{callstack}

The Ultimate Guide to React Native Optimization

2023 Edition

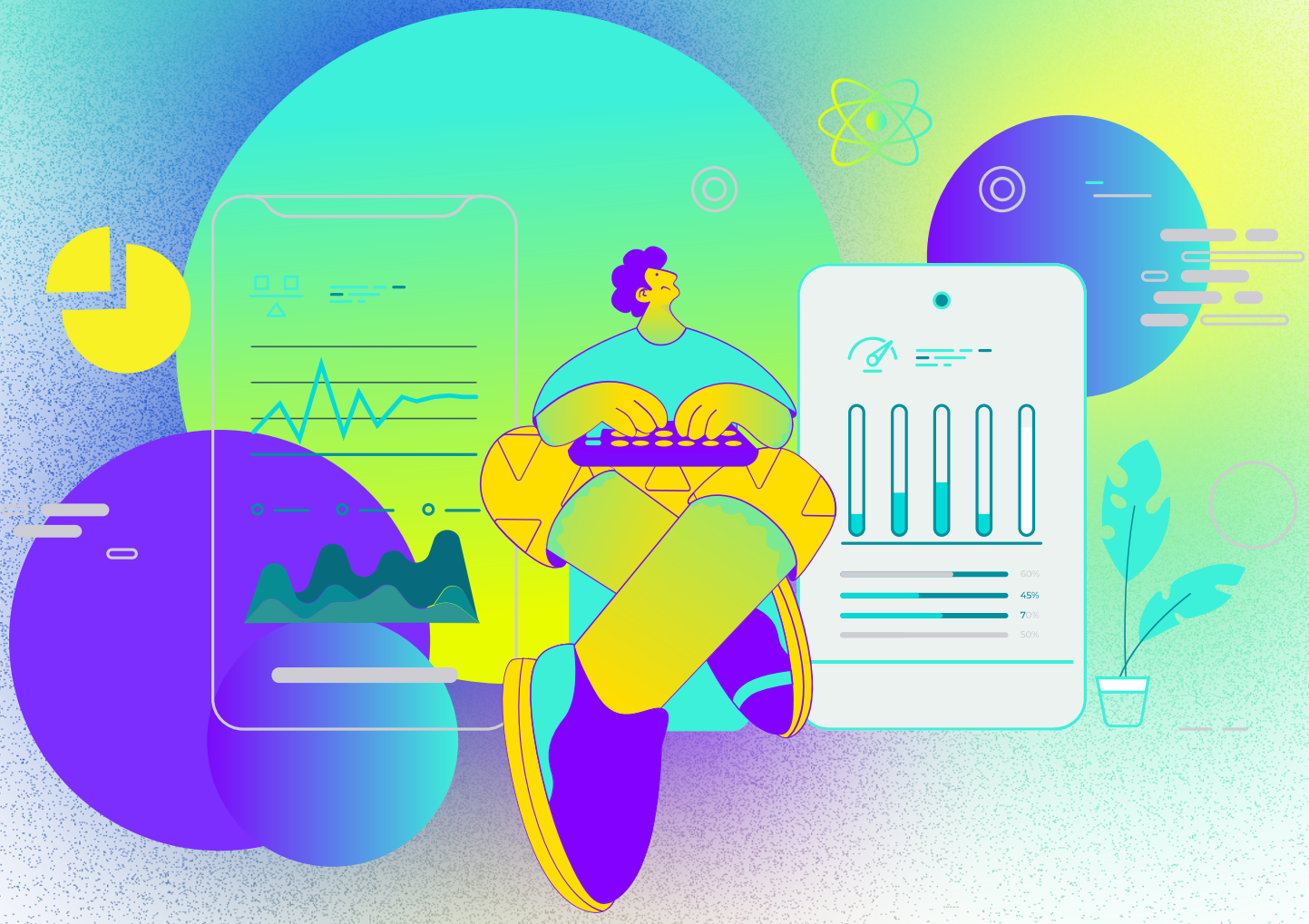


Table of Contents

How This Guide Is Organized 3
Introduction to React Native Optimization 6

Part one

Pay attention to UI re-renders 10
Use dedicated components for certain layouts 26
Think twice before you pick an external library 36
Always remember to use libraries dedicated to the mobile platform 42
Find the balance between native and JavaScript 48
Animate at 60FPS - no matter what 55
Replace Lottie with Rive 66
Optimize your app's JavaScript bundle 75

Part two

Always run the latest React Native version to access the new features 82
How to debug faster and better with Flipper 93
Avoid unused native dependencies 99
Optimize your application startup time with Hermes 105
Optimize your Android application's size with these Gradle settings 113
Experiment with the New Architecture of React Native 120

Part Three

Run tests for key pieces of your app 132
Have a working Continuous Integration (CI) in place 143
Don't be afraid to ship fast with Continuous Deployment 153
Ship OTA (Over-The-Air) when in an emergency 167
Make your app consistently fast 174
Know how to profile iOS 188
Know how to profile Android 196

Thank you 207
Authors 208

How This Guide Is Organized

Optimizing the React Native app is a complex process where you need to take various aspects into account – from implementation through using the latest React Native features to testing and continuous deployment.

This guide is a comprehensive source of tactics, tricks, tips, tools, and best practices to help you deliver an optimized React Native app.

We not only focus on the technological aspects of React Native optimization. We also underline the impact of each technological aspect on business continuity.

This guide contains best practices for optimizing:

- Stability
- Performance
- Resource usage
- User experience
- Maintenance costs
- Time-to-market

All these aforementioned aspects have a particular impact on the revenue-generating effectiveness of your apps. Such elements as stability, performance, and resource usage are directly related to improving the ROI of your products because of their positive impact on the user experience.

With a faster time-to-market, you can stay ahead of your competitors, whereas an easier and quicker maintenance process will help you reduce your spending on that particular process.

What this guide looks like and the topics it covers

This guide is divided into three parts:

The first part is about improving performance through understanding the React Native implementation details and knowing how to maximize them. This part covers the following topics:

1. Pay attention to UI re-renders
2. Use dedicated components for certain layouts
3. Think twice before you pick an external library
4. Always remember to use libraries dedicated to the mobile platform
5. Find the balance between native and JavaScript
6. Animate at 60FPS - no matter what
7. Replace Lottie with Rive
8. Optimize your app's JavaScript bundle

The second part is about improving performance by using the latest React Native features or turning some of them on. This part describes the following topics:

1. Always run the latest React Native version to access the latest features
2. How to debug faster and better with Flipper
3. Avoid unused native dependencies
4. Optimize your Android application startup time with Hermes
5. Optimize your Android application's size with Gradle settings
6. Experiment with the New Architecture of React Native

The third part is about enhancing the stability of the application by investing in testing and continuous deployment. This part tackles the following topics:

1. Run tests for key pieces of your app
2. Have a working Continuous Integration (CI) in place
3. Don’t be afraid to ship fast with Continuous Deployment
4. Ship OTA (Over-the-Air) in an emergency
5. Make your app consistently fast
6. Know how to profile iOS
7. Know how to profile Android

The structure of each section looks like this:

Issue

This part describes the main problem with React Native performance.

Solution

This part outlines how that problem may affect your business and what the best practices are to solve it.

Benefits

This part focuses on the business benefits of our proposed solution.

Introduction to React Native Optimization

React Native takes care of the rendering. But performance is still key.

With React Native, you create components that describe how your interface should look like. During the runtime, React Native turns them into platform-specific native components. Rather than talking directly to the underlying APIs, you focus on the user experience of your application.

However, that doesn't mean all the applications developed with React Native are equally fast and offer the same level of user experience.

Every declarative approach (including React Native) is built with imperative APIs. And you have to be careful when doing things imperatively.

When you're building your application the imperative way, you carefully analyze every callsite to the external APIs. For example, when working in a multi-threaded environment, you safely write your code in a thread, being aware of the context and resources that the code is looking for.

Despite all the differences between the declarative and imperative ways of doing things, they have a lot in common. Every declarative abstraction can be broken down into a number of imperative calls. For example, React Native uses the same APIs to render your application on iOS as native developers would use themselves.

React Native unifies performance but doesn't make it fast out of the box!

While you don't have to worry about the performance of the underlying iOS and Android APIs calls, how you compose the components can make all the difference. All your components will offer the same level of performance and responsiveness.

But is “same” a synonym for “best”? It’s not.

That’s when our checklist comes into play. Use React Native to its full potential. As discussed before, React Native is a declarative framework and takes care of rendering the application for you. In other words, you don’t dictate how the application will be rendered.

Your job is to define the UI components and forget about the rest. However, that doesn’t mean that you should take the performance of your application for granted. In order to create fast and responsive applications, you have to think the React Native way. You have to understand how the framework interacts with the underlying platform APIs.

If you need help with performance, stability, user experience, or other complex issues - contact us!

As React Native Core Contributors and leaders of the community, we will be happy to help.

PART 1

Improve performance by understanding the details of React Native implementation.

In this section, we will dive deeper into the most popular performance bottlenecks and the React Native implementation details that contribute to them. This will not only be a smooth introduction to some of the advanced React Native concepts, but it will also let you improve the stability and performance of your application by performing small tweaks and changes.

The following part is focused on the first point from the checklist of performance optimization tactics: UI re-renders. It's a very important part of the React Native optimization process because it allows for the reduction of the device's battery usage which translates into a better user experience for your app.

PART 1 | CHAPTER 1

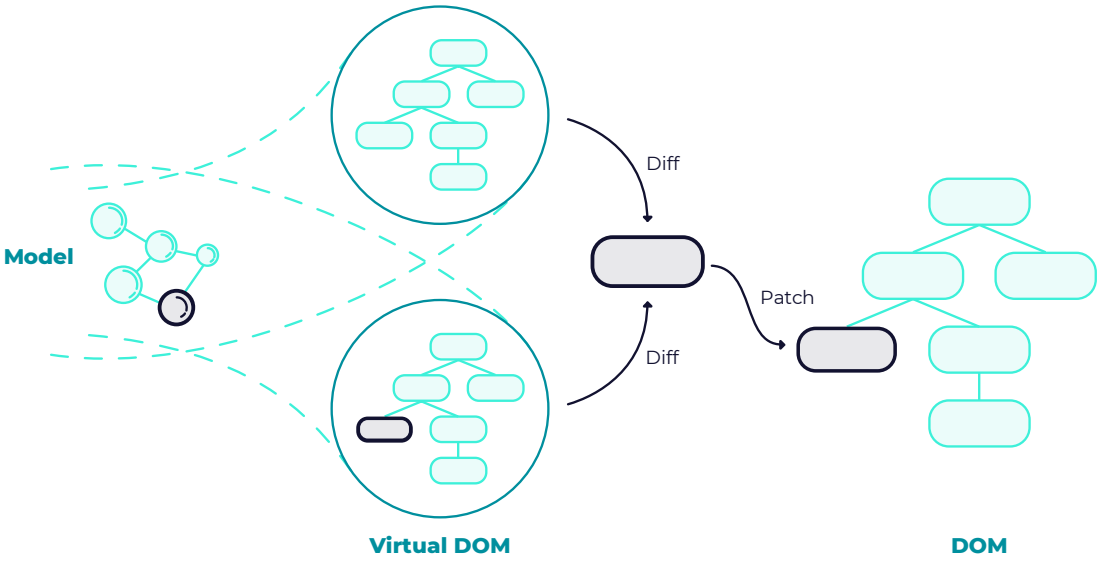
Pay attention to UI re-renders

Optimize the number of state operations and remember about memoized components to make your app work faster with fewer resources.

Issue: Incorrect state updates cause extraneous rendering cycles or the device is just too slow.

As discussed briefly, React Native takes care of rendering the application for you. You have to define all the components you need and compose the final interface out of these smaller building blocks. In that approach, you don't control the application rendering lifecycle.

In other words, when and how to repaint things on screen is purely React Native's responsibility. React looks out for the changes you have done to your components, compares them, and, by design, only performs the required and smallest number of actual updates.



By default, a component can re-render if its parent is re-rendering or the props are different. This means that your component's render method can sometimes run, even if their props didn't change. This is an acceptable tradeoff in most scenarios, as comparing the two objects (the previous and current props) would take longer.

Negative impact on performance, UI flicker, and FPS decrease

While the above heuristics are correct most of the time, performing too many operations can cause performance problems, especially on low-end mobile devices.

As a result, you may observe your UI flickering (when the updates are performed) or frames dropping (while there's an animation happening and an update is coming along).

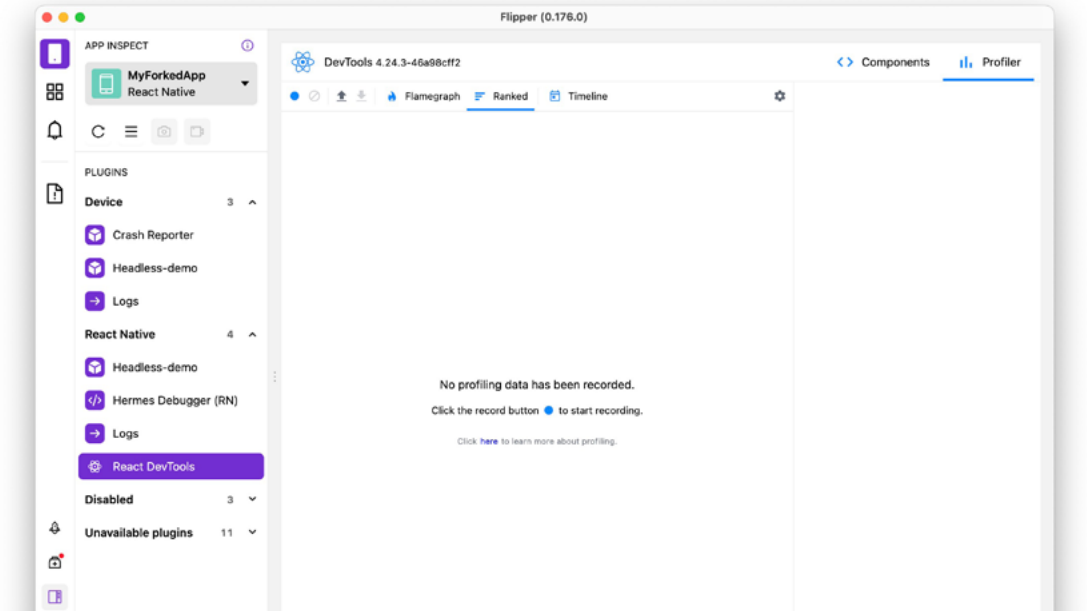
Note: Performing premature optimizations may have a counter-positive effect. Try looking at performance issues as soon as you spot dropped frames or undesired performance within your app.

As soon as you see any of these symptoms, it is the right time to look a bit deeper into your application lifecycle and look for extraneous operations that you would not expect to happen.

How do we know what to optimize?

When it comes to performance optimization, we want to make decisions based on data. The data comes from measuring performance using specialized tools. The process is often referred to as profiling. There are many tools available that can help us with profiling our React Native apps: react-devtools, why-did-you-render, Profiler, and others. For this exercise, we'll use Flipper, a platform for debugging iOS, Android, and React Native apps. It has React DevTools Profiler integrated as a plugin that can produce a flame graph of the React rendering pipeline as a result of profiling. We can leverage this data to measure

the re-rendering issues of the app. Once you've downloaded Flipper, select React DevTools:



Here is the code we're about to profile:

```
import React, { useEffect, useState } from 'react';
import { View } from 'react-native';

const App = () => {
  const [value, setValue] = useState('');
  const backgroundStyle = {
    backgroundColor: '#fff',
    flex: 1,
    marginTop: 80,
  };

  useEffect(() => {
    setTimeout(() => {
      setValue('update 1');
    }, 3000);
    setTimeout(() => {
      setValue('update 2');
    }, 5000);
  }, []);

  return (
    <View style={backgroundStyle}>
      <ColoredView />
    </View>
  );
};
```

```
const ColoredView = () => {
  const style = { backgroundColor: 'red', padding: 10 };

  return <View style={style} />;
};

export default App;
```

In the Flipper app, make sure the "Record why each component rendered while profiling" option is enabled in the settings icon and hit the round blue button to start profiling. After around 5 seconds, hit the round red button to stop profiling. Your profile will look something like this:

This profile shows us how much time a certain component took to render, how many times it re-rendered, and what was the cause of it. In our case, `ColoredView` rendered 2 times due to the parent component being re-rendered. This might give us a hint that there's an unexpected performance implication of the code associated with `ColoredView`. Using this knowledge, we can apply tailored solutions to avoid the extra re-renders.

Taking a look at the performance flame graph for the first time may be slightly intimidating. To understand React DevTools more in-depth, this video from Ben Awad is good at explaining it. Don't forget to watch this talk by Alex at React Native EU, which explains how we can use flame graph to identify and fix the issues. Also, visit the official react website for detailed information on React Profiler.

Solution: Optimize the number of state operations and remember to use memoized components when needed.

There're a lot of ways your application can turn into unnecessary rendering cycles and that point itself is worth a separate article. Here, we will focus on two common scenarios - using a controlled component, such as TextInput and global state.

Controlled vs uncontrolled components

Let's start with the first one. Almost every React Native application contains at least one TextInput that is controlled by the component state as per the following snippet:

```
import React, { useState } from 'react';
import { TextInput, StyleSheet } from 'react-native';

const UselessTextInput = () => {
  const [value, setValue] = useState('Text');

  const onChangeText = (text) => {
    setValue(text);
  };

  return (
    <TextInput
      accessibilityLabel="Text input field"
      style={styles.textInput}
      onChangeText={onChangeText}
      value={value}
    />
  );
};

const styles = StyleSheet.create({
  textInput: {
    height: 40,
    borderColor: 'gray',
    borderWidth: 1,
  },
});

export default UselessTextInput;
```

Read more: https://snack.expo.dev/@callstack-snack/textinput-example

The above code sample will work in most cases. However, on slow devices, and in situations where the user is typing really fast, it may cause a problem with the view updates.

This problem is caused by React Native's asynchronous nature. To better understand what is going on here, let's first take a look at the order of standard operations that occur while the user is typing and populating your `<TextInput />` with new characters.

Diagram that shows what happens while typing TEST

As soon as the user starts inputting a new character into the native input, an update is sent to React Native via the `onChangeText` prop (operation 1 on the above diagram). React processes that information and updates its state accordingly by calling `setState`. Next, a controlled component synchronizes its JavaScript value with the native component value (operation 2 on the above diagram).

There are benefits to such an approach. React is a source of truth that dictates the value of your inputs. This technique lets you alter the user input as it happens, by e.g. performing a validation, masking it, or completely modifying it.
Unfortunately, the aforementioned approach, while ultimately cleaner and more compliant with the way React works, has one downside and it is most noticeable when there are limited resources available and/or the user is typing at a very high rate.

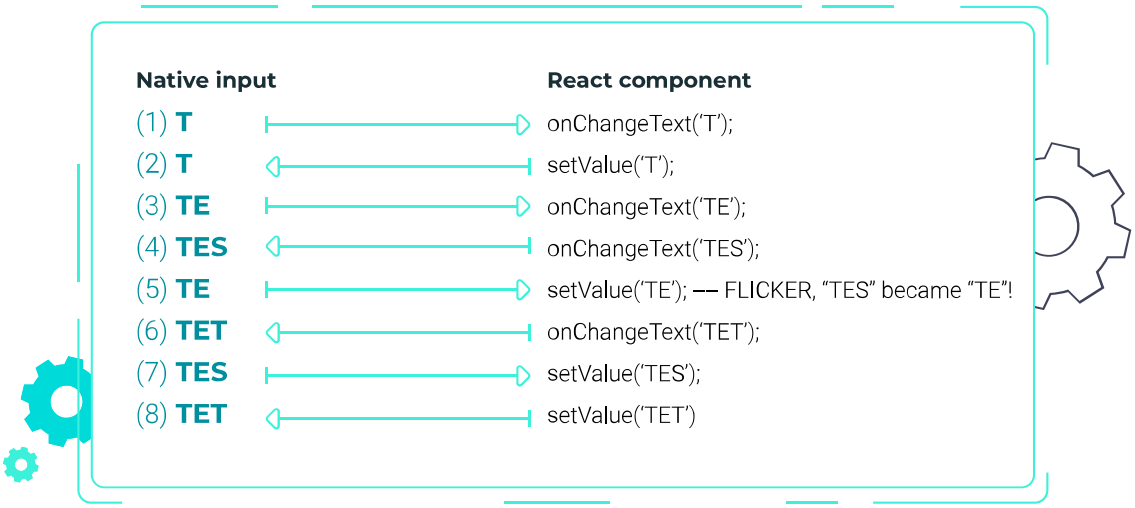


Diagram that shows what happens while typing TEST too fast

When the updates via `onChangeText` arrive before React Native synchronized each of them back, the interface will start flickering. The first update (operation 1 and operation 2) performs without issues as the user starts typing T.

Next, operation 3 arrives, followed by operation 4. The user typed E & S while React Native was busy doing something else, delaying the synchronization of the letter E (operation 5). As a result, the native input will temporarily change its value back from TES to TE.

Now, the user was typing fast enough to actually enter another character when the value of the text input was set to TE for a second. As a result, another update arrived (operation 6), with the value of TET. This wasn't intentional - the user wasn't expecting the value of its input to change from TES to TE.

Finally, operation 7 synchronized the input back to the correct input received from the user a few characters before (operation 4 informed us about TES). Unfortunately, it was quickly overwritten by another update (operation 8), which synchronized the value to TET - the final value of the input.

The root cause of this situation lies in the order of operations. If operation 5 was executed before operation 4, things would have run smoothly. Also, if the user didn't type T when the value was TE instead of TES, the interface would flicker but the input value would remain correct.

One of the solutions for the synchronization problem is to remove the value prop from TextInput entirely. As a result, the data will flow only one way, from the native to the JavaScript side, eliminating the synchronization issues that were described earlier.

```
import React, { useState } from 'react';
import { Text, TextInput, View, StyleSheet } from 'react-
native';

const PizzaTranslator = () => {
  const [value, setValue] = useState('');

  const onChangeText = (text) => {
    setValue(text);
  };

  return (
    <View style={styles.container}>
      <TextInput
        accessibilityLabel="Text input field"
        placeholder="Type here to translate!"
        onChangeText={onChangeText}
        defaultValue={value}
        style={styles.textInput}
      />
      <Text style={styles.label}>
        {value
          .split(' ')
          .map((word) => word && '🍕')
          .join(' ')}
      </Text>
    </View>
  );
};

export default PizzaTranslator;
const styles = StyleSheet.create({
  container: {
    padding: 10,
  },
  textInput: {
    height: 40,
```

```
    },
    label: {
      padding: 10,
      fontSize: 42,
    },
  },
});
```

Read more: https://snack.expo.dev/@callstack-snack/handling-text-input

However, as pointed out by @nparashuram in his YouTube video (which is a great resource to learn more about React Native performance), that workaround alone isn't enough in some cases. For example, when performing an input validation or masking, you still need to control the data that the user is typing and alter what ends up being displayed within TextInput.

Global state

Another common reason for performance issues is how components are dependent on the application's global state. The worst case scenario is when the state change of a single control like `TextInput` or `CheckBox` propagates the render of the whole application. The reason for this is a bad global state management design.

First, your state management library should take care of updating components only when a defined subset of data had changed - this is the default behavior of the redux connect function.

Second, if your component uses data in a different shape than what is stored in your state, it may re-render, even if there is no real data change. To avoid this situation, you can implement a selector that would memorize the result of the derivation until the set of passed dependencies changes.

```
      import { createSelector } from 'reselect';

const getVisibilityFilter = (state) => state.visibilityFilter;
const getTodos = (state) => state.todos;

const getVisibleTodos = createSelector(
  [getVisibilityFilter, getTodos],
  (visibilityFilter, todos) => {
    switch (visibilityFilter) {
      case 'SHOW_ALL':
        return todos;
     case 'SHOW_COMPLETED':
        return todos.filter((t) => t.completed);
      case 'SHOW_ACTIVE':
        return todos.filter((t) => !t.completed);
      default:
        return todos;
    }
  },
);

const mapStateToProps = (state) => ({
  todos: getVisibleTodos(state),
});

const VisibleTodoList = connect(mapStateToProps)(TodoList);

export default VisibleTodoList;
```

A typical example of selectors with edux state management library

A common bad performance practice is the belief that a state management library can be replaced by using a custom implementation based on `React Context`. It may be handy at the beginning because it reduces the boilerplate code that state management libraries introduce. But using it without proper memoization will lead to huge performance drawbacks. You will probably end up refactoring state management to `Redux`, because it will turn out that it is easier than the implementation of custom selectors to your current solution.

You can also optimize your application on a single component level. Using `React.memo` or `React.useMemo` will likely save you a lot of re-renders - the React Profiler can tell you precisely how many. Try not to implement these techniques in advance, because it may be premature optimization. In rare cases, memoization can lead to the app being less

performant due to increased memory usage. Which is impossible to measure with JS tooling. Always make sure to profile the “before” and “after” of your changes to have certainty it makes the app faster.

Atomic State

The pattern of a single store - a top-down mental model - as initially promoted by Redux is moving towards a more atomic design.

Why is it moving away from a single store?

- It’s often overkill for some apps.
- It’s too verbose.
- You end up using memoization techniques to avoid re-renders.
- Top-down is straightforward but leads to poor performance.
- More states are needed when the app grows, and each has its own problems and sub-states, such as handling local UI states (loadings, errors, messages, etc).

If you don’t pay attention to the Redux store, it tends to absorb all the states, leading to a monolithic structure that’s quite hard to reason with. The top-down build places most of the states at the top of the component; because of this, a state update from a parent component could produce re-renders to its children. Ideally, if possible, the state should be local to the component so it can be reused - this means building from the bottom-up.

To start thinking about this bottom-up pattern, we want to build from the smaller components, often called atoms. We don’t build the component starting from the parent (or “root” container element), we need to look at all the elements that make up the component. We then have to start from the atom and add the right state action, if needed.

Let’s say we have a TODO list with some filters: “show all”, “just the active ones”, or “just the completed ones”. We identify the top component, the `TodoList`, but we don’t start here. We first identify the children and start building those smaller components, so that later we can mix

them and build a complex element. We need to make some data visible in one component which will be managed by another one. To avoid a parent state and passing down the data and actions (top-down), we are going to use a state manager. This state manager will be in charge of storing the data, making it accessible, and providing actions/modifiers, because we are moving to a bottom-up approach. We are going to use some libraries that will help us.

Zustand

Zustand is often used as a top-down pattern but given the simplicity and unopinionated library, we can use it to build a component bottom-up.

Store

We create the obj where the filter will be and the modifiers will be exposed:

```
export const useHomeStore = create((set) => ({
  filter: filterType.all,
  showAll: () => set({ filter: filterType.all }),
  showOnlyCompleted: () => set({ filter: filterType.completed
}),
  showOnlyActive: () => set({ filter: filterType.active }),
}));
```

Show all filter button

Here we update the ticket and this item so any children won't suffer a re-render:

```
const ShowAllItem = () => {
  const showAll = useHomeStore((state) => state.showAll);
  return <Menu.Item onPress={showAll} title="Show All" />;
};
```

Todo item

Here because we are looking at a filter, `TodoItemList` will only re-render when it changes.

```
export const TodoItemList = ({ item }) => {
  const filter = useHomeStore((state) => state.filter);
  if (!shouldBeShown(filter, item.done)) {
    return null;
  }

  return (
    <View>
      <Text>{item.title}</Text>
      <Text>{item.description}</Text>
    </View>
  );
};
```

Jotai

Jotai was built with this bottom-up approach in mind, so its syntax is minimal and simple. Using the concept of atom as a “store” then you use different hooks to make the atom readonly or mutable.

Store

We use `useAtomValue` to read the filter value and `useSetAtom` to set a new value. This is especially useful when performance is a concern.

```
const filter = atom(filterType.all);

export const useCurrentFilter = () => useAtomValue(filter);
export const useUpdateFilter = () => useSetAtom(filter);
```

FilterMenuItem

```
const FilterMenuItem = ({ title, filterType }) => {
  const setUpdateFilter = useUpdateFilter();
  const handleShowAll = () => setUpdateFilter(filterType);

  return <Menu.Item onPress={handleShowAll} title={title} />;
};
```

TodoItem

```
export const TodoItemList = ({ item }) => {
  const filter = useCurrentFilter();

  if (!shouldBeShown(filter, item.done)) {
    return null;
  }

  return (
    <View>
      <Text>{item.title}</Text>
      <Text>{item.description}</Text>
    </View>
  );
};
```

Using this bottom-up approach, we can prevent state changes on the parent component, and produce re-renders to its children. In addition, it often leads to less overuse of memoization.

Benefits: Fewer resources needed and a faster application.

You should always keep the performance of your app in the back of your head. Beware of optimizing too fast. Some say that premature optimization is the root of all evil. They're not entirely correct, but also not entirely wrong. Premature memoization may lead to more memory usage and provide only a fraction of improvement compared to the effort taken. That's why measuring and identifying impactful problems to tackle is so important.

Most hard-to-solve performance issues are caused by bad architectural decisions around state management. Different libraries juggle different sets of tradeoffs. Make sure you and your team understand them and pick the tool you're most productive with. For some, it may be the verbosity of Redux. But others may prefer the ease of Zustand that avoids having to think about extra re-renders.

With all these steps in mind, your application should perform fewer operations and need smaller resources to complete its job. As a result, this should lead to lower battery usage and more satisfaction from interacting with the interface.

PART 1 | CHAPTER 2

Use dedicated components for certain layouts

Find out how to use dedicated higher-ordered React Native components to improve the user experience and the performance of your apps

Issue: You are unaware of the higher-order components that are provided with React Native.

In a React Native application, everything is a component. At the end of the component hierarchy, there are so-called primitive components, such as Text, View, or TextInput. These components are implemented by React Native and provided by the platform you are targeting to support the most basic user interactions.

When we're building our application, we compose it out of smaller building blocks. To do so, we use primitive components. For example, in order to create a login screen, we would use a series of TextInput components to register user details and a Touchable component to handle user interaction. This approach is true from the very first component that we create within our application and holds true through the final stage of its development.

On top of primitive components, React Native ships with a set of higher-order components that are designed and optimized to serve a certain purpose. Being unaware of them or not using them can potentially affect your application performance, especially as you populate your state with real production data. A bad performance of your app may seriously harm the user experience. In consequence, it can make your clients unsatisfied with your product and turn them towards your competitors.

Not using specialized components will affect your performance and UX as your data grows.

If you're not using specialized components, you are opting out of performance improvements and risking a degraded user experience when your application enters production. It is worth noting that certain issues remain unnoticed while the application is developed, as mocked data is usually small and doesn't reflect the size of a production database. Specialized components are more comprehensive and have a broader API to cover than the vast majority of mobile scenarios.

Solution: Always use a specialized component, e.g. FlatList for lists.

Let's take long lists as an example. Every application contains a list at some point. The fastest and dirtiest way to create a list of elements would be to combine `ScrollView` and `View` primitive components. However, such an example would quickly become problematic when the data grows. Dealing with large data-sets, infinite scrolling, and memory management was the motivation behind `FlatList` - a dedicated component in React Native for displaying and working with data structures like this.

