# **Create Feature Flags For Your React App with Firebase Remote Config**

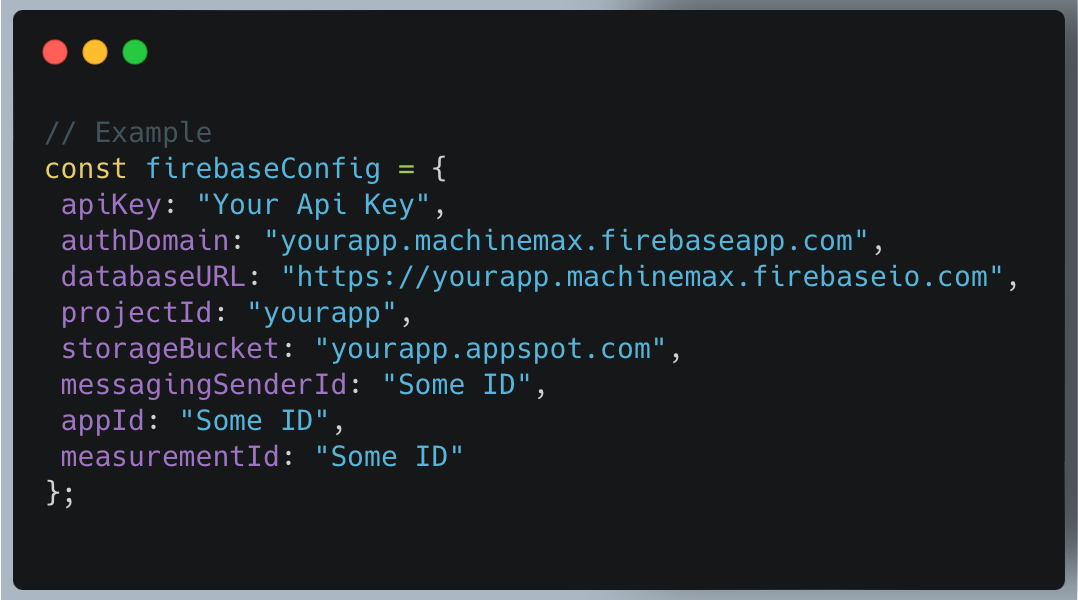
As the name suggests, feature flags are used to turn certain features on and off in your application. You can add feature flags for A/B testing by enabling the beta feature for some users. You can use them as fail-safe switches in case something goes wrong in production. Or you can use them to hide your work-in-progress functionality. Here I'll show you how to implement feature flags using Firebase Remote Config and React.

# **1. Initialize Firebase 🔥**

We need to install 2 packages for this demo. If you have already setup Firebase in your project, jump to the next step.

