme teda po troche aktualizovať baterku o *Redux* a pozrieme sa na jeho jednotlivé stavebné bloky zblízka. Najprv preto: * odstránime každú zmienku o stave z aktuálnej aplikácie * odstránime vlastnosti z komponentov

Okrem toho vytvoríme súbor `helper.js`, do ktorého presunieme funkciu `toggleCamera()` z komponentu `App`. Aby však všetko opäť fungovalo, tak potrebujeme nahradiť volanie prekladovej funkcie `t`, ktorú sme v komponentoch získali pomocou hook-u `useTranslation()`, za volanie `i18n.t()` nad objektom `i18n`, ktorý importneme z lokálneho modulu `i18n.js`:

```
import i18n from "./i18n";
```

8.3.2 Installation

Ešte predtým, ako začneme, nainštalujeme samotný Redux. Je to otázka dvoch balíkov: `redux` a `react-redux`:

```
$ npm install redux react-redux
```

8.3.3 Creating the Store with Reducer

Začneme tým, že vytvoríme sklad, v ktorom sa bude nachádzať stav aplikácie. Keďže k nemu bude môcť pristúpiť ľubovoľná časť aplikácie, je dobrou praxou ho osamostatniť do osobitného súboru s názvom `store.js`, ktorý sa bude nachádzať v priečinku `src/redux/`.

Stav bude aj naďalej reprezentovaný stavovou premennou `isOn` ako doteraz a po spustení bude mať nastavenú hodnotu `false`.

Aby sme sklad mohli vytvoriť, potrebujeme vytvoriť aspoň jednu **reducer** funkciu.

Ako sme už hovorili, reducer je "pure" funkcia, takže na základe vstupu vždy vráti rovnaký výstup bez akýchkoľvek bočných následkov. To tiež znamená, že reducer by nemal volať asynchrónne funkcie.

Reducer dostane dva argumenty funkcie:

- aktuálny stav, a
- akciu, ktorá bola vyvolaná.

Reducer vždy vráti nový stav aplikácie.

V našom prípade si vystačíme s jedným reducer-om, ktorý uložíme do súboru so skladom. Pre náš prípad bude teda vyzerať takto:

```
// in redux/store.js
function torchReducer(state = {isOn: false}, action){
    return state;
}
```

> **Poznámka**
> Ak budete sledovať odporúčania v súvislosti s *Redux*-om a organizovaním projektu, stretnete sa s tým, že všetky *reducer* funkcie sú umiestnené v samostatnom súbore `reducers.js`. V našom prípade máme len jeden *reducer* a bude umiestnený v rovnakom súbore, ako sklad.

Zadefinovali sme akurát podobu počiatočného stavu, kde bude mať premenná `isOn` hodnotu `false`.

Teraz môžeme vytvoriť nový sklad:

```
// in redux/store.js
import { createStore } from "redux";

export const store = createStore(torchReducer);
```

8.3.4 Subscribing to State Changes

Sklad je síce vytvorený, ale jednotlivé komponenty zatiaľ nemajú prístup k jeho obsahu. Je teda potrebné zabezpečiť distribúciu jednotlivých zmien pre jednotlivé komponenty.

Za týmto účelom je potrebné zabaliť hlavný komponent aplikácie do samostatného komponentu s názvom `Provider`, ktorý dostane ako parameter objekt skladu:

```
// App.js
import { Provider } from "react-redux";
import { createStore } from "redux";
import { store } from '../redux/store';

export function App() {
    return (
        <Provider store={store}>
            <View style={styles.container}>
                <ImageSwitch></ImageSwitch>
                <ButtonSwitch></ButtonSwitch>
            </View>
        </Provider>
    );
}
```

Komponent `Provider` nemá vizuálnu podobu. Zabezpečí však distribúciu zmien všetkým dotknutým komponentom (svojim potomkom), ktorí sa nachádzajú vo vnútri tohto komponentu.

> **Poznámka**
> Na účel distribúcie údajov všetkým prihláseným komponentom sa používa `Context`, o ktorom sme hovorili už skôr. Je to logické - `Provider` je vlastne rodičom všetkých komponentov, ktoré sa v ňom nachádzajú, čo vie využiť `Context` doručením nového

stavu komponentom nachádzajúcim sa v celom podstrome.