Compare the performance of adding a new list element based on ScrollView

```
import React, { useCallback, useState } from 'react';
import { ScrollView, View, Text, Button, StyleSheet } from
'react-native';

const objects = [
  ['avocado', '🥑'],
  ['apple', '🍏'],
  ['orange', '🍊'],
  ['cactus', '🌵'],
  ['eggplant', '🍆'],
  ['strawberry', '🍓'],
  ['coconut', '🥥'],
];

const getRandomItem = () => {
  const item = objects[~~(Math.random() * objects.length)];

  return {
    name: item[0],
    icon: item[1],
```

```
    id: Date.now() + Math.random(),
  };
};

const _items = Array.from(new Array(5000)).map(getRandomItem);

const List = () => {
  const [items, setItems] = useState(_items);

  const addItem = useCallback(() => {
    setItems([getRandomItem()].concat(items));
  }, [items]);

  return (
    <View style={styles.container}>
      <Button title="add item" onPress={addItem} />
      <ScrollView>
        {items.map(({ name, icon, id }) => (
          <View style={styles.itemContainer} key={id}>
            <Text style={styles.name}>{name}</Text>
            <Text style={styles.icon}>{icon}</Text>
          </View>
        ))}
      </ScrollView>
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    marginTop: 30,
  },
  itemContainer: {
    borderWidth: 1,
    margin: 3,
    padding: 5,
    flexDirection: 'row',
  },
  name: {
    fontSize: 20,
    width: 150,
  },
  icon: {
    fontSize: 20,
  },
});

export default List;
```

Read more: https://snack.expo.dev/@callstack-snack/scrollview-example

to a list based on `FlatList`.

```
import React, { useCallback, useState } from 'react';
import { View, Text, Button, FlatList, StyleSheet } from
'react-native';

const objects = [
  'avocado 🥑',
  'apple 🍏',
  'orage 🍊',
  'cactus 🌵',
  'eggplant 🍆',
  'strawberry 🍓',
  'coconut 🥥',
];
const getRandomItem = () => {
  const item = objects[~~(Math.random() * objects.length)].
split(' ');

  return {
    name: item[0],
    icon: item[1],
    id: Date.now() + Math.random(),
  };
};

const _items = Array.from(new Array(5000)).map(getRandomItem);

const List = () => {
  const [items, setItems] = useState(_items);

  const addItem = useCallback(() => {
    setItems([getRandomItem()].concat(items));
  }, [items]);

  const keyExtractor = useCallback(({ id }) => id.toString(),
[]);

  const renderItem = useCallback(
    ({ item: { name, icon } }) => (
      <View style={styles.itemContainer}>
        <Text style={styles.name}>{name}</Text>
        <Text style={styles.icon}>{icon}</Text>
      </View>
    ),
    [],
  );

  return (
    <View style={styles.container}>
      <Button title="add item" onPress={addItem} />
      <FlatList
```

```
          data={items}
          keyExtractor={keyExtractor}
          renderItem={renderItem}
        />
      </View>
    );
};

const styles = StyleSheet.create({
  container: {
    marginTop: 30,
  },
  itemContainer: {
    borderWidth: 1,
    margin: 3,
    padding: 5,
    flexDirection: 'row',
  },
  name: {
    fontSize: 20,
    width: 150,
  },
  icon: {
    fontSize: 20,
  },
});

export default List;
```

Read more: https://snack.expo.dev/@callstack-snack/flatlist-example

The difference is significant, isn't it? In the provided example of 5000 list items, the `ScrollView` version does not even scroll smoothly.

At the end of the day, `FlatList` uses `ScrollView` and `View` components as well. What's the deal then?

Well, the key lies in the logic that is abstracted away within the `FlatList` component. It contains a lot of heuristics and advanced JavaScript calculations to reduce the amount of extraneous renderings that happen while you're displaying the data on screen and to make the scrolling experience always run at 60FPS. Just using `FlatList` may not be enough in some cases. `FlatList` performance optimizations rely on not rendering elements that are currently not displayed on the screen.

The most costly part of the process is layout measuring. `FlatList` has to measure your layout to determine how much space in the scroll area should be reserved for upcoming elements.

For complex list elements, it may slow down the interaction with `FlatList` significantly. Every time `FlatList` approaches to render the next batch of data, it will have to wait for all the new items to render to measure their height.

However, you can implement `getItemHeight()` to define the element height up-front without the need for measurement. It is not straightforward for items without a constant height. You can calculate the value based on the number of lines of text and other layout constraints.
We recommend using the `react-native-text-size` library to calculate the height of the displayed text for all list items at once. In our case, it significantly improved the responsiveness for scroll events of `FlatList` on Android.

FlashList as a successor to FlatList

As already discussed, `FlatList` drastically improves the performance of a huge list compared to `ScrollView`. Despite proving itself as a performant solution, it has some caveats.

There are popular cases where developers or users have encountered, for instance, blank spaces while scrolling, laggy scrolling, and a list not being snappy, almost on a daily basis. `FlatList` is designed to keep certain elements in memory, which adds overhead on the device and eventually slows the list down, and blank areas happen when `FlatList` fails to render the items fast enough.

We can, however, minimize these problems to some extent by following the tips here, but still, in most cases, we want more smoothness and snappy lists. With `FlatList`, the JS thread is busy most of the time and we always fancy having that 60FPS tag associated with our JS thread when we're scrolling the list.

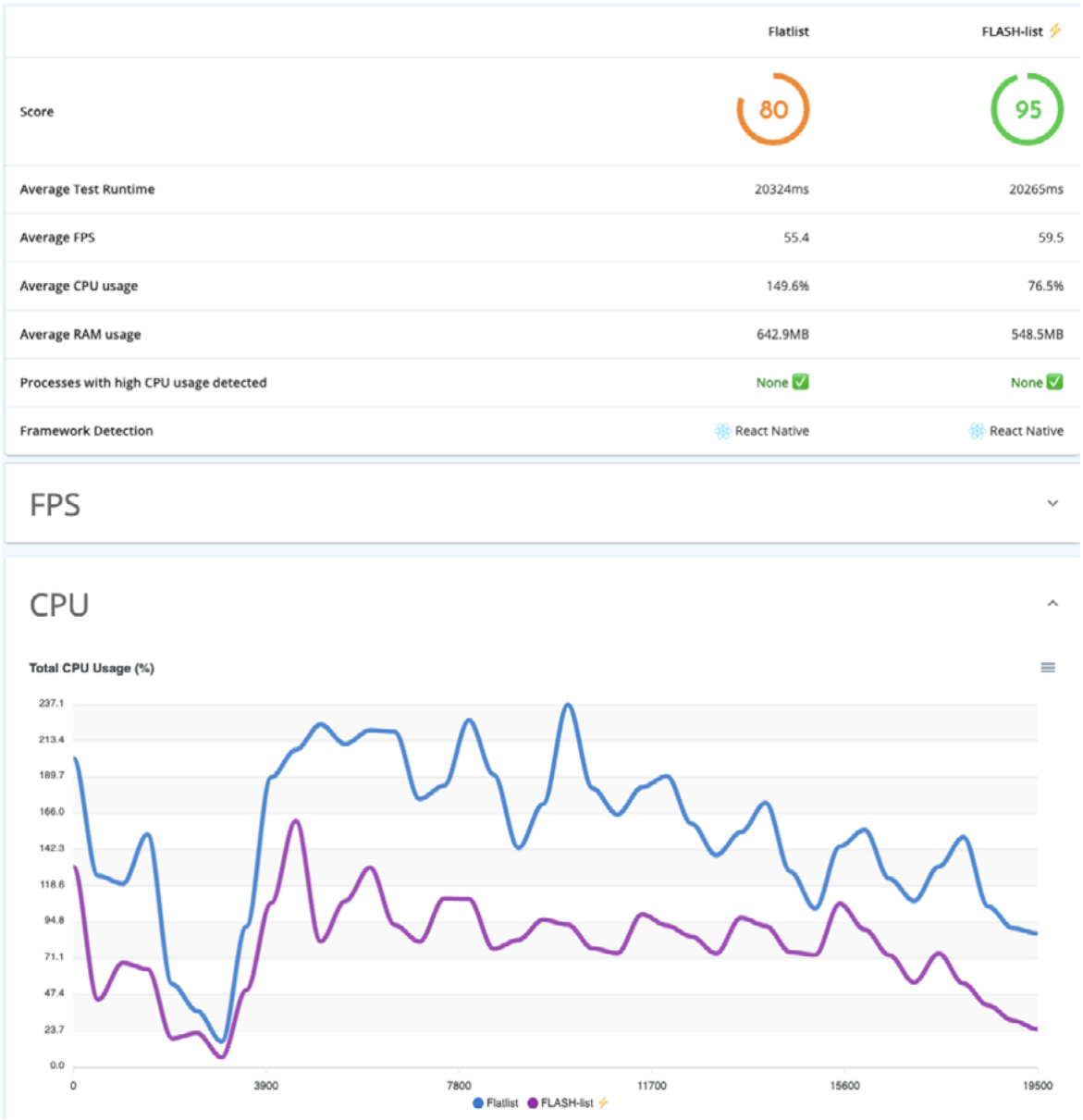
So how should we approach such issues? If not `FlatList`, then what? Luckily for us, the folks at Shopify developed a pretty good drop-in replacement for `FlatList`, known as `FlashList`. The library works on top of RecyclerListView, leveraging its recycling capability and fixing common pain points such as complicated API, using cells with dynamic heights, or first render layout inconsistencies.

FlashList recycles the views that are outside of the viewport and re-uses them for other items. If the list has different items, `FlashList` uses a recycle pool to use the item based on its type. It's crucial to keep the list items as light as possible, without any side effects, otherwise, it will hurt the performance of the list.

There are a couple of props that are quite important with `FlashList`. First is `estimatedItemSize`, the approximate size of the list item. It helps `FlashList` to decide how many items to render before the initial load and while scrolling. If we have different-sized items, we can average them. We can get this value in a warning by the list, if we do not supply it on the first render and then use it forward. The other way is to use the element inspector from the dev support in the React Native app. The second prop is `overrideItemLayout`, which is prioritized over `estimatedItemSize`. If we have different-sized items and we know their sizes, it's better to use them here instead of averaging them.

Let's talk about measuring `FlashList`. Remember to turn on release mode for the JS bundle beforehand. `FlashList` can appear to be slower than `FlatList` in dev mode. The primary reason is a much smaller and fixed `windowSize` equivalent. We can leverage `FlashList`'s built-in callback functions to measure the blank area `onBlankArea` and list load time `onLoad`. You can read more about available helpers in the Metrics section of the documentation.

We can also use Bamlab's Android Performance Profiler, which gives us the results for FPS on the release builds in the form of a performance report. It also creates a nice-looking graph of CPU usage over the period of profiling, so we can verify how certain actions affect this metric.



Performance report from Bamlab's Android Performance Profiler

There's also a [React Native Performance Monitor Flipper plugin]. We can use it to measure the FPS of the JS and UI threads in debug builds. Make sure to turn off the DEV mode when profiling.

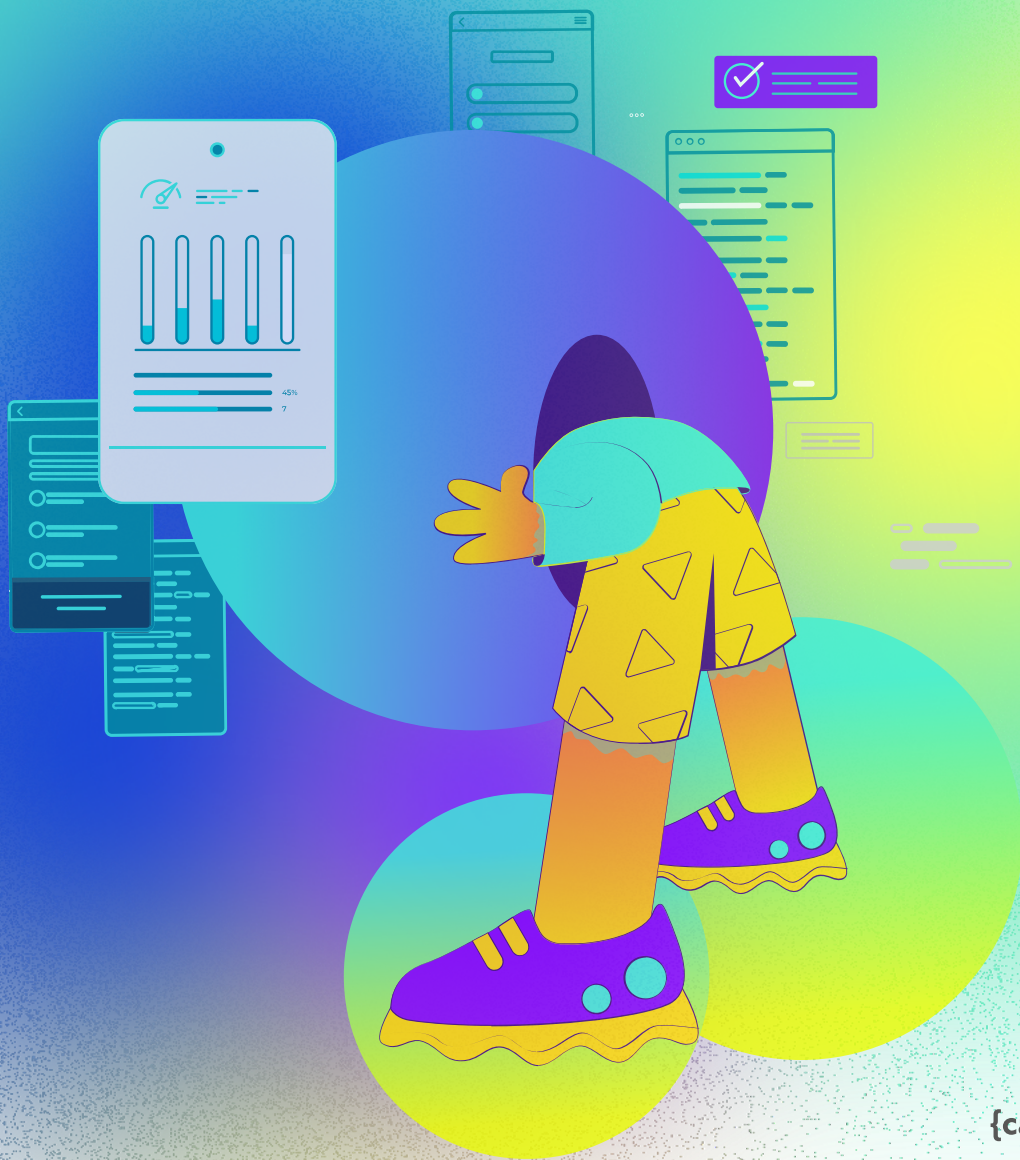
Benefits: Your app works faster, displays complex data structures, and you opt-in for further improvements.

Thanks to using specialized components, your application will always run as fast as possible. You can automatically opt-in to all the performance optimizations performed by React Native and subscribe for

further updates. At the same time, you also save yourself a lot of time reimplementing the most common UI patterns from the ground up, sticky section headers, pull to refresh - you name it. These are already supported by default if you choose to go with `FlashList`.

PART 1 | CHAPTER 3

Think twice before you pick an external library



How working with the right JavaScript libraries can help you boost the speed and performance of your apps.

Issue: You are choosing libraries without checking what is inside

JavaScript development is like assembling applications out of smaller blocks. To a certain degree, it is very similar to building React Native apps. Instead of creating React components from scratch, you are on the hunt for the JavaScript libraries that will help you achieve what you had in mind. The JavaScript ecosystem promotes such an approach to development and encourages structuring applications around small and reusable modules.

This type of ecosystem has many advantages, but also some serious drawbacks. One of them is that developers can find it hard to choose from multiple libraries supporting the same use case.

When picking the one to use in the next project, they often research the indicators that tell them if the library is healthy and well maintained, such as GitHub stars, the number of issues, contributors, and PRs.

What they tend to overlook is the library's size, number of supported features, and external dependencies. They assume that since React Native is all about JavaScript and embracing the existing toolchain, they will work with the same constraints and best practices they know from making web applications.

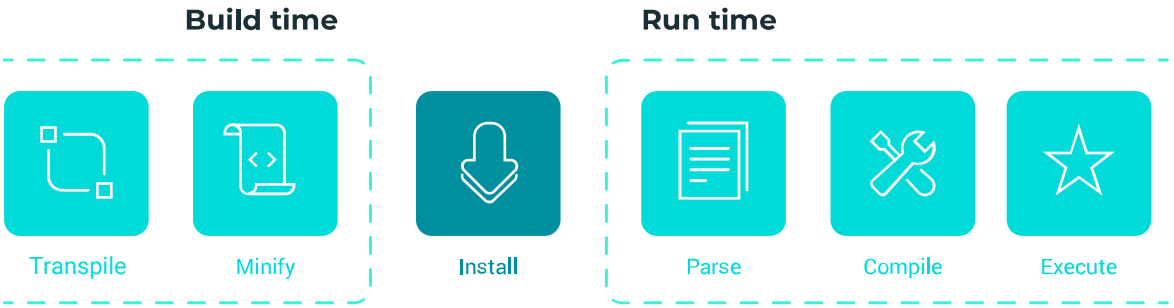
Truth is, they won't, as mobile development is fundamentally different and has its own set of rules. For example, while the size of the assets is crucial in the case of web applications, it is not as equally important in React Native, where assets are located in the filesystem.

The key difference lies in the performance of the mobile devices and the tooling used for bundling and compiling the application.

Although you will not be able to do much about the device limitations, you can control your JavaScript code. In general, less code means faster opening time. And one of the most important factors affecting the overall size of your code is libraries.

Complex libraries hamper the speed of your apps

Unlike a fully native application, a React Native app contains a JavaScript bundle that needs to be loaded into memory. Then it is parsed and executed by the JavaScript VM. The overall size of the JavaScript code is an important factor.



Interpretation with conventional engine engine

While that happens, the application remains in the loading state. We often describe this process as TTI – Time to Interactive. It is a time expressed in (well, hopefully) the milliseconds between when the icon gets selected from the application drawer and when it becomes fully interactive.

Unfortunately, Metro – the default React Native bundler – currently doesn't support tree shaking. If you're not familiar with this notion, read this article.

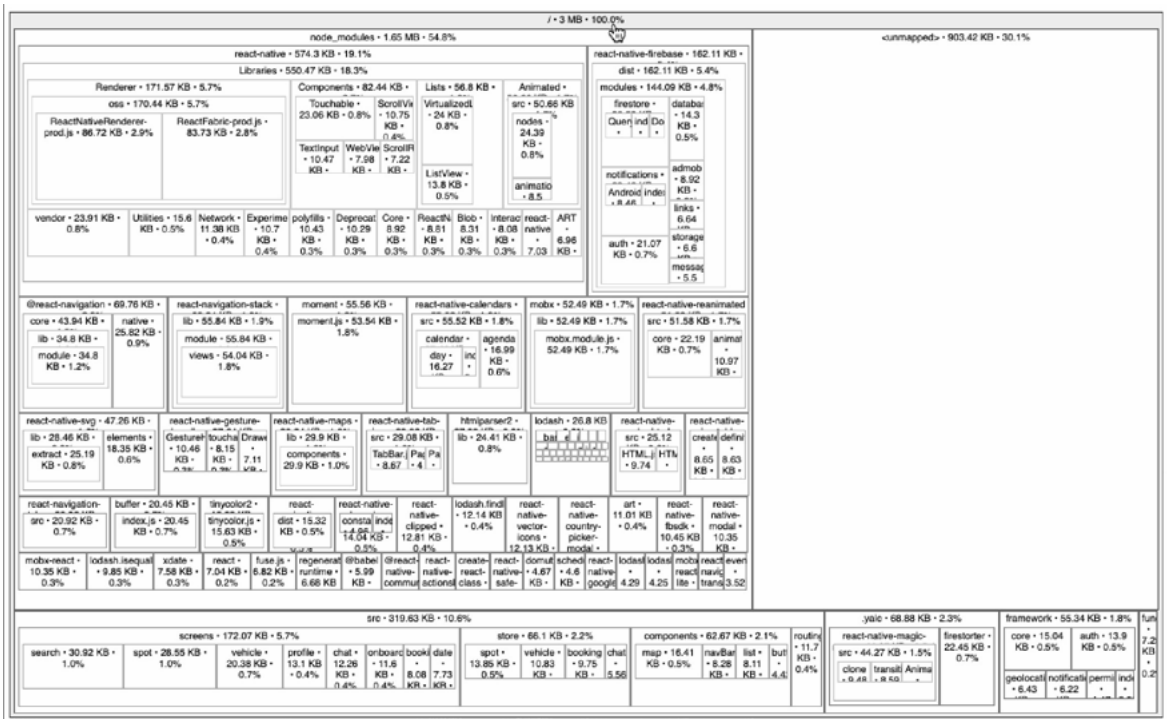
This means that all the code that you pull from NPM and import to your project will be present in your production JS bundle, loaded into

memory, and parsed.That can have a negative impact on the total startup time of your application.

What's worth pointing out is that it's not the case with Hermes engine, which automatically pages only necessary bytecode into memory. Read more in the Hermes chapter.

How do we analyze bundle size

Keeping tabs on your bundle size is very important. We can make use of the [react-native-bundle-visualizer]() to analyze the bundle with the help of GUI. We can get the details of any added library in the bundle; hence deciding if it's worth keeping or removing that library. This package produces output using the app bundle in the following form:



Solution: Be more selective and use smaller specialized libraries.

The easiest way to overcome this issue is to employ the right strategy for architecturing the project upfront.

If you are about to pull a complex library, check if there are smaller alternatives that have the functionality you're looking for.

Here's an example: One of the most common operations is manipulating dates. Let's imagine you are about to calculate an elapsed time. Rather than pulling down the entire `moment.js` library (67.9 KB) to parse the date itself,

```
import moment from 'moment';

const date = moment('12-25-1995', 'MM-DD-YYYY');
```

Parsing a date with moment.js

we can use `day.js` (only 2Kb) which is substantially smaller and offers only the functionality that we're looking for.

```
import dayjs from 'dayjs';

const date = dayjs('12-25-1995', 'MM-DD-YYYY');
```

Parsing a date with day.js

If there are no alternatives, a good rule of thumb is to check if you can import a smaller part of the library.
For instance, many libraries such as `lodash` have already split themselves into smaller utility sets and support environments where dead code elimination is unavailable.
Let's say you want to use lodash map. Instead of importing the whole library, (as presented here),

```
import { map } from 'lodash';

const square = (x) => x * x;
map([4, 8], square);
```

Using lodash map by importing the whole library

you could import only a single package:

```
import map from 'lodash/map';

const square = (x) => x * x;
map([4, 8], square);
```

Using lodash map by importing only single function

As a result, you can benefit from the utilities that are a part of the lodash package without pulling them all into the application bundle.
If you'd like to have constant insight into your dependencies' size impact, we highly recommend the import-cost VSCode extension. Or using the Bundlephobia website.

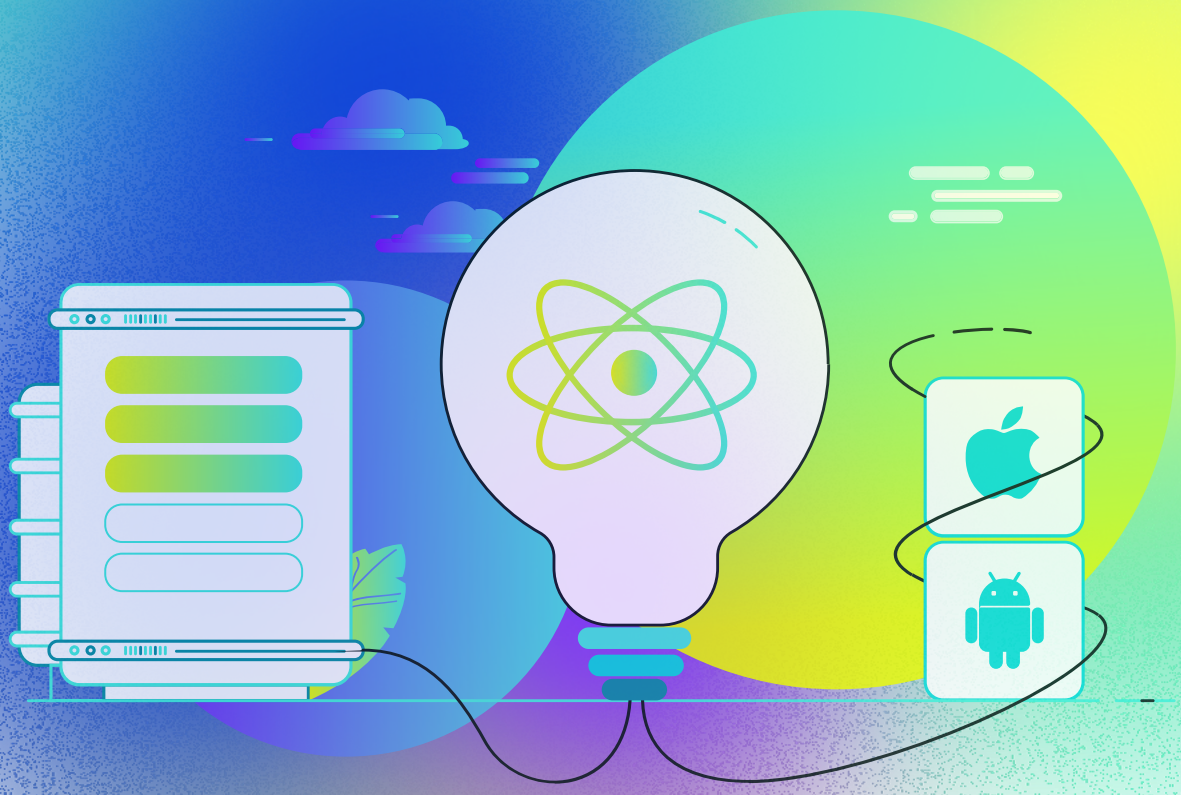
Benefits: Your app has a smaller footprint and loads faster.

Mobile is an extremely competitive environment, with lots of applications designed to serve similar purposes and fight over the same customers. Faster startup time, smoother interactions, and the overall look and feel might be your only way to stand out from the crowd.

You shouldn't downplay the importance of choosing the right set of libraries. Being more selective with third-party dependencies may seem irrelevant at first. But all the saved milliseconds will add up to significant gains over time.

PART 1 | CHAPTER 4

Always remember to use libraries dedicated to the mobile platform



Use libraries dedicated to mobile and build features faster on many platforms at once, without compromising on the performance and user experience.

Issue: You use web libraries that are not optimized for mobile.

As discussed earlier, one of the best things about React Native is that you can write the mobile application with JavaScript, reuse some of your React components, and do business logic with your favorite state management library.

While React Native provides web-like functionality for compatibility with the web, it is important to understand that it is not the same environment. It has its own set of best practices, quick wins, and constraints.

For example, while working on a web application, we don't have to worry too much about the overall CPU resources needed by our application. After all, most of the websites run on devices that are either plugged into the network or have large batteries.

It is not hard to imagine that mobile is different. There's a wide range of devices with different architectures and resources available. Most of the time, they run on a battery and the drain caused by the application can be a deciding factor for many developers.

In other words – how you optimize the battery consumption both in the foreground and background can make all the difference.

Not optimized libraries cause battery drain and slow down the app. The OS may limit your application's capabilities.

While React Native makes it possible to run the same JavaScript on mobile as in the browser, that doesn't mean you should be doing this every time. As with every rule, there are exceptions.

If the library depends heavily on networking, such as real-time messaging or offers the ability to render advanced graphics (3D structures, diagrams), it is very likely that you're better off going with a dedicated mobile library.

Mobile libraries were developed within the web environment in the first place, assuming the capabilities and constraints of the browser. It is very likely that the result of using a web version of a popular SDK will result in extraneous CPU and memory consumption.

Certain OSs, such as iOS, are known to be constantly analyzing the resources consumed by the application in order to optimize the battery life. If your application is registered to perform background activities and these activities take too much of the resources, the interval for your application may get adjusted, lowering the frequency of the background updates that you initially signed up for.

Solution: Use a dedicated, platform-specific version of the library.

Let's take Firebase as an example. Firebase is a mobile platform from Google that lets you build your apps faster. It is a collection of tools and libraries that enable certain features instantly within your app.

Firebase contains SDKs for the web and mobile – iOS and Android respectively. Each SDK contains support for Realtime Database.

Thanks to React Native, you can run the web version of it without major problems:

```
import { getDatabase, onValue, ref } from 'firebase/database';

const database = getDatabase();

onValue(ref(database, '/users/123'), (snapshot) => {
  console.log(snapshot.val());
});
```

An example reading from Firebase Realtime Database in RN

However, this is not what you should be doing. While the above example works without issues, it does not offer the same performance as the mobile equivalent. The SDK itself also contains fewer features – no surprises here, as web is different and there's no reason Firebase.js should provide support for mobile features.

In this particular example, it is better to use a dedicated Firebase library that provides a thin layer on top of dedicated native SDKs and offers the same performance and stability as any other native application out there.

Here's how the above example would look like:

```
import database from '@react-native-firebase/database';

database().ref('/users/123').on('value', (snapshot) => {
    console.log(snapshot.val());
});
```

An example reading from Firebase Realtime Database in RN

As you can see, the difference is minimal. In this case, the library authors did a great job mimicking the API to reduce the potential confusion while switching back and forth between the web and mobile context.

Benefits: Provide the fastest and most performant support with no harm to the battery life.

React Native is all about giving you control and freedom to choose how you want to build your application.

For straightforward aspects and maximum reusability, you can choose to go with the web version of the library. This will give you access to the same features as in the browser with relatively low effort.

For advanced use cases, you can easily extend React Native with a native functionality and talk directly to the mobile SDKs. Such an escape hatch is what makes React Native extremely versatile and enterprise-ready. It allows you to build features faster on many platforms at once, without compromising on the performance and user experience – something other hybrid frameworks cannot claim.

PART 1 | CHAPTER 5

Find the balance between native and JavaScript

Seek the harmony between native and JavaScript to build fast-working and low-maintenance apps.

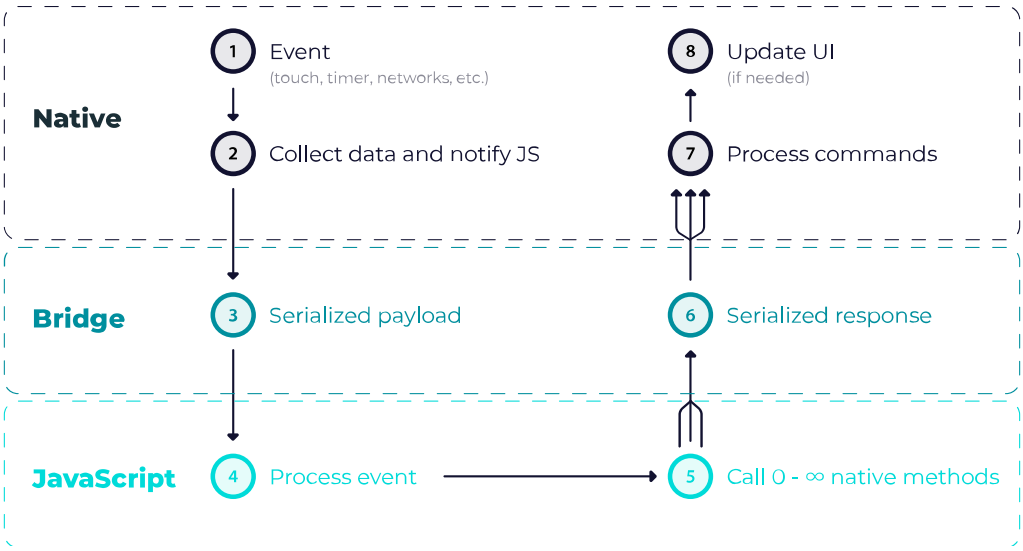
Issue: While working on native modules, you draw the line in the wrong place between native and JavaScript abstractions

When working with React Native, you're going to be developing JavaScript most of the time. However, there are situations when you need to write a bit of native code. For example, you're working with a third-party SDK that doesn't have official React Native support yet. In that case, you need to create a native module that wraps the underlying native methods and exports them to the React Native realm.

All native methods need real-world arguments to work. React Native builds on top of an abstraction called a bridge, which provides bidirectional communication between JavaScript and native worlds.

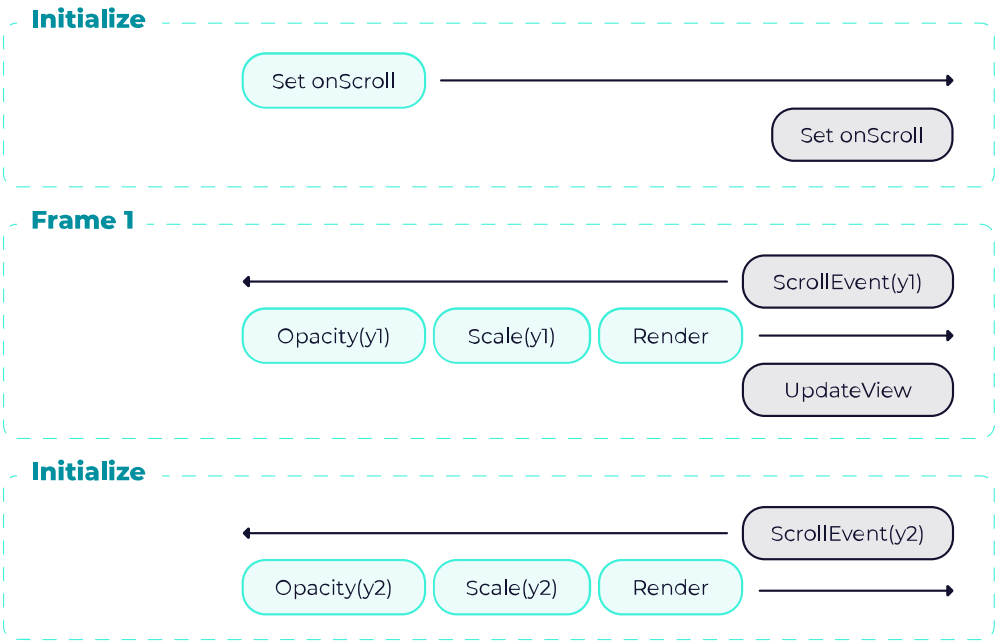
Note: There's an ongoing effort to move away from asynchronous bridge communication to a synchronous one. You can read more about it in the New Architecture chapter.

