KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 2

Basic React Native Components

Mateusz Pawełkiewicz

1.10.2025

React Native provides a set of **basic components** for building a mobile application's interface. Here are the most important ones:

- **View** – The basic container component for building the UI (the equivalent of <div> on the web). It serves as a **view container** where other components can be arranged.
- **Text** – A component used for displaying text. Every piece of text in the application must be inside a <Text> component. This allows applying text styles and is a platform requirement (you cannot just insert raw text without <Text>).
- **Image** – A component for displaying images (both from the network and local assets). It allows specifying the image source via the source property (e.g., source={require('path/to/image.png')} for local assets or source={{uri: 'https://address/image.jpg'}} for remote images). It can be styled (sizes, borders, etc.) similarly to other components.
- **ScrollView** – A container component that enables scrolling its content. We use it when we want to be able to scroll a larger amount of components or content within the screen. However, remember that ScrollView renders **all** its children at once, so it can be inefficient for very long lists (in such cases, we use list components, which we'll discuss later).
- **Pressable** – A modern touchable component that can react to different stages of touch (press, hold, release, etc.). It serves as a universal replacement for older touchable components (like TouchableOpacity, TouchableHighlight, etc.). Example usage:

  JavaScript

  ```
  <Pressable onPress={() => console.log('Pressed!')}>
    <Text>Touch me</Text>
  </Pressable>
  ```

  The code above will create an element that reacts to touch, displaying text inside it. Pressable also allows defining callbacks for events like onPressIn, onPressOut, onLongPress, etc., giving great control over behavior during user interaction.

The components above are the foundation of the interface in React Native – most screens will be built from View as containers, Text for displaying labels, Image for graphics, and scrollable elements (ScrollView or lists). We handle user interactions using Pressable or higher-level components based on it (e.g., <Button> internally relies on Pressable).

It's worth noting that React Native also offers many other components (e.g., **TextInput** for text fields, **Switch** for toggles, **Modal** for displaying modal windows, etc.), but in this lecture, we focus on the basic ones needed for layout and lists.

Flexbox – Arranging Elements in a Container

The default mechanism for arranging components in React Native is **Flexbox**. It allows for conveniently defining the interface layout along two axes – main and cross – similar to Flexbox in CSS on the web.

Note: In React Native, Flexbox has a few different default values than in CSS – e.g., the default layout direction (flexDirection) is **column** instead of row. This means that unless specified otherwise, components in a View will be arranged **vertically (one below the other)**.

The main **Flexbox properties** we use are:

- **flexDirection** – Specifies the direction elements are arranged in the container (main axis). Possible values are:
    - "column" (default) – Arranges children from top to bottom (vertically).
    - "row" – Arranges children from left to right (horizontally).
    - (Also "column-reverse" and "row-reverse" for reversing the order) .
- **justifyContent** – Defines the alignment of elements **along the main axis** (i.e., in the direction set by flexDirection). Typical values:
    - flex-start (default) – Elements at the beginning of the axis (left side or top of the container).
    - center – Elements centered along the main axis of the container.
    - space-between, space-around, space-evenly – Various ways to evenly distribute free space between elements.
- **alignItems** – Defines alignment **along the cross axis** (i.e., perpendicular to the main one). Example values:
    - stretch (default) – Stretches elements to fill the space on the cross axis (e.g., usually fills the full width when flexDirection: column).
    - flex-start / flex-end – Alignment to the start or end of the cross axis (e.g., to the left or right edge of the container in a column layout).
    - center – Centering elements on the cross axis (e.g., horizontally centering elements in a column).

Thanks to the combination of these properties, we can easily create "row" or "column" layouts and control the arrangement of elements (e.g., center something, distribute evenly, align to an edge, etc.). For example, to center elements both vertically and horizontally inside a container, we set the style:

JavaScript
```
<View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
  {/* ... */}
</View>
```

In the example above, the <View> container with justifyContent: 'center' and alignItems: 'center' will cause its children to be centered in both directions.

