KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 1

Introduction to mobile app development with Expo and React Native

Mateusz Pawełkiewicz

1.10.2025

React Native Architecture – Bridge and Native UI

React Native has a unique architecture that connects the JavaScript world with native user interface components. We write the application code in JavaScript or TypeScript (e.g., using React), but the interface is rendered using **native views** (UI components specific to iOS/Android).

How is this possible? The core is the so-called **Bridge** – an intermediary layer between the JavaScript code and the platform-specific native code. The Bridge is an asynchronous message-passing mechanism (in JSON format) between the JavaScript "world" and the native "world".

In practice, this means that the application logic runs in a JavaScript environment (a JavaScript engine is launched inside the mobile app, e.g., **JavaScriptCore** on iOS or an included JavaScriptCore/Hermes on Android), and when something needs to be drawn on the screen or a native function needs to be called – a message is sent via the Bridge to the native part of the application.

Multithreading: When an RN application starts, several threads responsible for different tasks are running.

- The **Main thread (UI)** is responsible for rendering views and handling user interactions – this is where native components and responses to touch, scroll, etc., operate.
- Concurrently, a **JavaScript thread** runs, executing our business code (application logic written in React/JS).
- There is also a **shadow thread**, which calculates view layouts – RN uses the **Yoga** engine to calculate the flexbox layout in a separate thread, after which the results are passed to the native thread, which draws the UI elements based on them.

This architecture allows the interface to remain smooth – rendering and touch handling happen independently, and any heavy computations in JS do not immediately block the UI.

Bridge – JS to Native Communication: The bridge is a key element of RN – it's the communication channel between the JS code and native modules. It uses serially transmitted, serialized messages (typically JSON) – allowing code written in a different language (JS) to "talk" to Objective-C/Swift or Java/Kotlin code.

For example, if an event occurs in a native module (e.g., a button press or an accelerometer event), it will be passed as a message to JS, where our application can handle it. Then, if a UI change is needed as a result, the JS side sends a message via the Bridge with information about the change (e.g., "change label text to X"), which goes to the native UI thread and calls the appropriate methods to update the views.

This entire process is asynchronous and batched – RN packs many operations together before sending them across the Bridge to minimize communication overhead.

It's worth noting that this approach (asynchronous messages) provides great flexibility but has some performance limitations. In extreme cases (e.g., a high volume of interactions and UI updates at once), the Bridge can become a bottleneck – every piece of information must be serialized, sent, and deserialized, which can cause delays or UI "stutter" at high event frequencies.

The RN team is working on a **new architecture (Fabric)**, which eliminates the traditional Bridge in favor of a synchronous JSI interface.

In summary, the React Native architecture allows us to write an application in React/JS that runs in its own runtime environment, with the interface rendered from native components thanks to the Bridge mechanism. This approach gives the best of both worlds – rapid development in JavaScript with the ability to use native UI elements and device APIs.

Environment Setup and Tools

Before we start creating an application, we must prepare the appropriate development environment:

- **Node.js** – An up-to-date Node.js (LTS version recommended) is required, which will be used to run Expo tools and the JavaScript bundler (Metro). The npm package manager is installed along with Node, which we will use.
- **Expo CLI / Expo tools** – Expo provides a command-line tool for managing projects. It is no longer necessary to install it globally – it's recommended to use it via **npx**, which ensures the latest version is used (e.g., `npx expo start`). The expo package contains a lightweight CLI available via the expo (or npx expo) command.
    - **Note:** The global `expo-cli` tool used to exist, but `npx create-expo-app` and `npx expo` have now replaced that approach.
- **Code Editor** – It's worth installing your favorite programming editor. Visual Studio Code is recommended (due to good TypeScript, ESLint, etc. support).
- **Expo Go (on your mobile device)** – The **Expo Go** application for your phone/tablet (available in the App Store and Google Play). It allows you to run Expo apps in development mode on a physical device by connecting to your development server. Install Expo Go on your device – it will be used for testing applications without needing to build full packages.
- **Android Studio (for Android emulator)** – If you don't have a physical device or want to test on an emulated Android, install **Android Studio** along with the Android SDK. During installation, make sure to select the components: Android SDK, Android SDK Platform, Android Virtual Device (AVD). Android Studio will be used to create and configure virtual Android devices (AVDs) on which you can run the application. After installing Android Studio, it's also a good idea to add the `ANDROID_HOME` environment variable pointing to the SDK location and add the `platform-tools` (adb) to your PATH – this will allow Expo to automatically detect the emulator.
- **Xcode (for iOS simulator)** – **(Applies to macOS users)**. To emulate the application on an iPhone, you need Xcode (available in the Mac App Store). After installing Xcode,

you can run the **iOS Simulator** and test the application. Note: The iOS Simulator is only available on macOS; Windows/Linux users can test on a physical iPhone via Expo Go but cannot run the native simulator.