As a result, JavaScript can execute native APIs and pass the necessary context to receive the desired return value. The communication itself is asynchronous – it means that while the caller is waiting for the results to arrive from the native side, the JavaScript is still running and may already be up for another task.



The number of JavaScript calls that arrive over the bridge is not deterministic and can vary over time, depending on the number of interactions that you do within your application. Additionally, each call takes time, as the JavaScript arguments need to be stringified into JSON, which is the established format that can be understood by these two realms.

For example, when the bridge is busy processing the data, another call will have to block and wait. If that interaction was related to gestures and animations, it is very likely that you have a dropped frame – the operation wasn't performed causing jitters in the UI.



Certain libraries, such as `Animated` provide special workarounds. In this case, use `NativeDriver`, which serializes the animation, passes it once upfront to the native thread, and doesn't cross the bridge while the animation is running – preventing it from being subject to accidental frame drops while other work is happening.
That's why it is important to keep the bridge communication efficient and fast.

More traffic flowing over the bridge means less space for other things

Passing more traffic over the bridge means that there is less space for other important things that React Native may want to transfer at that time. As a result, your application may become unresponsive to gestures or other interactions while you're performing native calls.

If you are seeing a degraded UI performance while executing certain native calls over the bridge or seeing substantial CPU consumption, you should take a closer look at what you are doing with the external libraries. It is very likely that there is more being transferred than should be.

Solution: Use the right amount of abstraction on the JS side – validate and check the types ahead of time.

When building a native module, it is tempting to proxy the call immediately to the native side and let it do the rest. However, there are cases, such as invalid arguments, that
end up causing an unnecessary round-trip over the bridge only to learn that we didn't provide the correct set of arguments.

Let's take a JavaScript module that proxies the call straight to the underlying native module.

```
import { NativeModules } from 'react-native';
const { ToastExample } = NativeModules;

export const show = (message, duration) => {
  ToastExample.show(message, duration);
};
```

Bypassing arguments to the native module

In the case of an incorrect or missing parameter, the native module is likely to throw an exception. The current version of React Native doesn't provide an abstraction for ensuring the JavaScript parameters and the ones needed by your native code are in sync. Your call will be serialized to JSON, transferred to the native side, and executed.

That operation will perform without any issues, even though we haven't passed the complete list of arguments needed for it to work. The error will arrive when the native side processes the call and receives an exception from the native module.

In such a scenario, you have lost some time waiting for the exception that you could've checked for beforehand.

```
import { NativeModules } from 'react-native';
const { ToastExample } = NativeModules;

export const show = (message, duration) => {
  if (typeof message !== 'string' || message.length > 100) {
    throw new Error('Invalid Toast content');
  }

  if (!Number.isInteger(duration) || duration > 20000) {
    throw new Error('Invalid Toast duration');
  }

  ToastExample.show(message, duration);
};
```

Using the native module with arguments validation

The above is not only tied to the native modules themselves. It is worth

keeping in mind that every React Native primitive component has its native equivalent and component props are passed over the bridge every time there's a rendering happening – or is it? It's not always the case when a component re-renders. React Native renderer is smart enough to "diff" the parts of our JS React component hierarchy and only send enough information through the bridge, so that the native view hierarchy is updated.

This is the case when styling components like e.g. View or Text using the `style` prop. Let's take a look at the following example using inline `styles`.

```
import React from 'react';
import { View } from 'react-native';

const App = () => {
  return (
    <View
      style={{
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
      }}>
      <View
        style={{
          backgroundColor: 'coral',
          width: 200,
          height: 200,
        }}
      />
    </View>
  );
};

export default App;
```

Read more: https://snack.expo.dev/@callstack-snack/inline-styled-view

Even though the `style` prop is passed as an inline object, it doesn't cause us any performance issues. Neither when we dynamically change the styles based on props, nor when we re-render the App component. View passes its props almost directly to the underlying native representation. And thanks to the React Native renderer, no matter how often

we re-render this component on the JS side, only the smallest amount of data necessary to update the style prop will be passed through the bridge.

In React Native we have nicer ways to deal with styling and it's through StyleSheet API — a dedicated abstraction similar to CSS StyleSheets. Although it provides no performance benefits, it's worth calling it out for the ease of development and maintenance. When we develop our app in TypeScript or Flow, StyleSheet is well typed and makes it possible for our code editors to auto-complete.

Benefits: The codebase is faster and easier to maintain

Whether you're facing any performance challenges right now, it is smart to implement a set of best practices around native modules as the benefits are not just about the speed but also the user experience. Sure, keeping the right amount of the traffic flowing over the bridge will eventually contribute to your application performing better and working smoothly. As you can see, certain techniques mentioned in this section are already being actively used inside React Native to provide you a satisfactory performance out of the box. Being aware of them will help you create applications that perform better under a heavy load.

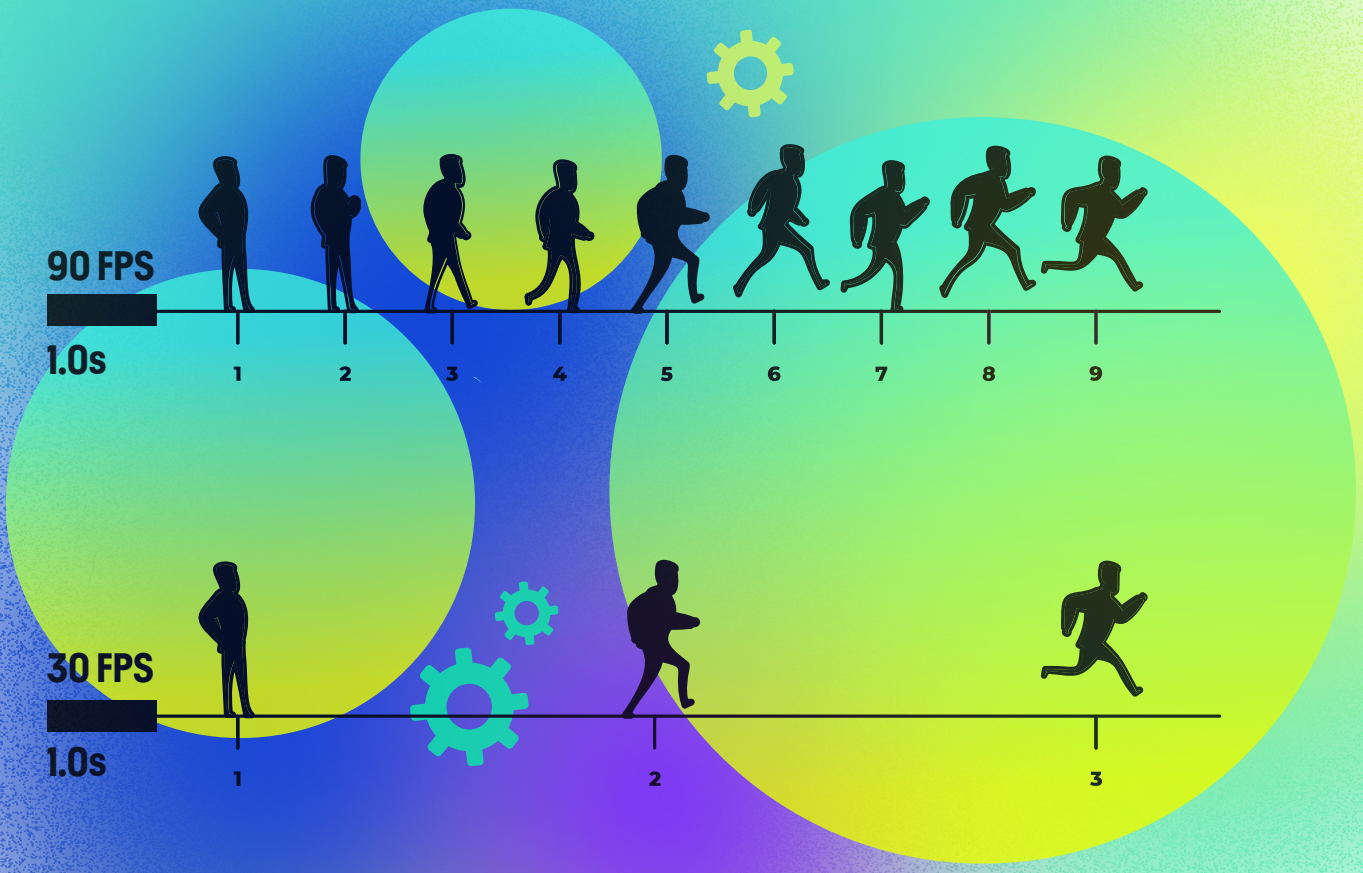
However, one additional benefit that is worth pointing out is the maintenance.

Keeping the heavy and advanced abstractions, such as validation, on the JavaScript side will result in a very thin native layer that is nothing more but just a wrapper around an underlying native SDK. In other words, the native part of your module is going to look more like a copy-paste from the documentation – comprehensible and specific.

Mastering this approach to the development of native modules is why a lot of JavaScript developers can easily extend their applications with additional functionality without specializing in Objective-C or Java.

PART 1 | CHAPTER 6

Animate at 60FPS - no matter what



Use native solutions to achieve smooth animations and a gesture-driven interface at 60FPS.

Issue: JS-driven animations are occupying the bridge traffic and slowing down the application.

Mobile users are used to smooth and well-designed interfaces that quickly respond to their interactions and provide prompt visual feedback. As a result, applications have to register a lot of animations in many places that will have to run while other work is happening.

As we know from the previous section, the amount of information that can be passed over the bridge is limited. There's currently no built-in priority queue. In other words, it is on you to structure and design your application in a way that both the business logic and animations can function without any disruptions. This is different from the way we are used to performing animations. For example, on iOS, the built-in APIs offer unprecedented performance and are always scheduled with the appropriate priority. Long story short - we don't have to worry too much about ensuring they're running at 60FPS.

With React Native, this story is a bit different. If you do not think about your animations top-down beforehand and choose the right tools to tackle this challenge, you're on track to run into dropped frames sooner or later.

Janky or slow animations affect the perception of the app, making it look slow and unfinished

In today's sea of applications, providing a smooth and interactive UI might be one of your only ways to win over customers who are looking to choose the app to go.

If your application fails to provide a responsive interface that works well with the user interactions (such as gestures), not only may it affect new customers, but also decrease the ROI and user sentiment.

Mobile users like the interfaces that follow them along and that look top-notch and ensure the animations are always running smoothly is a fundamental part that builds such an experience.

Solution: If it's possible, use native and correct animations.

One-off animations

Enabling the usage of the native driver is the easiest way of quickly improving your animations' performance. However, the subset of style props that can be used together with the native driver is limited. You can use it with non-layout properties like transforms and opacity. It will not work with colors, height, and others. Those are enough to implement most of the animations in your app because you usually want to show/hide something or change its position.

```
const fadeAnim = useRef(new Animated.Value(0)).current;

const fadeIn = () => {
  Animated.timing(fadeAnim, {
    toValue: 1,
    duration: 1000,
    useNativeDriver: true, // enables native driver
  }).start();
};

// [...]

<Animated.View style={{ opacity: fadeAnim }} />
```

Enabling the native driver for opacity animation

For more complex use cases, you can use the `React Native Reanimated` library. Its API is compatible with the basic `Animated` library and introduces a set of fine-grained controls for your animations with

a modern hooks-based interface. More importantly, it introduces the possibility to animate all possible style props with the native driver. So animating height or color will no longer be an issue. However, transform and opacity animations will still be slightly faster since they are GPU-accelerated. You can play with different combinations in this reanimated playground.

Gesture-driven animations

The most desired effect that can be achieved with animations is being able to control animation with a gesture. For your customers, this is the most enjoyable part of the interface. It builds a strong sentiment and makes the app feel very smooth and responsive. Plain React Native is very limited when it comes to combining gestures with native driven animations. You can utilize ScrollView scroll events to build things like a smooth collapsible header.

For more sophisticated use cases, there is an awesome library—React Native Gesture Handler—which allows you to handle different gestures natively and interpolate those into animations. You can build a swipeable element by combining it with Animated. While it will still require JS callbacks, there is a remedy for that!

The most powerful pair of tools for gesture-driven animations is using Gesture Handler combined with Reanimated. They were designed to work together and give the possibility to build complex gesture-driven animations that are fully calculated on the native side.

Reanimated API supports synchronous JavaScript execution on the UI thread using the concept of worklets. The library's runtime spawns a secondary JS context on the UI thread that is then able to run JavaScript functions in the form of said worklets. Now using your imagination and leveraging Reanimated, you can create wonderful animations at full available speeds.

```
import React from 'react';
import { StyleSheet, View } from 'react-native';
import { PanGestureHandler } from 'react-native-gesture-
handler';
import Animated, {
  useAnimatedGestureHandler,
  useAnimatedStyle,
  useSharedValue,
  withSpring,
} from 'react-native-reanimated';

const Snappable = (props) => {
  const startingPosition = 0;
  const x = useSharedValue(startingPosition);
  const y = useSharedValue(startingPosition);

  const animatedStyle = useAnimatedStyle(() => {
    return {
      transform: [{ translateX: x.value }, { translateY:
y.value }],
    };
  });

  const eventHandler = useAnimatedGestureHandler({
    onStart: (event, ctx) => {
      ctx.startX = x.value;
      ctx.startY = y.value;
    },
    onActive: (event, ctx) => {
      x.value = ctx.startX + event.translationX;
      y.value = ctx.startY + event.translationY;
    },
    onEnd: (event, ctx) => {
      x.value = withSpring(startingPosition);
      y.value = withSpring(startingPosition);
    },
  });

  return (
    <PanGestureHandler onGestureEvent={eventHandler}>
      <Animated.View style={animatedStyle}>{props.children}</
Animated.View>
    </PanGestureHandler>
  );
};

const Example = () => {
  return (
    <View style={styles.container}>
      <Snappable>
        <View style={styles.box} />
      </Snappable>
```

```
      </View>
    );
};

export default Example;

const BOX_SIZE = 100;

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  box: {
    width: BOX_SIZE,
    height: BOX_SIZE,
    borderColor: '#F5FCFF',
    alignSelf: 'center',
    backgroundColor: 'plum',
    margin: BOX_SIZE / 2,
  },
});
```

Read more: https://snack.expo.dev/@callstack-snack/gesture-with-animation

Low-level handling of gestures might not be a piece of cake, but fortunately, there are third-party libraries that utilize the mentioned tools and expose the prop `callbackNode`. It's an `Animated.Value` that's derived from specific gesture behavior. Its value range is usually from 0 to 1, which follows the progress of the gesture. You can interpolate the values to animated elements on the screen. A great example of the libraries that expose `CallbackNode` are `reanimated-bottom-sheet` and `react-native-tab-view`.

```
import * as React from 'react';
import { StyleSheet, Text, View } from 'react-native';
import Animated from 'react-native-reanimated';
import BottomSheet from 'reanimated-bottom-sheet';
import Lorem from './Lorem';

const { Value, interpolateNode: interpolate } = Animated;

const Example = () => {
  const gestureCallbackNode = new Value(0);

  const renderHeader = () => (
    <View style={styles.headerContainer}>
      <Text style={styles.headerTitle}>Drag me</Text>
    </View>
  );

  const renderInner = () => (
    <View style={styles.innerContainer}>
      <Animated.View
        style={{
          opacity: interpolate(gestureCallbackNode, {
            inputRange: [0, 1],
            outputRange: [1, 0],
          }),
          transform: [
            {
              translateY: interpolate(gestureCallbackNode, {
                inputRange: [0, 1],
                outputRange: [0, 100],
              }),
            },
          ],
        }}>
        <Lorem />
        <Lorem />
      </Animated.View>
    </View>
  );

  return (
    <View style={styles.container}>
      <BottomSheet
        callbackNode={gestureCallbackNode}
        snapPoints={[50, 400]}
        initialSnap={1}
        renderHeader={renderHeader}
        renderContent={renderInner}
      />
    </View>
  );
};
```

```
export default Example;

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  headerContainer: {
    width: '100%',
    backgroundColor: 'lightgrey',
    height: 40,
    borderWidth: 2,
  },
  headerTitle: {
    textAlign: 'center',
    fontSize: 20,
    padding: 5,
  },
  innerContainer: {
    backgroundColor: 'lightblue',
  },
});
```

Read more: https://snack.expo.dev/@callstack-snack/interpolation

Giving your JS operations a lower priority

It is not always possible to fully control the way animations are implemented. For example, React Navigation uses a combination of `React Native Gesture Handler` and `Animated` which still needs JavaScript to control the animation runtime. As a result, your animation may start flickering if the screen you are navigating to loads a heavy UI. Fortunately, you can postpone the execution of such actions using `InteractionManager`. This handy helper allows long-running work to be scheduled after any interactions/animations have completed. In particular, this allows JavaScript animations to run smoother.

Note: In the near future, you'll be able to achieve similar behavior with React itself on a renderer level (with Fabric) using the `startTransition` API. Read more about it in the New Architecture chapter.

```
import React, { useState, useRef } from 'react';
import {
  Text,
  View,
  StyleSheet,
  Button,
  Animated,
  InteractionManager,
  Platform,
} from 'react-native';
import Constants from 'expo-constants';

const ExpensiveTaskStates = {
  notStared: 'not started',
  scheduled: 'scheduled',
  done: 'done',
};

const App = () => {
  const animationValue = useRef(new Animated.Value(100));
  const [animationState, setAnimationState] = useState(false);
  const [expensiveTaskState, setExpensiveTaskState] = useState(
    ExpensiveTaskStates.notStared,
  );

  const startAnimationAndScheduleExpensiveTask = () => {
    Animated.timing(animationValue.current, {
      duration: 2000,
      toValue: animationState ? 100 : 300,
      useNativeDriver: false,
    }).start(() => {
      setAnimationState((prev) => !prev);
    });
    setExpensiveTaskState(ExpensiveTaskStates.scheduled);
    InteractionManager.runAfterInteractions(() => {
      setExpensiveTaskState(ExpensiveTaskStates.done);
    });
  };

  return (
    <View style={styles.container}>
      {Platform.OS === 'web' ? (
        <Text style={styles.infoLabel}>
          !InteractionManager works only on native platforms.
Open example on
          iOS or Android!
        </Text>
      ) : (
        <>
          <Button
            title="Start animation and schedule expensive task"
```

```
            onPress={startAnimationAndScheduleExpensiveTask}
          />
          <Animated.View
            style={[styles.box, { width: animationValue.current
}]}>
            <Text>Animated box</Text>
          </Animated.View>
          <Text style={styles.paragraph}>
            Expensive task status: {expensiveTaskState}
          </Text>
        </>
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    paddingTop: Constants.statusBarHeight,
    padding: 8,
  },
  paragraph: {
    margin: 24,
    fontSize: 18,
    textAlign: 'center',
  },
  box: {
    backgroundColor: 'coral',
    marginVertical: 20,
    height: 50,
  },
  infoLabel: {
    textAlign: 'center',
  },
});

export default App;
```

Read more: https://snack.expo.dev/@callstack-snack/interaction-manager

In practice, you can show a placeholder, wait for the animation to finish, and then render the actual UI. It would help your JavaScript animations to run smoother and avoid interruptions by other operations. It's usually smooth enough to provide a great experience.

Remember, that using JS-driven animations is bound to have significantly worse performance than native run on UI thread. Making it hard to achieve max FPS available for the device. Strive for platform-native animations whenever possible. And treat APIs such as InteractionManager for cases, where heavy UI updates may interfere with gestures or animations. They may fight for the same resource—native UI thread.

Benefits: Enjoy smooth animations and a gesture-driven interface at 60FPS.

There's no one single right way of doing animations in React Native. The ecosystem is full of different libraries and approaches to handling interactions. The ideas suggested in this section are just recommendations to encourage you to not take the smooth interface for granted.

What is more important is painting that top-down picture in your head of all interactions within the application and choosing the right ways of handling them. There are cases where JavaScript-driven animations will work just fine. At the same time, there are interactions where native animation (or an entirely native view) will be your only way to make it smooth.

With such an approach, the application you create will be smoother and snappy. It will not only be pleasant for your users to use but also for you to debug and have fun with it while developing.

> **"By adding delightful animations to your app, users tend to be much more forgiving. If done carefully, animations in React Native can perform great and improve the perceived performance of the app to the user."**
>
> **William Candillon**
> Chief Technology Officer at 28msec

PART 1 | CHAPTER 7

Replace Lottie with Rive

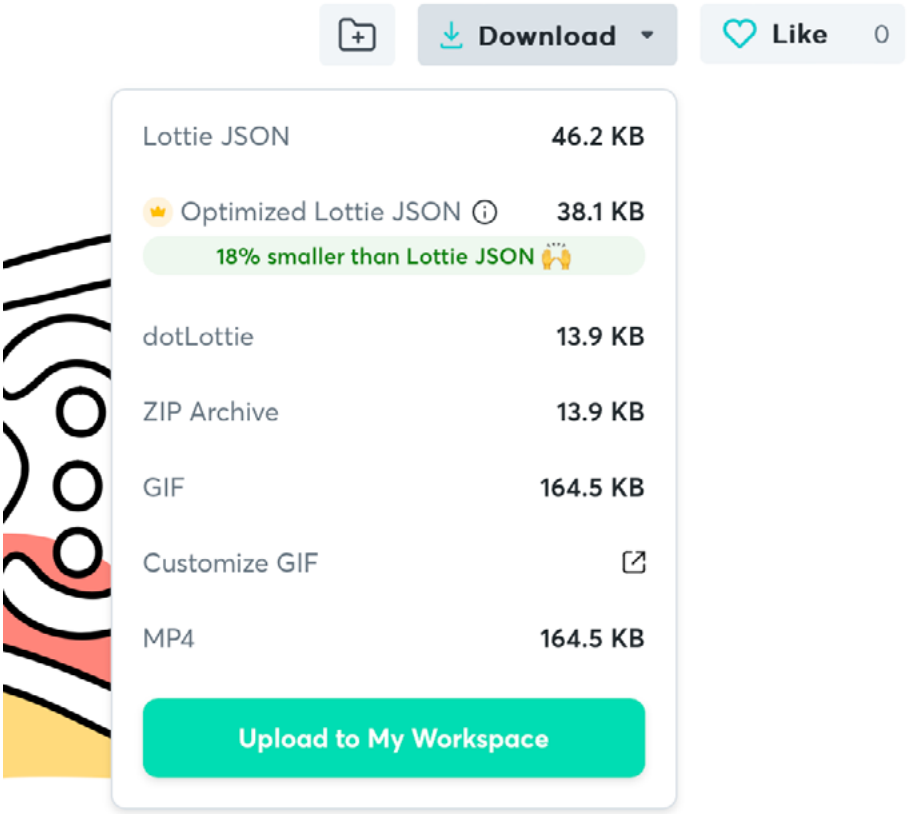
Leverage state machines to provide robust interactive animations at 60FPS

Issue: Real-time animations suffering from low FPS, file size, and not being robust

When we talk about interactive mobile apps, we often think there'll be using certain user driven animations. For example, we can think of getting a nice order placed animation when we complete a checkout. A more complex example would be onboarding steps where the user has to tap on various buttons to move forward and, in most cases, each step shows a nice animation. So how can developers implement such behavior? One approach is using GIFs. If we have 3 onboarding steps, then we will want to have 3 GIFs. And often this solution is good enough performance- and UX-wise. But what if we need more fidelity in our animations? Or when it needs to be high quality, maybe on a full screen? GIFs can quickly add a few megabytes to our app's bundle.

So what other options do we have? Let's talk about Lottie. A mobile client for Android and iOS, which was created by the developers at Airbnb to leverage LottieFiles, which is JSON-based animation exported using the plugin BodyMoving from Adobe AfterEffects. They have a pretty good community with lots of free-to-use animations. Check it out here.

If we look at the React Native Lottie library, it is pretty popular and well-received by the community. We can control the animation using the progress prop or use imperative methods such as play. There are other useful props that we can use to suit our needs. Let's compare the size of a Lottie JSON animation and a corresponding GIF.



Source: https://lottiefiles.com/128635-letter-d

If we compare, JSON is 46.2 KB and the GIF is 164.5 KB, which is almost 4 times as much as JSON. We can further reduce the size of JSON using the Optimized Lottie JSON but that's a paid solution. Now if we recall the above approach of having separate GIFs for each onboarding step, we can now use Lottie instead of GIFs since it's smaller. We can also use the remote resources of Lottie JSON instead of having it in a bundle but an extra effort will be required to keep it available for offline purposes.

We can still do better, but how good will that be if we use a single animation and control it using triggers? For example, if we tap on a button, the state of the animation changes for the next step. Let's say we want to change the size or do other customizations, we can't do it in the editor provided by LottieFiles. Instead, we will have to import that in Adobe AE and then do the adjustments and re-export it as JSON. We can, however, customize the color for layers in the web editor. Not everyone has expertise in using Adobe AE and has an animator to

consult, e.g. if we're working on our pet project, we will want to adjust any animation in the web editor.

There are other factors around performance associated with Lottie which we will discuss in the next section.

Solution: Leverage developer-friendly tools which offer better FPS with less file size.

There's a new tool in town called Rive. It aims to be an alternative to Lottie by providing everything Lottie does and then some. It also ships a web editor to customize real-time animations on the go. The editor allows the user to build interactive animations which have the capability to interact based on the inputs provided by the user. Having trigger-based animations is a great win, especially for mobile platforms.

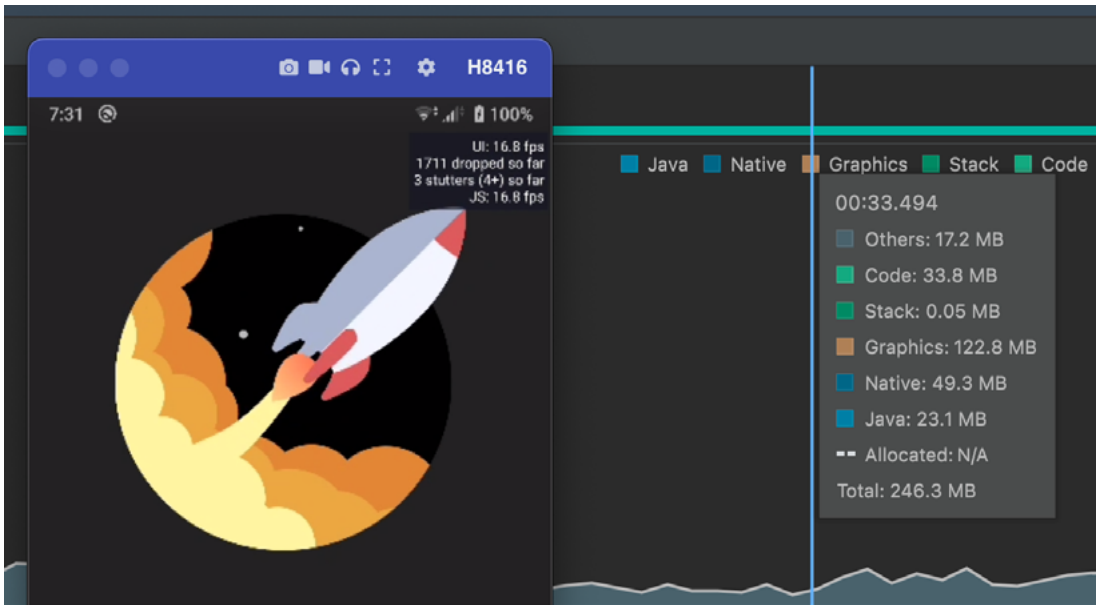
Remember the approach we took for the onboarding steps animations? Now if we use Rive animations, we can leverage its state machine, triggers, and user inputs to use a single animation file for our onboarding steps. It really helps in improving the developer experience and the cherry on top: the size of the animation file is very small compared to Lottie JSON. We also don't have to keep different files for each onboarding step; hence saving some KBs for the bundle as well.

```
import React, { useRef } from 'react';
import { Button, SafeAreaView, StyleSheet } from 'react-
native';
import Rive, { RiveRef } from 'rive-react-native';

const stateMachineOne = 'State Machine 1';

const App = () => {
  const riveRef = useRef<RiveRef>(null);
  const isRunning = useRef(false);

  const onIsRunning = () => {
    isRunning.current = !isRunning.current;
    riveRef.current?.setInputState(
      stateMachineOne,
      'isRunning',
      isRunning.current,
    );
  };

  const onSideKick = () => {
    riveRef.current?.fireState(stateMachineOne, 'sideKick');
  };

  return (
    <SafeAreaView>
      <Rive
        ref={riveRef}
        resourceName="character"
        style={styles.character}
        stateMachineName={stateMachineOne}
        autoplay
      />
      <Button title="Is Running" onPress={onIsRunning} />
      <Button title="SideKick" onPress={onSideKick}
color="#3e3e3e" />
    </SafeAreaView>
  );
};

export default App;

const styles = StyleSheet.create({
  character: {
    width: 400,
    height: 400,
  },
});
```

As we see, adding complex animation is developer-friendly. We can also add triggers or input-based animations quickly. All we need is the information regarding the state machine, inputs, and triggers. If we have an animator who built a beautiful animation for our project on Rive Editor, we can ask them to pass along the info. Otherwise, we can always look up this info in the web editor ourselves.

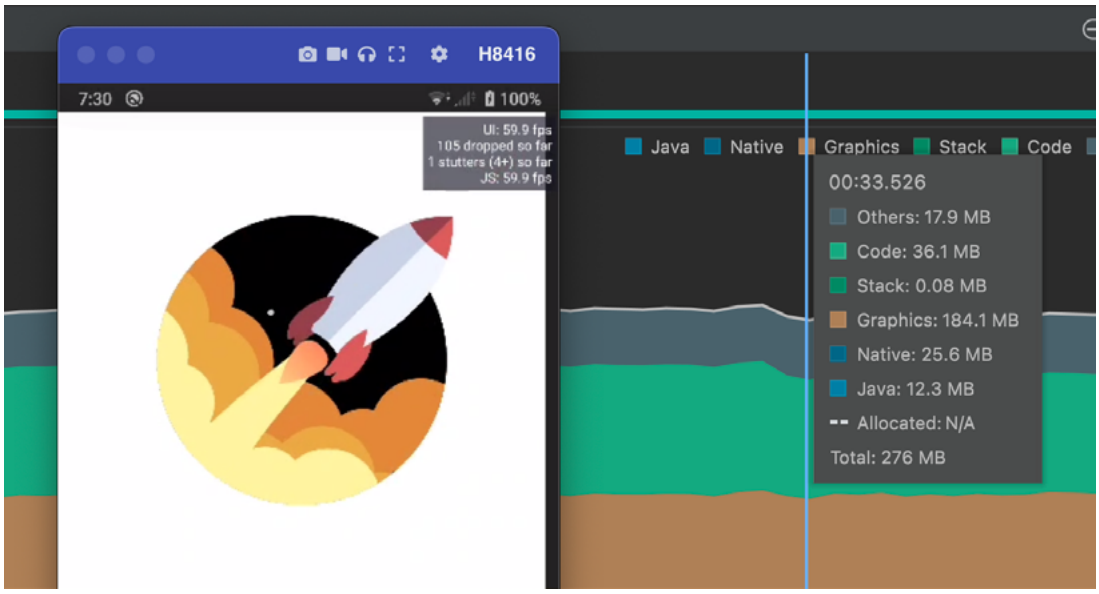
Now let's talk about performance in terms of FPS, CPU, and memory consumption. We will do a comparison of an animation that is built for Lottie with the same animation built for Rive. This tweet shows benchmarks done for the web platform. We'll extend this by using the same animations for Rive and Lottie on our React Native app.