- **flex property** – In addition to the above, we often use the flex property (a number) given to the container's children. It allows specifying what portion of the available space an element will occupy. Example: if we have three columns in a row with styles flex: 1, flex: 2, flex: 3, they will occupy 1/6, 2/6, and 3/6 of the row's width, respectively (because 1+2+3=6).
- **Main axis vs. cross axis:** It's important to understand that with flexDirection: 'column', the main axis is **vertical** (top to bottom), and the cross axis is **horizontal**. Conversely, with flexDirection: 'row', the main axis is horizontal, and the cross axis is vertical. That's

why, for example, justifyContent: 'center' with flexDirection: 'row' will center **horizontally**, but with flexDirection: 'column' – **vertically**.

- **gap (spacing between elements):** In earlier versions of RN, creating space between elements required manipulating margins. Since React Native 0.71, native support for the **gap** property was introduced – similar to CSS, it allows easily setting spacing between all elements inside a flex container. For now, only pixel values are supported (e.g., gap: 10 means a 10px gap between adjacent children of the container). Support for percentages and other units is planned for the future. Thanks to gap, you can avoid adding margins to each element and, for example, the problem of double spacing inside – Flexbox itself distributes the space evenly between elements. For example:

  JavaScript

  ```
  <View style={{ flexDirection: 'row', gap: 8 }}>
    {/* ... children in a row with 8px spacing ... */}
  </View>
  ```

  This will cause all elements inside to be separated by an 8-pixel gap (inner edges), without additional margin on the outsides.

- **Flex Wrap:** By default, a Flexbox container in RN does not wrap elements to a new line – if elements don't fit, they will simply be "cut off" (off-screen). To change this, we use the flexWrap: 'wrap' style on the container. Then, children that don't fit in one row/column will automatically move to the next line/column. In conjunction with flexWrap, the **alignContent** property can also be useful, defining the alignment of the entire lines relative to the container (e.g., alignContent: 'center' will center the arranged lines within the container's space). alignContent only works when flexWrap is enabled and we have more than one line of elements.

Responsiveness: Adapting to Different Screens

- **Flexbox and responsiveness:** Flexbox itself was designed to facilitate the creation of **responsive layouts** for various screen sizes. In practice, many interfaces can be built to automatically stretch or shrink to fill the available space (thanks to using flex and percentage properties for width/height). Often, there's no need to manually calculate pixels for different devices – elements with flex: 1 will divide the space themselves, and elements defined with, e.g., width: '50%' will take up half of the parent, regardless of the device.
- **Dimensions API:** Sometimes, however, we need to know the screen size or adjust the layout conditionally (e.g., a different layout for a phone vs. a tablet). React Native provides the **Dimensions** module to get the application window dimensions. Example usage:

  JavaScript

```
import { Dimensions } from 'react-native';
const windowWidth = Dimensions.get('window').width;
const windowHeight = Dimensions.get('window').height;
```

Thanks to this, we can, for example, check `if (windowWidth > 600) { /* large screen - e.g., tablet */ }` and conditionally change the layout. However, it should be noted that the values from `Dimensions.get()` are **constant for the application's lifetime**, unless we subscribe to change events. Various situations (screen rotation, split-screen mode, foldable devices) can change the dimensions while the app is running. You can attach a listener: `Dimensions.addEventListener('change', handler)` to react to size changes, but there is a more convenient approach – a **hook**.

- **useWindowDimensions:** React Native offers the **useWindowDimensions()** hook, which provides always-current window dimensions and automatically updates the values when the window size or screen orientation changes. Usage is simple:

  JavaScript

  ```
  import { useWindowDimensions } from 'react-native';
  const { width, height } = useWindowDimensions();
  ```

  Such code inside a React component will get the current screen `width` and `height`, and as soon as a change occurs (e.g., the user rotates the device from portrait to landscape), it causes the component to re-render with the new dimensions.

- **Recommendation:** In modern applications, we prefer `useWindowDimensions` over the static `Dimensions.get()`, precisely because it automatically reacts to size changes and fits better with React's reactive paradigm.
- **Example of responsiveness:** Let's say we want to build a grid of tiles that has one tile per row on narrower screens and two side-by-side on wider ones. We can achieve this, for example, like this:

  JavaScript

  ```
  const { width } = useWindowDimensions();
  const columns = width > 500 ? 2 : 1;
  ...
  <FlatList
    data={elements}
    numColumns={columns}
    renderItem={...}
  />
  ```

  Here we determine the number of list columns based on the screen width – above 500 pixels, we'll have 2 columns (e.g., a tablet in landscape), and below – 1 column (a typical phone). This way, the interface adapts automatically. You can also conditionally apply other styles: e.g., for a large screen, set larger margins, a larger font, etc., using regular conditional statements in the component's code.