- **Watchman (macOS)** – Optionally on macOS, it's recommended to install the Watchman tool (Facebook) for listening to file changes, which speeds up the Metro bundler. It can be installed, for example, via Homebrew (`brew install watchman`).

After installing the tools above, we can verify the configuration. Check the Node (`node -v`) and npm (`npm -v`) versions in your terminal. Make sure Android Studio has at least one virtual device (AVD) defined. On macOS, check if the iOS Simulator opens via Xcode.

Expo Developer Tools: After starting the project (with the `expo start` command), Expo opens an interactive menu in the console (the **Terminal UI**) and (optionally) a browser page with developer tools. From the Terminal UI, we have keyboard shortcuts for common actions, including: `a` – run on Android Emulator, `i` – run on iOS Simulator, `w` – run a preview in the web browser (Expo can also build a web version).

When running on an Android Emulator for the first time, you may be asked to **launch a dedicated emulator** – Expo will detect installed AVDs and let you choose. It's also worth logging into the Google Play store on the emulator and updating the Expo Go app within it (on emulators without Google services, Expo CLI may also offer to install Expo Go via `expo client:install:android`).

Connection Mode – LAN vs Tunnel: By default, the Expo Dev Server listens on your computer's local IP address (LAN mode). The device with Expo Go must be on the same WiFi network to connect via the QR code or address. If the computer and phone are not on the same network (e.g., you are using a public network or have connection issues), Expo allows using **Tunnel** mode – it tunnels traffic through external servers. To use it, start the server with the command `expo start --tunnel` or press `shift + t` in the console/select the **Tunnel** option while the server is running. In tunnel mode, after scanning the QR code, Expo Go connects via a generated URL (e.g., using ngrok).

Keep in mind that the Tunnel connection is significantly slower than local – refreshing the app takes longer. Therefore, it's recommended to use tunneling only when necessary, and preferably use an emulator or a physical device on the same local network for a faster development cycle.

Creating a New Expo Project

With the environment ready, we can create our first Expo project. Expo provides a convenient project creator: **create-expo-app**. In the latest versions, we use it via **npx**.

For example, to create a new project named **MyApplication**, we execute in the terminal:

```
npx create-expo-app@latest MyApplication --template blank
```

The command above will create a MyApplication directory with the basic structure of an Expo app. We used the --template blank parameter here, which indicates we want to use the "Blank" template (an empty project with minimal dependencies, without pre-configured navigation).

Expo provides several official templates: the default (default) already includes TypeScript configuration and example tab navigation (Expo Router), **blank** – a clean minimal project, **blank (TypeScript)** – a clean project with TypeScript enabled, **tabs** – a project with a ready-made tab router and TypeScript, and the **bare-minimum** template – a minimal project with generated android/ios directories (for a potential **"eject"** to a native project).

In our case, we chose blank – just the absolute basics. **Note:** You can create a project with TypeScript from the start, e.g., with the --template blank-typescript command, or add TS configuration later. New versions of Expo (SDK 49+) default to initializing a project with TS enabled if they detect .tsx files. The Expo documentation confirms: the default template already has TypeScript and recommended tools configured.

After creating the project, let's look at the folder and file structure:

- **App.js / App.tsx** – The main application file, containing the React component that serves as the entry point. In an Expo project, this is typically App.js (or App.tsx if using TS). This component is automatically registered and run by Expo. In the blank template, we'll see simple code displaying the text: "Open up App.js to start working on your app!" (or a similar welcome message). Our goal will be to edit this file.
- **package.json** – The file defining our node project: application name, version, npm dependencies (generated by create-expo-app), scripts (e.g., start, android, ios etc. for running). The generated package.json already includes the expo dependency (correlated with a specific Expo SDK version), react, and react-native (versions appropriate for the Expo SDK), and several scripts to facilitate work.
- **app.json / app.config.js** – The Expo configuration file (more on this in the next chapter). By default, create-expo-app creates a static app.json file with basic settings (including the app's "name" and "slug"). This file is used by Expo, among other things, to generate native files (during a potential prebuild) and to configure the app in Expo Go.
- **assets/** – A directory for static assets, such as images and fonts. By default, we'll find things like the app icon (adaptive-icon.png) and the splash screen background image (splash.png) here. We can place our own graphics here, for example, which we'll load via require in the RN code.
- **node_modules/** – A directory with installed JavaScript dependencies (npm packages required by our application). After creating the project and installing dependencies (which create-expo-app does automatically, unless we used the --no-install option), node_modules will contain expo, react, react-native packages, and many others needed for the Expo ecosystem to function.
- **tsconfig.json** – (if using TypeScript) The TypeScript configuration file. In the blank-typescript template, it will be present and properly configured for RN. If switching to TS independently, creating such a file with the correct options is required.