Chýba nám však ešte prihlásenie sa na odoberanie akcií (udalostí) súvisiacich s aktualizovaním stavu `isOn`. Na prihlásenie sa komponentu k odberu akcie použijeme hook `useSelector()`. V prípade komponentu `ButtonSwitch` bude jeho použitie vyzerať takto:

```
import { useSelector } from "react-redux";

export function ButtonSwitch() {
    const { t } = useTranslation();
    const isOn = useSelector(function(state){
        return state.isOn;
    });

    return (
        <Button
            onPress={null}
            title={isOn === true ?
                t("Turn Off") : t("Turn On")}
        />
    );
}
```

Podobne aktualizujeme aj komponent `ImageSwitch`:

```
import { useSelector } from "react-redux";

export function ImageSwitch() {
    const isOn = useSelector(function(state){
       return state.isOn;
    });

    const image = isOn
        ? require("../assets/bulb_on.png")
        : require("../assets/bulb_off.png");

    return (
        <Pressable onPress={null}>
            <Image source={image} />
```

```
        </Pressable>
    );
}
```

Ak teraz spustíme aplikáciu, bude reagovať na zmenu stavu. Tú teraz vieme zmeniť iba natvrdo v počiatočnom stave z hodnoty `false` na `true` a opačne. Je však možné vidieť, že aplikácia sa naozaj zmení na základe hodnoty tohto stavu.

Ako je možné vidieť, aktuálnu hodnotu stavu nemáme od predka získanú pomocou vlastností (properties), ale priamo zo skladu pomocou hook-u `useSelector()`.

8.3.5 Actions

Takže aktualizácia aplikácie na základe stavu nám už funguje. Teraz sa poďme pozrieť na to, ako vytvoriť akciu (udalosť) vedúcu k zmene stavu.

Obecne je možné akciu charakterizovať ako udalosť, ktorá nastala. Je to teda objekt opisujúci vzniknutú udalosť.

Akcia v sebe nesie dva typy informácií:

- **typ akcie**, a
- **metaúdaje** opisujúce alebo nesúce detaily akcie (vzniknutej udalosti).

Ak sa pozrieme na bežné udalosti, nájdeme jasné paralely. Napr. ak za udalosť pokladáme pohyb myšou (typ udalosti), tak jej metaúdaje budú napr. aktuálna poloha myši (`[x, y]`) a možno aj stav tlačítiek. Ak bol stlačený kláves (typ udalosti), tak metaúdaj bude určite obsahovať kód klávesy, resp. kláves, ktorý/é boli stlačené.

V našom prípade môžeme obecne rozmýšľať o týchto typoch akcií:

- `BUTTON_PRESSED` - keď dôjde k stlačeniu tlačidla,
- `IMAGE_PRESSED` - keď dôjde k stlačeniu obrázka, a
- `STATE_CHANGED` - keď reálne dôjde k zmene stavu (k zmene dôjsť nemusí aj po stlačení, ak aplikácia nemá povolený prístup ku kamere)

Ako som už spomínal, akcia je objekt. Jediný kľúč, ktorý je povinný, je typ akcie `type`.

Akcia samotná nemusí so sebou niesť údaje, ale samozrejme môže. To je vlastne aj náš prípad, nakoľko akcie `BUTTON_PRESSED` a `IMAGE_PRESSED` údaje niesť nepotrebujú, ale akcia `STATE_CHANGED` bude niesť so sebou nový stav.

Typy akcií osamostatníme do osobitného súboru s názvom `redux/actions.js`. V ňom vytvoríme samotné konštanty reprezentujúce jednotlivé typy. Tento prístup so sebou nesie niekoľko výhod:

- nedôjde k zbytočnému preklepu, ak je typ akcie reprezentovaný reťazcom, čo môže mať za následok znefunkčnenie aplikácie,
- ochrana pred preklepmi, k čomu nám samozrejme pomôže aj vývojové prostredie, ktoré bude názvom premenných rozumieť,
- možnosť zdokumentovať všetky typy akcií, ktoré môžu byť v aplikácii vytvorené, na jednom mieste,
- podpora v nástrojoch pre vývojárov.