- **Note on different pixel densities:** It's worth mentioning that the dimensions returned by RN (e.g., in Dimensions or useWindowDimensions) are dimensions in **dp points**, not raw device pixels. This means, for example, a width of 375 on an iPhone means 375 points, which might correspond to 750 physical pixels on a 2x Retina display. Fortunately, we usually don't have to worry about this, as RN itself converts element sizes to device pixels, maintaining a consistent look. However, if we need to know the pixel scale, there is the PixelRatio module, but it's not often needed in the context of typical responsiveness (adjusting the layout).

In summary, **responsiveness** in React Native is achieved mainly through **flexible styles** (Flexbox, percentage dimensions) and possibly through **dynamically checking window dimensions** (the useWindowDimensions hook, conditional columns, conditional styles). There is no CSS Media Queries mechanism here as in web development – we must rely on JavaScript and our own logic to detect, for example, a "tablet mode". Some UI libraries do this for us, but at the RN level, it's worth knowing the tools mentioned above.

Styles: StyleSheet vs Utility approach (NativeWind)

Styling in React Native is done using JavaScript style objects, inspired by CSS. We have two main approaches to defining styles:

1. StyleSheet (built-in RN module)

We use static style objects, usually created by StyleSheet.create(). For example:

JavaScript
```
const styles = StyleSheet.create({
  container: {
    backgroundColor: '#fff',
    padding: 16,
    borderRadius: 8
  },
  title: {
    fontSize: 18,
    fontWeight: 'bold'
  }
});
...
<View style={styles.container}>
  <Text style={styles.title}>Title</Text>
</View>
```

StyleSheet plays a role similar to CSS stylesheets – it allows defining a set of styles and referring to them by name. In practice, StyleSheet.create() simply returns plain objects with locked values, which can offer minor optimizations (RN can operate on these styles more efficiently, assuming they won't change).

Advantages of the StyleSheet approach:

- **Readability and structure:** We give styles names (object keys), which enforces some organization. Styles are collected in one place, easy to edit.
- **Validation and errors:** StyleSheet.create() checks the correctness of property names and values – in case of a typo, it will warn us at runtime.
- **Performance:** Defined styles are immutable – upon component re-render, they can be easily compared (because it's the same object), which aids optimization. Additionally, RN can internally communicate such styles to the native code only once, instead of with every use.

Disadvantages:

- **File/line bloat:** Creating many styles expands the code. For simple components, we have a lot of boilerplate – first define styles, then use them.
- **Lack of dynamism:** StyleSheet is static. If we want to change a style based on props or state (e.g., background color dependent on a variable), we often end up with a construction like style={[styles.base, condition && styles.highlight]} or inline styles. StyleSheet doesn't offer a mechanism for generating styles "on the fly" based on variables beyond such tricks.

2. Utility-first (e.g., NativeWind – Tailwind CSS for RN)

This approach involves using pre-made utility classes, similar to the Tailwind CSS library on the web. Instead of defining our own styles, we compose them from predefined shortcuts. An example using NativeWind (a library implementing Tailwind in RN):

JavaScript
```
import { Text, View } from 'react-native';
import { styled } from 'nativewind';
const Box = styled(View);
...
<Box className="bg-white p-4 rounded-lg">
  <Text className="text-lg font-bold text-blue-500">Text</Text>
</Box>
```

Here we use the className attribute just like in HTML – the NativeWind library translates these classes into actual RN styles. E.g., bg-white sets backgroundColor: '#fff', p-4 is padding: 16 (assuming 1 unit in Tailwind is 4px, which translates to 4 points in RN), rounded-lg is some predefined borderRadius, etc..