- **Other config files:** e.g., **babel.config.js** (Babel configuration, usually generated automatically with the babel-preset-expo preset), **.gitignore** (list of ignored files for git), etc.. – These files generally don't require attention at the start.

We can now run the prepared project. In the terminal, navigate to the project folder (cd MyApplication) and run the development server with the command:

```
npx expo start
```

This command will start the local Metro bundler server listening on port 8081 and prepare our application to run. In the terminal, you will see the Expo Dev Tools interface with some information. A **QR code** will appear, which we can scan using the Expo Go app on our phone to run our application on a physical device.

Alternatively, use the shortcuts: press a to automatically start the application on a running Android emulator (Expo will try to open an AVD and install/run the Expo Go client in it), or i (macOS only) to run the application in the iOS Simulator. You can also press w to run the application as a web page (Expo will then use webpack – but remember that not all native APIs work on the web).

If you are scanning the QR code with your phone, make sure the phone and computer are on the same WiFi network. Otherwise, use the previously mentioned tunnel mode. With a direct (LAN) connection, the Expo Go app should detect our server and load the application.

First Run: After the application loads (either on the emulator or device), we should see a welcome screen with a message (e.g., "Open up App.js to start working on your app!" and a few tips). This is the content of the default App component. Congratulations – you've run your first app in Expo □!

Expo Project Configuration (app.json, app.config.ts, .env)

Now that the project is created, it's worth understanding how we configure various aspects of the application. Expo uses a special configuration file that defines, among other things, the application name, package identifiers, icons, splash screens, and other settings. Additionally, we often use .env files to store environmental configuration (e.g., different API endpoints for development vs. production) and linting/formatting tools to maintain code quality.

Expo Configuration File (app.json / app.config.js)

In the project's root directory, there is an **app.json** file (or alternatively **app.config.js/ts**) which contains the Expo configuration. In its simplest form, app.json might look like this:

JSON
```
{
  "name": "My app",
```

```
  "slug": "my-app"
}
```

Above, only the application name and the so-called **slug** (a unique project identifier, used e.g. in Expo links) are defined. In practice, this config can contain many more fields – e.g., screen orientation, icons and splash image (start screen), URL schemes (deep linking), app permissions, information for the Google Play/Apple App Store, etc.. The full list of available properties is defined by the Expo schema (the documentation refers to the app.json reference).

It's important that the configuration file **must** be placed in the project's root directory, next to package.json, because Expo CLI reads it automatically.

If we use the JSON format, all settings are static. However, Expo allows using a JavaScript/TypeScript file as configuration – just create app.config.js or app.config.ts. When such a file exists, it is preferred over app.json and allows for dynamic configuration generation.

Advantages of dynamic config: In an app.config.js/ts file, we can write normal code (with some limitations). We can, for example, make certain values dependent on environment variables, combine config from several sources, or use comments. Such a file exports a configuration object. For example, we could write:

JavaScript
```
// app.config.js
const myValue = 'My App';
module.exports = {
  name: myValue,
  version: process.env.MY_CUSTOM_PROJECT_VERSION || '1.0.0',
  extra: {
    fact: 'kittens are cool',
  },
};
```

Here we use the MY_CUSTOM_PROJECT_VERSION environment variable (if defined) or default to '1.0.0' as the app version. We also added an **extra** field with our own data (an example fact). All fields from extra will be embedded in the application and available at runtime through the **expo-constants** module. They can be read, for example, like this:

```
import Constants from 'expo-constants'; console.log(Constants.expoConfig.extra.fact); // will print "kittens are cool"
```

Data from extra (and most of the configuration) is available in Constants.expoConfig during app runtime. This allows us to pass, for example, different API keys for different environments (by setting them in extra depending on an ENV variable at build time).

