KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – Lecture 10

Security, Accessibility and i18n

Mateusz Pawełkiewicz

1.10.2025

Mobile application security

OWASP Mobile Top 10. The OWASP organization publishes a list from time to time.**Top 10**The most important mobile app security threats. The current list includes misuse of platform mechanisms, insecure data storage, weak authentication, data validation errors, insecure network communication, cryptographic deficiencies, and other aspects (M1–M10). This list serves as a guide for developers, highlighting areas that require special attention when building mobile apps.

Key security practices: With respect to the OWASP Top 10, it is worth using proven practices to secure your application:

- **Keeping secrets out of your code:**It is necessary**avoid hardcoding secrets**(API keys, passwords, etc.) in the application code or in unsecured configuration files. Such data should be stored securely (e.g., retrieved from the server after authentication or saved in the device's encrypted storage). Otherwise, an attacker can easily extract this information, for example, by reverse engineering the application. This is a common mistake – it is often**storing keys and secrets in code**applications. Instead, we use secure platform storage.
- **Secure data storage:**Mobile devices are often vulnerable to physical loss or malware, so**sensitive local data must be properly secured**. Use**platform secure warehouses**– on iOS it is**Keychain**, and on Android**Keystore**– for storing passwords, session tokens, etc. Data stored there is protected at the hardware and system level (e.g., encrypted, linked to the lock screen). Avoid storing sensitive information in regular preferences, files, or SQLite databases without encryption.**Minimize local storage**– write down only what is necessary, and**clear data on logout**user. Clearing data (cache, session data) after logging out ensures that the next person using the device does not gain access to other people's information.
- **Logs without sensitive information:**When debugging an application, it is easy to leave user information, tokens, or other sensitive data in the logs.**Avoid logging confidential data**in the production version of the application. Even if Android restricts access to logcat for third-party applications from a certain version onwards, on many devices privileged system applications can still read the logs. The principle is simple:**log only what is necessary**, preferably static messages, and before publishing, remove or disable detailed debug logging. If you must log something for diagnostic purposes,**mask sensitive areas**(e.g., only show the last 4 digits of the card number) to prevent full data leakage. It's worth using debug log removal mechanisms in the production build (e.g., through tools like ProGuard/R8 on Android, which can remove login calls).

Secure network communication (MITM and pinning). Communication between the application and the server should always be properly secured to prevent**eavesdropping (man-in-the-middle attack, MITM)**. It is necessary**use HTTPS (TLS)**for all communication with the backend and**verify SSL certificates**. The application cannot accept arbitrary certificates (e.g., self-signed ones) without verification – such errors allow an attacker to impersonate the server. According to OWASP, one of the main threats is**unsecured**

communication, e.g., no TLS or accepting any certificate, which allows MITM. Therefore, the standard is to use HTTPS with correct certificate validation*THAT*.

In addition, it is recommended practice to**pinning certificates**, i.e., pinning a server's certificate or public key in the application. This involves the application storing a trusted certificate pattern or public key, and when establishing a TLS connection, it checks whether the server certificate matches this pattern. If it doesn't, it terminates the connection even if the system's list of trusted certificate authorities would accept it. Pinning protects against someone obtaining a fake certificate from a certificate authority or intercepting traffic on the device (e.g., using their own trusted certificate).**Pinning implementation**requires updating the application when the server certificate expires or changes – this should be taken into account (for example, you can pin the public key, which changes less frequently than the entire certificate). OWASP recommends using pinning wherever possible as an additional layer of defense. To summarize:**always use transport encryption (TLS)** i **handle certificate errors correctly**(don't ignore them), and in critical applications consider pinning.

Rate limiting. Mobile applications often use APIs that are vulnerable to abuse – for example, attempts to brute-force password guessing, spamming endpoints, or generating excessive traffic.**Rate limiting**is a mechanism for limiting the frequency of calls to a given action within a specified time period. Although it is typically implemented on the server side, a mobile developer should consider it in the system design. For example:**limiting the number of login attempts**(e.g., several unsuccessful attempts result in a temporary blocking of subsequent attempts) prevents dictionary attacks. Limits can also apply, for example, to the number of API requests per minute from a single token or IP address. According to security recommendations,**applying limits**and appropriate error/response codes (e.g. HTTP 429 Too Many Requests) is crucial for protection against automated attacks and excessive service load. From the mobile application perspective, it is also worth locally**throttle**certain user actions (e.g., not allowing 100 requests per second to be sent from the UI, even if the API cuts it off). These mechanisms do not affect ordinary users, but significantly increase the security and resilience of the system.