Advantages of the utility/Tailwind approach:

- **Development speed:** You can style a component very quickly, writing styles in one line instead of declaring an object. For those familiar with Tailwind on the web, the classes are intuitive and consistent between projects.
- **Design consistency:** Tailwind enforces the use of a certain scale (e.g., only specific font sizes, spacings). This often helps maintain a consistent application appearance and adherence to a design system.

- **Fewer files/less separation:** No need to switch between the style file and the component code – styles (classes) are close to the JSX markup, which some prefer (it's easier to associate a style with a component because you see it right next to it).

Disadvantages of the utility-first approach:

- **Long class strings in code:** A component might end up with a very long string of Tailwind classes, which can be hard to read, especially if they are conditional (e.g., appending classes depending on state). The code becomes wider instead of longer – which not everyone likes (though one could argue it's a matter of taste).
- **Limited palette:** Tailwind has specific values defined (e.g., colors, sizes). When we need something custom (e.g., an unusual shade of color or a non-standard margin), we must either add our own extension to the configuration or resort to a direct style. This means not all cases can be covered by ready-made utilities – sometimes we still have to write a piece of regular style.
- **Abstraction on abstraction:** For people who know CSS or RN styling well, Tailwind can be an extra layer to learn. RN's `StyleSheet` is already an abstraction of CSS, and Tailwind/NativeWind is another abstraction on top of that (special classes, configuration). Some programmers prefer full control and writing styles explicitly in JavaScript rather than relying on class strings interpreted by a library.
- **Performance:** In the context of RN, performance differences between using `StyleSheet` and, e.g., NativeWind are not large for simple screens. NativeWind works during rendering, parsing classes (probably with some optimization, e.g., compiling these styles at startup). `StyleSheet`, on the other hand, creates ready-made styles earlier. In huge lists, utility-first could theoretically be minimally slower on the first render (because it has to process many style strings). However, in practice, both methods are used successfully in production environments.

Summary of choice: The choice between `StyleSheet` and the utility approach depends on preferences and project requirements:

- **`StyleSheet`** gives full control, ease of creating custom design systems in code (e.g., defining custom color sets, spacings as JS constants). It is native to RN, so it doesn't require additional dependencies.
- **NativeWind/Tailwind** speeds up work, especially if one knows Tailwind – many things can be done "out of the box" with a single string of classes. It works well for rapid prototyping or when we want to maintain consistency with a web project using Tailwind.
- However, one must remember that RN is not the web – there's no CSS cascading or media queries, so some of Tailwind's advantages from the web don't apply here (e.g., Tailwind on the web greatly helps manage responsiveness via breakpoints – in RN, we do this in JS anyway; it helps fight CSS cascade – in RN, there is no cascade).

In many projects, approaches can be successfully combined. E.g., using Tailwind (NativeWind) for prototyping or layout, and switching to `StyleSheet` for very specific things or large components. The key is **consistency** – it's best to stick to one style within a project so the team doesn't have to juggle two systems.

Lists: FlatList, SectionList, Keys, and Render Optimization

In mobile applications, we often display lists of data – be it a list of contacts, messages, posts, etc. React Native offers special list components designed for performance:

- **FlatList** – The basic list component, which efficiently renders a scrollable list of single-level items.
- **SectionList** – A variation of the list that supports division into sections with headers (e.g., a list grouped by letters of the alphabet, where each section has a title). It works similarly to FlatList but accepts data grouped into sections.

How do these components differ from the previously discussed ScrollView? Primarily, in **performance with a large number of items**. ScrollView renders **all** children at once, so if we had 1000 items, they would all be created in memory (which can be slow and use a lot of RAM).

FlatList/SectionList work differently: they **render only the items currently visible on the screen** (plus some buffer before and after the screen). As the user scrolls, items that go out of view are removed (or "recycled"), and those that appear are created on the fly. This is a similar concept to "virtual lists" or RecyclerView on Android.

Using FlatList: It requires passing at least two things: a data source (data) and a function that renders an item (renderItem). Example:

JavaScript
```
const data = [{ id: '1', title: 'First' }, { id: '2', title: 'Second' }, ...];
<FlatList
  data={data}
  renderItem={({ item }) => <Text>{item.title}</Text>}
  keyExtractor={item => item.id}
/>
```

Here we pass an array of data objects and render each item as a <Text> with its title. We also used keyExtractor to inform the list that the unique key for each item is the id field of the object.