Note on sensitive data: The configuration file (whether JSON or JS) is included in the application and available in it as plain text. Therefore, **you should not put secrets there** (e.g., secret API keys, passwords). Although Expo automatically filters certain fields when sharing the config at runtime (e.g., sensitive keys like Code Signing certificates are excluded), the config is generally for public settings. Sensitive data is better kept outside the application or

by using secure storage mechanisms (Expo offers, e.g., EncryptedStorage or secrets configuration in EAS).

Environment Variables (.env)

Many applications need to separate configuration for different environments – e.g., a different API URL during development versus on production. Instead of "hardcoding" such values in the code, good practice dictates using **environment variables**.

This is simplified in Expo projects – Expo CLI can automatically read .env files. To use it, we create a .env file in the project's root directory and define key-value pairs in separate lines, e.g.:

```
EXPO_PUBLIC_API_URL=https://staging.example.com EXPO_PUBLIC_API_KEY=abc123
```

The EXPO_PUBLIC_ prefix: Expo intentionally requires that variables intended to be available in the application must start with EXPO_PUBLIC_. Only such variables from the .env file will be loaded and injected into the application. This makes it clear that any variable visible to the app **will be publicly accessible** (after bundling the code). In our example, we defined a base API URL and some key.

Now in the JS/TS code, we can read these values via the process.env object:

```
const apiUrl = process.env.EXPO_PUBLIC_API_URL; const apiKey = process.env.EXPO_PUBLIC_API_KEY;
```

When we run the application with expo start, Expo CLI will, on the fly, replace all references to process.env.EXPO_PUBLIC_API_URL with the corresponding string from the .env file (e.g., https://staging.example.com). Importantly, this happens during bundling – we don't need to use additional libraries like dotenv at all; Expo handles it natively. During development, even changes to .env can be picked up without a full CLI restart (they only require reloading the app, e.g., shake > Reload).

The prefix name EXPO_PUBLIC_ is not accidental – it's meant to remind you that such variables **are not private**. Never put secret keys in .env without this prefix hoping they won't end up in the app!. Expo won't load variables without this prefix by default anyway, and those with the prefix are compiled into the application's JavaScript bundle (meaning any user could read them from the app files).

A good pattern is to keep secrets only on the server-side, and only place public keys or identifiers in the mobile app. If we need different configurations (dev/prod), we can use multiple .env files (Expo supports the standard .env, .env.local, .env.production, etc. according to priority) or use the **EAS Secrets** functionality when building the app via EAS (Expo Application Services). For the purposes of our course, however, it's enough to know that .env with EXPO_PUBLIC_ is a convenient way to provide configuration to the code.

ESLint and Prettier Tools – Code Quality

When writing applications in JavaScript/TypeScript, it's worth using tools that help maintain code quality.

- **ESLint** is a static linter – it analyzes code and can catch errors or non-recommended constructs before we even run the app.
- **Prettier** is an automatic code formatter – it ensures a uniform formatting style (indentation, semicolons, quotes, etc.), which makes the code consistent regardless of who wrote it.

We can easily integrate both tools in an Expo project.

ESLint: New Expo projects don't have an ESLint config file by default, but Expo provides a simple way to add one. Execute the command:

```
npx expo lint
```

This will install the necessary ESLint dependencies and create a configuration file (since SDK 53, this is eslint.config.js compatible with the new **"Flat Config"** format). The default configuration extends the recommended set of Expo rules (eslint-config-expo). After creating the config, you can edit it as needed – e.g., adjust the list of ignored files or add your own rules. To check the code with the linter, run npx expo lint again – it should scan the files and list any warnings or rule violation errors. It's worth integrating ESLint with your editor – e.g., in VS Code, install the ESLint extension so problems are underlined in real-time as you type.

Prettier: Prettier can be used on its own (e.g., configuring format on save in VSCode), but it's best to integrate it with ESLint, so that breaking formatting rules is treated as a warning/error. Installation is simple – we add the Prettier packages and integration:

```
npx expo install prettier eslint-config-prettier eslint-plugin-prettier --dev
```

The command above (taken from the Expo documentation) installs Prettier and two components: **eslint-config-prettier** (sets up ESLint to disable rules that conflict with Prettier) and **eslint-plugin-prettier** (allows treating formatting problems as ESLint errors).