Súbor `actions.js` sa bude nachádzať v priečinku `src/` a bude vyzerať nasledovne:

```
export const BUTTON_PRESSED = 'BUTTON_PRESSED';
export const IMAGE_PRESSED = 'IMAGE_PRESSED';
export const STATE_CHANGED = 'STATE_CHANGED';
```

> **Poznámka**
> Aj v súvislosti s reprezentáciou akcií v projekte sa môžete stretnúť s rozličným prístupom. Najčastejšie sa dá stretnúť s dvoma súbormi, kde v jednom sa nachádzajú samotné typy akcií (náš prípad) a v druhom sa nachádzajú kostry akcií, resp. funkcie, ktoré objekty akcií vracajú.

Akcie zatiaľ vytvoríme ako vo forme objektov, ktoré vložíme do príslušných komponentov, ktoré ich budú odosielať (dispatch):

- akciu `BUTTON_PRESSED` vložíme do komponentu `ButtonSwitch`:

```
import { BUTTON_PRESSED } from "../redux/actions";

const action = {
    type: BUTTON_PRESSED
}
```

- akciu `IMAGE_PRESSED` vložíme do komponentu `ImageSwitch`:

```
import { IMAGE_PRESSED } from "../redux/actions";

const action = {
```

```
    type: IMAGE_PRESSED
}
```

- akciu `STATE_CHANGED` budeme posielať zvnútra funkcie `toggleCamera()`, takže si ju zatiaľ pripravíme v súbore `heleper.js`:

```
import { STATE_CHANGED } from "../redux/actions";

const action = {
    type: STATE_CHANGED,
    isOn: null  // new state goes here
}
```

8.3.6 Dispatching the Action

Na odoslanie akcie z komponentov priamo do skladu potrebujeme poznať vhodnú funkciu. Od verzie 7.1 Redux obsahuje na tento účel hook s názvom `useDispatch()`, ktorý použijeme aj my. Tento hook vráti funkciu `dispatch()`, ktorú použijeme na komunikáciu s vytvoreným skladom:

```
import { useDispatch } from "react-redux";
import { BUTTON_PRESSED } from "../redux/actions";

export function ButtonSwitch() {
    const dispatch = useDispatch();
    const action = {
        type: BUTTON_PRESSED
    }

    return (
        <Button
            onPress={function() {
                dispatch(action);
            }}
            title="Turn On"
        />
    );
}
```

8.3.7 Handling the Action with Reducer

Po stlačení tlačidla bude doručená akcia typu `BUTTON_PRESSED` do skladu, kde sa stane argumentom reducer funkcií, v našom prípade funkcie `torchReducer()`. Tá je aktuálne prázdna, takže ju aktualizujeme tak, aby vedela zareagovať na každý typ akcie jeho zalogovaním:

```
function torchReducer(state = { isOn: false }, action) {
    console.log(">> reducer invoked");

    switch (action.type) {
        case BUTTON_PRESSED:
            console.log(">> button pressed");
            return state;

        case IMAGE_PRESSED:
            console.log(">> image pressed");
            return state;

        case STATE_CHANGED:
            console.log(">> state changed");
            return state;

        default:
            console.warn(
                `Action type "${action.type}" was not handled.`
            );
            return state;
    }
}
```

> **Poznámka**
> Je dôležité mať aj vetvu `default` pre prípady, kedy nedôjde k ošetreniu danej akcie. To môže byť želané, ale rovnako tak neželané. Je teda dobré takýto prípad aspoň zalogovať, aby sme sa vyhli prípadným problémom.

8.3.8 Dispatching and Handling the Action STATE_CHANGED

Ako už bolo uvedené skôr, reducer je "pure" funkcia. Z toho vyplýva, že ošetrenie akcie by malo byť veľmi rýchle a bez bočných efektov alebo volania asynchrónnych funkcií. Pre nás to znamená, že v reducer funkcii nebudeme volať objekt baterky, aby sa rozsvietil, prípadne zhasol. To totiž so sebou nesie niekoľko bočných funkcionalít, ako je asynchrónne volanie alebo overovanie prístupu ku kamere na platforme Android.