Lottie



Lottie playing our animation at roughly17 FPS both on JS and UI threads

Rive



Rive playing our animation at roughly 60 FPS both on JS and UI threads

We benchmarked this on a Sony Xperia Z3 model using an Android profiler and Perf monitor that is shipped with the React Native app. We disabled the DEV mode so that our JS thread doesn't throttle because of it.

Both images show a tiny window right above the rocket with details of FPS on both the UI and JS thread. The results show that the Rive animation runs almost at 60FPS whereas Lottie runs at 17FPS.

Now let's focus on the right part of both images, which is the Memory consumption detailed view. If we look closely, there are mainly three portions: Java, Native, and Graphics. Java represents the memory allocated for Java or Kotlin code. The Native represents the memory allocated from C or C++ code. The graphics represent the memory used for the graphics buffer queues to display pixels on the screen. In Java, Lottie uses almost 23 MB and Rive uses almost 12 MB of RAM. In Native, Lottie uses 49 MB and Rive uses 25 MB of RAM. In Graphics, Lottie consumes 123 MB, whereas Rive uses 184 MB of RAM. The total memory consumption of Lottie is 246 MB and Rive is 276 MB.

	JAVA	NATIVE	GRAPHICS
MEMORY CONSUMPTION	Java or Kotlin code	C or C++ code	pixels on the screen
LOTTIE	23 MB	49 MB	123 MB
RIVE	12 MB	25 MB	184 MB

The results show that Rive outperforms Lottie in all departments except Graphics. The end user expects the app to run at 60FPS to enjoy a smooth user experience. If one has to do a trade-off between memory consumption and FPS, they might go with FPS as most of the devices have enough memory to exercise the app's needs.

Benefits: A reduced regression cycle while developing a feature and a happy user base

If we opt-in to a world without state machines, the developers will be implementing the logic in their code. And each time there is a change in the interactivity of the animation, devs will be required to re-work their code. This is not a good developer experience.

Rive's state machines give designers the power to think as if they were coding and structure the state machine for an animation that will interact after being given a certain input. Now the developer can use that animation and implement the interactivity firing the inputs on the animation and be done with it. If this animation needs to be changed with the same inputs, the dev only needs to replace the animation source and that's it. More info here

Almost 18.7% of people uninstall the app due to storage issues. This hurts the company's ROI. Developers should always pay attention to reducing the bundle size and the storage utilized by their app. In a tweet

by Rive's CEO, the Lottie file was around 24.37 KB and the same Rive file was around 2 KB. At the end of the day, each KB saved adds up to a reduced app size. We always want to choose a library that best fulfills our needs by providing a better developer experience, ease of API, and a smooth experience for the end user.

PART 1 | CHAPTER 8

Optimize your app's JavaScript bundle

Issue: Metro, the default JS bundler for React Native, produces a bundle that's too large.

React Native application's logic is mostly located in the JavaScript code which runs in the JavaScript engine (JavaScriptCore or Hermes). But before loading JavaScript code into the app, it should be bundled, usually into a single JS file or sometimes to multiple files. React Native provides a built-in tool for JavaScript code bundling called Metro.

Solution: Use external plugins or swap it with third-party bundlers.

Metro takes in an entry file and various options and gives you back a single JavaScript file that includes all your code and its dependencies, also known as a JavaScript bundle. According to official docs, Metro speeds up builds using a local cache of transformed modules out of the box. Metro trades configurability for performance, whereas other bundles like Webpack are the other way around. So when your project needs custom loaders, use symlinks or the proven Webpack configuration for bundling JavaScript code and splitting app logic. There are a few alternative bundlers that could be used in React Native apps and provide more configuration features. Each of them have some benefits and limitations

Re.Pack

Re.Pack is a Webpack-based toolkit to build your React Native application with full support of the Webpack ecosystem of loaders, plugins, and support for various features like symlinks, aliases, code splitting, etc. Re.Pack is the successor to Haul, which served a similar purpose but balanced a different set of tradeoffs and developer experience.
The ecosystem part of Webpack is crucial for many developers, since it's the most popular bundler of the web, making the community behind loaders and plugins its key advantage. Thanks to that pluggability, it provides ways to improve the build process and Webpack's overall

performance. At least for the parts that are not connected to the internal module graph building and processing. Such parts would be, e.g. JavaScript and TypeScript transpilation or code minification. You can replace Babel transpiler and Terser minifier with faster alternatives like ESBuild thanks to the esbuild-loader or swc with swc-loader.

Another Webpack feature that helps our apps achieve better performance is reducing the amount of code in the final bundle with tree shaking. Tree shaking is a dead code elimination technique done by analyzing the import and export statements in the source code and determining which code is actually used by the application. Webpack will then remove any unused code from the final bundle, resulting in a smaller and more efficient application. The code that's eligible for tree shaking needs to be written in ECMAScript modules (import and export statements) and mark itself as side-effect free through `package.json` "`sideEffects: false`" clause.

The very first issue on Metro bundler's GitHub is about symlink support. It remains open today, as there are various reasons Metro is blocked to introduce this functionality with the desired DX. There are ways to mitigate that particular shortcoming, but they require extra configuration. Webpack, on the other hand, as virtually any other bundler, and doesn't have this issue. Re.Pack uses the Webpack bundler under the hood so it provides symlinks functionality out-of-the-box. Which could be invaluable from the developer experience point of view of some workflows like monorepos.

Re.Pack also offers the ability to use asynchronous chunks to split your bundle into multiple files and load them on-demand. Which can improve initial loading times if you're using the JavaScriptCore engine. However, it won't provide that much value when used with Hermes, which leverages the memory mapping technique for dynamic reading only the necessary parts of the bundle's bytecode directly from the RAM. But there's a twist to that! Webpack doesn't really care whether you load the dynamic chunk from the filesystem or remote. Hence it allows for dynamic loading code that's never been there in the app bundle in the

first place. Directly from a remote server or a CDN. Now this can help you with reducing not only the initial load time, but also the precious app size.

On top of that, Webpack 5 introduced support for the concept of Module Federation. It's a functionality that allows for code-splitting and sharing the split code parts (or chunks) between independent applications. It also helps distributed and independent teams to ship large applications faster. Giving them the freedom to choose any UI framework they like and deploy independently, while still sharing the same build infrastructure. Re.Pack 3 supports this functionality out-of-the-box.

All these configurations and flexibility affect the build process. The build speed is a little bit longer than the default Metro bundler due to customization options. Also, the Fast Refresh functionality is limited compared to the Metro bundler. The Hot Module Replacement and React Refresh features require the full application reloaded with Webpack and Re.Pack, but they are supported by Metro.
If you don't need the huge customization that the Webpack ecosystem offers or don't plan to split your app code, then you may as well keep the default Metro bundler.

react-native-esbuild
One of the main benefits of react-native-esbuild is fast builds. It uses the ESBuild bundler under the hood which has huge improvements in bundling performance even without caching. It also provides some features like tree shaking and is much more configurable compared to the Metro bundler. ESBuild has its own ecosystem with plugins, custom transformers, and env variables. This loader is enabled by default for `.ts, .tsx, .mts`, and `.cts` files, which means ESBuild has built-in support for parsing TypeScript syntax and discarding the type annotations. However, ESBuild does not do any type checking so you will still need to run type check in parallel with ESBuild to check types. This is not something ESBuild does itself.
Unfortunately, react-native-esbuild has some tradeoffs, so it is very important to select the right bundler by paying attention to them as well.

It doesn't support Hermes, which could be a crucial point for some projects. And it does not have Fast Refresh or Hot Module Replacement, but this library supports live reload instead.

rnx-kit

An interesting extension to Metro is Microsoft's rnx-kit. It is a package with a huge variety of React Native development tools. It also has a custom bundler that works on top of the Metro bundler enhancing it.
As you already know, Metro does not support symlinks. rnx-kit provides the ability to fully work with symlinks. One more benefit compared to Metro is the tree shaking functionality out-of-the-box.
Metro supports TypeScript source files, but it only transpiles them to JavaScript. Metro does not do any type-checking. rnx-kit solves this problem. Through the configuration, you can enable type-checking. Warnings and errors from TypeScript appear on the console.
Also, rnx-kit provides duplicate dependencies and cyclic dependencies detection out-of-the-box. This could be very useful to reduce the size of the bundle which leads to better performance and prevents cyclic dependencies issues.

Benefits: Ship less JavaScript to your users. Save developers' time when bundling.

The choice of a bundle tool depends on the specific case. It is impossible to select only one bundler for all the apps. If you need customization options provided by the Webpack ecosystem or plan to split your app code, then we would suggest using Re.Pack for its widely customizable configuration, a huge amount of loaders, plugins maintained by the community, and a lot of features like symlinks, aliases, etc. compared to other bundle tools. If the Webpack ecosystem feels overhead, then it is better to stay with the default Metro bundler or try to use other bundler options like react-native-esbuild and rnx-kit which also provides some benefits like decreased build time, using esbuild under the hood, symlinks, and typescript support out-of-the-box. But be careful and always pay attention to the tradeoffs that come with a new bundling system.

If you need help with performance, stability, user experience, or other complex issues - contact us!

As React Native Core Contributors and leaders of the community, we will be happy to help.

PART 2

Improve performance by using the latest React Native features.

React Native is growing fast and so is the number of features

Last year, developers contributed more than 3670 commits to the React Native core. The number may seem impressive, but, in fact, it's even larger, since it doesn't include the smaller contributions made under the React Native Community organization (9678 commits).

All that proves that React Native is developing at a really healthy pace. Contributions made by both the community and Meta enable more and more advanced use cases of the framework. A great example of that is Hermes – an entirely new JavaScript engine built and designed specifically for React Native and Android. Hermes aims to replace the JavaScriptCore, previously used on both Android and iOS. It also brings a lot of enterprise-grade optimizations by improving your Android application's performance, start-up time, and overall size reduction.

In this section, we will show you some of the features you can turn on right now to start your optimization process. We also encourage you to keep track of all the new React Native features to make sure you use the framework to its full potential.

PART 2 | CHAPTER 1

Always run the latest React Native version to access the new features

Upgrade your app to the latest version to get more features and better support.

Issue: You are running an old and unsupported version of React Native and depriving yourself of new improvements and features

Keeping your application up to speed with the frameworks you use is crucial. That is why you should subscribe to the latest features, performance improvements, and security fixes.
The JavaScript ecosystem is particularly interesting in this aspect, as it moves really quickly. If you don't update your app regularly, chances are your code will end up being so far behind that upgrading it will become painful and risky.

Every day, developers from all around the world introduce new features, critical bug fixes, and security patches. On average, each release includes around 500 commits.

Over the years, React Native has grown significantly, thanks to open-source contributors and Meta's dedication to improving the ecosystem. Here are some highlighted crucial features that have been introduced to React Native over the course of its releases:

Fast Refresh

To improve the developer experience and velocity, the React team introduced a feature called Fast Refresh to React Native. This lets you quickly reflect the code on the device, whenever you save the file instead of building or reloading the app. It is smart enough to decide when to do a reload after we fix an error or just render otherwise.
A tip here: the local state of functional components and hooks is preserved by default. We can override this by adding a comment to that

file: // `@refresh reset`. Whereas, class components are remounted without preserving the state.

Auto-Linking

Whenever we add native code to our React Native app as a dependency, it needs to be linked. Previously linking was done manually or using react-native link dep-name. React Native CLI introduced auto-linking so the devs didn't need to link a library themselves. After a couple of years, when the re-architecture of React Native was released to the community, a need arose to auto link fabric components and turbo modules, which was handled gracefully by the CLI team. They released an update to the community to help the developer experience and velocity.

Flipper

Introduced as a recommended way of debugging React Native apps, Flipper is loaded with awesome tools such as ReactDevtools, Network Inspector, Native Layout Inspector, and others. We can also view the Metro and Device logs right in Flipper. The community has appreciated such a feature, and one example of that is a performance plugin that we can install in Flipper to measure the performance of the React Native apps.

LogBox

React Native redesigned its error and warning handling system. They had to do a ground-up redesign of its logging system and the developer experience is much better because of it. Developers can easily trace the cause of an error using code frames and component stacks. Syntax error formatting helps to understand the issue more quickly with the aid of syntax highlighting. Log Notifications show warnings and logs at the bottom of the screen instead of covering the whole screen.

Hermes

A new JS engine created by Meta to improve the performance of React Native apps in terms of CPU usage, memory consumption, app size, and Time To Interactive (TTI). Initial support was launched for Android

devices, but after two years, support was extended to iOS as well. After a couple of months, the previously used garbage collector for Hermes GenGC was replaced with a new one called Hades - a concurrent garbage collector. The Meta team saw improvements of CPU-intensive workloads by 20-50%. Later on, the team decided to ship a bundled Hermes instead of downloading it from NPM. This was done to avoid confusion between what version of Hermes is compatible with React Native. Also, both Hermes and React Native use the same JSI code which makes it hard to maintain. Now whenever a version of React Native is released, a version of Hermes can be released as well, making sure that both are fully compatible.

New Architecture

This one has its own chapter.

In the React Native ecosystem, it's common that libraries are not backward- compatible. New features often use goodies not available in the previous versions of the framework. This means that if your application runs on an older version of React Native, you are eventually going to start missing out on the latest improvements.

@react-native-community/cli	react-native
^10.0.0	^0.71
^9.0.0	^0.70
^8.0.0	^0.69
^7.0.0	^0.64
^6.0.0	^0.68
^5.0.0	^0.65,^0.66,^0.67
^4.0.0	^0.62, ^0.63
^3.0.0	^0.61
^2.0.0	^0.60
^1.0.0	^0.59

That's why keeping up with the newest React Native upgrades are the only way to go.

Unfortunately, there is some serious work associated with upgrading your React Native code with every new release. Its amount will depend on the number of underlying changes to the native functionalities and core pieces. Most of the time, you have to carefully analyze and compare your project against the latest version and make the adjustments on your own. This task is easier if you're already comfortable with moving around the native environment. But if you're like most of us, it might be a bit more challenging.

For instance, it may turn out that the modules and components you used in your code are no longer part of the react-native core.

It would be because of the changes introduced by Meta during a process called the *LEAN CORE* link. The goals of the effort were to:

- Make the react-native package smaller, more flexible, and easier to maintain by extracting some parts of the core and moving them to the react-native-community repository,
- Transfer the maintenance of the extracted modules to the community.

The process accelerated the growth of particular modules and made the whole ecosystem better organized. But it also had some negative effects on the react-native upgrading experience. Now, you have to install the extracted packages as an additional dependency, and until you do, your app will not compile or crash at runtime.
However, from a developer's perspective, the migration to community packages is usually nothing more than introducing a new dependency and rewriting imports.

Another important issue is the support of third-parties. Your code usually relies on external libraries and there's a risk that they might also be incompatible with the latest React Native version.

There are at least two ways to solve this problem:

- Wait for the project maintainers to perform the necessary adjustments before you upgrade.
- Look for alternatives or patch the modules yourself – by using a handy utility called patch-package or creating a temporary fork with the necessary fixes.

Running on an old version means shipping with issues that may discourage your users

If you are running on an older version, it is likely that you are lagging behind your competition that uses the latest versions of the framework.

The number of fixes, improvements, and advancements in the React Native framework is really impressive. If you're playing a game of catch up, you are opting out of a lot of updates that would make your life a lot easier. The workload and the cost involved in making regular upgrades are always offset by the immediate DX enhancements.

In this section, we present some of the well-established practices to make upgrading React Native to the newer version easier.

Solution: Upgrade to the latest version of React Native (we'll show you how).

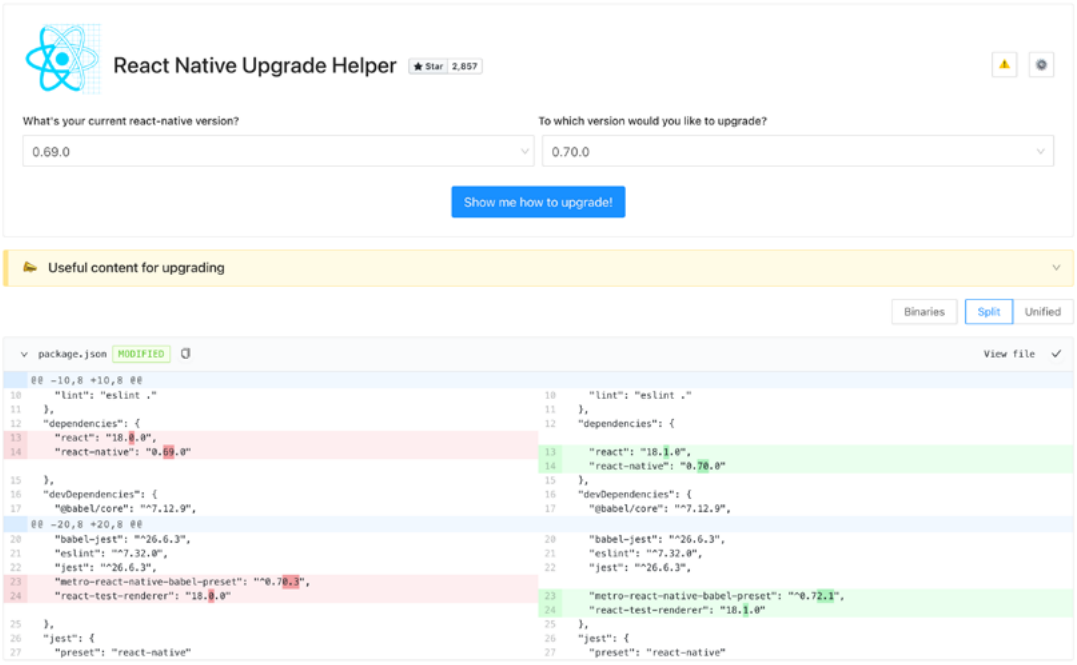
Upgrading React Native might not be the easiest thing in the world. But there are tools that can simplify the process and take most of the problems away.

The actual amount of work will depend on the number of changes and your base version. However, the steps presented in this section can be applied to every upgrade, regardless of the state of your application.

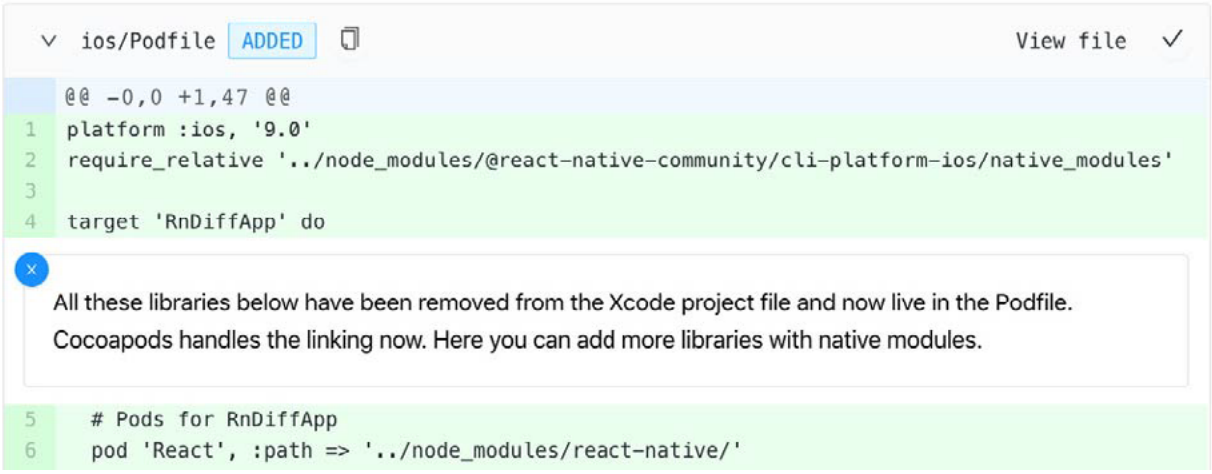
Preparing for the upgrade

React Native Upgrade Helper is a good place to start. On a high level, it gives you an overview of the changes that have happened to React

Native since the last time you upgraded your local version.



To do so, the helper compares bare React Native projects created by running npx `react-native init` with your version and the one you're upgrading to. Next, it shows the differences between the projects, making you aware of every little modification that took place in the meantime. Some changes may be additionally annotated with special information that will give more context on why something has happened.

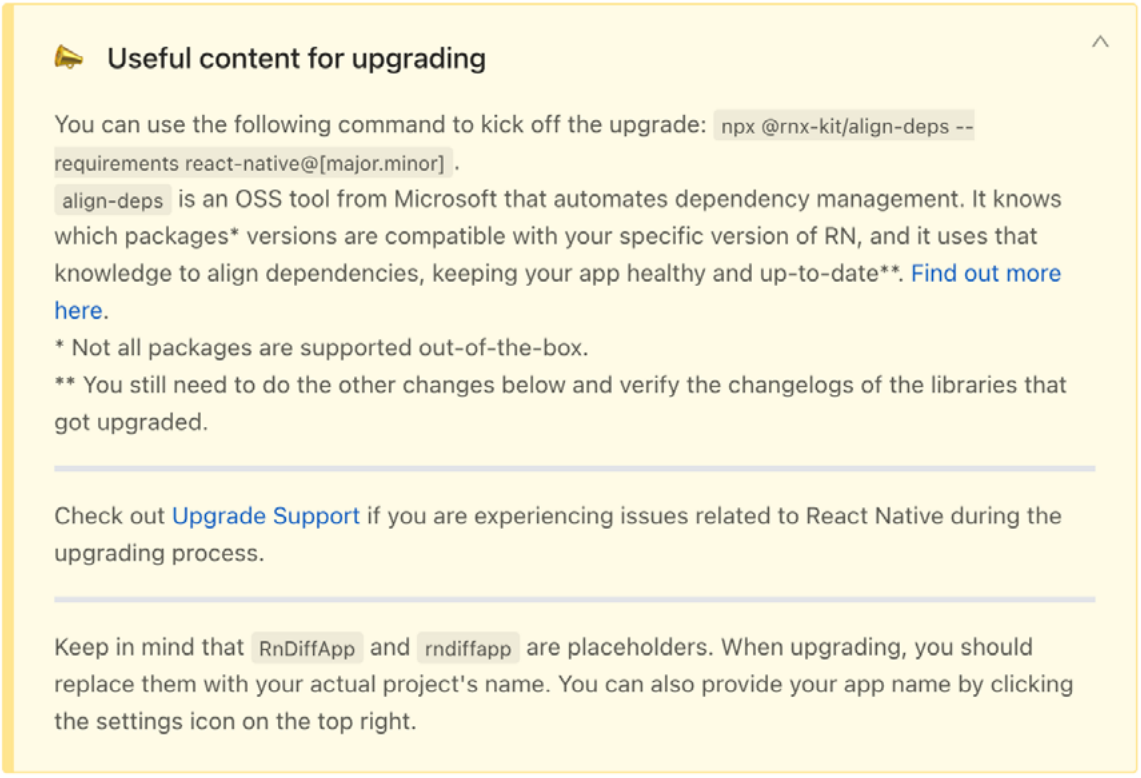


Additional explanation of the more interesting changes to user files

Having a better overview of the changes will help you move faster and act with more confidence.

Note: Having more context is really important as there is no automation in place when it comes to upgrading - you will have to apply the changes yourself.

React Native Upgrade Helper also suggests useful content to read while upgrading. In most cases that includes a dedicated blog post published on the React Native blog as well as the raw changelog.



Useful content to read while upgrading React Native to a newer version

We advise you to read the recommended resources to get a better grip on the upcoming release and learn about its highlights.

Thanks to that, you will not only be aware of the changes, but you will also understand the reasoning behind them. And you will be ready to open up your project and start working on it.

Applying the JavaScript changes

The process of upgrading the JavaScript part of React Native is similar to upgrading other JavaScript frameworks. Our recommendation here is to perform upgrades step-by-step - bumping one library at a time. As a rule of thumb, once you have upgraded a library, save your work at that point in a commit and then move on to the next library. In our opinion, this approach is better than upgrading everything at once as it gives you more control and makes catching regressions much easier.

The first step is to bump the React and React Native dependencies to the desired versions and perform the necessary changes (including breaking changes). To do so, you can look up the suggestions provided by React Native Upgrade Helper and apply them manually. Once it's completed, make sure to reinstall your `node_modules`.

Note: When performing the upgrade, you may see a lot of changes coming from iOS project files (everything inside .xcodeproj, including .pbxproj). These are files generated by Xcode as you work with your iOS part of React Native application. Instead of modifying the source file, it is better to perform the changes via the Xcode UI. This was the case with upgrading to React Native 0.60 and the appropriate operations were described in this issue.

Finally, you should try running the application. If everything is working - perfect. The upgrade was smooth and you can call it a day! On a more serious note though – now you should check if there are newer versions of other dependencies you use! They may be shipping important performance improvements.

Unfortunately, there's also another more pessimistic scenario. Your app may not build at all or may instantly crash with a red screen. In that case, it is very likely that some of your third-party dependencies

are not working properly, as in some cases the dependencies include native code which supports new OS features, so you need to make them compatible with your React Native version.

Note: If you have a problem with your upgrades, you can check the Upgrade Support project. It is a repository where developers share their experience and help each other solve some of the most challenging operations related to upgrading.

Upgrading third-party libraries

In most cases, it's your React Native dependencies that you should look at first. Unlike regular JavaScript/React packages, they often depend on native build systems and more advanced React Native APIs. This exposes them to potential errors as the framework matures into a more stable API.

If the error occurs during the build time, bumping the dependency to its latest version usually makes it work. But it may not always be the case. To make sure the version of React Native you're upgrading to is compatible with your dependencies, use the align-deps project by Microsoft developers. It allows you to keep your dependencies on the right version based on the requirements and by leveraging the presets of rules. It also has a CLI, so you can wire it up to your CI and ensure that no one in your repo or monorepo will inadvertently introduce incompatible versions of packages and break the app.

Once your application builds, you are ready to check the changelog and make yourself familiar with the JavaScript changes that happened to the public API. If you overlook this step, it can result in runtime exceptions. Using Flow or TypeScript should help you ensure that the changes were applied properly.

As you can see, there is no magic trick that would fix all the errors and

upgrade the dependencies automatically. This is mostly manual work that has to be done with patience and attention. It also requires a lot of testing to ensure that you didn't break any features along the way. Fortunately, there are tools like align-deps that help you avoid at least some of the manual work, improving the upgrading experience significantly.

Benefits: You're running the latest versions which translates to more features and better support.

Upgrading to the latest React Native version shouldn't be different from keeping your other frameworks and libraries up to date. Apart from critical performance and security improvements, new React Native releases also address the latest underlying changes to iOS and Android. That includes the breaking changes that apply to mobile phones, such as when certain APIs get deprecated.

Here is an example: In 2019, Google announced that all Android applications submitted to Google Play after August 1, 2019 had to be 64-bit. In order to continue developing your application and shipping new features, you had to upgrade to React Native 0.59 and perform the necessary adjustments.

Upgrades like this are really critical to keeping your users satisfied. After all, they would be disappointed if the app started to crash with the newer version of the operating system or disappeared from the App Store. There might be some additional workload associated with every release, but staying up to date will pay back with happier users, more stable apps, and a better development experience.

PART 2 | CHAPTER 2

How to debug faster and better with Flipper

Establish a better feedback loop by implementing Flipper and have more fun while working on your apps.

Issue: You're using Chrome Debugger or some other hacky way to debug and profile your React Native application.

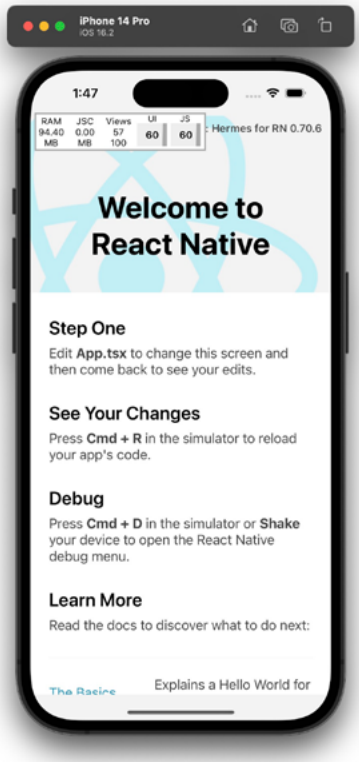
Debugging is one of the more challenging parts of every developer's daily work and finding out what is wrong can be very frustrating. We usually try to fix bugs as soon as possible, especially when they are critical and make an app unfunctional. Time is an important factor in that process and we usually have to be agile to quickly solve the issues. However, debugging React Native is not very straightforward, as the issue you are trying to solve may occur on different levels. Namely, it may be caused by:

- JavaScript - your application's code or React Native,
- Native code - third-party libraries or React Native itself.

When it comes to debugging native code, you have to use the tools built into Android Studio and Xcode.

When it comes to debugging JavaScript code, you may encounter several difficulties. The first and most naive way to debug is to write console.logs in your code and check the logs in the terminal. This method works for solving trivial bugs only or when following the divide and conquer technique. In all other cases, you may need to use an external debugger.

By default, React Native ships with built-in debugging utilities.



The most common one is Google Chrome Debugger. It allows you to set breakpoints in your code or preview logs in a handier way than in a terminal. Unfortunately, using the
Chrome Debugger may lead to hard-to-spot issues. It's because your code is executed in Chrome's V8 engine instead of a platform-specific engine, such as JSC or Hermes.

The instructions generated in Chrome are sent via Websocket to the emulator or device. It means that you cannot really use the debugger to profile your app so it detects the performance issues. It can give you a rough idea of what might cause the issues, but
you will not be able to debug the real cause due to the overhead of WebSocket message passing.

Another inconvenience is the fact that you cannot easily debug network requests with the Chrome Debugger (it needs additional setup and still has its limitations). In order to debug all possible requests, you have to open a dedicated network debugger using the emulator's developer menu. However, its interface is very small and inconvenient due to the size of the emulator's screen.

From the developer menu, you can access other debugging utilities, such as layout inspector or performance monitor. The latter is relatively convenient to use, as it's displaying only a small piece of information. However, employing the former is a struggle because of the limited workspace it provides.

Spending more time on debugging and finding performance issues means a worse developer experience and less satisfaction

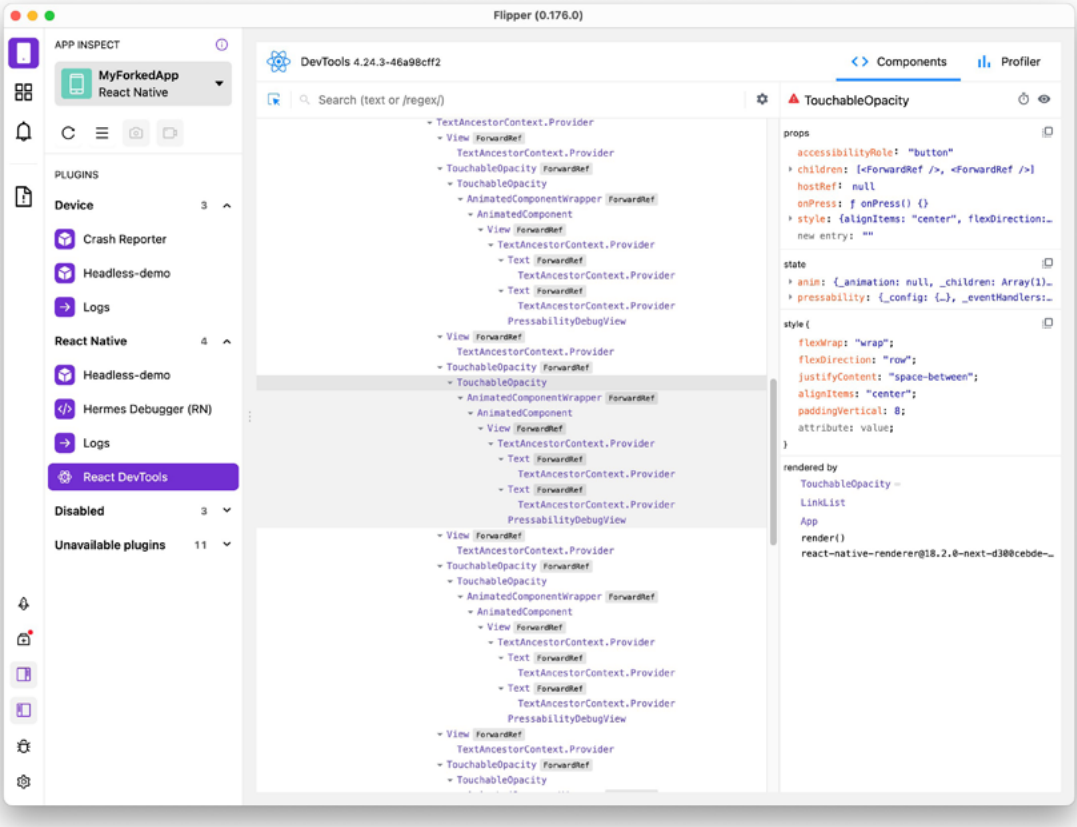
Unlike native developers, the ones working with React Native have access to a wide range of debugging tools and techniques. Each originates from a different ecosystem, such as iOS, Android, or JS. While it may sound great at first, you need to remember that every tool requires a different level of expertise in the native development. That makes the choice challenging for the vast majority of JavaScript developers.