Next, you need to modify the ESLint configuration to include Prettier. For the new eslint.config.js (Flat Config), the documentation suggests:

JavaScript
```
const expoConfig = require('eslint-config-expo/flat');
const eslintPluginPrettierRecommended = require('eslint-plugin-prettier/recommended');
module.exports = [expoConfig, eslintPluginPrettierRecommended, { ignores: ['dist/*'] }];
```

If you are using the classic .eslintrc.js, the configuration would look something like this:

JavaScript
```
module.exports = {
  extends: ['expo', 'prettier'],
  plugins: ['prettier'],
  ignorePatterns: ['/dist/*'],
```

```
  rules: {
    'prettier/prettier': 'error'
  }
};
```

In the above, we set that we extend the base Expo config and add the `prettier` extension (which disables conflicting ESLint rules), activate the `prettier` plugin, and add the `'prettier/prettier': 'error'` rule – this means that if the code does not meet Prettier's formatting rules, it will be reported as an ESLint error. (You can use 'warn' instead of 'error' if you prefer warnings) .

From now on, after running `npx expo lint`, the linter will also catch formatting inconsistent with Prettier. It's also good to add a **.prettierrc** file with your own Prettier configuration (e.g., if you want to use double quotes, semicolons, etc.). If we don't, Prettier will use its default settings.

In practice, having ESLint + Prettier, we can automatically format code on every file save (VSCode: “Format on save” option) and be sure that the entire team adheres to the same coding standards. This prevents many errors and maintains code cleanliness.

Running the Application: Emulator, Device, Logs, Fast Refresh

With the project and tools configured, let's focus on efficiently running and testing the application.

- **Android Emulator:** If you've configured Android Studio and an AVD, you can start the emulator and press the `a` key in the Expo Dev Tools. Expo will automatically locate the installed Android SDK and try to run the application on the emulator. The first time, this might take a moment – Expo will install the **Expo Go** app on the emulator (if it's not already installed) and connect to our Metro server. If everything goes well, you'll see the emulator window with the running application (it should display the Expo/Hello World welcome screen). It's a good idea to log into your Google account on the emulator and update Expo Go via the Play Store – this will facilitate future launches and ensure compatibility with the latest SDK.
- **iOS Simulator:** On macOS, launch the **iOS Simulator** (e.g., from the Xcode menu: Open Developer Tool -> Simulator). When it's active, press `i` in the Expo Dev Tools – the project should open in the simulator. Expo will automatically build the JS **bundle** and display the application. If the simulator is freshly installed, Expo CLI might ask to install the Expo client – confirm and wait. After a moment, the application should be running on the virtual iPhone.
- **Physical Device:** Alternatively, you can test directly on your phone/tablet. Make sure the device and computer are on the same WiFi network. Run `expo start` and scan the displayed QR code with your camera (iOS) or via the **Scan QR Code** option in Expo Go (Android). The Expo Go app will connect to your server and run the project. This

method is especially appreciated when we want to test features requiring physical hardware (e.g., camera, sensors) – Expo Go provides access to the device's API without additional configuration.

- **Application Logs:** When running in development mode, Expo listens for logs from the application. Every `console.log` in the RN code will appear in the Dev Tools console (terminal). Runtime errors will also be displayed there (along with a stack trace). In development mode, RN also displays an **error overlay** on the application screen, making debugging easier. You can also open the developer menu and select **Remote JS Debugging** – this will connect to a debugger (by default in the Chrome browser), where you can use the devtools (console, debugger). However, since Expo SDK 48+, it's recommended to use the new **Expo Developer Tools** debugger or Flipper. But for starters, the console and its messages should be sufficient. If you need to display native logs (e.g., Android's logcat or Xcode's console), Expo CLI provides the `expo run:android`/`expo run:ios` commands after a prebuild, but for pure Expo, this is rarely needed – most things are logged in Metro.
- **Developer Menu:** On both the emulator and a physical device, you can open the so-called **Developer Menu**. It contains useful options (Reload, enabling/disabling Fast Refresh, debug, performance monitor, etc.). To invoke it:
    - **Android:** Shake the device or (emulator) use the shortcut **Ctrl+M**.
    - **iOS:** Shake the device or (simulator) use **Cmd+D**.
    - This menu allows you, among other things, to toggle **Fast Refresh** mode. Fast Refresh should be enabled by default – this means that any code change in the editor will cause an automatic reload of that part of the application on the device within ~1 second, preserving component state (if possible). If changes don't appear, check the menu to ensure **Enable Fast Refresh** is active (if you see **Disable Fast Refresh**, it means it's already on).
- **Fast Refresh** significantly speeds up iteration – after saving a file, you immediately see the effect in the application, without needing to manually refresh. In the developer menu, you will also find the **Reload** option (reloads the entire application), **Debug Remote JS** (mentioned earlier), **Performance Monitor** (displays FPS and CPU/GPU usage, useful for optimization), and others. Familiarize yourself with them, as this is your "toolbox" during RN development.
- **Hot Reload vs. App State:** It's worth adding that Fast Refresh in new versions of RN tries to preserve component state during a refresh. This means if you have a component with a `useState` hook and only change the markup, the state will not reset. However, this isn't always possible (e.g., changing the component structure might cause a full reload). When you need a full application restart, you can use the Reload option from the menu or simply stop and restart `expo start`.