Preto po stlačení tlačidla a vytvorení akcie typu `BUTTON_PRESSED` zavoláme funkciu `toggleTorch()`. Dôoležité však bude upraviť jej kód, kedy v prípade, že sa baterka má naozaj rozsvietiť, vyšle funkcia akciu `STATE_CHANGED` s hodnotou nového stavu v premennej `isOn`.

Najprv teda zavoláme funkciu `toggleTorch()` po stlačení tlačidla:

```
// components/ButtonSwitch.js
import { toggleTorch } from "../helper";

return (
    <Button
      onPress={function () {
          dispatch(action);
          toggleCamera();
      }}
      title={isOn === true ? t("Turn Off") : t("Turn On")}
  />
);
```

Následne upravíme kód funkcie `toggleTorch()` tak, aby pri úspešnom zmene stavu baterky bola táto informácia distribuovaná do skladu:

```
// helper.js
if (cameraAllowed) {
    const state = store.getState();
    await Torch.switchState(!state.isOn);
    store.dispatch({
        type: STATE_CHANGED,
        isOn: !state.isOn,
    });
}
```

Nakoniec už len upravíme príslušnú vetvu v reducer funkcii, kde vytvoríme nový stav na základe hodnoty získanej z akcie:

```
case STATE_CHANGED:
  console.log(">> state changed");
    return {
        ...state,
        isOn: action.isOn,
    };
```

Ak teraz aplikáciu vyskúšame, bude fungovať tak, ako má - po stlačení tlačidla dôjde nie len k rozsvieteniu blesku na zariadení, ale aj k zmene stavu komponentov na základe aktuálneho stavu v sklade. Jediné, čo zostáva aktualizovať, je ošetrenie stlačenia obrázka v komponente `ImageSwitch`, ktoré bude vyzerať podobne, ako v prípade komponentu `ButtonSwitch`:

```
// components/ImageSwitch.js
import { toggleTorch } from "../helper";

return (
    <Button
      onPress={function () {
          dispatch(action);
            toggleTorch();
        }}
        title={isOn === true ? t("Turn Off") : t("Turn On")}
    />
);
```

8.4 Conclusion

Dnes sme si predstavili knižnicu *Redux*, pomocou ktorej sme uchovali stav našej aplikácie. Jej použitie na malých projektoch nemá veľký význam, ale oceníte ju najmä vtedy, ak vytvárate veľké a komplexné aplikácie.

Nabudúce nadviažeme na túto tému a pozrieme sa na ďalšie možnosti perzistencie údajov vo vašich aplikáciách.

Prednáška 9

Data Persistence

perzistencia údajov pomocou AsyncStorage, SQLite, Realm,

9.1 What is Persistence About?

Trvácnosť údajov po reštarte aplikácie

- údaje alebo nastavenia budú v stave, v ktorom používateľ aplikáciu opustil

9.2 Data Persistence and Security

Ak teda chceme, aby údaje aplikácie prežili jej reštart, poprípade reštart celého zariadenia, musia byť niekde na zariadení fyzicky uložené. Otázkou je teda aj bezpečnosť uloženia týchto údajov, a teda:

- prístup k týmto údajom priamo zo súborového systému
- bezpečnosť uloženia samotných údajov, a hlavne
- bezpečnosť uloženia citlivých údajov.

Persisted vs unpersisted — persisted data is written to the device's memory, which lets the data be read by your app across application launches without having to do another network request to fetch it or asking the user to re-enter it. But this also can make that data more vulnerable to being accessed by attackers. Unpersisted data is never written to memory-–so there's no data to access!

Upozornenie
Never store sensitive API keys in your app code. Anything included in your code could be accessed in plain text by anyone inspecting the app bundle. Tools like react-native-dotenv[a] and react-native--config[b] are great for adding environment-specific variables like API endpoints, but they should not be confused with server-side environment variables, which can often contain secrets and api keys.

[a] https://github.com/goatandsheep/react-native-dotenv
[b] https://github.com/luggit/react-native-config/

9.3 Application Sandbox

Mobilné platformy sa preto snažia o ochranu zdrojov aplikácie jej izolovaním od iných aplikácií. To znamená, že jedna aplikácia nemôže vojsť do priečinku súborového systému, ktorý patrí inej aplikácii. Vďaka tomu je možné aplikácie chrániť hlavne od škodlivých aplikácií, ktoré sa snažia dostať ku kritickým súborom, ako napr.