yarn add firebase

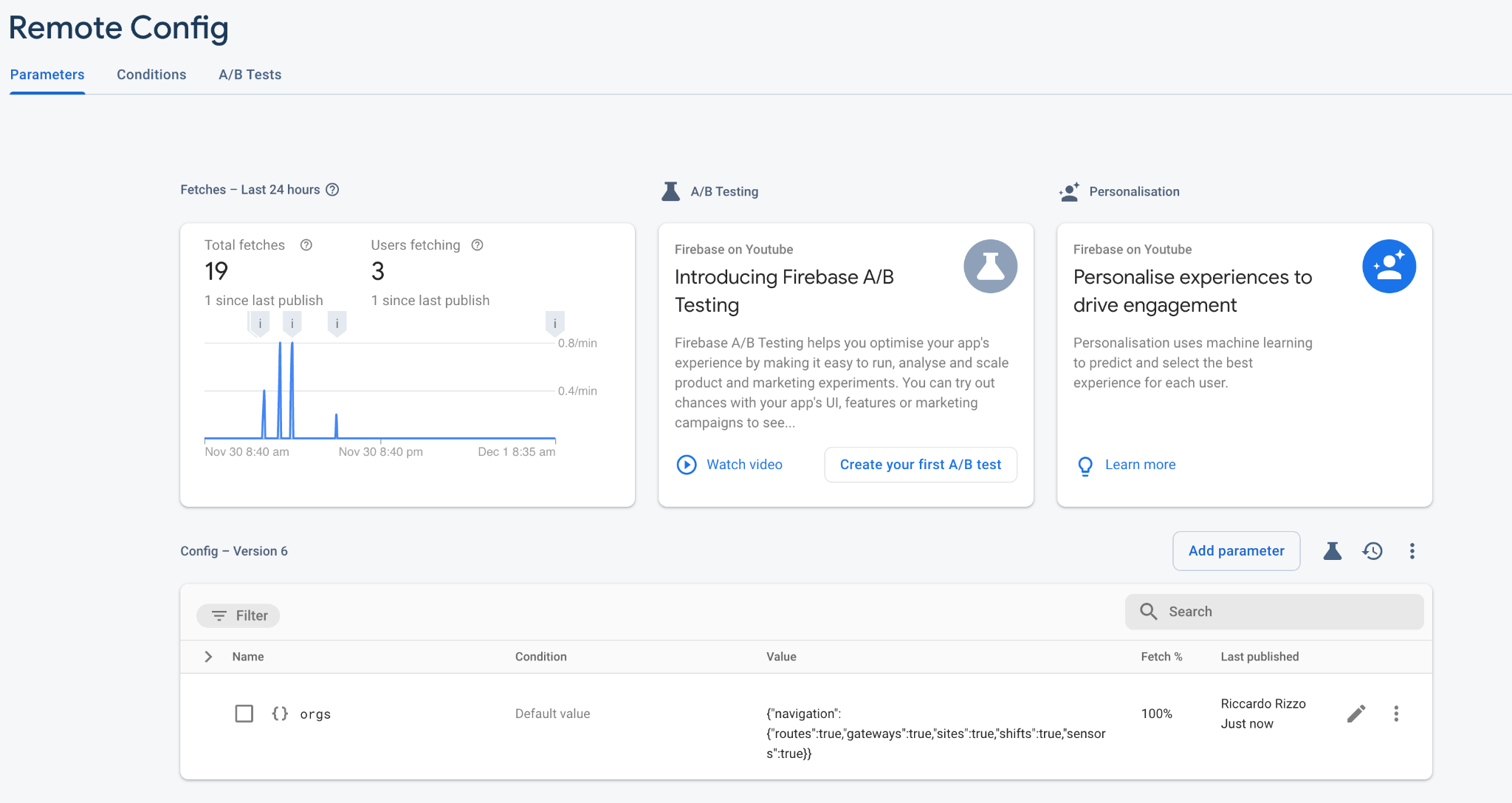
Then, create a file firebase.tsx to initialise the Firebase instance along with other Firebase services you need, which would be remote-config in our case.Head over to your [Firebase Console](https://console.firebase.google.com/) and create a new project if you don’t have one. Then go to the Project Settings page to get your Firebase SDK configs and paste it into firebase.tsx .



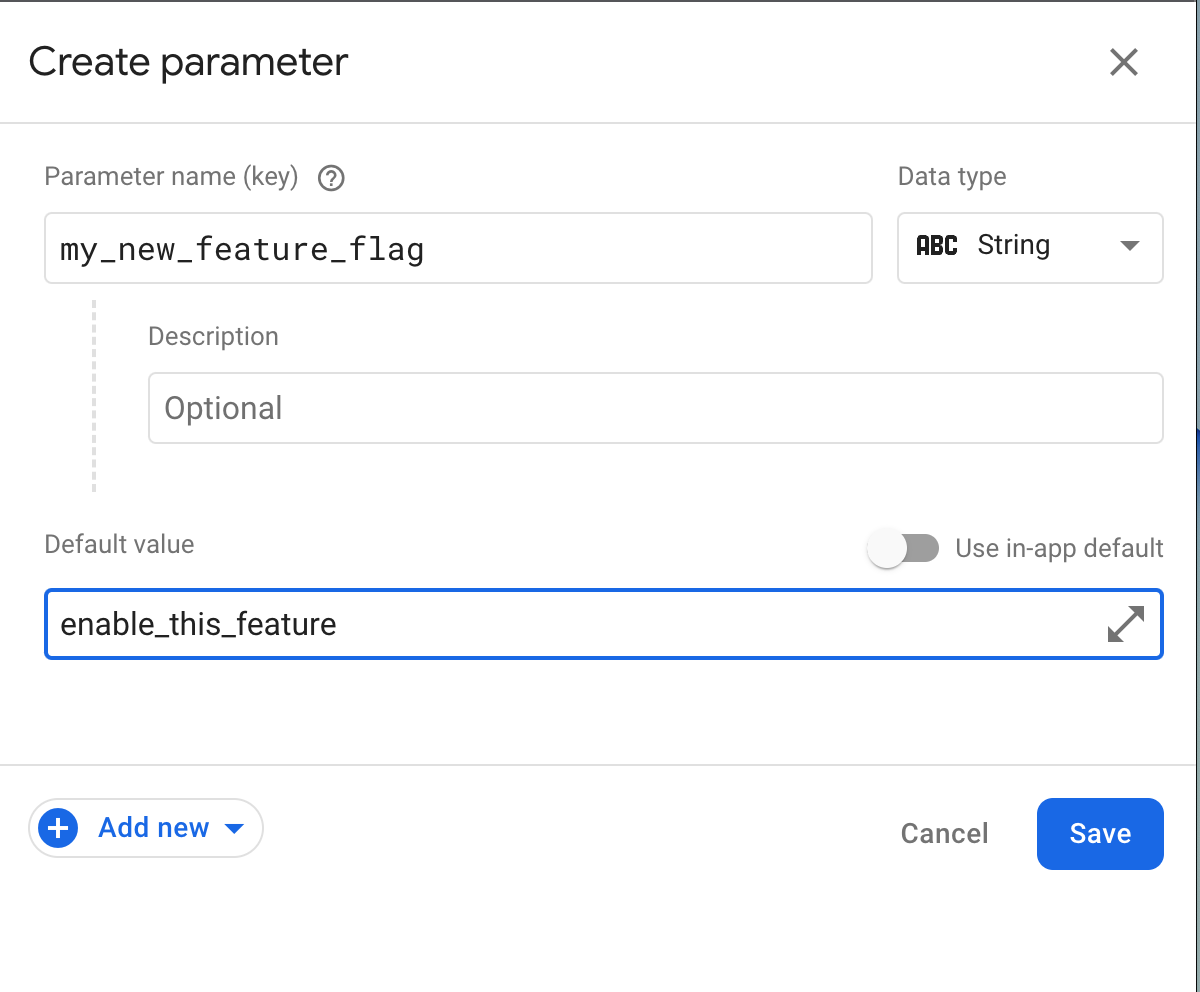
With the firebase instance initialised, we can pass it around and invoke all the available services.