Demo: "Hello, RN" Screen – Component, SafeAreaView, First Styles

Now that our application is running, let's make our first changes – we'll create a simple **“Hello, RN”** screen. This will show how to define components in React Native, use SafeAreaView, and apply styles.

Open the App.js (or App.tsx) file. By default, you'll see something like:

JavaScript
```
<View style={styles.container}>
  <Text>Open up App.js to start working on your app!</Text>
  <StatusBar style="auto" />
</View>
```

Instead, let's modify the code to display “Hello, RN”. We'll also use the **SafeAreaView** component at the top of the hierarchy.

SafeAreaView is a component provided by RN (on iOS) or the **react-native-safe-area-context** library (Expo includes it), which works similarly to a regular View but automatically adds appropriate **paddings** so that the content is placed within the so-called **safe area** of the screen. In other words, it protects content from overlapping with the notch, status bar, or rounded edges.

It's recommended to wrap the main screen content in SafeAreaView, so that on new iPhones, our text doesn't hide under the notch, and on Android, it isn't covered by the status bar.

We will also change the component to a **functional** one using TypeScript and define it as type React.FC. Here is an example implementation:

TypeScript
```
import React from 'react';
import { SafeAreaView, Text, StyleSheet } from 'react-native';

export const App: React.FC = () => {
  return (
    <SafeAreaView style={styles.container}>
      <Text style={styles.text}>Hello, RN!</Text>
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  text: {
    fontSize: 24,
  },
});
```

A few explanations for the code above:

- We used SafeAreaView instead of a regular View as the top-level container. This ensures that on devices with a notch, the content will be automatically offset from the screen edges. (In Expo, the SafeAreaView component comes from the react-native-safe-area-context library, which works on both iOS and Android – we don't need to install it additionally, as Expo bundles it) .
- Inside, we placed a Text component with the text "Hello, RN!". For readability, we added the styles.text style, which sets a larger font size (24).
- Our App component was declared as React.FC (React Functional Component). This is not mandatory, but helpful in TypeScript – it specifies that this is a React component with correctly defined props/state types (in this case, without props).
- We used a named export (export const App...), although export default function App() { ... } would work just as well. In Expo, if we use a named export, we should still have an indication in package.json of which file is the main one (by default, Expo expects App.(j|t)sx). In our example, we'll stick with the named export for demonstration.
- We used **StyleSheet.create** to define styles. StyleSheet in RN is a built-in API that helps create styles similarly to CSS, but written as JS objects.
- The container style sets flex: 1 (meaning the container fills the entire height – this is important for SafeAreaView to work correctly), a white background, and centers the elements (alignItems, justifyContent set to 'center' – similar to CSS Flexbox).
- The text style only sets the font size. In RN, we define styles using JavaScript objects, where attribute names are camelCase instead of kebab-case (e.g., backgroundColor instead of background-color). Most properties align with CSS (e.g., color, fontSize, margin, padding, etc.), which makes learning styling in RN easier. We can create styles via StyleSheet (which may offer optimizations) or inline as plain objects (e.g., <Text style={{ fontSize: 24 }}>).
- Make sure you import SafeAreaView from the correct module. In pure RN, it's in react-native (works only on iOS 11+), whereas in Expo, it's recommended to import from react-native-safe-area-context for full cross-platform compatibility. In the code above, for simplicity, we import from react-native – in newer versions of Expo, this will be substituted with the implementation from safe-area-context anyway.

Save the changes and watch **Fast Refresh** in action – the application should update automatically. On the screen, you will see a white background with the text "Hello, RN!" in the center. If you are testing on an iPhone with a notch, you'll notice the text doesn't stick to the top edge – this is the effect of SafeAreaView (the content is in the safe area).

Congratulations, you've just implemented your first screen in React Native! □ This is a simple example, but it contains the fundamentals: a functional component, styling, and using the special SafeAreaView to protect against notches/status bars.

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.