Keys are very important – just like with lists in React on the web, a key enables rendering optimization and unique identification of an item. FlatList will by default look for a key field in the data (i.e., item.key) and use it; if not found, it will use the array index as the key. Using the index is not recommended, as it can lead to incorrect transitions/animations if the order changes or items are added/removed. Therefore, we either name the id field key, or – as above – use keyExtractor to explicitly point to the property containing the identifier.

SectionList is very similar in use, with the difference that data has a structure of an array of sections, e.g.:

JavaScript
```
const sections = [
  { title: 'A', data: ['Alice', 'Albert'] },
  { title: 'B', data: ['Barbara', 'Boleslaw'] }
];
<SectionList
  sections={sections}
  renderItem={({ item }) => <Text>{item}</Text>}
  renderSectionHeader={({ section }) => <Text>{section.title}</Text>}
/>
```

Here each section has a header (rendered by `renderSectionHeader`) and its own list of data. `SectionList` itself will take care of displaying the sections in the correct order.

List Optimization – Keys and Memoization: We've already discussed keys – they must be unique and stable for the data. The second aspect of performance is **avoiding unnecessary renders of items**, e.g., when state unrelated to a given item changes.

React Native's `FlatList` is internally implemented as a `PureComponent`, which means it does a shallow check of props, and if they haven't changed, it doesn't re-render the entire list. However, we still pass some props to each list item (e.g., `item`) and often an anonymous rendering function. To limit the rendering of individual **list items** when not needed, we can use several techniques:

- **`React.memo` for the list item component:** If we have a separate component defined for the item (e.g., `<ListItem item={...} />`), it's worth wrapping it in `React.memo`. Then, if the same item appears again (e.g., when scrolling up/down) and its props haven't changed, React will skip re-rendering that component.
- **`renderItem` function with `useCallback`:** We often pass `renderItem` as a function literal (`renderItem={({item}) => ... }`). This creates a new function on every render of the parent, which might make the list think something has changed. It's better to extract this function outside the JSX or use the `useCallback` hook to define it, e.g.:

  JavaScript

  ```
  const renderItem = useCallback(({ item }) => <ListItem item={item} />, []);
  <FlatList data={data} renderItem={renderItem} ... />
  ```

  This way, the function reference will be constant between renders (as long as the dependencies in the `[]` array don't change).

- **`extraData` prop:** `FlatList` has an `extraData` prop, through which we can pass additional data that affects the list's rendering. This is mainly used when, for example, the list component is a `PureComponent` and wouldn't normally notice changes in some external state. `extraData={selectedId}` – an example from the documentation – causes the list to know it needs to re-render if the `selectedId` value changes. If our list depends on some state outside of `data` (e.g., a selected item), we should pass it in `extraData`.

- **getItemLayout:** For very long lists with fixed item heights, getItemLayout can be used – it allows calculating an item's scroll position without rendering it (for fast scrolling). This is rather advanced usage – I mention it for awareness.

In the context of our lecture, the most important things to remember are: **FlatList/SectionList are efficient and preferred for lists, keys must be unique, and list components should be memoized to avoid unnecessary refreshes**. In practice, if the list is simple, RN will manage on its own. However, if we notice performance drops when scrolling (lags), it's worth analyzing if, for example, we are not passing a new function/object in renderItem every time, causing a re-render. The solution is often the aforementioned React.memo or PureComponent for the list item.

- **Separator and ListHeader:** It's also worth knowing that FlatList has props that make life easier, e.g., ItemSeparatorComponent (a component that draws a separator between list items), ListHeaderComponent (a list header displayed before the items – useful, e.g., for putting something at the top, like a user profile before a list of posts), or ListFooterComponent (similarly at the bottom). You can also easily add pull-to-refresh (onRefresh + refreshing) and infinite scroll (onEndReached). All this makes FlatList a very versatile tool.

Icons and Fonts in RN

The aesthetics of a mobile application often require the use of icons (e.g., menu icon, action icons) and custom fonts. React Native supports system fonts by default, but we can add our own, and we implement icons through libraries.