- prístupové údaje k webovým službám, ktoré sú uložené v nastaveniach,
- údaje uložené v databáze,
- fotky, videá a zvukové nahrávky vytvorené kamerou a mikrofónom, alebo obecne
- vytvorené/uložené zdroje aplikácie.

The Android platform takes advantage of the Linux user-based protection to identify and isolate app resources. This isolates apps from each other and protects apps and the system from malicious apps. To do this, Android assigns a unique user ID (UID) to each Android application and runs it in its own process.

problém - "rootnuté" zariadenia

9.4 Where are the Data

Všetky aplikácie majú údaje uložené v predvolenom priečinku s cestou `/data/data/<package name>/`. Predvolene sa tu budú nachádzať databázy aplikácie, nastavenia a pod. V prípade, že aplikácia potrebuje viac miesta, existuje špeciálny priečinok na SD karte s názvom `Android/data/<package`

`name>`.

> **Upozornenie**
> Na SD kartu môžu svoje údaje ukladať všetky aplikácie. Na karte však nie sú žiadne reštrikcie pre ukladanie údajov! To môže viesť jednak k zmätku v názvosloví a organizácii súborov a priečinkov, ako aj k nebezpečenstvu získania údajov takto uložených. Rovnaký problém môže nastať aj v prípade root aplikácií, ale na úrovni celého zariadenia.

Organizácia priečinku s údajmi pre aplikáciu, môže vyzerať nasledovne:

- `databases/` - here go the app's databases
- `lib/` - libraries and helpers for the app
- `files/` - other related files
- `shared_prefs/` - preferences and settings
- `cache/` - well, caches

9.5 AsyncStorage

Async Storage je jednoduché, **aynchrónne**, **nešifrované**, perzistentné úložisko typu **slovník** (key-value storage) pre React-Native aplikácie. Donedávna bol súčasťou `react-native`, dnes je už z neho samostatný projekt[1].

> **Poznámka**
> **Async Storage** umožňuje ukladať len reťazce (údaje typu `string`). Takže ak chceme ukladať aj iné typy údajov alebo obecne objekty, musíme ich predtým serializovať. Pre serializáciu do JSON-u je možné pri ukladaní použiť `JSON.stringify()` a pri načítavaní zasa `JSON.parse()`.

Async Storage is the React Native equivalent of Local Storage from the web.

Async Storage nie je zdieľaný medzi aplikáciami! Každá aplikácia má v rámci vlastného sandbox-u vlastný *Async Storage* a nemá prístup k údajom iných aplikácií.

[1] https://react-native-async-storage.github.io/async-storage/

9.5.1 Do Use Async Storage, When...

Používať *Async Storage* sa oplatí vtedy, keď:

- Persisting non-sensitive data across app runs
- Persisting Redux state
- Persisting GraphQL state
- Storing global app-wide variables

9.5.2 Don't Use Async Storage for...

Vyhnite sa používaniu *Async Storage*, keď:

- token storage
- secrets

9.5.3 Installation

```
$ yarn add @react-native-async-storage/async-storage
```

> **Poznámka**
> Pre staršie verzie *ReactNative* bude potrebné aj linkovať:
> ```
> $ react-native link \
> @react-native-async-storage/async-storage
> ```

9.5.4 Usage

Pred použitím treba importnúť:

```
import AsyncStorage \
    from '@react-native-async-storage/async-storage';
```

Pre ukladanie alebo aktualizovanie (prepisovanie) údajov je možné použiť funkciu `.setItem(key, value)` napr. takto:

```
async function storeData(key, value){
    try {
        await AsyncStorage.setItem(key, value);
    } catch (e) {
        // saving error
```

```
    }
}
```

V prípade, že je potrebné uložiť objekt, spravíme z neho reťazec pomocou volania `JSON.stringify()`:

```
await AsyncStorage.setItem(key, JSON.stringify(value));
```

Na čítanie údajov je možné použiť funkciu `.getItem(key)`, ktorá vráti:

- promise, v ktorom sa nachádza uložená hodnota, alebo
- `null`, ak hodnota pre daný kľúč nie je

```
async function getData(key){
    try{
        return await AsyncStorage.getItem(key);
    }catch(e){
        // read error
    }
}
```