Inconvenient tooling usually decreases the velocity of the team and frustrates its members. As a result, they are not as effective as they could be, affecting the quality of the app and making the releases less frequent.

Solution: Turn on Flipper and start debugging.

Wouldn't it be great to have one comprehensive tool to handle all of the above use cases? Of course, it would! And that's where Flipper comes into play!

Flipper is a debugging platform for mobile apps. It also supports React Native as its first-class citizen. Launched in September 2019, Flipper has been shipped by default with React Native since version 0.62.



Source: https://fbflipper.com/docs/features/react-native

It is a desktop app with a convenient interface, which directly integrates with your application's JS and native code. This means that you no longer have to worry about JS runtime differences and the performance caveats of using the Chrome Debugger. It comes with a network inspector, React DevTools, and even a native view hierarchy tool.

What's more, Flipper lets you preview logs from native code and track native crashes, so you don't have to run Android Studio or Xcode to check what is happening on the native side!

Flipper is easily extensible, so there is a high chance it will be enriched with a wide range of useful plugins developed by the community. At this point, you can use Flipper for tasks such as detecting memory leaks, previewing the content of Shared Preferences, or inspecting loaded images. Additionally, Flipper for React Native is shipped with React

DevTools, Hermes debugger, and Metro bundler integration.

What's most exciting is that all the needed utilities are placed in one desktop app. This minimizes context switches. Without Flipper, a developer debugging an issue related to displaying the data fetched from the backend had to use the Chrome Debugger (to preview logs), in-emulator network requests debugger, and probably in-emulator layout inspector, or a standalone React Devtools app. With Flipper, all those tools are available as built-in plugins. They are easily accessible from a side panel and have similar UI and UX.

Benefits: You have more fun working with React Native and establish a better feedback loop.

A better debugging process makes your app development cycle faster and more predictable. As a result, your team is able to produce more reliable code and spot any kind of issues much easier.

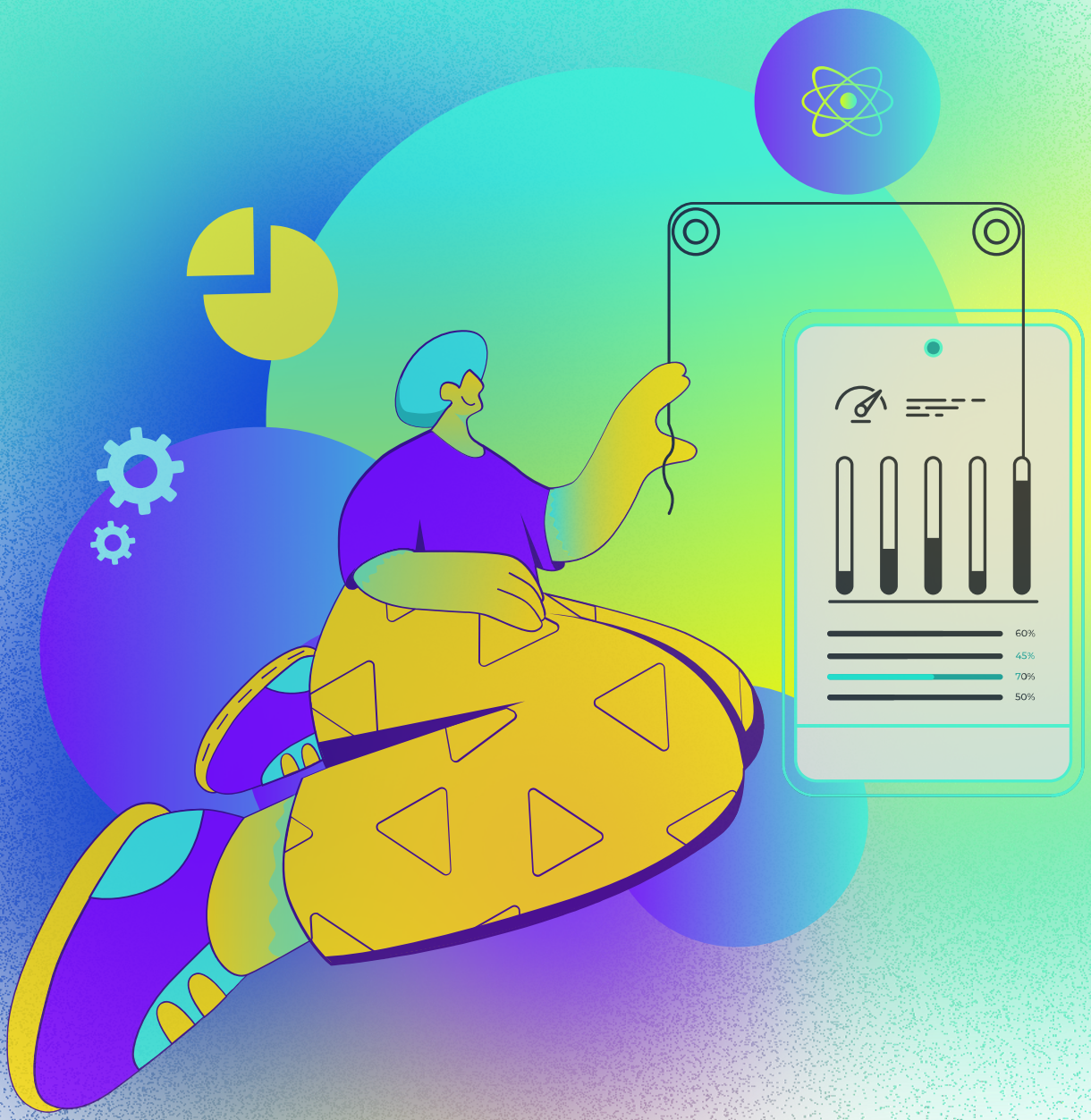
Having all the debugging utilities in one interface is definitely ergonomic and does not disrupt any interactions with an emulator or device. The process will be less burdensome for your team and that will positively impact the velocity of the product development and bug fixing.

"Feel like a functionality is missing in Flipper? Good news! Flipper is easily extensible and has a comprehensive guide on how to write custom plugins in React. Why not build your own?"

Alexandre Moureaux
App performance specialist at BAM

PART 2 | CHAPTER 3

Avoid unused native dependencies



Improve the Time to Interactive of your app by removing the linking of unused dependencies.

Issue: You have a lot of dependencies in your project but you don't know if you need all of them

Every bit of native code we use in our apps has a runtime cost asociated with reading, loading, and executing said code. The more native dependencies our apps have, the slower it is for apps to start, which impacts the TTI (Time to Interactive) metric. And makes your users wait more to start interacting with your app.

In our React Native apps, we often rely on dependencies that load Kotlin, Java, Swift, ObjectiveC, JavaScript, and recently more often, even C++. Those dependencies are declared in the package.json file, which allows for a JavaScript bundler to correctly discover and, well, bundle their JS parts into the final application. It may be counterintuitive at first, but this declaration in the JavaScript toolchain influences the native side as well. And the reason for that is the "autolinking" feature of the React Native CLI.

Autolinking allows us to link native dependencies in React Native automatically, without ever touching native toolings like Cocoapods and Gradle. If you're not familiar with how the Android or iOS toolchain works in terms of using external dependencies, you might be asking "What's linking native dependencies?" When dealing with native binaries, be it either in C/C++, ObjectiveC, or Java, linking is a way for the native toolchain to understand where to find the actual code that's associated with the 3rd-party dependency we want our code to use. What's important is that it's necessary and for a long time we, React Native developers, needed to do this step manually. Since React Native 0.60, we have an automated way of doing this thanks to the React Native CLI.

One important thing to know about how autolinking works is that it

crawls your package.json and then node_modules in search of native code. The tool doesn't know whether you're actively using the library that ships native code or not. It will be linked anyway. What does this mean? All the native dependencies will be linked and available in our app bundle. As a result, we'll end up with an increased bundle size and degraded TTI, as the mobile OS will spend more time loading the native binaries, showing your users a splash screen a bit longer.

Solution: Find and remove unused dependencies.

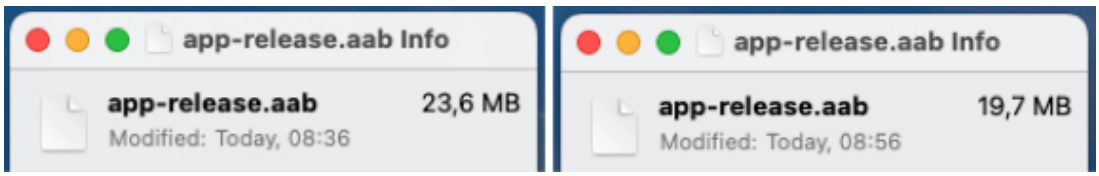
To find the unused dependencies in our project, we can use the depcheck library. It's a great tool for analyzing the project's dependencies to see how each one of them is used, which dependencies are useless, and which dependencies are missing from `package.json`. To use depcheck, we need to run `npx depcheck` in the root of our project. An example of the results looks like this:

```
 Unused dependencies
* lottie-react-native
* react-native-gesture-handler
* react-native-maps
* react-natave-reanimated
* react-native-video
* react-native-webview

 Unused devDependencies
* @babel/core
* @babel/runtime
* @react-native-community/eslint-config
* @tsconfig/react-native
* @types/jest
* @types/react-test-renderer
* babel-jest
* jest-circus
* metro-react-native-paper-preset
* typescript
```

Example output of the depcheck library

Dev dependencies likely won't end up in the JS bundle unless there is any native code inside, so in this example, there is no need to focus on them. But the results can show us that we have a few unused dependencies - and what's more important, in this case, these dependencies

are relying on some native code. Now we have to remove them and it's done! In the example app, removing unused dependencies from the screenshot above occurred with the following result in the bundle size:

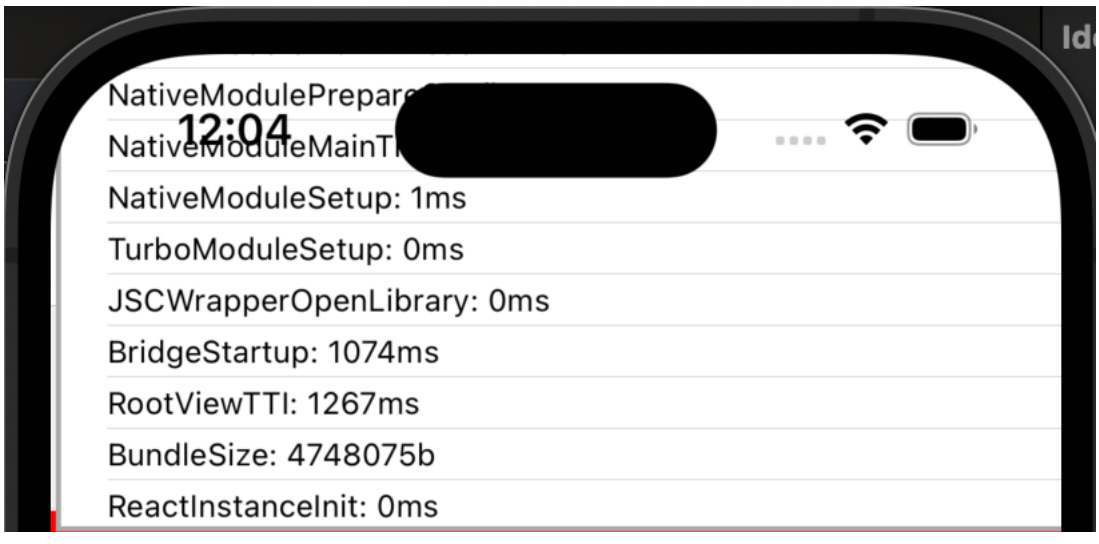


Comparision of bundle sizes before and after removing unused native dependencies

It's worth mentioning that on the Android device, there was a noticeable improvement in the Time to Interactive, which was reduced by 17% in this case.
You may be wondering how we can measure the TTI. There are a few ways to do it. Whichever you choose, remember to always measure on a release version of the app when dealing with absolute numbers.

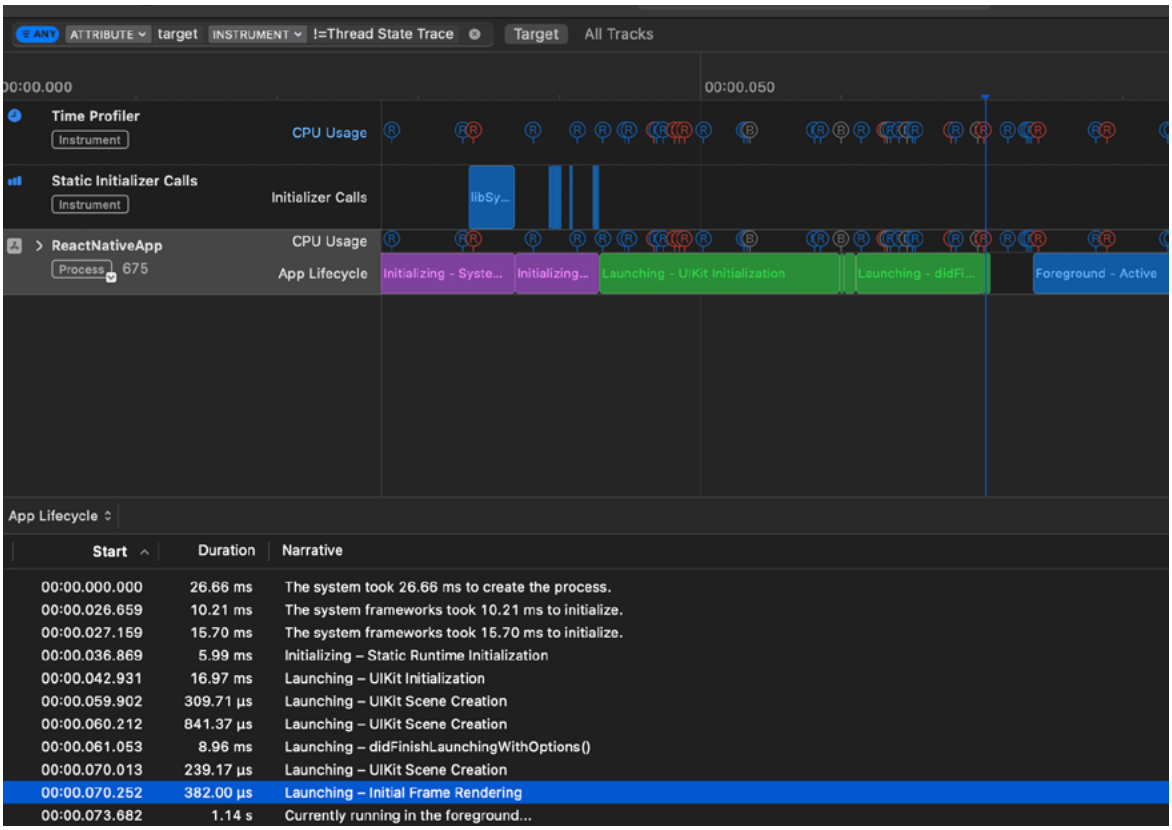
One way is to use a stopwatch and measure the time the app took to show the first screen, as shown here. It's entirely manual, but it will often get the job done for one-off measurements.

If we want to get a more detailed output, we can make use of Perfetto for Android. For iOS, we can enable Perf Monitor from the DevMenu and double-tap on the monitor window to expand. The output will look like this:



Performance monitor on the iOS simulator

We can also use App launch from Xcode instruments. All you need is to install a release build through profiling to your real device. Then select App Launch from the window that will appear automatically once the build is installed. Hit the record button, and once the app has launched,



Example usage of Xcode's App Launch tool

stop recording. You will get an output similar to this:
There are two phases when calculating app launch time on iOS. The first one is called pre-main time, and it's the time before the main function of the app is executed. It's marked with the purple area on the graph above - all the work needed to launch the app correctly, like initialization and the linking of libraries happens in this phase.

The second phase, called post-main-time, is the time between executing the app's main function and presenting the first interactable view to the user. It's marked with the green color on the graph above. The total app launch time is the sum of both of these metrics. If you want to learn more about testing app launch time, here's a good read on this topic.

It's worth mentioning that there are lots of third-party tools helping developers to gain a bunch of performance information from apps already submitted to Google Play and App Store. The most popular are Firebase Performance Monitoring, Sentry, and DataDog. The key advantage of using one of these tools is gaining data about performance from many different devices.

Benefits: A smaller bundle size and faster Time to Interactive.

Removing a few unused native dependencies ended up reducing both the size of the app bundle and TTI by around 17%. Providing only resources needed by the app can improve the Time to Interactive metric, making users less likely to uninstall your app from their devices due to excessive load time.

It's worth remembering that although autolinking is a great and powerful feature, it can be overzealous when it comes to linking code our app doesn't really use. Make sure to keep your dependencies up to date and clean up unused ones during refactorings.

> **"There are so many tricky parts to making a great native app, and to lower the barrier to entry, React Native can abstract away things that you might want to come back and check on later once you've got your app up and running – this ebook does a solid job of helping you understand how to really get from good to great."**
>
> **Orta Therox**
> CocoaPods creator, TypeScript core contributor

PART 2 | CHAPTER 4

Optimize your application startup time with Hermes

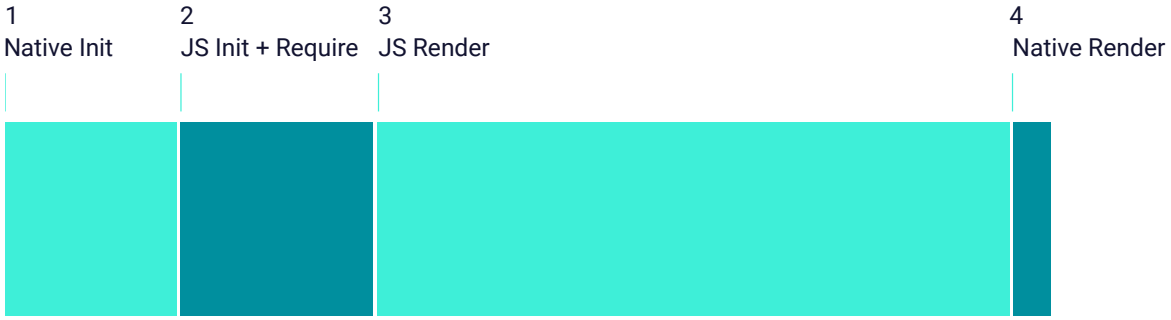
Achieve a better performance of your apps with Hermes.

Issue: You're loading a lot of Android packages during the startup time which is unnecessary. Also, you're using an engine that is not optimized for mobile.

Users expect applications to be responsive and load fast. Apps that fail to meet these requirements can end up receiving bad ratings in the App Store or Play Store. In the most extreme situations, they can even get abandoned in favor of their competition.

There is no single definition of the startup time. It's because there are many different stages of the loading phase that can affect how "fast" or "slow" the app feels. For example, in the Lighthouse report, there are eight performance metrics used to profile your web application. One of them is Time to Interactive (TTI), which measures the time until the application is ready for the first interaction.

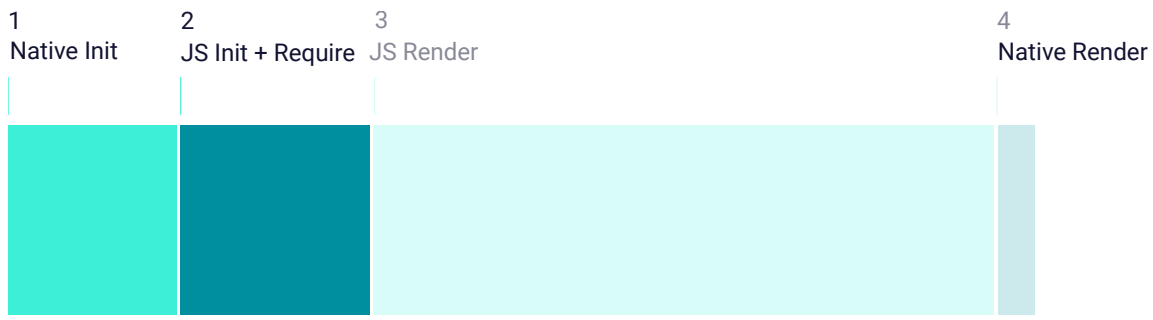
There are quite a few things that happen from the moment you press the application icon from the drawer for the first time.



The loading process starts with a native initialization which loads the JavaScript VM and initializes all the native modules (1 in the above diagram). It then continues to read the JavaScript from the disk and loads it into the memory, parses, and starts executing (2 in the above diagram). The details of this operation were discussed earlier in the section about choosing the right libraries for your application.

In the next step, React Native starts loading React components and sends the final set of instructions to the UIManager (3 in the above diagram). Finally, the UIManager processes the information received from JavaScript and starts executing the native instructions that will result in the final native interface (4 in the above diagram).

As you can see in the diagram below, there are two groups of operations that influence the overall startup time of your application.



The first one involves the first two operations (1 and 2 in the diagram above) and describes the time needed for React Native to bootstrap (to spin up the VM and for the VM to execute the JavaScript code). The other one includes the remaining operations (3 and 4 in the diagram above) and is associated with the business logic that you have created for your application. The length of this group is highly dependent on the number of components and the overall complexity of your application.

This section focuses on the first group – the improvements related to your configuration and not the business logic itself.

If you have not measured the overall startup time of your application or have not played around with things such as Hermes yet - keep on reading.

Long startup times and slow UX can be one of the reasons your app gets a bad rating and ends up being abandoned.

Creating applications that are fun to play with is extremely important, especially considering how saturated the mobile market already is. Now, all mobile apps have to be not only intuitive, they also should be pleasant to interact with.

There is a common misconception that React Native applications come with a performance trade-off compared to their native counterparts. The truth is that with enough attention and configuration tweaks, they can load just as fast and without any considerable difference.

Solution: Turn on Hermes to benefit from a better performance.

While a React Native application takes care of a native interface, it still requires JavaScript logic to be running at runtime. To do so, it spins off its own JavaScript virtual machine. Until recently, it used JavaScript-Core (JSC). This engine is a part of WebKit—which powers the Safari browser—and by default is only available on iOS. For a long time, it made sense for React Native to use JSC for running JavaScript on Android as well. It's because using the V8 engine (that ships with Chrome) could potentially increase the differences between Android and iOS, and make sharing the code between the platforms way more difficult.

JavaScript engines need to perform various complicated operations. They constantly ship new heuristics to improve the overall performance, including the time needed to load the code and then execute it. To do so, they benchmark common JavaScript operations and challenge the CPU and memory needed to complete this process.
Most of the work of developers handling the JavaScript engines is being tested against the most popular websites, such as Facebook or Twitter. It is not a surprise that React Native uses JavaScript in a different way. For example, the JavaScript engine made for the web doesn't have to worry much about the startup time. The browser will most likely already be running at the time of loading a page. Because of that, the engine can shift its attention to the overall CPU and memory consumption, as web applications can perform a lot of complex operations and computations, including 3D graphics.

As you could see on the performance diagram presented in the previous section, the JavaScript virtual machine consumes a big chunk of the app's total loading time. Unfortunately, there is little you can do about it unless you build your own engine. That's what the Meta team ended up doing.

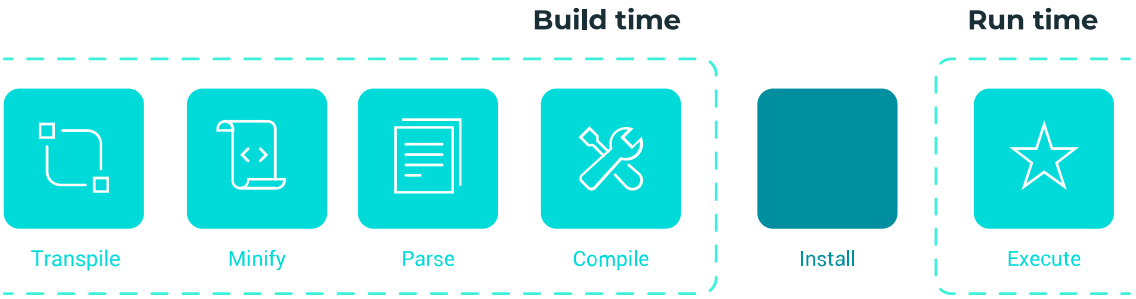
Meet Hermes - a JavaScript engine made specifically with React Native in mind. It is optimized for mobile and focuses on relatively CPU-insensitive metrics, such as application size and memory consumption. Chances are you're already using it! As of v0.70, React Native has been shipping with Hermes turned on by default. Which marks an important milestone in the engine's stability.
It's come a long way from the bare-bones Android-only engine open-sourced in 2019, with a carefully curated set of supported JS features—due to size constraints—through finding low-size-footprint ways of adding more EcmaScript spec features, like Proxies and Intl, until making it available for macOS and iOS.

Today Hermes is still small enough (~2 MB) to provide significant improvements to apps' TTI and gives us a set of features rich enough to be used in most of the apps out there.

Before we go into the details of enabling Hermes in existing React Native applications, let's take a look at some of its key architectural decisions.

Bytecode precompilation

Typically, the traditional JavaScript VM works by parsing the JavaScript source code during the runtime and then producing the bytecode. As a result, the execution of the code is delayed until the parsing completes. It is not the same with Hermes. To reduce the time needed for the engine to execute the business logic, it generates the bytecode during the build time.



It can spend more time optimizing the bundle using various techniques to make it smaller and more efficient. For example, the generated bytecode is designed in a way so that it can be mapped in the memory without eagerly loading the entire
file. Optimizing that process brings significant TTI improvements as I/O operations on mobile devices tend to increase the overall latency.

No JIT

The majority of modern browser engines use just-in-time (JIT) compilers. It means that the code is translated and executed line-by-line. However, the JIT compiler keeps track of warm code segments (the ones that appear a few times) and hot code segments (the ones that run many times). These frequently occurring code segments are then sent to a compiler that, depending on how many times they appear in the program, compiles them to the machine code and, optionally, performs some optimizations.

Hermes, unlike the other engines, is an AOT(ahead-of-time) engine. It means that the entire bundle is compiled to bytecode ahead of time. As a result, certain optimizations that JIT compilers would perform on hot code segments are not present.

On one hand, it makes the Hermes bundles underperform in benchmarks that are CPU-oriented. However, these benchmarks are not really comparable to a real-life mobile app experience, where TTI and application size takes priority.

On the other hand, JIT engines decrease the TTI as they need time to parse the bundle and execute it in time. They also need time to "warm up". Namely, they have to run the code a couple of times to detect the common patterns and begin to optimize them.

If you want to start using Hermes on Android, make sure to turn the enableHermes flag in android/app/build.gradle to true:

```
project.ext.react = [
  entryFile: "index.js",
  enableHermes: true
]
```

For iOS, turn the `hermes_enabled` flag to true in ios/Podfile:

```
use_react_native!(
    :path => config[:reactNativePath],
    # to enable hermes on iOS, change `false` to `true` and
then install pods
    :hermes_enabled => true
)
```

In both cases, whenever you switch the Hermes flag, make sure to re-build the project according to instructions provided in the native files. Once your project is rebuilt, you can now enjoy a faster app boot time and likely smaller app size.

Benefits: A better startup time leads to a better performance. It's a never-ending story.

Making your application load fast is an ongoing effort and its final result will depend on many factors. You can control some of them by tweaking both your application's configuration and the tools it uses to compile the source code.

Turning Hermes on is one of the things that you can do today to

drastically improve certain performance metrics of your app, mainly the TTI.

Apart from that, you can also look into other significant improvements shipped by the Meta team. To do so, get familiar with their write-up on React Native performance. It is often a game of gradual improvements that make all the difference when applied at once. The React Native core team has created a visual report on benchmarking between stock RN and Hermes-enabled RN: see here.

As we have mentioned in the section on running the latest React Native, Hermes is one of those assets that you can leverage as long as you stay up to date with your React Native version.

Doing so will help your application stay on top of the performance game and let it run at a maximum speed.

PART 2 | CHAPTER 5

Optimize your Android application's size with these Gradle settings

Improve TTI and reduce memory usage and the size of your app by adjusting Proguard rules to your projects.

Issue: You are not enabling Proguard for release builds and creating APK with code for all CPU architectures. You ship a larger APK.

At the beginning of each React Native project, you usually don't care about the application size. After all, it is hard to make such predictions so early in the process. But it takes only a few additional dependencies for the application to grow from a standard 5 MB to 10, 20, or even 50 MB, depending on the codebase.

Should you really care about app size in the era of super-fast mobile internet and WiFi access everywhere? Why does a bundle size grow so rapidly? We will answer those questions in this section. But first, let's have a look at what a typical React Native bundle is made of.

By default, a React Native application on Android consists of:

- four sets of binaries compiled for different CPU architectures,
- a directory with resources such as images, fonts, etc.,
- a JavaScript bundle with business logic and your React components,
- other files.

React Native offers some optimizations that allow you to improve the structure of the bundle and its overall size. But they are disabled by default.

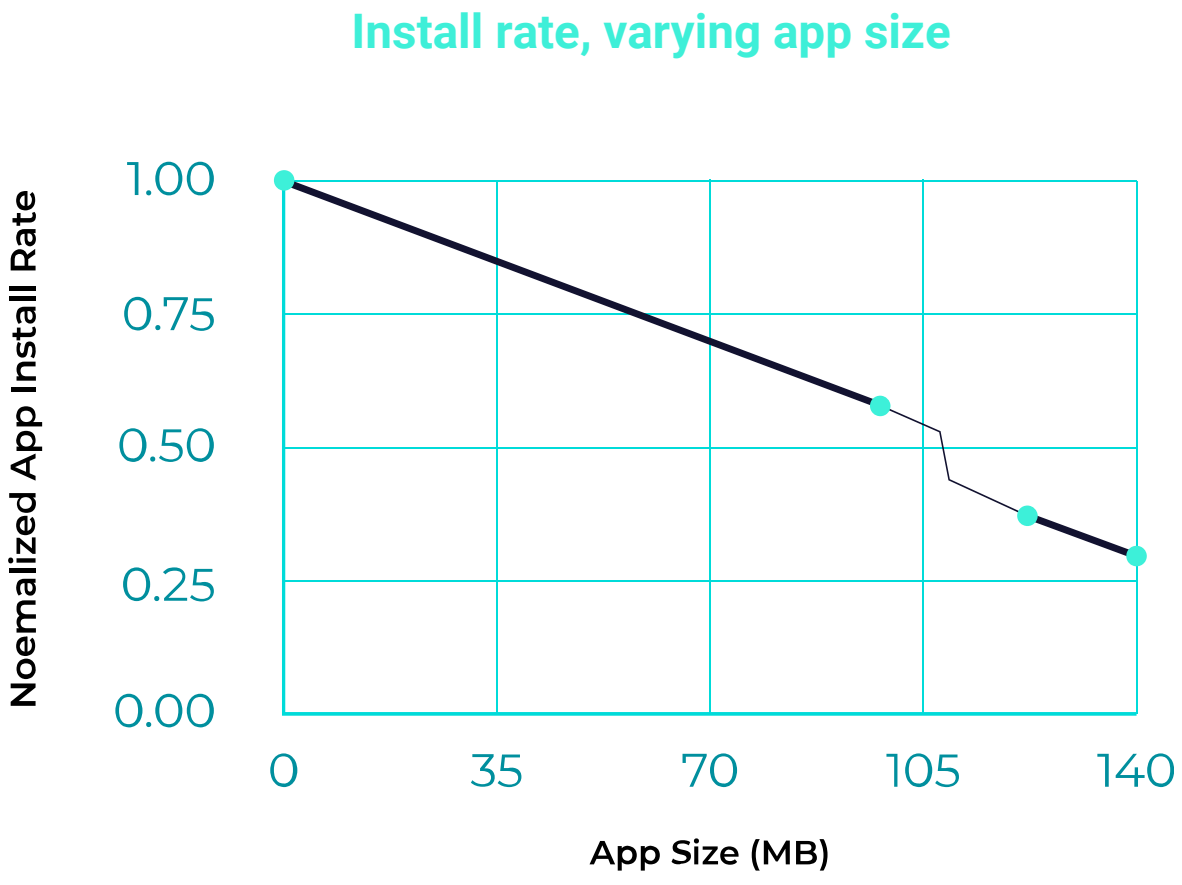
If you are not using them effectively, especially when your application grows, you are unnecessarily increasing the overall size of your application in bytes. That can have a negative impact on the experience of

your end users. We discuss it in the next section.