# **2. Create Remote Flags in Firebase 🚩**

In your Firebase console, go to the Remote Config section. You can find it under the "Grow" section on the left menu. If this is your first time setting up, you should see the following. Enter an argument key, which will be our feature flag name, and its default value, which will be a semantic versioning string ^1.0.0.



Click **Add parameter** and you should reach this page.



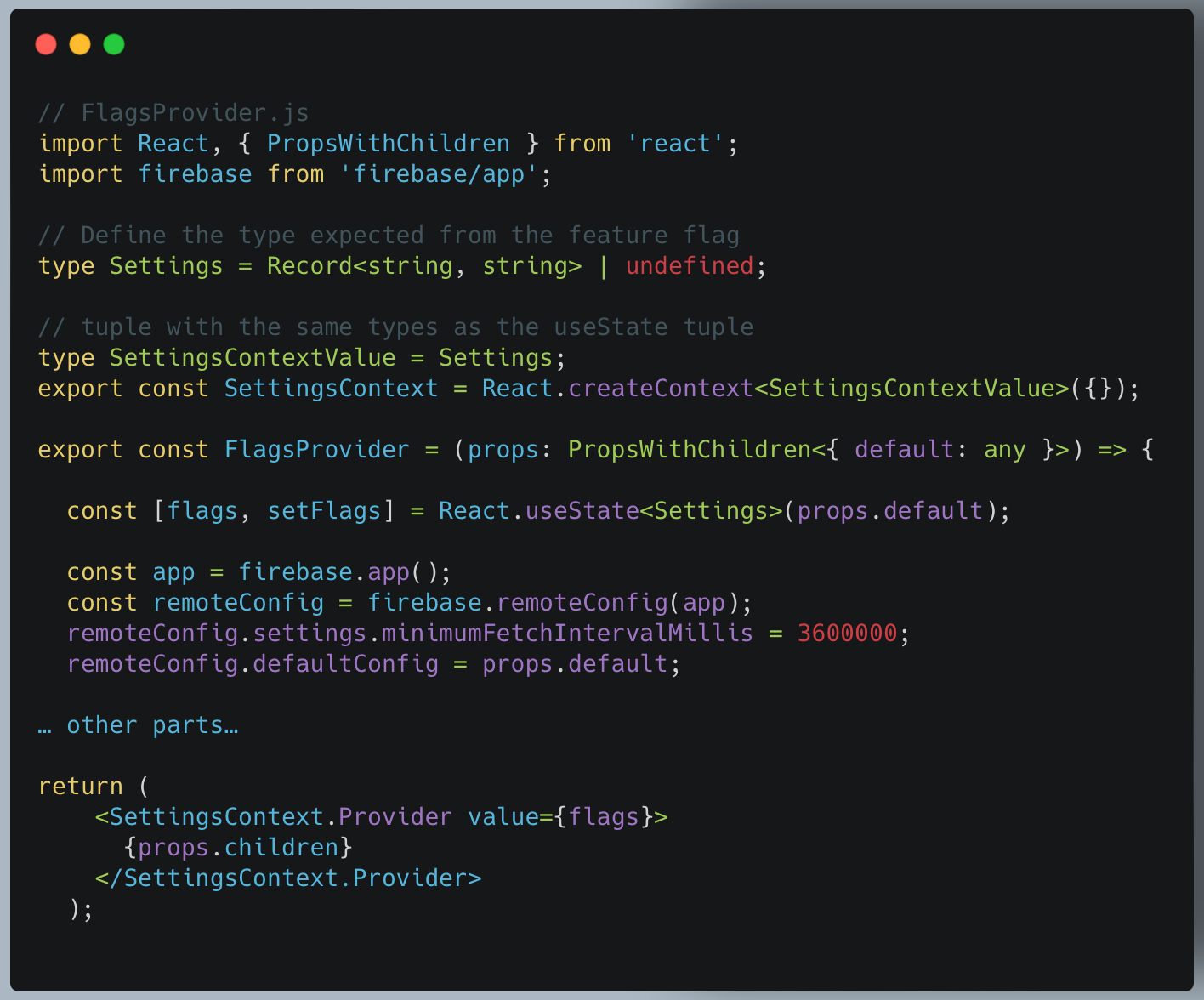
Add your parameter name and the default value and then click **Publish changes** and you’ll have your first feature flag ready to use!