Ako je vidieť, API je veľmi jednoduché a pripomína `localStorage`. Okrem uvedených funkcií ešte obsahuje:

- `.mergeItem(key, value)` - Merges an existing value stored under key, with new value, assuming both values are stringified JSON.
- `.removeItem(key)` - Removes item for a `key`.
- `.getAllKeys()` - Returns all keys known to your App, for all callers, libraries, etc.
- `.multiGet(keys)` - Fetches multiple key-value pairs for given array of keys in a batch.
- `.multiSet(keyValuePairs)` - Stores multiple key-value pairs in a batch.
- `.clear()` - Removes whole AsyncStorage data, for all clients, libraries, etc. You probably want to use removeItem or multiRemove to clear only your App's keys.

9.6 Advanced

Na platforme *Android* je `AsyncStorage` nastavený na veľkosť *6MB*. V prípade prekročenia tohto limitu dôjde k chybe `database or disk is full`.

This 6MB limit is a sane limit to protect the user from the app storing too

much data in the database. This also protects the database from filling up the disk cache and becoming malformed (endTransaction() calls will throw an exception, not rollback, and leave the db malformed). You have to be aware of that risk when increasing the database size. We recommend to ensure that your app does not write more data to AsyncStorage than space is left on disk. Since AsyncStorage is based on SQLite on Android you also have to be aware of the SQLite limits.

V prípade, že potrebujete zvýšiť tento limit, upravte obsah súboru `android/gradle.properties` nastavením hodnoty voľby s názvom `AsyncStorage_db_size_in_MB`. Napr. zvýšenie limitu na *10MB* bude vyzerať takto:

```
AsyncStorage_db_size_in_MB=10
```

9.7 Data Security

Ako už bolo uvedené, `AsyncStorage` ponúka **nešifrované** perzistentné úložisko. Skúsme sa preto pozrieť, ako v skutočnosti vyzerá na disku zariadenia.

Začnime tým, že vylistujeme priečinok `/data/data/` a pokúsime sa identifikovať názov balíčka našej aplikácie s využitím jej názvu, ktorý by sa v balíčku mal nachádzať (napr. `com.persistence-1`).

Po vojdení dovnútra v ňom uvidíme štruktúru priečinkov, ako bola spomenutá vyššie. V našom prípade vojdeme priamo do priečinku `databases/`, nakoľko pracujeme s databázou `AsyncStorage`.

Nájdeme tu súbor `RKStorage`, ktorý keď vylistujeme príkazom `cat`, tak sa bude veľmi ponášať na dump databázy. Dokonca v ňom nájdeme aj text, ktorý sme uložili.

Stiahneme súbor von zo zariadenia príkazom

```
$ adb pull /data/data/com.persistence/databases/RKStorage
```

Ak nad ním spustíme príkaz `file`, tak sa dozvieme, o aký typ súboru ide:

```
$ file RKStorage
RKStorage: SQLite 3.x database, user version 1, last written
using SQLite version 3008010
```

Tým pádom je jasné, že sa jedná o štandardný databázový súbor *SQLite*, čo znamená, že nemáme problém s ním manipulovať pomocou príkazu `sqlite3`

ako na klientskom počítači, tak priamo v *Android* zariadení. Takže nám nerobí žiadny problém akokoľvek upravovať, či kradnúť údaje takto uložené:

```
sqlite> .tables
android_metadata      catalystLocalStorage

sqlite> .schema
CREATE TABLE android_metadata (locale TEXT);
CREATE TABLE catalystLocalStorage (
    key TEXT PRIMARY KEY,
    value TEXT NOT NULL);

sqlite> select * from catalystLocalStorage;
message|this is AsyncStorage

sqlite> update catalystLocalStorage
    set value='you have been hacked'
    where key='message';

sqlite> select * from catalystLocalStorage;
message|you have been hacked
```

9.8 SQLite

Knižnicu SQLite netreba extra predstavovať, pretože vďaka svojim vlastnostiam sa s ňou dá stretnúť na mnohých miestach. Zo stránky projektu sa dozvieme základné vlastnosti:

- SQLite je knižnica napísaná v jazyku C, ktorá poskytuje malý, rýchly, samostatný, vysoko dostupný, plnohodnotný SQL databázový engine.
- SQLite je v súčasnosti najpoužívanejším databázovým enginom, pretože je zabudovaný priamo v mobilných zariadeniach, v mnohých počítačoch, resp. operačných systémoch a býva súčasťou mnohých aplikácií, ktoré denne používame.