A bigger APK size means more time needed to download from the app store and more bytecode to load into memory

It's great that you and your team operate on the latest devices and have fast and stable access to the internet. But you need to remember that not everyone has the same luxury. There are still parts of the world where network accessibility and reliability are far from perfect. Projects such as Starlink already improve that situation, but that will take time to cover the most remote areas out there.

Right now, there are still markets where every megabyte of traffic has its price. In those regions, the application's size directly impacts the conversion, and the installation/ cancellation ratio increases along with the app size.



Source: https://segment.com/blog/mobile-app-size-effect-on-downloads/

It is also a common belief that every well crafted and carefully designed application not only provides a beautiful interface but is also optimized for the end device. Well, that is not always the case. And because the Android market is so competitive, there is a big chance that a smaller alternative to those beautiful yet large apps is already gaining more traction from the community.

Another important factor is device fragmentation. The Android market is very diverse in that respect. There are more than 20 popular manufacturers, each releasing an array of devices every year. Contributing to a relatively significant share of mid to low-end devices, which account for over 60% of all smartphone sales annually. And those devices may face issues when dealing with bigger APKs.

As we have stressed already, the startup time of your application is essential. The more code the device has to execute while opening up your code, the longer it takes to launch the app and make it ready for the first interaction.

Now, let's move to the last factor worth mentioning in this context – device storage.

Apps usually end up taking up more space after the installation. Sometimes they may even not fit into the device's memory. In such a situation, users may decide to skip installing your product if that would mean removing other resources such as applications or images.

Solution: Flip the boolean flag enableProguardInReleaseBuilds to true, adjust the Proguard rules to your needs, and test release builds for crashes. Also, flip enableSeparateBuildPerCPUArchitecture to true.

Android is an operating system that runs on plenty of devices with different architectures, so your build must support most of them. React Native supports four: `armeabi-v7a`, `arm64-v8a`, `x86`, and `x86_64`.

While developing your application, Gradle generates the APK file that can be installed on any of the mentioned CPU architectures. In other words, your APK (the file outputted from the build process) is actually four separate applications packaged into a single file with `.apk` extension. This makes testing easier as the application can be distributed onto many different testing devices at once.

Unfortunately, this approach has its drawbacks. The overall size of the application is now much bigger than it should be as it contains the files required by all architectures. As a result, users will end up downloading extraneous code that is not even compatible with their phones.

Thankfully, you can optimize the distribution process by taking advantage of App Bundles when releasing a production version of your app.

App Bundle is a publishing format that allows you to contain all compiled code and resources. It's all due to the fact that Google Play Store Dynamic Delivery will later build tailored APKs depending on the end users' devices.

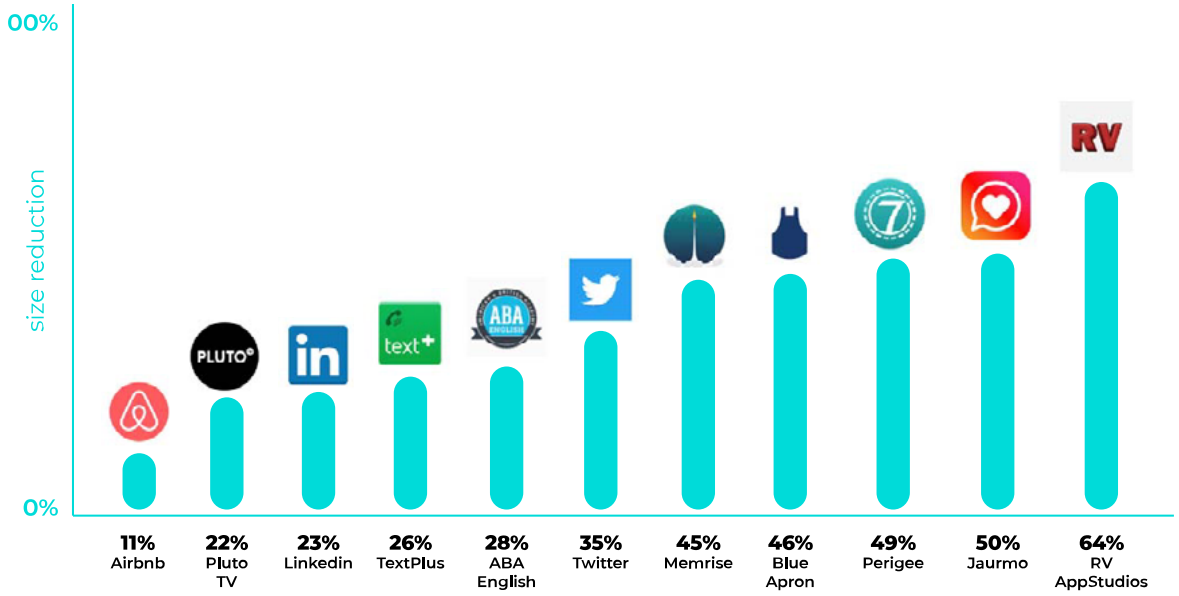
To build App Bundle, you have to simply invoke a different script than usual. Instead of using `./gradlew assembleRelease`, use `./gradlew bundleRelease`, as presented here:
The main advantage of the Android App Bundle over builds for multiple

```
cd android
./gradlew bundleRelease
```

Building a React Native app as App Bundle

architectures per CPU is the ease of delivery. After all, you have to ship only one artifact and Dynamic Delivery will do all the magic for you. It also gives you more flexibility on supported platforms.
You don't have to worry about which CPU architecture your end user's device has. The average size reduction for an app is around 35%, but in

some cases, it can be even cut in half, according to the Android team.



Source: https://medium.com/google-developer-experts/exploring-the-android-app-bundle-ca16846fa3d7

Another way of decreasing the build size is by enabling Proguard. Proguard works in a similar way to dead code elimination from JavaScript - it gets rid of the unused code from third-party SDKs and minifies the codebase.

However, Proguard may not work out-of-the-box with some projects and usually requires an additional setup to achieve optimal results. In this example, we were able to reduce the size of the mentioned 28 MB build by 700 KB. It is not much, but it is still an improvement.

```
def enableProguardInReleaseBuilds = true
```

Enabling proguard in android/app/build gradle

Another good practice is keeping your eye on resources optimization. Each application contains some svg or png graphics that can be optimized using free web tools.

Reducing redundant text from svg and compressing png images can save some bytes when your project has already too many of them.

Benefits: A smaller APK, slightly faster TTI, and slightly less memory used.

All the mentioned steps are worth taking when you're struggling with a growing application size. You will achieve the most significant size reduction by building the app for different architectures. But the list of possible optimizations doesn't stop there.

By striving for a smaller APK size, you will do your best to reduce the download cancellation rate. Also, your customers will benefit from a shorter Time To Interactive and be more inclined to use the app more often.

Finally, you will demonstrate that you care about every user, not only those with top-notch devices and fast internet connections. The bigger your platform gets, the more important it is to support those minor groups, as every percent of users translates into hundreds of thousands of actual users. If you'd like to learn more about optimizing Android, check the Android Profiling chapter.

PART 2 | CHAPTER 6

Experiment with the New Architecture of React Native

Leverage the capabilities of the new rendering system inside your app.

Issue: Your app is using old architecture without the concurrent features of React 18.

Maybe it's better to say "current" architecture since it's still mostly used by production apps. This term refers to how React Native's two realms (Native and JS) communicate with each other. Both new and old architecture is based on the communication between JavaScript and the native side. Currently, this communication is handled by the bridge. Let's go over its limitations in order to easier understand the problems that the New Architecture is trying to solve.

- It is asynchronous: the JavaScript side submits data to a bridge and waits for the data to be processed by the native side
- It's single-threaded (that's why it's important to not overload the JS thread and execute animations on the UI thread).
- It adds additional overhead when it comes to the serialization of data from JSON objects.

The bridge is still working fine for most use cases. However, when we start to send a lot of data over the bridge, it may become a bottleneck for our app. This problem can be seen when rendering a lot of components in a long list. In the case when the user scrolls fast, there will be a blank space caused by the communication between the JS and native sides being asynchronous. Essentially what happens is that we are having a "traffic jam" on our bridge with objects waiting to be serialized. The same issue with the bridge being "overloaded" can be seen in native modules sending a lot of data back and forth.

This bottleneck, together with providing a type safe way of communicating between native and JS, are the main things that the new architecture is trying to solve. However, not everything about new architecture is as good as it may seem. We will also get into the drawbacks that it brings.

Solution: Migrate your app to New Architecture.

What is New Architecture?

Starting from React Native 0.68 developers can leverage new capabilities of the framework. The New Architecture relies on a series of tools which are key components to the new experience, two most important ones are: Fabric and TurboModules. The first one is a new rendering system and the second one is a new way of writing native modules. We will get into details later in this section.
Codegen and JSI are two new tools improving developer experience. They are essential to understand how the new architecture works.
Codegen drastically improves DX by generating a lot of native boilerplate code and ensuring type safety. And JSI, a C++ API for interacting with any JS engine.

Note: Prior to React Native v0.71+ it's possible but way harder to deploy the New Architecture in an app. So that's the version we are suggesting for you to upgrade.

Codegen
A code generation tool that makes JS source of truth by automating the compatibility between JS and native side. It allows to write statically typed JS (called JS Spec) which is then used to generate the interface files needed by Fabric native components and TurboModules. Spec consists of a set of types written in TypeScript or Flow that defines all the APIs provided by the native module. Codegen ensures type-safety as well as compile-time type safety, which means smaller code and faster execution as both realms can trust each other around validating the data every time. To find out more about it, refer to the docs.

JSI
JSI is the foundation of the New Architecture, a C++ API for interacting with any JS engine. In contrast to the bridge which was asynchronous,

JSI is synchronous which allows for invoking native functions faster. It lets JavaScript to hold references to C++ host objects and invoke methods directly on them. This removes the major overhead of asynchronous communication between JS and native by serializing objects using the bridge.

Fabric

Fabric is React Native's new concurrent rendering system, a conceptual evolution of the legacy render system. The core principle is to unify more render logic in C++ to better leverage interoperability between platforms. Host Components like View, Text, etc. are now lazily initialized, resulting in faster startups. Fabric allows us to take advantage of the features introduced in React 18.

TurboModules

This is a new way of writing native modules that also leverages the power of JSI, allowing for synchronous, and an order of magnitude faster data transfer from native to JS and vice versa. It is a rewrite of the communication layer between JavaScript and platform native modules like Bluetooth, Biometrics, etc. It also allows for writing native code for both platforms using C++ and introduces the lazy loading of modules to speed up your app startup time.

Bridgeless mode

For the time being React Native is not removing the bridge completely, it’s still there allowing developers to gradually adopt new architecture and JSI. However usage of the bridge needs to be explicitly stated (by importing it), so not yet migrated modules won’t work. Meta is planning to soon allow apps to run in completely bridge-less mode. Which will result in faster app startup due to removing the overhead of loading the bridge every time the app starts.

How to turn on New Architecture

In order to turn on the New Architecture in your app, you need to update your app to at least React Native 0.68, however, we recommend upgrading to at least React Native 0.71+ because a lot of improvements

have been added.

To migrate your app to the New Architecture, follow these steps:

1. Upgrade your app to at least React Native 0.71+, you can use https://react-native-community.github.io/upgrade-helper/
2. Check all third-party libraries that your app is depending on and the important thing is that: all of them need to be migrated! This might be a long-time blocker for a lot of apps out there. Components that are not yet compatible will show a red box - Unimplemented component: `<ComponentName>` - and you will likely notice them. In that case, please let the library maintainers know about it, as this will speed up the adoption.
3. [Android] Set `newArchEnabled=true` in `gradle.properties`.
4. [iOS] Run `RCT_NEW_ARCH_ENABLED=1 pod install` inside the iOS folder.

Benefits: You are able to leverage all the latest features including React 18, Fabric, TurboModules, and JSI.

Now that you know the basics of how the New Architecture works, let's go over the benefits.

Performance

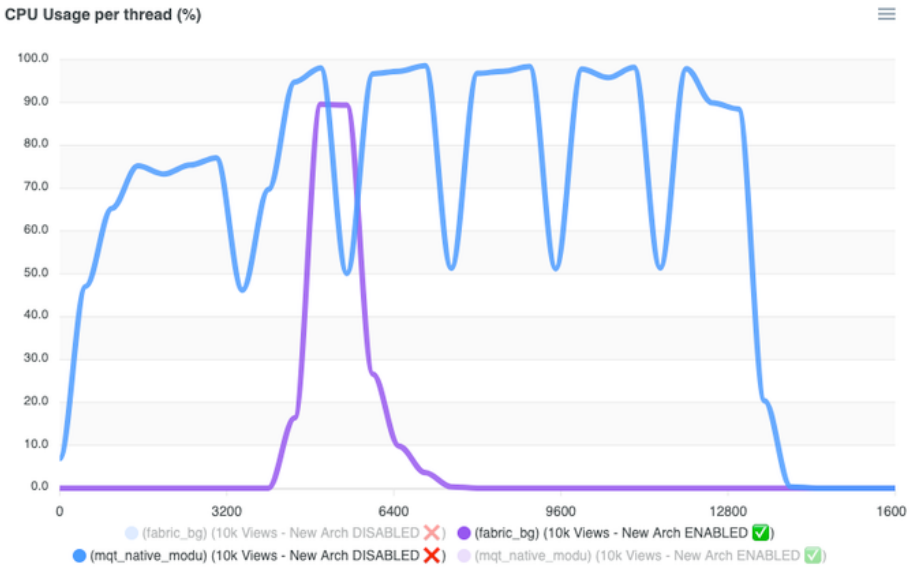
Due to the synchronous nature of the new architecture, while communicating with the native side, there will be some performance improvements. The app's startup time will be significantly reduced as every native module will be lazily-loaded. Once the bridgeless mode will be available it will also remove the overhead of loading the bridge at startup. However, not every scenario proves this, in some of the benchmarks architecture performance is worse.

Meta's goal was not to make new architecture X times faster than the old one. Apart from removing major bottlenecks they wanted to create a new solid foundation which would allow for new capabilities that could not be developed using previous architecture. Migration of the Facebook app took over a year and they haven't noticed any significant performance improvements nor regressions that are perceivable by the end user. However, this doesn't mean that performance improvements won't come in the future. Now that they reworked internals they have

a great foundation to build upon.
Let's go over some performance benchmarks by Alexandre Moureaux from BAM. Here is the link to the source: https://github.com/reactwg/react-native-new-architecture/discussions/85

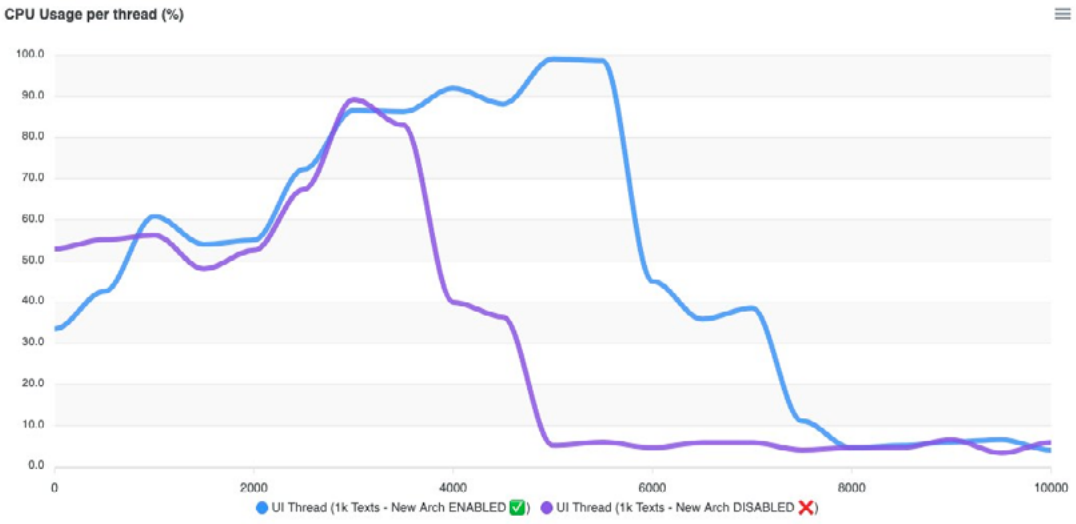
Benchmark of rendering 10K views

In this case new architecture proves that it's more efficient than the old one. Using on average less CPU but more RAM.



Benchmark of rendering 2K Text components

In this scenario, the old architecture is faster, mainly because of heavier UI thread consumption.



The official response from the React Native team is that their internal benchmarks while rolling out the New Architecture to users was neutral across all React Native surfaces in the Facebook app on both Android and iOS. As stated by Samuel Susla in this discussion thread, "In the last years, we conducted dozens of tests in production on millions of devices to assure performance was neutral."
So in most use cases, you can expect a neutral performance impact without any performance regressions. And keep in mind that the New Architecture is getting better every single day with many developers contributing to the repository, so the results may be totally different by the time you are reading this.

Future readiness

New Architecture allows your app to leverage Concurrent React features. Which improves UI responsiveness, provides Suspense for data fetching to handle complex UI loading schemes, and ensures your app is ready for any further React innovations that will be built on top of its new concurrent engine introduced in React 18.

Let's see how we can leverage React18's `startTransition` API in order to prioritize between two state updates. In our example, a button click can be considered an urgent update whereas the `NonUrgentUI` can be considered a non-urgent update. To tell React about a non-urgent update, we can wrap the `setState` in the `startTransition` API. This allows React to prepare a new UI and show the old UI until a new one is prepared. In our example, we wrapped `setNonUrgentValue` in `startTransition` and told React that `nonUrgentValue` is a transition and not so urgent, it may take some time. We've also added a conditional `backgroundColor`. When you run this example, you will see that once you click on the button, the view will retain its old UI for e.g., if we start at value 1, the UI will be green.

Once you click on the button, the Value text UI will be updated but the UI for the container will still remain green until the transition is completed and the color will change to red due to the new UI being rendered. That's the magic of React's concurrent rendering.

To understand it better, assume that wrapping an update in `startTransition` renders it in a different universe. We don't see that universe directly but we can get a signal from it using the `isPending` variable returned from the `useTransition` hook. Once the new UI is ready, both universes merge together to show the final UI.

```
import React from 'react';
import { Button, StyleSheet, Text, View } from 'react-native';

const dummyData = Array(10000).fill(1);

const NonUrgentUI = ({ value, isPending }) => {
  const backgroundStyle = {
    backgroundColor: value % 2 === 0 ? 'red' : 'green',
  };

  return (
    <View>
      <Text>Non urgent update value: {isPending ? 'PENDING' :
value}</Text>
      <View style={[styles.container, backgroundStyle]}>
        {dummyData.map((_, index) => (
          <View key={index} style={styles.item} />
        ))}
      </View>
    </View>
  );
};

const ConcurrentStartTransition = () => {
  const [value, setValue] = React.useState(1);
  const [nonUrgentValue, setNonUrgentValue] = React.
useState(1);
  const [isPending, startTransition] = React.useTransition();

  const handleClick = () => {
    const newValue = value + 1;
    setValue(newValue);
    startTransition(() => {
      setNonUrgentValue(newValue);
    });
  };
  return (
    <View>
      <Button onPress={handleClick} title="Increment value" />
      <Text>Value: {value}</Text>
      <NonUrgentUI value={nonUrgentValue} isPending={isPending}
/>
    </View>
```

```
    );
};

export default ConcurrentStartTransition;

const styles = StyleSheet.create({
  container: {
    flexDirection: 'row',
    flexWrap: 'wrap',
  },
  item: {
    width: 10,
    height: 10,
  },
});
```

To understand it better, let's visualize the code snippet that we just went through. The image below shows a comparison of when we use `startTransition` and when we don't. Looking at the image, we see that React flushes the urgent update right off, which happens due to calling `setValue` without wrapping it in `startTransition`.

Next, we see that React shows the old UI (viewed in green) for the UI that depends on the nonurgent updates, which means the updates that are wrapped in `startTransition`. We also see a ***Pending*** text displayed, this is a way for React18 to tell us that the new UI depending on this state is not yet ready. Once it's ready, React flushes it and we don't see the ***Pending*** text anymore, and the view color changes to red.

On the other hand, if we don't use `startTransition`, React tries to handle both updates as urgent and flushes once both are ready. This certainly has a few downsides, such as the app trying to render some heavy UI all at once which may cause jarring effects for the users. With React18, we can handle this by delaying the updates that are not urgent.

There are some other noticeable features in React18 that you might want to check out by playing with the linked sandboxes from React's official website. See useDeferredValue and startTransition with Suspense.

Maintenance & Support

The React Native core team is committed to offer support for the 3 latest versions of React Native (you can check the support policy here: https://github.com/reactwg/react-native-releases#releases-support-policy). And the React core team plans new features built on the concurrent rendering engine. It's important to not stay behind, as the cost of paying the tech debt, will get higher in time. It's worth calling out that React Native is no different to any other software project in this regard. Not updating dependencies may not only cause your team to spend more time on this task when it's unavoidable. It can also expose your app to security vulnerabilities already patched in the upstream.

The React Native team has dedicated capacity to help the community

solve their app and library problems regarding new architecture adoption in close cooperation. Although it's not stable yet, it's worth considering starting to plan your migration early on so you can identify the problems and blockers that your app may be facing. 2023 is the best time to try it, evaluate, ask for support and give feedback. You can get a head start with fine support from the React Native core engineers in solving your app's build and runtime issues preventing you from migration. Once the new architecture gets stable and turned on by default, the core team will surely focus on further development of React Native. And hence the migration support time will be reduced.

If you need help with performance, stability, user experience, or other complex issues - contact us!

As React Native Core Contributors and leaders of the community, we will be happy to help.

PART 3

How to ship quicker with a stable development environment

React Native is great for shipping fast and with confidence, but are you ready for that?

These days, having a stable and comfortable development setup that encourages shipping new features and doesn't slow you down is a must. You have to ship fast and be ahead of your competitors.

React Native plays really well in such environments. For example, one of its biggest selling points is that it allows you to ship updates to your applications without undergoing the App Store submission. They're called Over-the-Air (OTA) updates.

The question is: is your application ready for that? Does your development pipeline accelerate the development and shipping features with React Native?

Most of the time, you would like the answer to be simply yes. But in reality, it gets complicated.

In this section, we present some of the best practices and recommendations that allow you to ship your apps faster and with more confidence. And it's not just about turning on the Over-the-Air updates, as most articles suggest. It's about building a steady and healthy development environment where React Native shines and accelerates innovation.

And that's what this part of our guide is all about.

PART 3 | CHAPTER 1

Run tests for key pieces of your app

Focus testing on key pieces of the app to have a better overview of new features and tweaks.

Issue: You don't write tests at all or write low-quality tests with no real coverage, and you only rely on manual testing.

Building and deploying apps with confidence is a challenging task. However, verifying if everything actually works requires a lot of time and effort – no matter if it is automated or not. Having somebody who manually verifies that the software works as expected is vital for your product.

Unfortunately, this process doesn't scale well as the amount of your app functionalities grow. It also doesn't provide direct feedback to the developers who write the code.
Because of that, it increases the time needed to spot and fix a bug.

So what do the developers do to make sure their software is always production-ready and doesn't rely on human testers? They write automated tests. And React Native is no exception. You can write a variety of tests both for your JS code – which contains the business logic and UI – and the native code that is used underneath.

You can do it by utilizing end-to-end testing frameworks, spinning up simulators, emulators, or even real devices. One of the great features of React Native is that it bundles to a native app bundle, so it allows you to employ all the end-to-end testing frameworks that you love and use in your native projects.

But beware, writing a test may be a challenging task on its own, especially if you lack experience. You might end up with a test that doesn't have a good coverage of your features. Or only to test positive behavior,

without handling exceptions. It's very common to encounter low-quality tests that don't provide too much value and hence, won't boost your confidence in shipping the code.

Whichever kind of test you're going to write, be it unit, integration, or E2E (short for end-to-end), there's a golden rule that will save you from writing the bad ones. And the rule is to "avoid testing implementation details." Stick to it and your test will start to provide value over time. You can't move as fast as your competition, chances of regressions are high, and apps can be removed from stores when receiving bad reviews. The main goal of testing your code is to deploy it with confidence by minimizing the number of bugs you introduce in your codebase. And not shipping bugs to the users is especially important for mobile apps, which are usually published in app stores.

Because of that, they are a subject of a lengthy review process, which may take from a few hours up to a few days. And the last thing you want is to frustrate your users with an update that makes your app faulty. That could lead to lower ratings and, in extreme cases, even taking the app down from the store.

Such scenarios may seem pretty rare, but they happen. Then, your team may become so afraid of having another regression and crash that it will lose its velocity and confidence.

Solution: Don't aim at 100% coverage, focus on key pieces of the app. Test mostly integration.

Running tests is not a question of "if" but "how". You need to come up with a plan on how to get the best value for the time spent. It's very difficult to have 100% lines of your code and dependencies covered. Also, it's often quite impractical.

Most of the mobile apps out there don't need a full test coverage of the code they write.

The exceptions are situations in which the client requires full coverage because of the government regulations they must abide by. But in such cases, you're probably already aware of the problem.

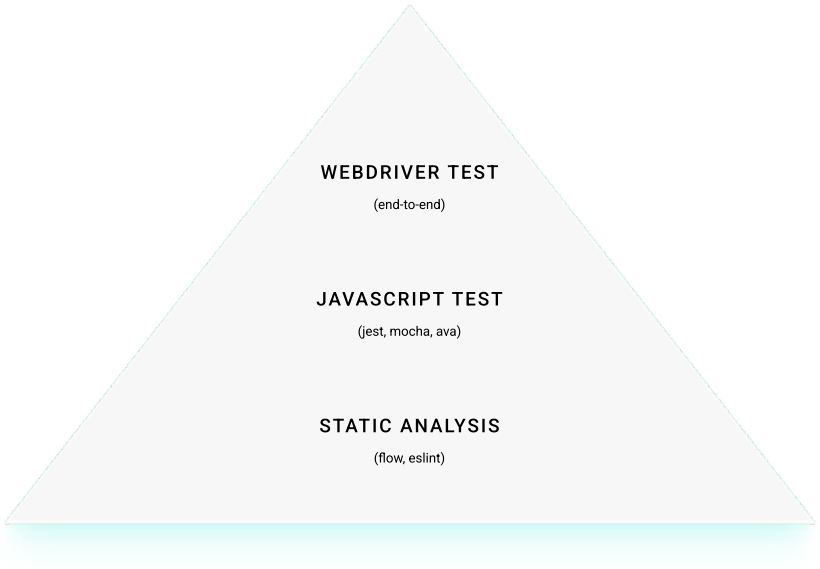
It's crucial for you to focus your time on testing the right thing. Learning to identify business-critical features and capabilities is usually more important than writing a test itself. After all, you want to boost confidence in your code, not write a test for the sake of it. Once you do that, all you need to do is decide on how to run it. You have quite a few options to choose from.

In React Native, your app consists of multiple layers of code, some written in JS, some in Java/Kotlin, some in Objective-C/Swift, and some even in C++, which is gaining adoption in the React Native core.

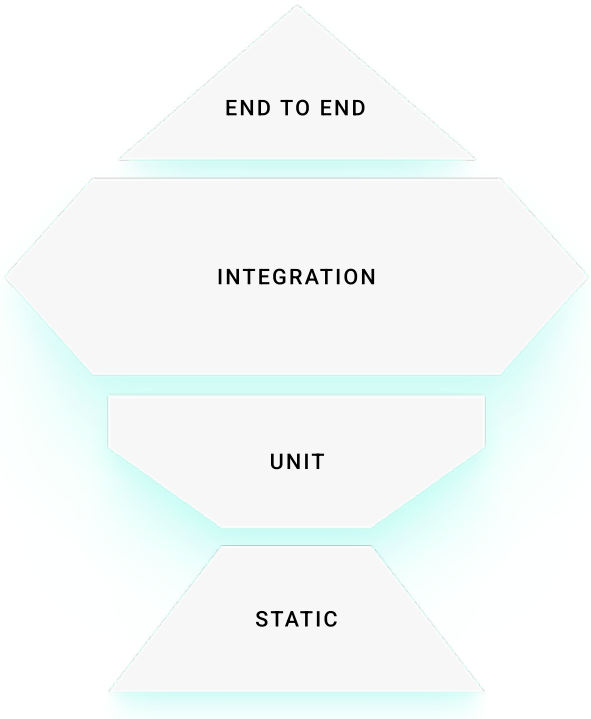
Therefore, for practical reasons, we can distinguish between:

- **JavaScript testing** – with the help of the Jest framework. In the context of React Native, if you think about "unit" or "integration" tests, this is the category they eventually fall into. From a practical standpoint, there is no reason for distinguishing between those two groups.
- **End-to-end app testing** – with the help of Detox, Appium, or another mobile testing framework you're familiar with.

Because most of your business code lives in JS, it makes sense to focus your efforts there.



Testing pyramid. Source: https://twitter.com/aaronabramov_/status/805913874704674816



Testing trophy. Source: https://twitter.com/kentcdodds/status/960723172591992832

JavaScript testing

Writing tests for utility functions should be pretty straightforward. To do so, you can use your favorite test runner. The most popular and recommended one within the React Native community is Jest. We'll also be referring to it in the following sections.

For testing React components, you need more advanced tools though. Let's take the following component as an example:

```
import React, { useState } from 'react';
import {
  View,
  Text,
  TouchableOpacity,
  TextInput,
  ScrollView,
} from 'react-native';

const QuestionsBoard = ({ questions, onSubmit }) => {
  const [data, setData] = useState({});

  return (
    <ScrollView>
      {questions.map((q, index) => {
        return (
          <View key={q}>
            <Text>{q}</Text>
            <TextInput
              accessibilityLabel="answer input"
              onChangeText={(text) => {
                setData((state) => ({
                  ...state,
                  [index + 1]: { q, a: text },
                }));
              }}
            />
          </View>
        );
      })}
      <TouchableOpacity onPress={() => onSubmit(data)}>
        <Text>Submit</Text>
      </TouchableOpacity>
    </ScrollView>
  );
};

export default QuestionsBoard;
```

It is a React component that displays a list of questions and allows for answering them. You need to make sure that its logic works by checking if the callback function is called with the set of answers provided by the user.

To do so, you can use an official `react-test-renderer` library from the React core team. It is a test renderer, in other words, it allows you to render your component and interact with its lifecycle without actually dealing with native APIs. Some people may find it intimidating and hard to work with because of the low-level API.

That's why the community around React Native came out with helper libraries, such as React Native Testing Library, providing us with a good set of helpers to productively write your high-quality tests.

A great thing about this library is that its API forces you to avoid testing the implementation details of your components, making it more resilient to internal refactors.