Application Accessibility

Accessibility, often abbreviated as**a11y**, is a feature of the application that allows it to be conveniently used by people with disabilities or limitations (e.g., vision, hearing, motor skills). When creating mobile applications, we must ensure that**every user**will be able to use it – in practice, this means support for screen readers (VoiceOver on iOS, TalkBack on Android), the ability to operate the app without looking at the screen or using touch, good color contrasts, scalable text, etc. In many countries, ensuring accessibility is not only good practice, but even a legal requirement (e.g., public apps must meet WCAG 2.1 level AA standards). Below, we will discuss the main aspects of accessibility in the context of apps (especially React Native, but the rules are similar for native apps).

Roles and labels of interface elements

Accessibility Labels. Every interface element that conveys information or is interactive should have**description for screen reader**In React Native, this is done using the

propertyaccessibilityLabel, accessibility labels are set in an analogous way in iOS/Android components. The screen reader reads this label aloud when the focus is on a given element. The label should briefly describe the element, e.g. for a button icon without text, setaccessibilityLabel="Open user settings"instead of just the icon name. In RN, if the element is text, this attribute defaults to the text content – but it often needs to be specified. Example in RN:

```
<Pressable accessibilityLabel="Open user settings" onPress={openSettings}>
<Text>Settings</Text>
</Pressable>
```

Here the button visually shows the "Settings" icon/text, but for screen readers we provide a fuller description**what does he do?**(e.g. "Open user settings").**Good labels**they say**what is it and what is it for**element, do not contain visual details (e.g. button color) – because it is irrelevant to a blind user.

Role (accessibilityRole). Each interface element has a role (type) – e.g., a button, a header, an image, a text field, etc. Specifying a role helps assistive technologies tell the user what they are dealing with. In React Native, this is done using propaccessibilityRoleExample values include:"button" (button), "header"(section heading),"image", "text", "link", "search", "checkbox"etc. We should set roles according to the purpose of the element - for example, if we have built our own touch component that acts as a button, let's give itaccessibilityRole="button"The reader will then announce it as a "button." In RN, interactive elements (TouchableOpacity, Pressable, etc.) are available by default and often treated as buttons, but to be sure, you can specify a role. Example:

```
<View accessible={true} accessibilityRole="button" onTouchEnd={...}>
<Text>Click me</Text>
</View>
```

Here is a normal viewViewhas been marked as available and its role is*button*, so VoiceOver/TalkBack will treat the whole thing as a button. On the other hand, purely decorative elements that**they shouldn't**be read, they should have their accessibility disabled (accessible={false}) so as not to “clutter” the user’s information.

Focus Order and Navigation

Focus order.A user who is blind or has difficulty interacting often navigates an application using hardware buttons or gestures that move**focus**successively between the elements. It is very important that**logical focus order**corresponds to the intuitive order of content. Typically, this order is consistent with the UI layout (e.g., top to bottom, left to right). In React Native (and native apps in general), the reading order corresponds to the order in which elements were added in the view hierarchy. Therefore, the layout should be structured in such a way that**sequence of elements in the code**was logical to read. For example, if something is on the screen on the left and then on the right, but is added in the code the other way around, the reader may read it in the wrong order.

For more difficult cases, platforms offer mechanisms for manual ordering. In the RN, there is an experimental propaccessibilityOrderallowing you to define a specific order of elements

when the default is incorrect. In Android, you can use the attribute natively`android:accessibilityTraversalAfter`, and in iOS arrange the elements in the correct order or use accessibility containers.**Availability containers**(e.g. grouping under`accessible={true}`on the parent in the RN) can cause the entire group to be read as one unit, which also affects the order. In summary: make sure that**navigation through elements is logical**, and use container mechanisms or platform guidelines to achieve this. Test your app with a screen reader,**stepping through the elements**– whether the focus jumps in a sensible way. If not, correct the order in the code or apply appropriate attributes.