# **3. Create FlagsProvider Component**

In order to share our feature flags across your app, we need to create a **FlagsProvider** which fetches all the remote flags we created above and share them through the Context API.

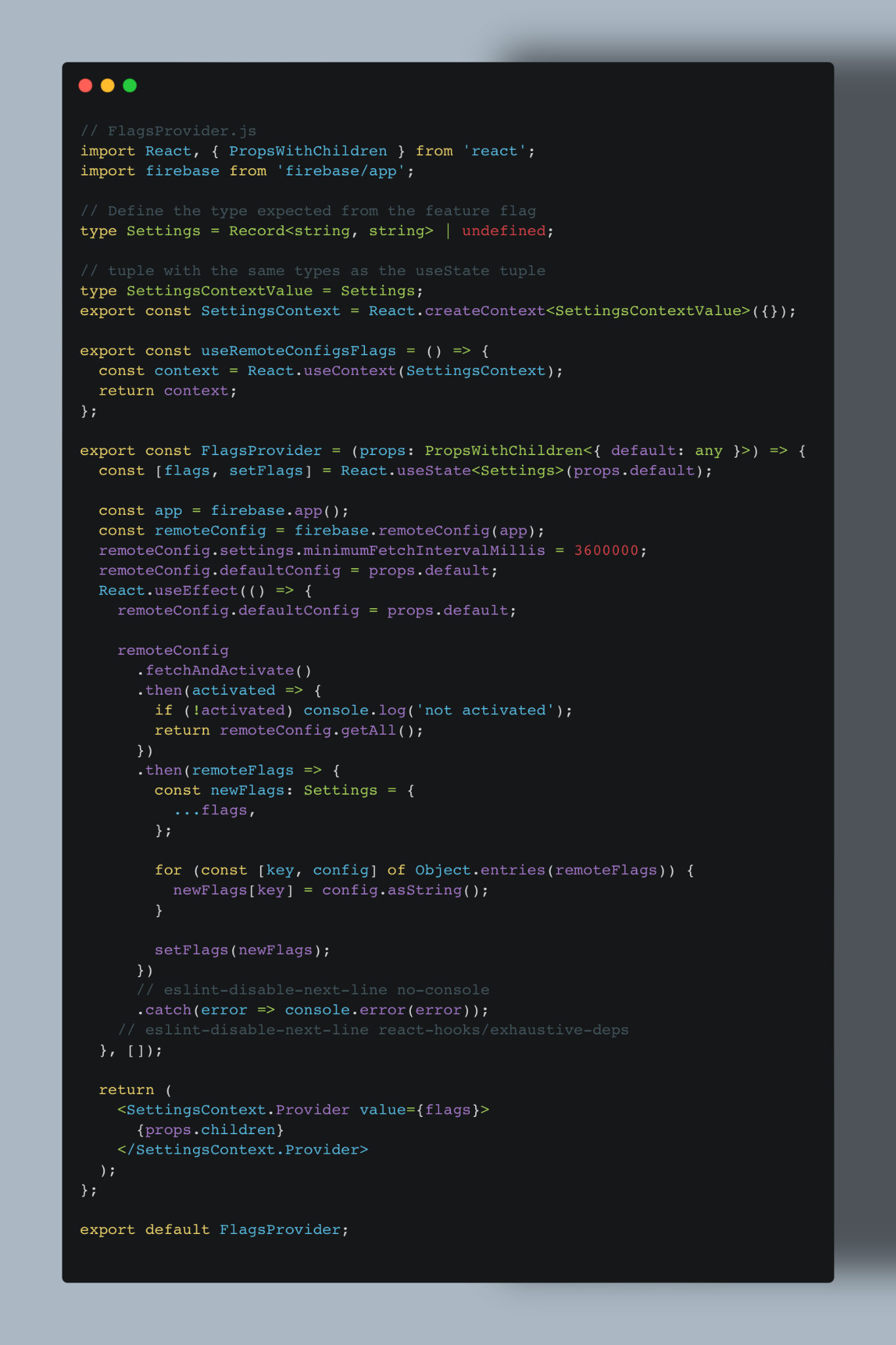
First, create a FlagsContext and get the remoteConfig instance. Since all the fetched remote configs will be cached in your browser’s IndexDB, we can adjust the maximum age of the cache in development, i.e. minimumFetchIntervalMillis . Obviously, you can also manually delete the cache yourself.