- **Icons:** A popular method is to use the **react-native-vector-icons** library (by Joel James, aka **oblador**). It's a set of several popular icon families (FontAwesome, Ionicons, Material Icons, etc.) gathered in one library. In practice, after installing the package (or using Expo, where it's pre-installed as @expo/vector-icons), we can import, for example, an Ionicons icon:

  JavaScript

  ```
  import Icon from 'react-native-vector-icons/Ionicons';
  // ...
  <Icon name="home" size={24} color="#000" />
  ```

  This <Icon> component is essentially a special text component that displays a character from the icon font corresponding to the name "home". The library automatically handles loading the appropriate fonts (e.g., the .ttf file with Ionicons) and makes them available as React components.

- In Expo, this is simplified – @expo/vector-icons is available right away and is based on react-native-vector-icons. Thanks to this, in Expo, you can immediately do import { Ionicons } from '@expo/vector-icons'; and use <Ionicons name="home" size={32} color="green" />.

- **Note:** If you are not using Expo, after installing react-native-vector-icons, you still need to link the icon fonts to the native project. In RN >=0.60, autolinking works – usually, npx react-native link or manually adding the font files to the Xcode and Android (assets) project is enough. Details are in the library's documentation.
- Alternatively, you can use icons in the form of PNG/SVG images. The **react-native-svg** library and its derivatives exist, allowing the import of SVG files as components (which is convenient for custom vector icons). However, for typical icons, **vector-icons** are the most convenient.
- **Fonts:** iOS and Android have their own default fonts (San Francisco on iOS, Roboto on Android, plus fallbacks). If the application's graphic design requires a custom font, it must be included. The process is as follows:
  1. **Add font files** to the project: e.g., .ttf or .otf files. For Android, they are usually placed in the android/app/src/main/assets/fonts/ folder (must be created if it doesn't exist). For iOS – added to the project directory in Xcode and ensure the fonts are listed in the Info.plist UIApp Fonts list.
  2. **Linking/Autolink:** In newer RN (>=0.60), autolinking should detect fonts in the android/assets directory automatically. Sometimes a minor configuration in react-native.config.js is needed. In iOS, after adding to the Xcode project and Info.plist, the fonts will be packed during compilation.
  3. **Using the font in styles:** When the font is available, we use the fontFamily style. E.g., if the font is named "CustomFont-Bold" (the family name defined inside the font file), we use: fontFamily: 'CustomFont-Bold' in the style.
- In **Expo**, it's different – Expo recommends using the **expo-font** library to dynamically load fonts at the application's start (they must be loaded asynchronously before showing the UI). In pure RN, however, after linking, the font is available natively right away.
- **Example:** We add two fonts, "OpenSans-Regular.ttf" and "OpenSans-Bold.ttf". After correct adding, in the RN code, we can use:

  JavaScript

  ```
  <Text style={{ fontFamily: 'OpenSans-Regular' }}>Sample text</Text>
  <Text style={{ fontFamily: 'OpenSans-Bold', fontSize: 18 }}>Bold text</Text>
  ```

  If everything is linked correctly, the text will appear in the appropriate font. In case of problems (e.g., the font doesn't work on Android in release), you need to check the file paths and cache (it may require recompiling the application).

To summarize:

- **Icons**: Easiest via ready-made icon libraries (vector icons). They provide hundreds of ready-to-use icons compliant with project standards.
- **Fonts**: Custom fonts can be used; they must be added to the project natively (or via Expo). Then we use fontFamily in text styles to apply them.

SafeAreaView and StatusBar – Safe Screen Areas

Modern devices, especially iPhones with a notch (camera cutout) or Androids with cutouts and rounded corners, have introduced the concept of a **safe area**. Interface content should fit within the screen area that is not obscured by hardware (notches, roundings) or system elements (status bar, navigation bars).

[Image: Two phone screens. Left: static margins creating gray bars. Right: `SafeAreaView` using full screen while avoiding notch.] *Drawing:* The illustration above shows two approaches. On the left – when we protect with a static margin, we lose a lot of space (gray bars). On the right – using `SafeAreaView` allows utilizing the full screen while avoiding collision with the notch at the top and the home bar at the bottom.