Color contrast

Text to background contrastmust be high enough to make the text readable for people with low vision or in low light conditions. WCAG 2.1 guidelines recommend a contrast of at least**4.5:1**for small text sizes (and 3:1 for large headings). This means that, for example, gray text on a light background may be unreadable. Text and background colors should be chosen so that the difference in brightness and color meets these criteria.**Check the contrast**using available tools – for example, the WebAIM website provides a Color Contrast Checker, where you can enter colors and obtain their contrast ratio. During the design phase, it's worth planning an appropriate color palette right away.

Additionally, mobile apps should respect high contrast modes if the system offers them. For example, Android has a high contrast text option – our app should not override system settings (do not block color changes). In iOS, you can use dynamic colors from the system palette (which automatically adjust to high contrast or dark/light modes). Generally,**avoid small, light gray letters on a white background**or color combinations such as dark red text on a black background, etc. Also ensure that interface elements can be distinguished without relying solely on color – e.g., highlight links (not just color) for color-blind people.**Sufficient contrast is the basis for visual accessibility.**

Dynamic text scaling (Dynamic Type)

Many users need**larger text**on the screen – e.g., for visually impaired people or even in certain conditions (small screen, strong sunlight). Mobile systems allow you to set your preferred font size.**Dynamic Type**is an iOS feature (and Android equivalents) that scales fonts in apps according to user preferences. As app developers, we need to**support text scaling**– that is, use text styles that are scalable and test our interface with larger fonts. In React Native, the standard component`<Text>`supports dynamic scaling if we don't block it. RN uses Apple's mechanism**Dynamic Type**automatically for system fonts, but the developer must ensure that the interface still looks correct with larger text. You absolutely should not set the font-size everywhere to a small value, ignoring the system settings.

Good practice: take the device, go to settings**Accessibility > Display & Text Size**(iOS) or**Accessibility > Text Size**(Android) and set the largest font, then run your app. Are all the texts still legible and fit on the screen? Are they not clipping or overlapping other elements? If so, you need to improve the layout (e.g., use scrollable components for long texts, give elements more flexibility). Providing dynamic text often means using relative units (e.g., EMS) instead of pixels when defining custom fonts. In RN, you can also query the

AccessibilityInfo module for the preferred**text size**and respond to changes as needed. Generally,**respect user settings**– someone intentionally set the text to large, so your app should respect that instead of, for example, displaying the text as an image that cannot be scaled.

To summarize the section on accessibility:**design the interface with everyone in mind**Label elements appropriately and ensure logical navigation, use clear contrasts, and allow for content scaling. Test the app using a screen reader (it's worth turning on VoiceOver/TalkBack and trying to use the app yourself) and other features (font change, high contrast). This will make the app easier to use.**more useful and friendly**not only for people with disabilities, but for all users (accessibility often improves the overall usability of a product).

Internationalization (i18n) in applications

i18n (internationalization)is the adaptation of the application to easily translate the interface into different languages and adapt it to different regions.**location (l10n)**is a specific translation and adaptation to a given language/culture. In the context of React Native (and JavaScript applications in general), a popular solution is the library**i18next**with integration**react-i18next**It allows you to manage and switch dictionaries for multiple languages on the fly.**Why is this important?**If an app has users in different countries, they expect an interface in their native language, as well as local formats for dates, numbers, units of measurement, and currencies. Lack of localization translates to a poor user experience and can even make the app unusable (for example, if someone doesn't speak English). Therefore, it's crucial to ensure i18n compliance during development.

i18next w React Native. Using i18next in RN is similar to using React for web. The application loads libraries`i18next` i `react-i18next`, initializes them with a list of supported languages and translation files (e.g., JSON files with keys and translated texts), and then, instead of hardcoding the texts in components, uses a translation function. Typical flow:

1. **Installation**: `npm install i18next react-i18next`(possibly additionally a user language detector, backend for loading files, if needed).
2. **Translation files**: Creating e.g. a folder`locals`, and in it JSON files for each language, e.g.`en.json`, `pl.json`. Inside, key-value objects, e.g.`{ "login": "Log In", "welcome": "Welcome {{name}}" }`for English and Polish equivalents`{ "login": "Log in", "welcome": "Hello {{name}}" }`.Keys can also have a nested structure (dots) for organizations.
3. **Initialization and configuration**: We are calling`i18next.init()`(or we use`useTranslation`hook from react-i18next) where we specify the available languages, the default language, the language detection method (e.g. based on device settings) and load prepared dictionaries. For example:

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import en from './locales/en.json';
import pl from './locales/pl.json';


i18n.use(initReactI18next).init({
```

```
  resources: { en: { translation: en }, pl: { translation: pl } },
lng: 'pl', // default language (e.g. set to Polish)
fallbackLng: 'en',// fallback language
  interpolation: { escapeValue: false }
});
```

After such configuration, the library provides translation context in the application.

4. **Use in components**: Thanks `react-i18next` we can call a hook `const { t } = useTranslation();` and then use `t('login')` Whether `t('welcome', { name: 'Jan' })` to get the translated text. Instead `<Text>Welcome</Text>` we write `<Text>{t('welcome', { name: userName })}</Text>` All interface content should be dictionaries-based, making it easy to switch languages throughout the application.

Plural (pluralization). Different languages have different plural forms, which must be taken into account in translations. i18next supports pluralization automatically – if we pass it to `t` parameter `count`, the library will select the appropriate form. In translation files, we then define texts with suffixes, e.g. keys: `"message_one": "You have 1 new message", "message_few": "You have {{count}} new messages", "message_many": "You have {{count}} new messages"`. For Polish language there are different forms depending on the number (1, 2-4, 5-21 etc.), so we can define `_one`, `_few`, `_many` etc. For English there are usually only two forms: `one` i `other` Then in the code we call `t('message', { count: ileWiadomosci })` without adding a suffix – i18next will select the appropriate variant based on the value `count` and rules of a given language. It is important to **use parameter `count`** with this name, because the library recognizes it. This allows us to avoid manually selecting grammatical forms – the library uses built-in rules (e.g., for Polish, exceptions are defined for numbers ending in 2, 3, 4, but not 12, 13, 14, etc.). This significantly simplifies the preparation of messages such as "X elements." It is also worth providing a translation for the number **0**, because, for example, in Polish there is often a separate construction ("0 messages" – here the form is the same as for the plural, but in other phrases it can be unique). I18next will use the form by default `other` for 0 unless we specifically define a key with a suffix `zero` for a given text.

Date and number formats. Internationalization is not only about words, but also **formatting dates, times, numbers and currencies** according to local customs. For example, the date 03/04/2025 means something different in the USA (April 3) than in Europe (April 3). Similarly, the number **1,234.56** is written as **1 234,56** in Poland (decimal point instead of a dot, space as a thousands separator). **Each region has its own formats** – even countries with the same language have different notation (e.g., US English vs. Canadian English have different date formats and units of measurement). Therefore, the application should use the appropriate locale (*local*) for formatting. We have a built-in JavaScript API **Intl** (Internationalization API) – objects such as `Intl.DateTimeFormat` i `Intl.NumberFormat` allow you to format dates and numbers according to the locale. i18next from version 21+ has built-in formatting functions using Intl. So, for example, you can place a placeholder with formatting in the translation text: `"today": "Today is {{date, datetime}}"` and pass it on `t('today', { date: new Date() })`, and the library will format the current date according to the settings. Of course, you need to define which formats you are interested in (e.g., shorter or longer date). Alternatively, you can use `Intl.DateTimeFormat(locale, options).format(new Date())` – e.g. for Polish `locale="pl-PL"`.

It is important that**set the correct locale with country code**, not just language. E.g.en-US vs one-GB– both are English, but US uses month/day/year, and GB uses day/month/year and a different currency. In turn, for multi-regional languages like Arabic, browsers can default to different regions (e.g. ar-SA vs. ar-EG), which affects, for example, the calendar or numbering system used. In i18next, you can force number/date formatting for a specific region by adjusting the formatter or specifying the locale. Generally, however, when an application uses**device language**, usually the system provides a code with the region, so you can use it.

Example of number formatting in i18next: in translation file:

"file_size": "File size: {{val, number}} MB"

If the user has a pl-PL locale, then('file_size', { val: 1234.5 })will return e.g.*"File size: 1,234.5 MB"*, and for en-US*"File size: 1,234.5 MB"*It's thanks toIntl.NumberFormatunderneath. Furthermore, you can pass formatting options, e.g.{{val, number(minimumFractionDigits: 2)}}so that there are always 2 decimal places. i18next also provides a shortcutcurrency, e.g. {{price, currency(USD)}}will display the dollar amount with symbols.

To sum up, **you should use locale-specific formatting mechanisms instead of gluing dates manually.**This ensures accuracy and adaptability to user habits. We also make sure to account for varying text lengths in different languages – for example, a translation can be significantly longer and needs to be accommodated within the UI (by designing flexible containers).

Offline-First Approach