Dôležité je povedať, že táto knižnica poskytuje *serverless* databázový engine.

Pre nás je zaujímavé, že SQLite je priamou súčasťou aj mobilnej platformy Android, kde je pomocou SDK možné priamo písať SQL príkazy. Existuje však mnoho rozširujúcich knižníc, ktoré pridávajú aj podporu ORM.

9.9 Realm

zástupca *NoSQL* databáz.

Literatúra

[1] *'Alert'*. Launches an alert dialog with the specified title and message. URL: https://reactnative.dev/docs/alert.

[2] *'ToastAndroid'*. React Native's ToastAndroid API exposes the Android platform's ToastAndroid module as a JS module. URL: https://reactnative.dev/docs/toastandroid.

[3] *3 Things to Know About Android as React Native Developer*. URL: https://desmart.com/blog/3-things-to-know-about-android-as-react-native-developer.

[4] *8 no-Flux strategies for React component communication*. In React, one of the first big issues that comes up is figuring out **how components should communicate with each other.** If you start to dig a little, you'll get a ton of answers. Sooner or later Flux will be mentioned, which will only raise new questions. So here are eight **simple** strategies for communicating between React components. URL: https://www.javascriptstuff.com/component-communication/.

[5] *A Complete React Redux Tutorial for Beginners (2019)*. Trying to understand Redux, it's really confusing how it all works. Especially as a beginner. So much terminology! Actions, reducers, action creators, middleware, pure functions, immutability, thunks… How does it all fit together with React to make a working app? URL: https://daveceddia.com/redux-tutorial/.

[6] *Application Sandbox*. URL: https://source.android.com/security/app-sandbox.

[7] *Async Storage*. Data storage system for React Native. URL: https://react-native-async-storage.github.io/async-storage/.

[8] *Building Hybrid Mobile Apps*. URL: https://www.wavemaker.com/learn/hybrid-mobile/building-hybrid-mobile-apps/.

[9] *Building React Native Apps — Expo or not?* Are you thinking of using React Native to build cross-platform apps? When you set out on a path

to build apps using React Native, you come across the question, should I use Expo or not? URL: https://medium.com/@adhithiravi/building-react-native-apps-expo-or-not-d49770d1f5b8.

[10] *Components and Props*. Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. This page provides an introduction to the idea of components. URL: https://reactjs.org/docs/components-and-props.html.

[11] *Context*. Context provides a way to pass data through the component tree without having to pass props down manually at every level. URL: https://reactjs.org/docs/context.html.

[12] *Core Components and Native Components*. URL: https://reactnative.dev/docs/intro-react-native-components.

[13] *DZone's Guide to Mobile Development 2015*. 2015. URL: https://dzone.com/guides/the-dzone-guide-to-mobile-development-2015-edition.

[14] *DZone's Guide to Mobile Development 2016*. 2016. URL: https://dzone.com/guides/mobile-application-development-11.

[15] *Expo SDK 39 is now available*. URL: https://dev.to/expo/expo-sdk-39-is-now-available-1lm8.

[16] *How the useEffect Hook Works (with Examples)*. URL: https://daveceddia.com/useeffect-hook-examples/.

[17] *How to Pass Data between React Components*. URL: https://www.pluralsight.com/guides/how-to-pass-data-between-react-components.

[18] *i18next documentation*. I18next is an internationalization-framework written in and for JavaScript. But it's much more than that. URL: https://www.i18next.com.

[19] *Images*. URL: https://reactnative.dev/docs/images.

[20] *Internationalization & Localization*. So let's say it turns out that you'll need to localize the product / service you're working on to certain countries. What are the main design aspects? URL: https://uxknowledgebase.com/internationalization-localization-d84795b7962c.

[21] *JSX Specification*. XML-like syntax extension to ECMAScript. URL: https://facebook.github.io/jsx/.

[22] *Leveling Up with React: Redux*. We'll learn how to manage state across an entire application efficiently and in a way that can scale without dangerous complexity. URL: https://css-tricks.com/learning-react-redux/.