A test for the `QuestionsBoard` component would look like this:

```
import { render, screen, fireEvent } from '@testing-library/
react-native';
import { QuestionsBoard } from '../QuestionsBoard';

test('form submits two answers', () => {
  const allQuestions = ['q1', 'q2'];
  const mockFn = jest.fn();

  render(<QuestionsBoard questions={allQuestions}
onSubmit={mockFn} />);

  const answerInputs = screen.getAllByLabelText('answer
input');

  fireEvent.changeText(answerInputs[0], 'a1');
  fireEvent.changeText(answerInputs[1], 'a2');
  fireEvent.press(screen.getByText('Submit'));

  expect(mockFn).toBeCalledWith({
    1: { q: 'q1', a: 'a1' },
    2: { q: 'q2', a: 'a2' },
  });
});
```

Test suite taken from the official RNTL documentation

You first render the `QuestionsBoard` component with your set of questions. Next, you query the tree by label text to access an array of questions, as displayed by the component. Finally, you set up the right answers and press the `submit` button.

If everything goes well, your assertion should pass, ensuring that the `verifyQuestions` function has been called with the right set of arguments.

Note: You may have also heard about a technique called "snapshot testing" for JS. It can help you in some of the testing scenarios, e.g. when working with structured data that may change slightly between tests. The technique is widely adopted in the React ecosystem because of its built-in support from Jest.

If you're into learning more about snapshot testing, check out the official documentation on the Jest website. Make sure to read it thoroughly, as `toMatchSnapshot` and `toMatchInlineSnapshot` are low-level APIs that have many gotchas.

They may help you and your team quickly add coverage to the project. And at the same time, snapshots make adding low-quality and hard-to-maintain tests too easy. Using helper tools like eslint-plugin-jest with its no-large-snapshots option, or snapshot-diff with its component snapshot comparison feature for focused assertions, is a must-have for any codebase that leverages this testing technique.

E2E tests

The cherry on top of our testing pyramid is a suite of end-to-end tests. It's good to start with a so-called "smoke test" – a test ensuring that your app doesn't crash on the first run. It's crucial to have a test like this, as it will help you avoid sending a faulty app to your users. Once you're done with the basics, you should use your E2E testing framework of choice to cover the most important functionalities of your apps.

These can be, for instance, logging in (successfully or not), logging out, accepting payments, and displaying lists of data you fetch from your or third-party servers.

Note: Beware that these tests are usually a bit harder to set up than the JS ones.

Also, they are more likely to fail because of the issues related to e.g. networking, file system operations or storage or memory shortage. What's more, they provide you with little information on why they do it. This test's quality (not only the E2E ones) is called "flakiness" and should be avoided at all cost, as it lowers your confidence in the test suite. That's why it's so important to divide testing assertions into smaller groups, so it's easier to debug what went wrong.

For the purpose of this section, we'll be looking at Detox – the most popular E2E test runner within the React Native community.

Before going any further, you have to install Detox. This process requires you to take some additional "native steps" before you're ready to run your first suite. Follow the official documentation as the steps are likely to change in the future.

Once you have successfully installed and configured Detox, you're ready to begin with your first test.

```
it('should display the questions', async () => {
  await devicePixelRatio.reloadReactNative();

  await element(by.text(allQuestions[0])).toBeVisible();
});
```

This quick snippet shown above would ensure that the first question is displayed.
Before that assertion is executed, you should reload the React Native instance to make sure that no previous state is interfering with the results.

Note: When you're dealing with multiple elements (e.g. in our case – a component renders multiple questions), it is a good practice to assign a suffix testID with the index of the element, to be able to query the specific one. This, as well as some other interesting techniques, is in the official Detox recommendation.

There are various matchers and expectations that can help you build your test suite the way you want to.

Benefits: You have a better overview of the new features and tweaks, can ship with confidence, and when the tests are green - you save the time of other people (the QA team).

A high-quality test suite that provides enough coverage for your core features is an investment in your team's velocity. After all, you can move only as fast as your confidence allows you to. And the tests are all about making sure you're heading in the right direction.

The React Native community is working hard to make testing as easy and pleasant as possible – for both your team and the QA teams. Thanks to that, you can spend more time innovating and pleasing users with flashy new functionalities, and not squashing bugs and regressions over and over again.

> **"By testing key features of an app via integration testing, developers can effectively identify and eliminate potential bugs, ultimately leading to a more confident and efficient development process."**
>
> 
>
> **Christoph Nakazawa**
> Senior Engineering Manager & Creator of Jest

PART 3 | CHAPTER 2

Have a working Continuous Integration (CI) in place

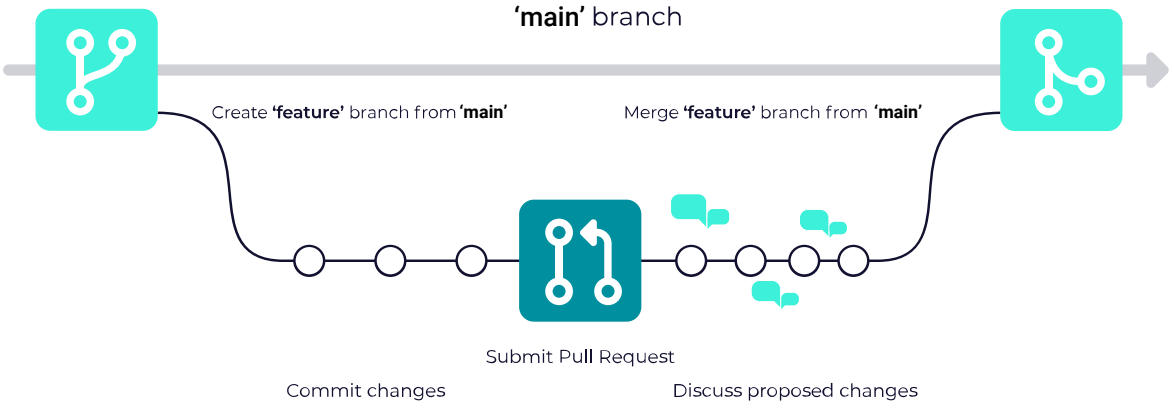
Use a CI provider to improve the building, testing, and distribution of your apps.

Issue A lack of CI or having an unstable one means a longer feedback loop - you don't know if your code works and you cooperate slowly with other developers.

As you have already learned from the previous section, covering your code with tests can be very helpful for increasing the overall reliability of your app. However, while testing your product is vital, it is not the only prerequisite on your way to shipping faster and with more confidence.

What is equally important is how quickly you detect the potential regressions and whether finding them is a part of your daily development lifecycle. In other words – it all comes down to the feedback loop.

For better context, let's take a look at the early days of the development process. When you're starting out, your focus is on shipping the first iteration (MVP) as fast as possible. Because of that, you may overlook the importance of the architecture itself. When you're done with the changes, you submit them to the repository, letting other members of your team know that the feature is ready to be reviewed.



An example of a workflow on Github, where changes are proposed in the form of a PR.

While this technique can be very useful, it is potentially dangerous on its own, especially as your team grows in size. Before you're ready to accept a PR, you should not only examine the code but also clone it to your environment and test it thoroughly. At the very end of that process, it may turn out that the proposed changes introduce a regression that the original author hasn't spotted.

The regression can occur because we all have different configurations, environments, and ways of working.

It's harder to onboard new members to your organization. You can't ship and test PRs and different contributions as they happen.

If you're testing your changes manually, you're not only increasing the chances of shipping regressions to production. You're also slowing down the overall pace of the development. Thankfully, with the right set of methodologies and a bit of automation, you can overcome this challenge once and for all.

This is when Continuous Integration (CI) comes into play. CI is a development practice where proposed changes are checked-in to the upstream repository several times a day by the development team. Next, they are verified by an automated build, allowing the team to detect changes early.

The automated builds are performed by a dedicated cloud-based CI provider that usually integrates from the place where you store your code. Most of the cloud providers available these days support GitHub, which is a Microsoft-owned platform for collaborating on projects that use Git as their version control system.

CI systems pull the changes in real-time and perform a selected set of tests to give you early feedback on your results. This approach introduces a single source of truth for testing and allows developers with different environments to receive convenient and reliable information.

approach introduces a single source of truth for testing and allows developers with different environments to receive convenient and reliable information.

Using a CI service, you not only test your code but also build a new version of the documentation for your project, build your app, and distribute it among testers or releases. This technique is called Continuous Deployment and focuses on the automation of releases. It has been covered in more depth in this section.

Solution: Use a CI provider such as Circle CI or EAS Build to build your application. Run all the required tests and make preview releases if possible.

There are a lot of CI providers to choose from, with the most popular being CircleCI, GitHub Actions and Bitrise. For React Native applications, another option is EAS Build, a service created by Expo with first-class support for React Native

We have selected CircleCI as our reference CI provider for the purpose of this section, as it has wide community adoption. In fact, there is actually an example project demonstrating the use of CI with React Native. You can learn more about it here. We will employ it later in this section to present different CI concepts.
We will also dive into EAS Build, which is designed to make building and testing applications as easy as possible. It also helps with distributing app binaries to the stores and internally amongst your team.

Note: A rule of the thumb is to take advantage of what React Native or React Native Community projects already use. Going that route, you can ensure that it is possible to make your chosen provider work with React Native and that the most common challenges have been already solved by the Core Team.

CircleCI

As with most CI providers, it is extremely important to study their configuration files before you do anything else.

Let's take a look at a sample configuration file for CircleCI, taken from the mentioned React Native example:

```
version: 2.1

jobs:
  android:
    working_directory: ~/CI-CD
    docker:
      - image: reactnativecommunity/react-native-android
    steps:
      - checkout
      - attach_workspace:
          at: ~/CI-CD
      - run: npm i -g envinfo && envinfo
      - run: npm install
      - run: cd android && chmod +x gradlew && ./gradlew
assembleRelease

workflows:
  build_and_test:
    jobs:
      - android
```

Example of .circleci/config.yml

The structure is a standard Yaml syntax for text-based configuration files. You may want to learn about its basics before proceeding any further.

Note: Many CI services, such as CircleCI or GitHub Actions, are based on Docker containers and the idea of composing different jobs into workflows. You may find many similarities between such services.

These are the three most important building blocks of a CircleCI configuration: `commands`, `jobs`, and `workflows`.

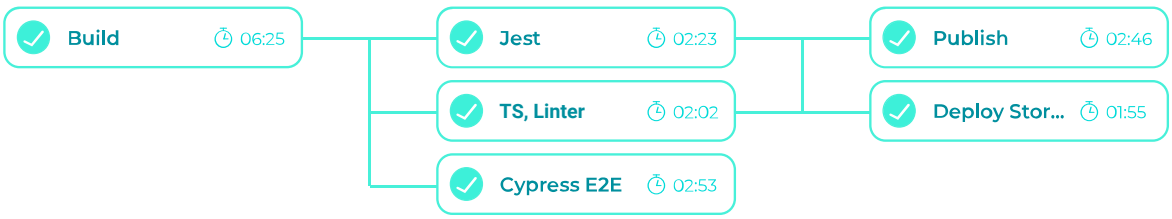
A command is nothing more than a shell script. It is executed within the specified environment. Also, it is what performs the actual job in the cloud. It can be anything, from a command to install your dependencies, such as `yarn install` (if you're using Yarn) to something more complex like `./gradlew assembleDebug` that builds Android files.

A *job* is a series of commands - described as steps - that is focused on achieving a single, defined goal. *Jobs* can be run in different environments, by choosing an appropriate *Docker* container.

For example, you may want to use a Node container if you need to run only your React unit tests. As a result, the container will be smaller, have fewer dependencies, and will install faster. If you want to build a React Native application in the cloud, you may choose a different container, e.g. with Android NDK/SDK or the one that uses OS X to build Apple platforms.

Note: To help you choose the container to use when running React Native tests, the team has prepared a react-native-android Docker container that includes both Node and Android dependencies needed to perform the Android build and tests.

In order to execute a `job`, it has to be assigned to a `workflow`. By default, `jobs` will be executed parallelly within a workflow, but this can be changed by specifying the requirements for a `job`.



Workflow contains jobs that can be grouped to run in a sequence or in parallel

You can also modify the jobs execution schedule by adding filters, so, for instance, a deploy job will only run if the changes in the code refer to the main branch.

You can define many workflows for different purposes, e.g. one for tests that would run once a PR is opened, and the other to deploy the new version of the app. This is what React Native does to automatically release its new versions every once in a while.

EAS Build

EAS stands for Expo Application Services. One of the services it provides is EAS Build. In contrast with a generic CI provider, it is designed for building and testing React Native applications, handles app-signing credentials and internal distribution amongst your team members, and also closely integrates with EAS Submit to automate app store submissions.

EAS Build is a managed service. This means many workflow steps that are traditionally manually defined in a generic CI provider are handled automatically. It also uses a new build environment for each build job with the tools such as the Java JDK, the Android SDK and NDK, Xcode, Fastlane, etc. already installed.

You can get started by following just a few steps below on your development computer. After these steps, you will have set up your project for EAS, configured your build profile, and started a build job.

Running an EAS Build

First, install the `eas-cli` package by running this command in your terminal:

```
npm install -g eas-cli
```

Or, you can use `npx eas-cli` to run the commands if you'd like to avoid installing a global dependency.

Setting up EAS Build

Next, initialize a configuration file for EAS Build and run this command in the root directory of your project:

```
eas build:configure
```

This command will create a file named `eas.json` in the root directory of your project. It contains the whole config for EAS to run properly.

```
{
   "cli":{
      "version":">= 3.3.2”
   },
   "build":{
      "development":{
         "distribution":"internal",
         "android":{
            "gradleCommand":":app:assembleDebug"
         },
         "ios":{
            "buildConfiguration":"Debug"
         }
      },
      "preview":{
         "distribution":"internal"
      },
      "production":{}
   },
   "submit":{
      "production":{}
   }
}
```

Example of eas.json

Info: Your project may need some additional configuration for cases like using a monorepo. Read this guide to see how to configure your build profile further.

`eas.json` is a text-based configuration file that uses standard JSON syntax. Inside, you will find a top-level field named `build`. This field contains a JSON object with all of the configuration that defines your

app's build profiles. A `build profile` is a named group of configuration options that describe the necessary parameters to create specific types of builds, like for internal distribution or an app store release.

The JSON object under `build` contains multiple profiles. By default, the `development, preview`, and `production` are the three build profiles provided. The `development profile` is for debug builds that include developer tools and is mainly used during development and testing. The `preview` profile doesn't include developer tools and is for builds intended to be shared with your team when testing your app in a production-like environment. Finally, the production profile is for release builds that are submitted to app stores.

You can define additional custom profiles under the build field for specific types of builds to best fit your project and team's needs. Additionally, each profile can also have platform-specific configuration for Android and iOS.

Running your build on EAS Build

Any type of build can be triggered from a single command. For example, you can use `preview` profile to share the app with your team for testing:

```
eas build --profile preview --platform all
```

Info: The `--platform all` option lets you build for Android and iOS at the same time.

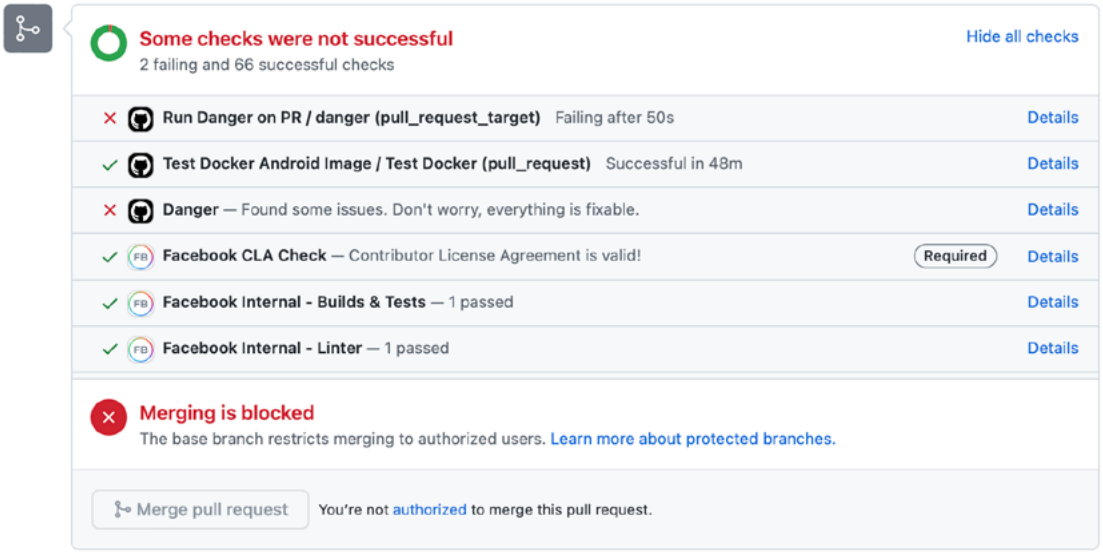
A build profile that has its `distribution` field set to "`internal`" (as shown in the example `eas.json` configuration earlier) configures EAS Build to provide shareable URLs. Once the build is complete, this URL can be shared with your teammates for internal distribution. By using the URL, any member of your team can download the app to their device.

You can modify a build profile any time during the development lifecycle of your project. In some cases, additional configuration may be

required or useful like when working on a project inside a monorepo or sharing configuration between different profiles.

Benefits: You get early feedback on added features, and swiftly spot the regressions. Also, you don't waste the time of other developers on testing the changes that don't work.

A properly configured and working CI provider can save you a lot of time when shipping a new version of an application.



GitHub UI reporting the status of CircleCI jobs, an example taken from React Native repository

By spotting errors beforehand, you can reduce the effort needed to review the PRs and protect your product against regressions and bugs that may directly decrease your income.

With a managed service tailored to Expo and React Native like EAS Build, caching JavaScript, Android, and iOS dependencies is done automatically. There is no need to configure your build steps for caching and all build jobs are accelerated by default.

PART 3 | CHAPTER 3

Don’t be afraid to ship fast with Continuous Deployment

Establish a continuous deployment setup to ship new features and verify critical bugs faster.

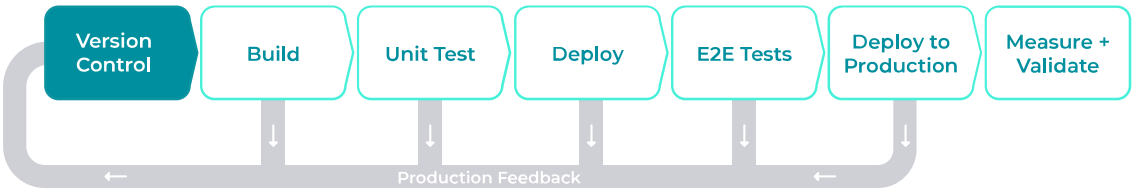
Issue: Building and distributing your apps manually is a complex and time-consuming process.

As you have learned in the previous section, automation of the critical pieces of the development lifecycle can help you improve overall development speed and security. The shorter the feedback loop, the faster your team can iterate on the product itself.

However, testing and development are only a part of the activities that you have to perform when working on a product. Another important step is the deployment – building and distributing the application to production. Most of the time, this process is manual.

The deployment takes time to set up and is far more complex than just running tests in the cloud. For example, on iOS, Xcode configures many settings and certificates automatically. This ensures a better developer experience for someone who's working on a native application. Developers who are used to such an approach often find it challenging to move the deployment to the cloud and set up such things as certificates manually.

The biggest downside of the manual approach is that it takes time and doesn't scale. In consequence, teams that don't invest in the improvements to this process end up releasing their software at a slower pace.



Continuous Deployment is a strategy in which software is released frequently through a set of automated scripts. It aims at building, testing, and releasing software with greater speed and frequency. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production.

You are not shipping new features and fixes as quickly as you should.

Building and distributing your application manually slows down your development process regardless of how big your team is. Even in small product teams of around 5 people, automated build pipelines make everyone's work easier and reduce unnecessary communication. This is especially important for remote companies.

Continuous Deployment also allows you to introduce standards and best practices focused on improving the overall performance of the application. Some of them have been previously discussed in this guide. With all the steps required for the deployment in a single place, you can ensure that all releases are done the same way and enroll company-wide standards.

Solution: Establish a continuous deployment setup that makes the build and generates the changelog. Ship to your users instantly.

When it comes to automating the deployment of mobile applications, there are a few established ways to go.

One way is to write a set of scripts from scratch by interacting with `xcode` and `gradle` directly. Unfortunately, there are significant differences between the tooling of Android and iOS and not many developers have enough experience to handle this automation. On top of that, iOS is much more complicated than Android due to advanced code signing and distribution policies. And as we have said before, if you are doing it manually, even Xcode cannot help you.

Another way is to use a pre-existing tool in which the developers have handled the majority of use cases. Our favorite one is *fastlane* - a set of modular utilities written in Ruby that let you build your iOS and Android applications by writing a set of instructions in a configuration file.

After you have successfully built your binaries, it is time to deploy them to their destination.

Again, you can either upload the files to the desired service (e.g. App Store) manually or use a tool that will take care of that for you. For the same reasons as before, we prefer to use an existing solution - in this case, AppCenter by Microsoft.

AppCenter is a cloud service with tooling for the automation and deployment of your application. Its biggest advantage is that many of the settings can be configured from the graphical interface. It is much easier to set up the App Store and Play Store deployments this way, rather than working with uploads from the command line.

Yet another way to automate deployment that we'd like to call out is EAS Submit is a cloud service created by Expo with first-class support for submitting React Native applications. It provides a fast workflow for you to easily submit your application to both stores. When used with EAS Build, it also provides automatic submissions out of the box.

For the purpose of this section, we will use *Fastlane* and *AppCenter* in *CircleCI* pipelines to fully automate the process of app delivery to the final users. Then, we will dive into the EAS Submit.

Note: Describing the ins and outs of the setup would make this section too long. That's why we have chosen to refer only to the specific documentation. Our goal is to provide you with an overview, and not a step-by-step guide, since the final config will be different for each project.

Next, you have to run the init command within the React Native project. We will run the `fastlane` command twice from each native folder. This is because React Native is actually two separate apps at a low level.

```
cd ./ios && fastlane init
cd ./android && fastlane init
```

As a result, this command will generate setup files in both `ios` and `android` folders. The main file in each folder would be called Fastfile and it's where all the `lanes` will be configured. In the `fastlane` nomenclature, a `lane` is just like a workflow - a piece that groups low-level operations that deploy your application.

Low-level operations can be performed by calling `actions` – predefined `fastlane` operations that simplify your workflow. We will show you how they function in the next section.

Setting up fastlane on Android

Now that you have successfully set up fastlane in your projects, you are ready to automate the deployment of our Android application. To do so, you can choose an Android specific action - in this case, `gradle`. As the name suggests, Gradle is an action that allows you to achieve similar results as with Android Gradle used standalone.

Our lane uses the `gradle` action to first clean the build folder, and then assemble the APK with signature based on passed params.

```
default_platform(:android)

project_dir = File.expand_path("..", Dir.pwd)

platform :android do
  lane :build do |options|
    if (ENV["ANDROID_KEYSTORE_PASSWORD"] && ENV["ANDROID_KEY_
PASSWORD"])
      properties = {
      "RELEASE_STORE_PASSWORD" => ENV["ANDROID_KEYSTORE_
PASSWORD"]
      "RELEASE_KEY_PASSWORD" => ENV["ANDROID_KEY_PASSWORD"]
      }
    end

    gradle(
      task: "clean",
      project_dir: project_dir,
      properties: properties,
      print_command: false
    )

    gradle(
      task: "assemble",
      build_type: "Release",
      project_dir: project_dir,
      properties: properties,
      print_command: false
    )
  end
end
```

Part of the android/fastlane/Fastfile that defines Android lane, named build

You should be able to run a lane build by implementing:

```
cd ./android && fastlane build
```

This should produce a signed Android APK.

Note: Don't forget to set environment variables to access keystore. These are RELEASE_STORE_PASSWORD and RELEASE_KEY_PASSWORD and have been set in the example presented above.

Setting up fastlane on iOS

With the Android build being automated, you're ready to move to iOS now. As we discussed earlier, iOS is a bit more complex due to the certification and provisioning profiles. They were designed by Apple to increase security. Fortunately, `fastlane` ships with a few dedicated `actions` that help us overcome these complexities.

You can start with the match action. It helps in managing and distributing iOS certificates and provisioning profiles among your team members. You can read about the idea behind match in the *codesigning.guide concept.*

Simply put, `match` takes care of setting up your device in a way that it can successfully build an application that will be validated and accepted by the Apple servers.

Note: Before you move any further, make sure that your init match for your project. It will generate the required certificates and store them in a central repository where your team and other automation tools can fetch them.

Another action that you could use apart from `match` is `gym`. Gym is similar to the Gradle action in a way that it actually performs the build of your application. To do so, it uses the previously fetched certificates and signs settings from `match`.

```
default_platform(:ios)

ios_directory = File.expand_path("..", Dir.pwd)
base_path = File.expand_path("..", ios_directory)
ios_workspace_path = "#{ios_directory}/YOUR_WORKSPACE.
xcworkspace"
ios_output_dir = File.expand_path("./output", base_path)
ios_app_id = "com.example"
ios_app_scheme = "MyScheme"

before_all do
  if is_ci? && FastlaneCore::Helper.mac?
    setup_circle_ci
  end
end

platform :ios do
  lane :build do |options|
    match(
      type: options[:type],
      readonly: true,
      app_identifier: ios_app_id,
    )

    cocoapods(podfile: ios_directory)

    gym(
      configuration: "Release",
      scheme: ios_app_scheme,
      export_method: "ad-hoc",
      workspace: ios_workspace_path,
      output_directory: ios_output_dir,
      clean: true,
      xcargs: "-UseModernBuildSystem=NO"
    )
  end
end
```

Part of ios/fastlane/Fastfile where iOS lane is defined

You should be able to run lane build by running the same command as for Android:

```
cd ./ios && fastlane build
```

This should produce an iOS application now too.

Deploying the binaries

Now that you have automated the build, you are able to automate the last part of the process - the deployment itself. To do so, you could use App Center, as discussed earlier in this guide.

Note: You have to create an account in the App Center, apps for Android and iOS in the dashboard and generate access tokens for each one of them. You will also need a special Fastlane plugin that brings an appropriate action to your toolbelt. To do so, run fastlane add_plugin appcenter.

Once you are done with configuring your projects, you are ready to proceed with writing the `lane` that will package the produced binaries and upload them to the App Center.

```
lane :deploy do
  build

  appcenter_upload(
    api_token: ENV["APPCENTER_TOKEN"],
    owner_name: "ORGANIZATION_OR_USER_NAME",
    owner_type: "organization", # "user" | "organization"
    app_name: "YOUR_APP_NAME",
    file: "#{ios_output_dir}/YOUR_WORKSPACE.ipa",
    notify_testers: true
  )
end
```

Part of ios/fastlane/Fastfile with upload lane

That's it! Now it is time to deploy the app by executing deploy lane from your local machine.

Integrating with CircleCI

Using all these commands, you are able to build and distribute the app locally. Now, you can configure your CI server so it does the same on every commit to *main*. To do so, you will use *CircleCI* – the provider we have been using throughout this guide.

Note: Running Fastlane on CI server usually requires some additional setup. Refer to official documentation to better understand the difference between the settings in local and CI environments.

To deploy an application from `CircleCI`, you can configure a dedicated `workflow` that will focus on building and deploying the application. It will contain a single job, called `deploy_ios`, that will execute our `fastlane` command.

```yaml
version: 2.1

jobs:
  deploy_ios:
    macos:
      xcode: 14.2.0
    working_directory: ~/CI-CD
    steps:
      - checkout
      - attach_workspace:
          at: ~/CI-CD
      - run: npm install
      - run: bundle install
      - run: cd ios && bundle exec fastlane deploy

workflows:
  deploy:
    jobs:
      - deploy_ios
```

Part of CircleCI configuration that executes Fastlane build lane

Pipeline for Android will look quite similar. The main difference would be the executor. Instead of a macOS one, a docker react-native-android Docker image should be used.

Note: This is a sample usage within CircleCI. In your case, it may make more sense to define filters and dependencies on other jobs, to ensure the deploy_ios is run at the right point in time.

You can modify or parametrize the presented lanes to use them for other kinds of deploys, for instance, for the platform-specific App Store. To learn the details of such advanced use cases, get familiar with the official Fastlane documentation.

EAS Submit

Before using EAS Submit to submit to the app stores, you need a developer account for each store to generate signing credentials for your React Native application.

Generating app-signing credentials

Whether or not you have experience with generating app-signing credentials, you can use EAS CLI to do the heavy lifting for you. The CLI will handle the app-signing credentials process.
For Android, you'll need to generate a keystore for your app. Run the following command to begin:

```
eas credentials
```

The CLI will now prompt you with a series of questions. Select `Android` when asked for the platform and then select `production` as the build profile. The keystore can be generated by selecting "`Keystore: Manage everything needed to build your project`" and "`Set up a new keystore`". The generated keystore is stored securely on EAS servers. For iOS, select iOS as the platform when prompted by the CLI. It will generate a provisioning profile and a distribution certificate by signing

into your Apple Developer account.
This will prepare your `production` build to submit it for review. You can run the following command without specifying any build profile name if the default `production` profile is present in the `eas.json` file:

```
eas build --platform all
```

After the build is complete, you will be able to download the app binaires for each platform. Downloading the binary is required for the Android submission process.

The Android submission process
After building your production build with EAS Build, to submit your app to the Google Play Store, you will need to:

- Create a Google Service Account and download its JSON private key. You can learn about this step in this guide.
- Once you have downloaded the JSON private key, provide its path in your `eas.json` file:

```
{
   "cli":{
      "version":">= 3.3.2"
   },
   "submit":{
      "production":{
         "android":{
            "serviceAccountPath":"../path/to/api-xyz.json",
            "track":"production"
         }
      }
   }
}
```

- Manually upload your app for the first time. This is a limitation of the Google Play Store API.
- The `eas build` command from the previous section will help you generate an app binary that you can directly upload to the Play Sto- retart start the submission process, run the command:

```
eas submit -p android
```

The command will lead you step by step through the process of submitting the application. It will ask you to select a binary to submit. After creating the Android build from the previous section, you can just select the new build.
It will also display a summary of the provided configuration and begin the submission process. After the submission process is complete, the build will be visible through the Google Play Console.

The iOS submission process
After building your production build with EAS Build, to submit your app to the Apple App Store, you will need to run the following command:

```
eas submit -p ios
```

The command will lead you step by step through the process of submitting the app. It will prompt you to:

- Log in to your Apple Developer account and select your team. You can also provide this information in `eas.json` by setting the appleId and appleTeamId fields under the submit production submission profile. The Apple ID password has to be set with the `EXPO_APPLE_PASSWORD` environment variable.
- Select a binary to submit. After creating the iOS build from the previous section, you can just select the new build.
- It will display a summary of the provided configuration and begin the submission process. After the submission process is complete, the build will be visible on the App Store Connect website.

Automating submissions
As a developer, you can save time when your app deployment process has evolved to the point where the app is automatically submitted to the app stores once a production build completes.
EAS Build gives you automatic submissions out of the box with the

`--auto-submit flag`. This flag tells EAS Build to pass the build along to EAS Submit upon completion. The flag will also try to use a submission profile with the same name as the selected build profile. If you don’t have a custom build profile, you can select and use the default production profile for creating a build and submitting it to the app stores.

When you run `eas build --auto-submit` you will be provided with a link to a submission details page, where you can track the progress of the submission. You can also find this page at any time from the Submissions section of the Expo dashboard for your project, and it is also linked from your submitted build’s detail page.

Benefits: A short feedback loop along with nightly or weekly builds lets you verify features faster and ship critical bugs more often.

With automated deployment, you no longer waste your time on manual builds and sending the artifacts to test devices or app stores. Your stakeholders are able to verify features faster and shorten the feedback loop even further. With regular builds, you will be able to catch or ship fixes to any critical bugs with ease

PART 3 | CHAPTER 4

Ship OTA (Over-The-Air) when in an emergency

Submit critical updates and fixes instantly through OTA.

Issue: Traditional ways of updating apps are too slow and you lose your precious time on them.

The traditional model of sending updates on mobile is fundamentally different from the one we know from writing JavaScript applications for other platforms. Unlike the web, mobile deployment is much more complex and comes with better security out-of-the-box. We have talked about that in detail in the previous section focused on the CI/CD.

What does it mean for your business?
Every update, no matter how quickly shipped by your developers, is usually going to wait some time while the App Store and Play Store teams review your product against their policies and best practices.

This process is particularly challenging in all Apple platforms, where apps are often taken down or rejected, because of not following certain policies or meeting the required standard for the user interface. Thankfully, the risk of your app being rejected with React Native is reduced to a minimum, as you're working on the JavaScript part of the application. The React Native Core Team ensures that all the changes done to the framework have no impact on the success of your application's submission.