React Native introduced the **`SafeAreaView`** component precisely to solve this problem. Its job is to **automatically add appropriate paddings** from the top and/or bottom (and possibly sides), so that the content we insert does not overlap the device's critical areas. `SafeAreaView` mainly concerns iOS devices (on Android, the status bar usually doesn't overlap the UI, unless we use a transparent status bar).

In practice, usage is simple: we wrap the main screen view in `<SafeAreaView style={{ flex: 1 }}> ... </SafeAreaView>`. This `flex: 1` is important for `SafeAreaView` to stretch to the full screen. Without it, `SafeAreaView` will adjust its height to its children and may not work as intended.

- **RN vs. community library:** The `SafeAreaView` component built into RN was eventually marked as **deprecated** in favor of the **`react-native-safe-area-context`** library, which provides an improved `SafeAreaView` and tools for reading safe area dimensions. In newer projects (e.g., generated by Expo or React Navigation), this library is often already installed. Its use is identical from a JSX perspective but offers more configuration options (e.g., the `edges` prop to specify which edges to consider).
- **`StatusBar`:** This is the system bar at the top of the screen (shows time, battery, etc.). React Native allows us to control its style via the **`StatusBar`** component/API. We can, for example, change the status bar's background color on Android, or change the icon style (light/dark) on iOS. Examples:
  - `<StatusBar barStyle="light-content" />` This will set a light color for the font/status icons (meaning the status bar text will be light – e.g., white text, good for dark backgrounds).
  - `<StatusBar backgroundColor="#61dafb" />` (This will work on Android) It sets a specific background color for the status bar. `<StatusBar ... />` is often placed at the top of the component tree (e.g., in `App.js`) or in each screen separately if different settings are required per screen.
  - You can also hide the status bar: `<StatusBar hidden={true} />`, although this is rarely recommended, as users generally want to see the clock/battery.
- **In conjunction with `SafeAreaView`:** `SafeAreaView` by default also considers the status bar area as "unsafe" – meaning it will add padding from the top exactly the height of the status bar. This makes our UI start below it. Sometimes we may want it differently (e.g., a background image should stretch under the status bar). In the `safe-area-context` library, you can then use the `edges={['left','right','bottom']}` option to skip the top edge.

To summarize:

- **SafeAreaView** – We use it so our interface doesn't overlap with notches and screen roundings. Mandatory on iPhone X and newer for the app to look correct.
- **StatusBar** – A component to change the appearance of the status bar. It allows adjusting the status bar's color scheme to our application (e.g., dark text on a light background or vice versa).

Example: Profile Screen with a List and Responsive Tiles

Finally, let's combine the discussed concepts in a mini-demo of a **User Profile** screen architecture. Let's assume we want to build a screen where profile information is at the top, and below it is a list of items (e.g., user's posts). Additionally, on larger screens, this list should display tiles in two columns (responsive design).

Screen Structure:

1. **Safe Container:** We'll use SafeAreaView as the root.
2. **Profile Header:** Inside SafeAreaView at the top, there can be a regular View containing:
   - Avatar image (Image),
   - Username (Text),
   - Possibly stats (posts, followers, etc. – this could be 3 columns with numbers and labels).
   - Example layout: flexDirection: 'row' for a row or column depending on the design.
3. **Post List:** Below the header, we place the user's post list. Since there could potentially be many posts, we'll use FlatList. Each list item could be, for example, a thumbnail of the post's image or the post title. For simplicity, let's assume they are image tiles (like a photo grid on an Instagram profile). We'll use the numColumns prop of FlatList to arrange the tiles in columns.