Our provider component should take in a default prop for setting the default values of your feature flags. We’ll save them in a flags state and pass it to our Provider below, so we can trigger a re-render when we set new flags.

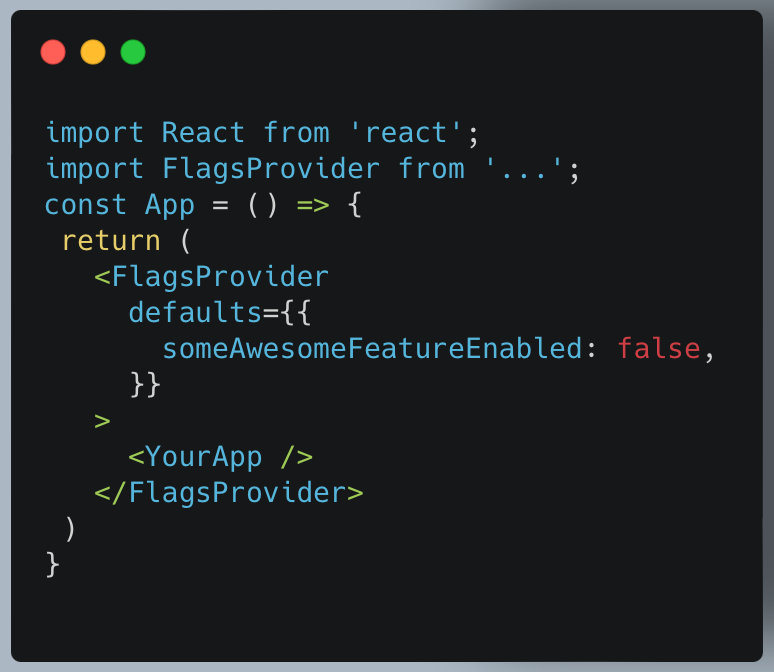


Next, we’ll add an useEffect hook. We need to assign our default prop as the default config. Then we fetch and activate all the remote flags. We would then loop through the remote flags and set our flags state.



The final version should be like this:

# **4. Wrap our App with FlagsProvider**

Finally, wrap the FlagsProvider to your root App and specify the defaults prop. The flag names **MUST** match the ones you set in your Firebase Console. Or else it won’t be able to fetch the corresponding flags.

# **How to Access your Flags**

To make our life easier, we are going to create a custom hook for retrieving your flags directly from your FlagsContext .



We are now able to use our feature flag directly from any components in our application.



# **All Done!**

This can be a developmental change experience. You can keep developing new features, and once they're ready, you can turn on the flag instead of redeploying the app. If you want to add more flags, just add them to Firebase and default props. Remote config can also be done in React Native. The API will be slightly different, but the concepts are the same.