Traditionally, applications have assumed a constant connection to the internet – this approach*online-first*. **Offline-first**is a design philosophy in which an application is able to run (at least partially) without access to the network. This means that**core app features work offline**, and when the internet is available, data synchronization occurs. In the context of mobile applications, this is an extremely valuable feature: users often lose coverage (subway, plane, weak signal) and would like to continue using the application. The offline-first application will then not display the message "no internet - nothing can be done", but**will allow you to continue viewing the data, make changes that will be saved and sent later**.

The most important components of the offline-first approach are**data cache**, **queue of changes to be synchronized** and **appropriate fallback UI**:

- **Data cache:**The application should**store local copies of your most important data**downloaded from the server to be able to display them when the connection is lost. When used for the first time or when connecting, the data is downloaded and saved in a local database, files (e.g. AsyncStorage in RN, SQLite database, Realm, etc.). In the absence of internet, the application uses**data from the cache**, allowing the user to access recently loaded information. Examples: a news app might cache recent articles so they can be read offline; maps might cache map tiles; an online

store might store a list of recently viewed products. It's important to implement**refresh mechanisms**– e.g., cache versioning or invalidating it to download newer data after the network is restored (so as not to be stuck with old information forever). Caching is the basis of offline mode – it makes our application**he has something to show**without internet.

- **Queue of mutations (operations to be performed):**Just reading data offline is one thing, but what if the user**will perform an action**offline – e.g., write a comment, fill out a form, add an object? The offline-first approach assumes that such operations will not be lost. Instead of rejecting the action,**the application saves them in the queue**and marks it for sending as soon as the connection returns. For example, a message sent in an offline messenger should be saved locally and marked as "pending." When the network returns, the application will automatically attempt to send these outstanding actions to the server (**background sync**). Such a queue should be**lasting**(persistent), meaning even if the user closes the application before the network is restored, synchronization should occur after restarting and reconnecting. Technically, this can be achieved, for example, by saving actions in a local database with a timestamp. After the network is restored – which can be detected by system events or a library such as NetInfo – the application**I'm trying to send**actions from the queue (in the background, invisible to the user). If the server confirms their receipt, it removes them from the local queue. Conflicts can occur here (e.g., the user changed something offline, and in the meantime another change occurred online) – in such cases, it is necessary to anticipate**conflict resolution strategy**(e.g. the last change wins, or we ask the user). However, the most important thing is to**no data entered offline was lost**Examples: a polling app allows offline voting and keeps votes in a queue to submit later; a task management app saves added/edited tasks offline and syncs them with the server at the next opportunity. When implementing this, it's also worth introducing**limit ponawiania**and a backoff mechanism (e.g. if sending fails, try again less frequently) and ensure security - data in the queue may also be sensitive, so it should be stored securely and transmitted after the network is restored in a secure manner (TLS).
- **Fallback UI:**The user should be**informed about offline status**and what the application does. Good practices include, for example:**marking content that comes from the cache**(to make it clear that they may be out of date), the message "You are offline - viewing recently downloaded data". If the user has performed an action (e.g. sent a message), it can be immediately shown in the interface (so-called**optimistic UI update**), but mark it, for example, with a gray color or a clock icon, that it is waiting to be sent. If synchronization fails, you should be able to resend it or inform the person about the problem.**Error handling and fallback mechanisms**are important: the application can, for example, if a queue cannot be sent for a long time, notify the user "Your data will be sent as soon as we regain connection" instead of keeping him in suspense. It should also have**emergency plan**: if we really can't load anything (first run offline, no cache) – we display a friendly message or limited functionality instead of a hang. OWASP (and other sources) recommend, for example,**cache fallback**– showing*"last known good state"*data when newer data cannot be downloaded. At least the user sees something instead of a blank screen. In addition,**inform about synchronization status**For example, as soon as the network is restored, you can display a toast saying "Connection restored – data is being synchronized." When

operations are successful, remove the checkmarks for pending items. This transparency increases user confidence in the application.

In summary, the offline-first application uses**local memory**as a source of truth, and the network is used to exchange data in the background. This requires more work to implement, but significantly improves the UX – the application is**fast**(because he reads locally) and**resistant to weak internet**Examples of successful implementations include the Trello mobile app, where boards and cards can be viewed and edited offline, and changes will be automatically synchronized once the network is restored. In the 2025 era, users will even**they are waiting**some amount of offline operation, and tools (e.g. Redux Offline libraries, WatermelonDB, Apollo Client with persist cache, etc.) make it much easier.

Example: i18n, availability and caching in practice

Finally, let's combine the above into a mini-demonstration. Let's imagine a simple login form in a React Native application, to which we'll add**multi-language support (i18n)** and **accessibility improvements**, and we will also implement**simple offline mechanism**for data list.

1. Adding i18n to the form: Let's say we have a form with fields for "Email," "Password," and a "Login" button. We want to support both English and Polish. We're using i18next:

We prepare translation files`en.json` i `pl.json`:

```
// pl.json
{
"login_title": "Login",
  "email_label": "Email",
"password_label": "Password",
"login_button": "Log in"
}
// en.json
{
  "login_title": "Login",
  "email_label": "Email",
  "password_label": "Password",
  "login_button": "Log In"
}
```

- We initialize i18next with these resources (similarly as described earlier, using`initReactI18next`). We set the default language to, for example, Polish, but also the detection mechanism (to, for example, match the phone's language).
- In the form component we use a hook`useTranslation()`Then, instead of hard-coding the text, we use`t('email_label')`, `t('password_label')`etc.:

```
import { useTranslation } from 'react-i18next';
...
const { t } = useTranslation();
return (
  <View>
```

```
  <Text style={styles.title}>{t('login_title')}</Text>
  <TextInput placeholder={t('email_label')} ... />
  <TextInput placeholder={t('password_label')} secureTextEntry ... />
  <Button title={t('login_button')} onPress={handleLogin} />
 </View>
);
```

- Now the interface will be in the selected language. To change the language, simply call`i18n.changeLanguage('en')`– all components using`t`will respond and display texts in English. Thanks to i18n, we can easily expand the application with additional languages without modifying the logic code.

2. Accessibility labels: Continuing our form, we will add attributes that make life easier for people using screen readers:

- **Label for title text:**If "Login" is a screen header, you could mark it with the header role. In RN, we could do`<Text accessibilityRole="header">...`.
- **Form fields:** element `<TextInput>`is not automatically read with the label (the placeholder is sometimes read, but it's better to provide your own). You can approach it in two ways: either use a component`<Label>`associated with the field (in native iOS/Android you can connect labels to fields), or in RN use`accessibilityLabel`. E.g.:

```
<TextInput
 placeholder={t('email_label')}
 accessibilityLabel={t('email_label')}
 ... />
```

  This will make VoiceOver read "Email, text edit field." Similarly for a password:

```
<TextInput
 placeholder={t('password_label')}
 accessibilityLabel={t('password_label')}
 secureTextEntry={true}
accessibilityHint="Password field. Text invisible while typing."
 ... />
```

  Here we also used`accessibilityHint`to add a hint that the text being typed will be masked (the reader might know this anyway, as secureTextEntry sets the role to password).

- **Login button:**Using the component`<Button>`in RN, it itself passes the text as a label. But if it were, for example, an icon instead of text, it would be necessary to add`accessibilityLabel={t('login_button')}`We also make sure the role is correct (Button usually already has a role button).
- **Grouping and order:**We check with a screen reader that the focus goes to the following order: "Login" title (header), Email field, Password field, Login button. For example, if the UI is in a column, it will be in that order. If the order were incorrect, we could wrap the fields in a container with`accessible={false}`or use the mentioned order attributes (but this is rather unnecessary here).
- **Button size and readability:**We'll make sure the touchscreen elements are large enough (at least 44x44pt according to Apple – easily achieved in RN if we use

standard buttons). Text and background colors – e.g., white "Log in" text on a blue button – should have a contrast ratio > 4.5:1. If the background is light, the text must be dark, and vice versa.

3. Simple offline cache for the list: Now, let's assume that after logging in, the application displays a list of, for example, recent news or products. We want to implement a cache + offline mechanism:

- **Data storage:**We will choose the simplest method –**AsyncStorage**(In RN, this is a module that provides a simple asynchronous key-value store.) When we fetch a list from the API, we also store it in AsyncStorage after a successful fetch. For example:

```
async function fetchData() {
  try {
    const response = await fetch(API_URL);
    const data = await response.json();
setItems(data);
    await AsyncStorage.setItem('itemsCache', JSON.stringify(data));
  } catch (err) {
    console.error(err);
  }
}
```

  This function retrieves data and saves the cache under the key 'itemsCache'.

- **Reading from cache when there is no internet:**We need to detect the offline state. We can use the library@react-native-community/netinfo– Provides network change events. If the user is offline (or the fetch fails with a network error), then:

```
const cached = await AsyncStorage.getItem('itemsCache');
if(cached) {
  setItems(JSON.parse(cached));
} else {
// No cache - show message about no data
}
```

  This means we display cached data (if any). It's worth marking this as offline data in the UI (e.g., a gray banner reading "Offline mode: showing cached data").

- **Cache update:**When reconnecting, we can automatically refresh the data. If we use NetInfo, we can listen whenisConnectedchanges to true and then executefetchData()again. After refreshing, the list shows the latest data and the cache is updated.
- **Change Queue (Simplicity):**If our list allows, for example, deleting items, and the user does it offline, we can save such an action in a queue (e.g., AsyncStorage.setItem 'queuedDeletes', etc.). For this simple example, we can assume that the list is offline read-only, and changes require the Internet (this simplifies matters). In a more complex app, we would use, for example, a library**Redux Offline**or wrote their own queue mechanism as described earlier.

- **Fallback UI:**In our example,**when there is no internet and no cache**, we should show the user, for example, a screen with the message "No connection. Try again later." instead of an empty list. When the cache is available, we will show the list with the cache +, for example, an offline icon. It's also a good idea to provide a "Try again" button for manual refresh, which will force a connection check.
- **Cache data security:**If the list contains very sensitive data, you should consider encrypting it before writing it to AsyncStorage (because AsyncStorage on iOS/Android is private to the app, but stored in a regular file). For our purposes, the demo can be skipped.

Such a simple mechanism implemented will ensure that the user**sees the last known data even without internet**, which significantly improves the experience. In a real application, you can go further – for example, use a library of the type**Realm**the**WatermelonDB**for local data storage and sync mechanisms. However, it is important to anticipate these offline scenarios at the design stage.

Literature:

1. **https://owasp.org/www-project-mobile-top-10/**(Access date: 1/10/2025)
2. **https://reactnative.dev/docs/accessibility**(Access date: 1/10/2025)
3. **https://www.i18next.com/overview/typescript**(Access date: 1/10/2025
4. **https://developer.android.com/training/articles/keystore**(Access date: 1/10/2025)
5. **https://developer.apple.com/documentation/security/keychain_services**(Access date: 1/10/2025)