[23] *Metro Bundler*. The JavaScript bundler for React Native. URL: https://facebook.github.io/metro/.

[24] *Persisting Data in React Native*. When building React Native applications, there are several ways to persist data, with each having its own strong advantage. In this piece, we shall discuss the most popu-

lar ways to persist data in our React Native application. URL: https://pusher.com/tutorials/persisting-data-react-native.

[25] *Picking your compileSdkVersion, minSdkVersion, and targetSdkVersion.* URL: https://medium.com/androiddevelopers/picking-your-compilesdkversion-minsdkversion-targetsdkversion-a098a0341ebd.

[26] *Props.* Most components can be customized when they are created, with different parameters. These created parameters are called 'props', short for properties. URL: https://reactnative.dev/docs/props.

[27] *Props and State in React Native explained in Simple English.* URL: https://codeburst.io/props-and-state-in-react-native-explained-in-simple-english-8ea73b1d224e.

[28] *React Container Lifecycle.* URL: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/.

[29] *React Native.* JavaScript-ový rámec pre tvorbu natívnych mobilných aplikácií. URL: https://reactnative.dev.

[30] *React Redux.* Official React bindings for Redux. URL: https://react-redux.js.org.

[31] *react-i18next documentation.* react-i18next is a powerful internationalization framework for React / React Native which is based on i18next. URL: https://react.i18next.com.

[32] *ReactJS: Share data between the components.* URL: https://medium.com/coding-in-depth/reactjs-share-data-between-the-components-de492b129086.

[33] *Redux - An Introduction.* Redux is one of the hottest libraries in front-end development these days. However, many people are confused about what it is and what its benefits are. URL: https://www.smashingmagazine.com/2016/06/an-introduction-to-redux/.

[34] *Security: Async Storage.* URL: https://reactnative.dev/docs/security#async-storage.

[35] *Setting up the development environment.* URL: https://reactnative.dev/docs/environment-setup.

[36] *SQLite.* domovská stránka knižnice SQLite. URL: https://www.sqlite.org/index.html.

[37] *Steps for Localizing React Native App.* The localization itself seems difficult for a React Native app because a usual app consists of three projects: the main JavaScript, Xcode project for iOS, and Android files. But in reality, it is not that complex. URL: https://ilyagru.github.io/steps-for-localizing-react-native-app.

[38] *The Component Lifecycle.* Each component has several "lifecycle methods" that you can override to run code at particular times in the

process. URL: https://reactjs.org/docs/react-component.html#the-component-lifecycle.

[39] Krasimir Tsonev. *React in patterns*. A book about common design patterns used while developing with React. It includes techniques for composition, data flow, dependency management and more. 2018. URL: https://krasimir.gitbooks.io/react-in-patterns/content/.

[40] *Understanding Expo for React Native*. A quick guide on what it does, and why it's so popular. URL: https://hackernoon.com/understanding-expo-for-react-native-7bf23054bbcd.

[41] *Using the Effect Hook*. The Effect Hook lets you perform side effects in function components. URL: https://reactjs.org/docs/hooks-effect.html.

[42] *What Are the Different Types of Mobile Apps? And How Do You Choose?* Anyone planning to build an app for their business will inevitably have to answer the question: which type of mobile app do we build? URL: https://clevertap.com/blog/types-of-mobile-apps/.

[43] *What are the popular types and categories of apps*. URL: https://thinkmobiles.com/blog/popular-types-of-apps/.

[44] *What is Mobile Development?* Interested in why you're developing on the platforms you do? Or why you're using the tools you have? URL: https://dzone.com/articles/what-is-mobile-development.

[45] *Where Android apps store data?* URL: https://android.stackexchange.com/a/47951.

Miroslav Biňas

Vývoj aplikácií pre chytré zariadenia

Poznámky ku prednáškam 2020

Vydala Technická univerzita v Košiciach, Letná 9, 042 00 Košice, Slovensko
http://www.tuke.sk

Vydanie: prvé
Náklad: 50 ks
Rozsah: 142 strán
Rok: 2021

Sadzba typografickým systémom LaTeX.
ISBN 978-80-553-3956-6


9 788055 339566