The code sketch might look like this:

JavaScript
```javascript
import { SafeAreaView, FlatList, Image, Text, View, useWindowDimensions } from 'react-native';

const ProfileScreen = () => {
  const { width } = useWindowDimensions();
  const numColumns = width > 600 ? 2 : 1; // two tiles side-by-side for width > 600
  const data = [...]; // array of objects representing posts, e.g., {id, imageUrl, title}

  const renderPost = ({ item }) => (
    <View style={{ flex: 1, margin: 4, backgroundColor: '#eee' }}>
      <Image source={{ uri: item.imageUrl }} style={{ width: '100%', height: 150 }} />
      <Text style={{ padding: 4 }}>{item.title}</Text>
    </View>
  );

  return (
    <SafeAreaView style={{ flex: 1 }}>
      {/* Profile Header */}
      <View style={{ padding: 16, alignItems: 'center' }}>
        <Image source={{uri: profile.avatarUrl}} style={{ width: 80, height: 80, borderRadius: 40 }} />
        <Text style={{ fontSize: 20, fontWeight: 'bold', marginTop: 8 }}>{profile.name}</Text>
        <Text style={{ color: 'gray' }}>@{profile.username}</Text>
      </View>

      {/* Post List */}
      <FlatList
        data={data}
        keyExtractor={(item) => item.id}
        renderItem={renderPost}
        numColumns={numColumns}
        contentContainerStyle={{ padding: 8 }}
        ListEmptyComponent={<Text>No posts.</Text>}
      />
    </SafeAreaView>
  );
};
```

Discussion: The code above:

- Uses `useWindowDimensions()` to detect the width. If width > 600, we set 2 columns (`numColumns=2`), otherwise 1 column.
- The profile header is a simple centered view with an image and texts.
- `FlatList` uses `contentContainerStyle` to add 8px padding around the entire list (this prevents tiles from touching the screen edges).
- Each tile (`renderPost`) is a `View` with a margin, containing an image and a title.
- We used `flex: 1` in the `View` style so that tiles in a row spread evenly across the full width.
- **Important:** When using `numColumns`, RN divides the data into rows itself, and our item should take `flex: 1` to fill its column.
- `ListEmptyComponent` shows the text "No posts." if the `data` array is empty.

Responsive Tiles: Thanks to `numColumns` and `useWindowDimensions`, on a phone, the user will see a single column of posts (one below the other, full width), and on a tablet in landscape, they will see two columns side-by-side. This could be extended further – e.g., for *very* wide screens (TV? large tablet), set 3 columns, etc. If needed, the logic can be more complex (e.g., breakpoints: >600px = 2 columns, >900px = 3 columns, etc.).

SafeArea and StatusBar in practice: Since we wrapped everything in `SafeAreaView`, on an iPhone with a notch, the avatar and username won't be hidden under the status bar – `SafeAreaView` will add padding at the top. If the header background were a different color, it would be wise to apply that color to `SafeAreaView` as well (`style` on `SafeAreaView: backgroundColor`), otherwise, there might be, for example, white space at the top (behind the notch). We didn't set the `StatusBar` in this component – the system status bar will be visible by default. However, we could globally in `App` set, for example, `<StatusBar barStyle="dark-content" backgroundColor="#fff" />` if we want dark icons on a light background to match the white background of `SafeAreaView`.

This demo screen illustrates the combined use of many concepts from the lecture: basic components (View, Text, Image), Flexbox (centering the header, tiles in a grid), responsiveness (dimensions hook and conditional columns), StyleSheet vs utility (here we used inline styles for brevity, which is equivalent to defining in StyleSheet), lists (FlatList with column optimization), icons/fonts (one could add, e.g., a profile edit icon next to the name), SafeAreaView, and StatusBar control.

Literature:

1. https://reactnative.dev/docs/getting-started (Access Date: 1.10.2025) - Official React Native documentation.
2. https://docs.expo.dev/ (Access Date: 1.10.2025) - Official Expo documentation.
3. https://docs.expo.dev/guides/new-architecture/ (Access Date: 1.10.2025) - Key Expo documentation explaining the New Architecture (Fabric).
4. https://docs.expo.dev/router/introduction/ (Access Date: 1.10.2025) - Expo Router documentation, the modern navigation standard (mentioned in the lecture).
5. https://docs.expo.dev/guides/environment-variables/ (Access Date: 1.10.2025) - Detailed description of managing environment variables (.env) in Expo.
6. https://reactnavigation.org/ (Access Date: 1.10.2025) - React Navigation documentation (an alternative, popular navigation library).
7. https://nodejs.org/ (Access Date: 1.10.2025) - Official Node.js website.
8. https://code.visualstudio.com/ (Access Date: 1.10.2025) - Official Visual Studio Code website