As a result, the submission process takes a while. And if you're about to ship a critical update, every minute counts.

Fortunately, with React Native, it is possible to dynamically ship your JavaScript changes directly to your users, skipping the App Store review process. This technique is often referred to as an over-the-air update. It lets you change the appearance of your application immediately, for all the users, following the technique that you have selected.

When critical bugs happen - minutes and hours can be critical. Don't wait to fix your end users' experience.

If your application is not OTA-ready, you risk it being left with a critical bug on many devices, for as long as Apple/Google reviews your product and allows it to be distributed.

Even though the review times have gotten much better over the years, it is still a good escape hatch to be able to immediately recover from an error that slipped through the testing pipeline and into production.

Solution: Implement OTA updates with App Center/ CodePush or EAS Update

As mentioned earlier, React Native is OTA-ready. It means that its architecture and design choices make such updates possible. However, it doesn't ship with the infrastructure to perform such operations. To do so, you will need to integrate a 3rd-party service that carries its own infrastructure for doing so.

There are two popular ways to implement OTA into your app. The first tool for OTA updates is CodePush, a service that is a part of Microsoft's App Center suite. The second tool is created by the Expo and is called EAS Update.

App Center/CodePush

Configuring the native side

To integrate CodePush into your application, please follow the required steps for iOS and Android, respectively. We decided to link to the official guides instead of including the steps here as they include additional native code to apply and that is very likely to change in the coming months.

Configuring the JavaScript side

Once you set up the service on the native side, you can use the JavaScript API to enable the updates and define when they should happen. One of the ways that enable fetching updates on the app startup is to use the codePush wrapper and wrap your main component.

```
import React from 'react';
import { View } from 'react-native';
import codePush from 'react-native-code-push';

const MyApp = () => <View />;

export default codePush(MyApp);
```

Basic CodePush integration

That's it! If you have performed all the changes on the native side, your application is now OTA-ready.

For more advanced use cases, you can also change the default settings on when to check for updates and when to download and apply them. For example, you can force CodePush to check for updates every time the app is brought back to the foreground and install updates on the next resume.

The following diagram code snippet demonstrates such a solution:

```
import React from 'react';
import { View } from 'react-native';
import codePush from 'react-native-code-push';

const MyApp = () => <View />;

export default codePush({
  updateDialog: true,
  checkFrequency: codePush.CheckFrequency.ON_APP_RESUME,
  installMode: codePush.InstallMode.ON_NEXT_RESUME,
})(MyApp);
```

Custom CodePush setup

Shipping updates to the application

After configuring CodePush on both JavaScript and the native side of React Native, it is time to launch the update and let your new customers enjoy it. To do so, we can do this from the command line, by using the App Center CLI:

```
npm install -g appcenter-cli
appcenter login
```

And then, a *release* command to bundle React Native assets and files and send them to the cloud:

```
appcenter codepush release-react -a <ownerName>/<appName>
```

Once these steps are complete, all users running your app will receive the update using the experience you configured in the previous section.

Note: Before publishing a new CodePush release, you will have to create an application in the App Center dashboard.

That will give you the *ownerName* and *appName* that you're looking for. As said before, you can either do this via UI by visiting App Center, or by using the App Center CLI.

EAS Update

One of the services provided by EAS is EAS Update. It provides first-class support for instant updates in React Native applications and is especially easy if you're already using Expo. EAS Update serves updates from the edge with a global CDN and uses modern networking protocols like HTTP/3 for clients that support them. It implements the Expo Updates protocol, an open, standard specification for instant updates. As with other Expo products, EAS Update provides superior DX and is often a pleasure to work with.

You will need to install the `eas-cli` package. We have already installed it in the "Have a working Continuous Integration (CI) in place" chapter.

Info: To get EAS Update working in your project with the bare React Native workflow, you need to set up Expo in your project. See the guide to make that work correctly.

Before proceeding with the further sections, make sure you have EAS Build configured in your project.

Info: To build your app you can use either EAS Build or another tool. If you don't want to use EAS Build, here's the guide on what to do.

Setting up EAS Update

Then you need to initialize your project for EAS Update:

```
eas update:configure
```

Creating a build

Follow the steps to create a build of your app using EAS Build or another tool of your choice. The new build will include the expo-updates native module, which will be responsible for downloading and launching your updates. Install the build on your device or an emulator or simulator.

Creating an update

Once you have installed the new build on your device, we're ready to send an update to it! Make a small, visible change to the JS part of your application to help you confirm when your device is running the new update. Now, you're ready to run the command to create an update and publish it to EAS.

```
eas update --branch production --message "Fixed a bug."
```

Once these steps are complete, all users running your app will receive the update with your changes. By default, `expo-updates` checks for updates in the background when an application launches, and this behavior is configuration. With the default behavior, though, terminate your application running on your device and launch it to fetch the new update, which usually takes a few short moments. Terminate and launch your application again and you should see your new changes live!

You can configure EAS Update in many different ways. EAS Update has the concept called `channels`, which are a way to send different updates to different builds of your application. There are more advanced concepts like `branches`, which enable bespoke deployment workflows inspired by those used by some of the largest mobile development teams in the world.

EAS Update is very handy when you need to deploy your changes to production, and it's also very handy during development. It's a very convenient and fast way to share your work with your teammates.

Benefits: Ship critical fixes and some content instantly to the users.

With OTA updates integrated into your application, you can send your JavaScript updates to all your users in a matter of minutes. This possibility may be crucial for fixing significant bugs or sending instant patches.

For example, it may happen that your backend will stop working and it causes a crash at startup. It may be a mishandled error - you never had a backend failure during the development and forgot to handle such edge cases.

You can fix the problem by displaying a fallback message and informing users about the problem. While the development will take you around one hour, the actual update and review process can take hours if not days. With OTA updates set up, you can react to this in minutes without risking the bad UX that will affect the majority of users.

PART 3 | CHAPTER 5

Make your app consistently fast

Use the DMAIC process to help you prevent regressing on app performance

Issue: Every once in a while after fixing a performance issue, the app gets slow again.

Customers have very little patience for slow apps. There is so much competition on the market that customers can quickly switch to another app. According to the Unbounce report, nearly 70% of consumers admit that page speed influences their willingness to buy. Good examples here are Walmart and Amazon—both of these companies noticed an increase in revenue by up to 1% for every 100 milliseconds of load time improvement. The performance of websites and mobile apps can thus noticeably impact businesses' performance.

It's becoming increasingly important to not only fix performance issues but also make sure they don't happen again. You want your React Native app to perform well and fast at all times.

Solution: Use the DMAIC methodology to help you solve performance issues consistently.

From the technical perspective, we should begin by avoiding any guesswork and base all decisions on data. Poor assumptions lead to false results. We should also remember that improving performance is a process, so it's impossible to fix everything at once. Small steps can provide big results.

This all leads us to the fact that developing an app is a process. There are some interactions that lead to results. And, what is most important, the processes can be optimized.

One of the most effective ways of doing that is using the DMAIC methodology. It's very data-driven and well-structured and can be used to improve React Native apps. The acronym stands for Define, Measure, Analyze, Improve, and Control. Let's see how we can apply each phase in our apps.

Define

In this phase, we should focus on defining the problem, what we want to achieve, opportunities for improvement, etc. It's important to listen to the customer's voice in this phase - their expectations and feedback. It helps to better understand the needs and preferences and what problems they are facing. Next, it is very important to measure it somehow. Let's say the customer wants a fast checkout. After analyzing the components, we know that to achieve this we need a swift checkout process, a short wait time, and smooth animations and transitions. All of these points can be decomposed into CTQ (Critical-to-Quality) that are measurable and can be tracked. For example, a short wait time can be decomposed into a quick server response and a low number of server errors.

Another handy tool is analyzing common user paths. With good tracking, we can analyze and understand what parts of the app are mostly used by the users.

In this phase, it's very important to choose priorities. It should end up with defining the order in which we will optimize things. Any tools and techniques for prioritizing will definitely help here.

Ultimately, we need to define where we want to go - we should define our goals and what exactly we want to achieve. Keep in mind that it all should be measurable! It's a good practice to put these goals in the project scope.

Measure

Since we already know where we want to go, it's time to assess the starting point. It's all about collecting as much data as possible to get the actual picture of the problem. We need to ensure the measurement process is precise. It's really helpful to create a data collection plan and engage the development team to build the metrics. After that, it's time to do some profiling.

When profiling in React Native, the main question is whether to do this on JavaScript or the native side. It heavily depends on the architecture of the app, but most of the time it's a mix of both.

One of the most popular tools is React Profiler, which allows us to wrap a component to measure the render time and the number of renders. It's very helpful because many performance issues come from unnecessary rerenders. Discover how to use it here:

```
import React, { Profiler } from 'react';
import { View } from 'react-native';

const Component = () => (
  <Profiler id="Component" onRender={(...args) => console.
log(args)}>
    <View />
  </Profiler>
);

export default Component;
```

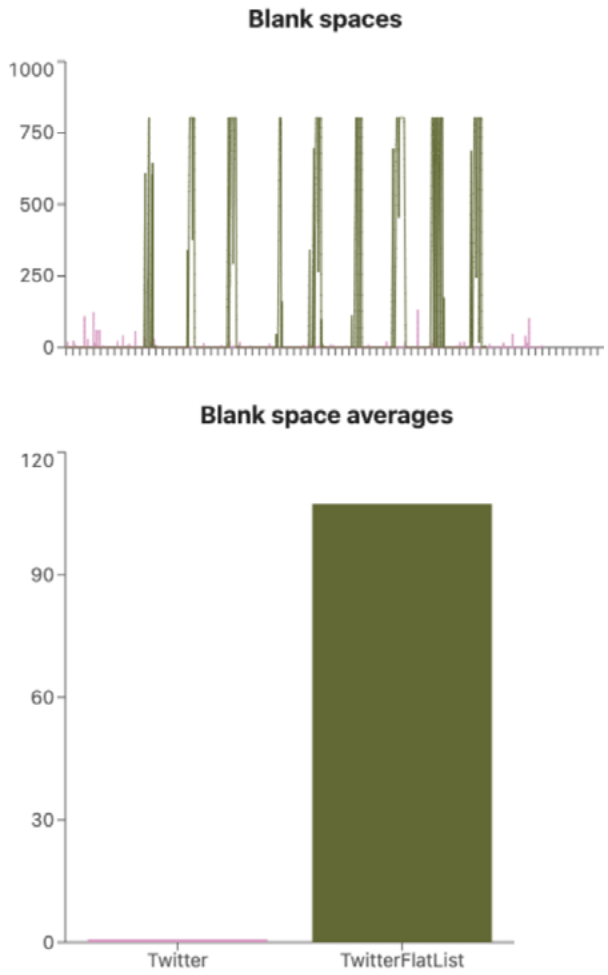
Using React Profiler API

It will output the data:

```
{
 id: "Component",
 phase: "mount",
 actualDuration: 1.3352311453,
 baseDuration: 0.95232323318,
 ...
}
```

Output of the React Profiler API

The second tool is a library created by Shopify - react-native-performance. It allows you to place some markers in the code and measure the execution time. There is also a pretty nice Flipper plugin that helps to visualize the output:



https://shopify.github.io/react-native-performance/docs/guides/flipper-react-native-performance

Speaking of Flipper, it has some more plugins that help us to measure the app performance and speed up the development process. We can use, e.g. React Native Performance Monitor Plugin for a Lighthouse-like experience or React Native Performance Lists Profiler Plugin.

On the native side, the most common method is using Native IDEs - Xcode and Android Studio. There are plenty of useful insights which can be analyzed and lead to some conclusions and results.

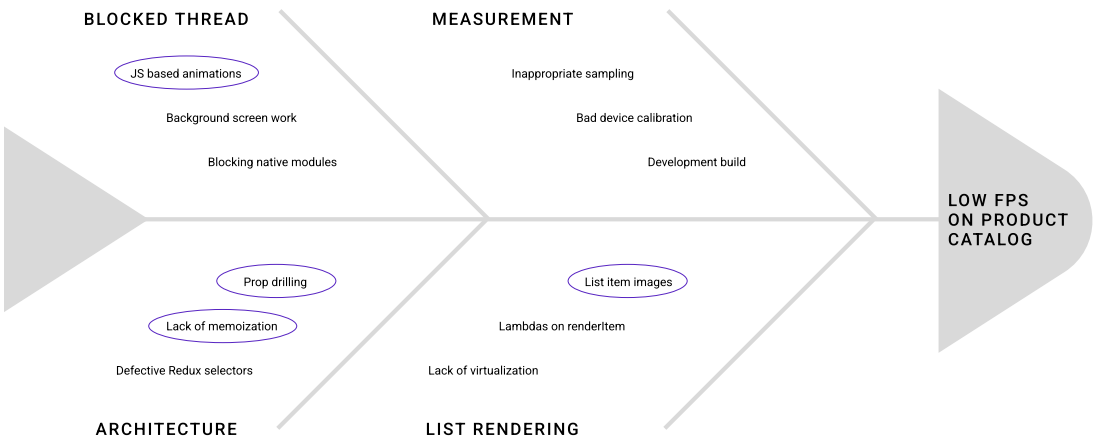
The most important aspect of this phase is measurement variation. Due to different environments, we have to be very careful when profiling. Even if the app is run on the same device, there might be some external factors that affect performance measurements. That's why we should base all the measurements on release builds.

Analyze

The goal of this phase is to find the root cause of our problem. It's a good idea to start with a list of things that could potentially cause the problem. A little brainstorming with a team is really helpful here.

One of the most popular tools to define a problem is called a cause and effect diagram. It looks like a fish and we should draw it from right to left. We start from the head and it should contain the problem statement - at this stage, we should already have it based on the Define phase. Then, we identify all the potential major causes of the problem and assign them to the fish bones. After that, we assign all the potential causes to each major cause. There are many things that could have an impact on performance. The list could get really long, so it's important to narrow it down. Outline the most important factors and focus on them.

Finally, it's time to test the hypothesis. For example, if the main problem is low FPS, and the potential major cause is related to list rendering, we can think of some improvements in the area of images in the list items. We need to design a test that will help us accept or reject the hypothesis - it will probably be some kind of proof of concept. Next, we interpret the results and decide if it was improved or not. Then we make a final decision.



Cause and effect diagram example

Improve

Now we know what our goal is and how we want to achieve it, it's time to make some improvements. This is the phase where optimization techniques start to make sense.

Before starting, it's a good idea to have the next brainstorming session and identify potential solutions. Depending on the root cause, there might be a lot of them. Based on the last example with images on the list item, we can think about implementing proper image caching and reducing unnecessary renders.

After outlining the solutions, it's time to pick the best one. Sometimes the solution that gives the best effects might be extremely costly, e.g. when it's necessary to make some architectural changes.

It's then time to implement the solution. After that, it's required to properly test it and we are done!

Control

The last step is the control phase. We need to make sure that everything works well now. The performance will degrade if it is not under control. People tend to blame devices, the used technology, or even users when it comes to bad performance. So what do we need to do to keep our performance on a high level?

We need to make sure that we have a control plan. We can use some of our work from the previous phases to make it. We should point out focal points, some measurement characteristics, acceptable ranges for indicators, and testing frequency. Additionally, it is a good practice to write down some procedures and what to do if we spot issues.

The most important aspect of the control phase is monitoring regressions. Until recently it was quite difficult to do that in React Native, but now we have plenty of options to improve our monitoring.

Real-time user monitoring

One way to keep the performance improvements we introduce in our apps is through real-time monitoring tools. Such as Firebase Performance Monitoring, which is a service that gives us some insights into performance issues in production. Or Sentry Performance Monitoring, which tracks application performance, collects metrics like throughput and latency, and displays the impact of errors across multiple services.

It's a great addition to any app builders that want to have insights on how the performance is distributed across all the devices that install their apps. Based on real user data.

Testing regressions as a part of the development process

Another way to keep performance regressions under control is through automated testing. Profiling, measuring, and running on various devices is quite manual and time-consuming. That's why developers avoid doing it. However, it gets too easy to unintentionally introduce performance regressions that would only get caught during QA, or worse, by your users. Thankfully, we have a way to write automated performance regression tests in JavaScript for React and React Native.

Reassure allows you to automate React Native app performance regression testing on CI or a local machine. In the same way you write your integration and unit tests that automatically verify that your app is still working correctly, you can write performance tests that verify that your app is still working performantly. You can think about it as a React performance testing library. In fact, Reassure is designed to reuse as much of your React Native Testing Library tests and setup as possible. As it's designed by its maintainers and creators.

It works by measuring certain characteristics—render duration and render count—of the testing scenario you provide and comparing that to the stable version measured beforehand. It repeats the scenario multiple times to reduce the impact of random variations in render times caused by the runtime environment. Then it applies a statistical analysis to figure out whether the code changes are statistically significant or not. As a result, it generates a human-readable report summarizing the results and displays it on CI or as a comment to your pull request.

The simplest test you can write would look something like this:

```
import React from 'react';
import { View } from 'react-native';
import { measurePerformance } from 'reassure';

const Component = () => {
  return <View />;
};

test('mounts Component', async () => {
  await measurePerformance(<Component />);
});
```

Code: Component.perf-test.tsx

This test will measure the render times of Component during mounting and the resulting sync effects. Let's take a look at a more complex example though. Here we have a component that has a counter and a slow list component:

```
import React from 'react';
import { Pressable, Text, View } from 'react-native';
import { SlowList } from './SlowList';

const AsyncComponent = () => {
  const [count, setCount] = React.useState(0);

  const handlePress = () => {
    setTimeout(() => setCount((c) => c + 1), 10);
  };

  return (
    <View>
      <Pressable accessibilityRole="button"
onPress={handlePress}>
        <Text>Action</Text>
      </Pressable>

      <Text>Count: {count}</Text>

      <SlowList count={200} />
    </View>
  );
};
```

And the performance test looks as follows:

```
import React from 'react';
import { screen, fireEvent } from '@testing-library/react-
native';
import { measurePerformance } from 'reassure';
import { AsyncComponent } from '../AsyncComponent';

test('AsyncComponent', async () => {
  const scenario = async () => {
    const button = screen.getByText('Action');

    fireEvent.press(button);
    await screen.findByText('Count: 1');

    fireEvent.press(button);
    await screen.findByText('Count: 2');

    fireEvent.press(button);
    fireEvent.press(button);
    fireEvent.press(button);
    await screen.findByText('Count: 5');
  };

  await measurePerformance(<AsyncComponent />, { scenario });
});
```

When run through its CLI, Reassure will generate a performance comparison report. It's important to note that to get a diff of measurements, we need to run it twice. The first time with a `--baseline` flag, which collects the measurements under the `.reassure/ directory`.

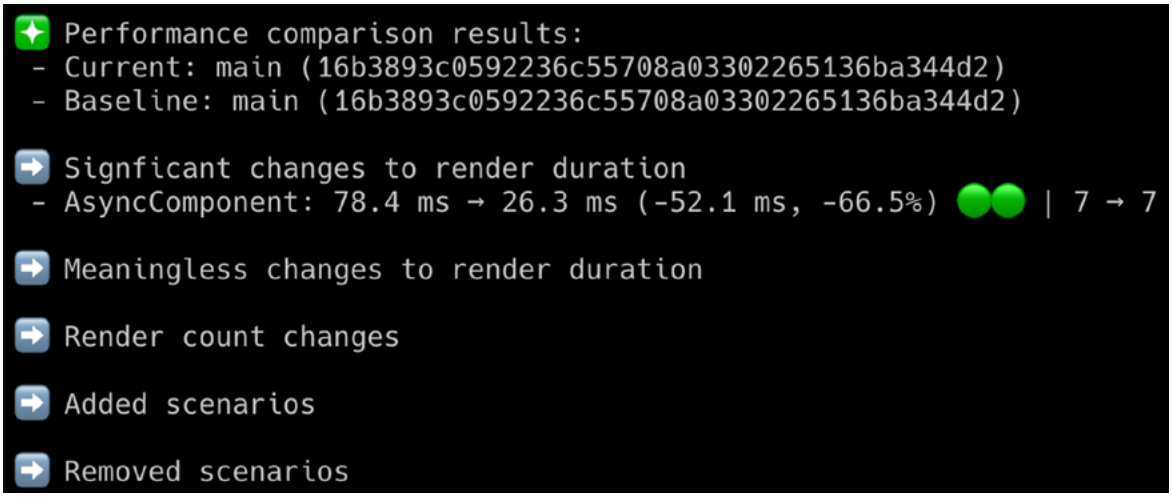
```
Running performance tests:
- Baseline: main (16b3893c0592236c55708a03302265136ba344d2)

PASS src/AsyncComponent.perf-test.tsx
  ✓ AsyncComponent (2982 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.769 s, estimated 5 s
Ran all test suites.

✅ Written Baseline performance measurements to .reassure/baseline.perf
```

After running this command, we can start optimizing our code and see how it affects the performance of our component. Normally, we would keep the baseline measurement and wait for performance regressions to be caught and reported by Reassure. In this case, we'll skip that step and jump straight into optimizing, because we just noticed a nice possibility to do so. And since we have our baseline measurement for reference, we can actually verify our assumptions and whether the improvement was real or only subjective.

The possibility we noticed is that the `<SlowList/>` component can be memoized, as it doesn't depend on any external variables. We can leverage `useMemo` for that case:

```
const slowList = useMemo(() => <SlowList count={200} />, []);
```

Once we're done, we can run Reassure a second time. Now without the `--baseline` flag.



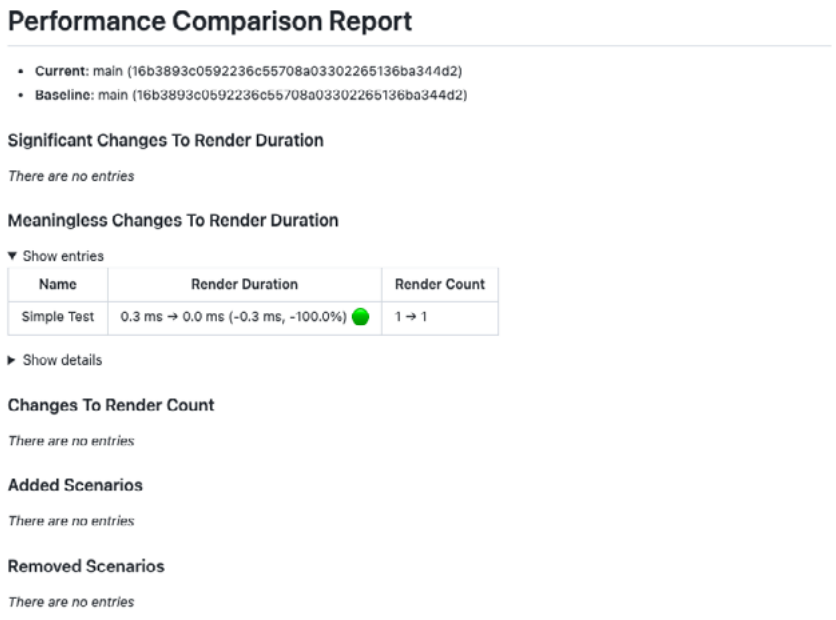
Performance comparison report from Reassure

Now that Reassure has two test runs to compare—the current and the baseline—it can prepare a performance comparison report. As you can notice, thanks to applying memoization to the `SlowList` component rendered by `AsyncComponent`, the render duration went from 78.4 ms to 26.3 ms, which is roughly a 66% performance improvement.

Test results are assigned to certain categories:

- **Significant Changes To Render Duration** shows a test scenario where the change is statistically significant and should be looked into as it marks a potential performance loss/improvement.
- **Meaningless Changes To Render Duration** shows test scenarios where the change is not statistically significant.
- **Changes To Render Count** shows test scenarios where the render count did change.
- **Added Scenarios** shows test scenarios that do not exist in the baseline measurements.
- **Removed Scenarios** shows test scenarios that do not exist in the current measurements.

When connected with Danger JS, Reassure can output this report in the form of a GitHub comment, which helps catch the regressions during code review.



Performance Comparison Report

- **Current:** main (16b3893c0592236c55708a03302265136ba344d2)
- **Baseline:** main (16b3893c0592236c55708a03302265136ba344d2)

Significant Changes To Render Duration

There are no entries

Meaningless Changes To Render Duration

▼ Show entries

Name	Render Duration	Render Count
Simple Test	0.3 ms → 0.0 ms (-0.3 ms, -100.0%)	1 → 1

▶ Show details

Changes To Render Count

There are no entries

Added Scenarios

There are no entries

Removed Scenarios

There are no entries

Report generated by Reassure with Danger JS

You can discover more use cases and examples in the docs.

Benefits: A well-structured and organized optimization process.

When working on an app, regardless of its size, it's important to have a clear path for reaching our goals. The main benefit of using DMAIC when optimizing React Native applications is a structured and direct approach. Without it, it may be difficult to verify what works (and why). Sometimes our experience and intuition are just enough. But that's not always the case.

Having a process like this allows us to focus on problem-solving and constantly increase productivity. Thanks to the DMAIC approach, performance optimization becomes a part of your normal development workflow. Making your app closer to being performant by default. Spotting the performance issues even before they hit your users.

No software is flawless. Bugs and performance issues will happen even if you're the most experienced developer on the team. But we can take action to mitigate those risks by using automated tools like Sentry, Firebase, or Reassure. Use them in your project and enjoy the additional confidence they bring to your projects. And the improved UX they bring to your users in turn.

"Performance regression monitoring is a critical process in the development and maintenance of mobile apps. Without it, small issues can go unnoticed and lead to significant performance degradation, negatively impacting the user experience and potentially decreasing user retention. Regular performance regression monitoring allows developers to proactively identify and fix issues before they become a problem for users, ensuring the app runs at optimal performance and providing a better experience for all users."

Michał Chudziak
Independent Consultant @michalchudziak.dev

PART 3 | CHAPTER 6

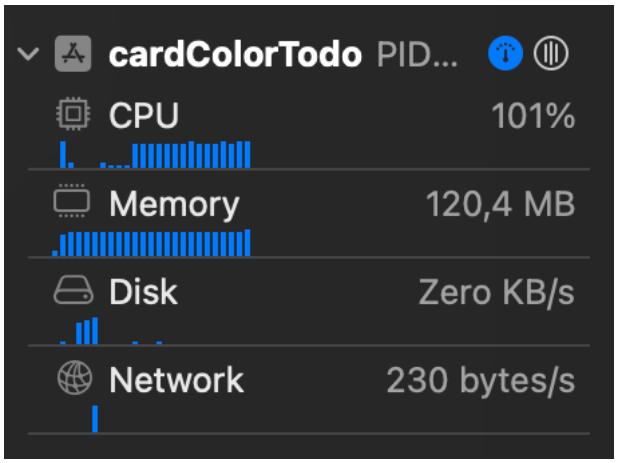
Know how to profile iOS

Improve your app with real-time metrics

Issue: It takes too much time to see the result of an action.

Profiling is essential to understanding the runtime performance of the app, through analysis that measures the memory or time complexity, frequency, and duration of function calls, etc. Getting all this information helps you to track down and provide proper solutions to keep your app healthy and your users engaged.

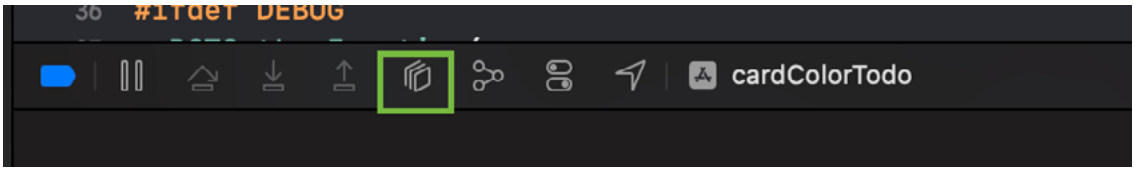
Xcode provides some basic tools to do the first report. You can monitor the CPU, Memory, and Network.



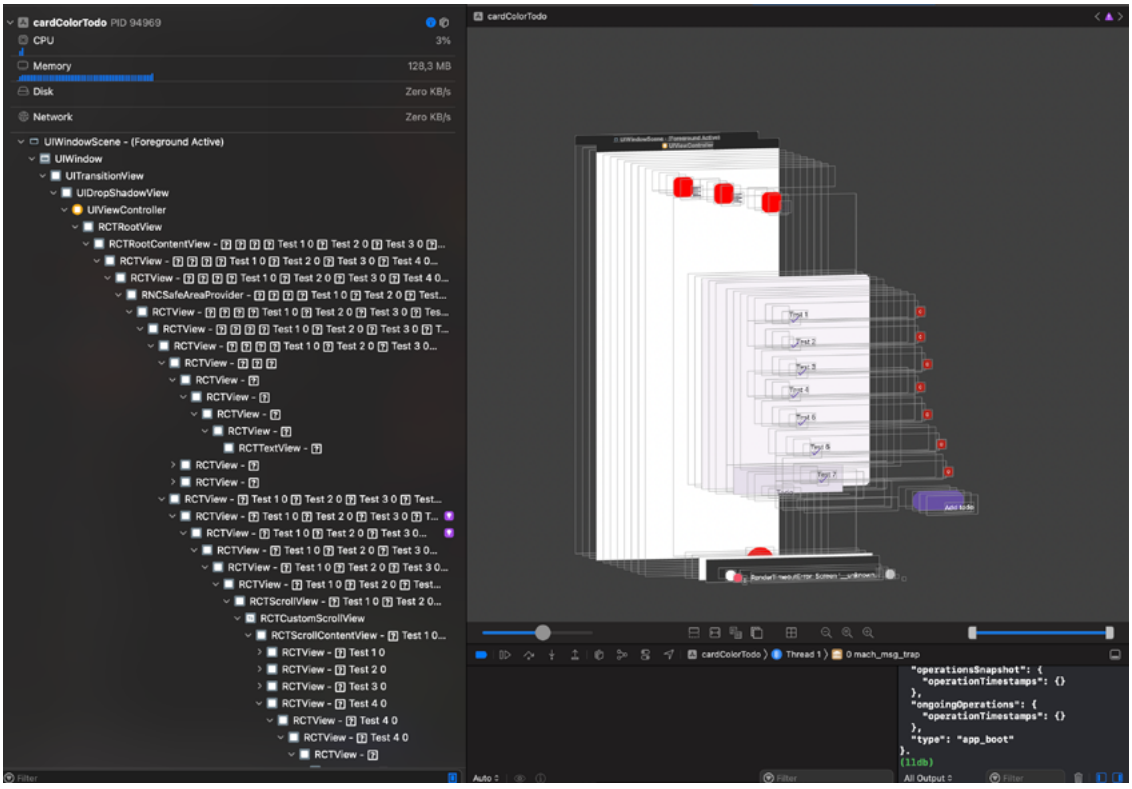
CPU Monitor measures the amount of work done. Memory Monitor is for observing the use of the app. All iOS devices use SSD for permanent storage, accessing this data is slower compared to RAM. Disk Monitor is for understanding your app's disk-writing performance. Network Monitor analyzes your iOS app's TCP/IP and UDP/IP connections.

You can tap on each of them to find more information.

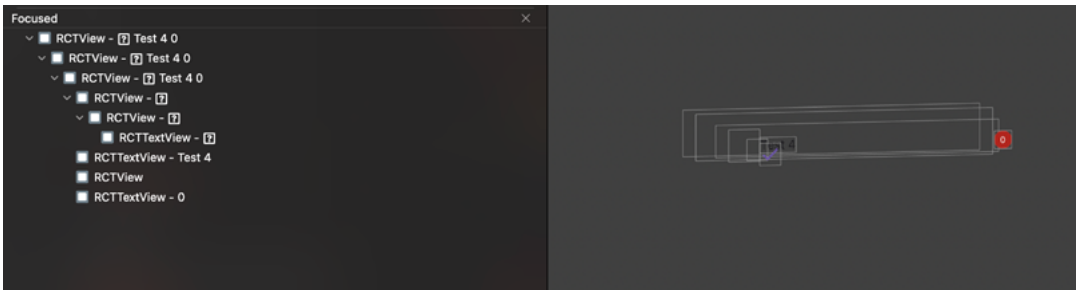
It also provides an extra monitor that isn't shown by default but can help you inspect your UI - it's the View Hierarchy.
When the app is running and you are on the screen you want to inspect, click on Debug View Hierarchy.



This will show your current UI in a 2D/3D model and the view tree.



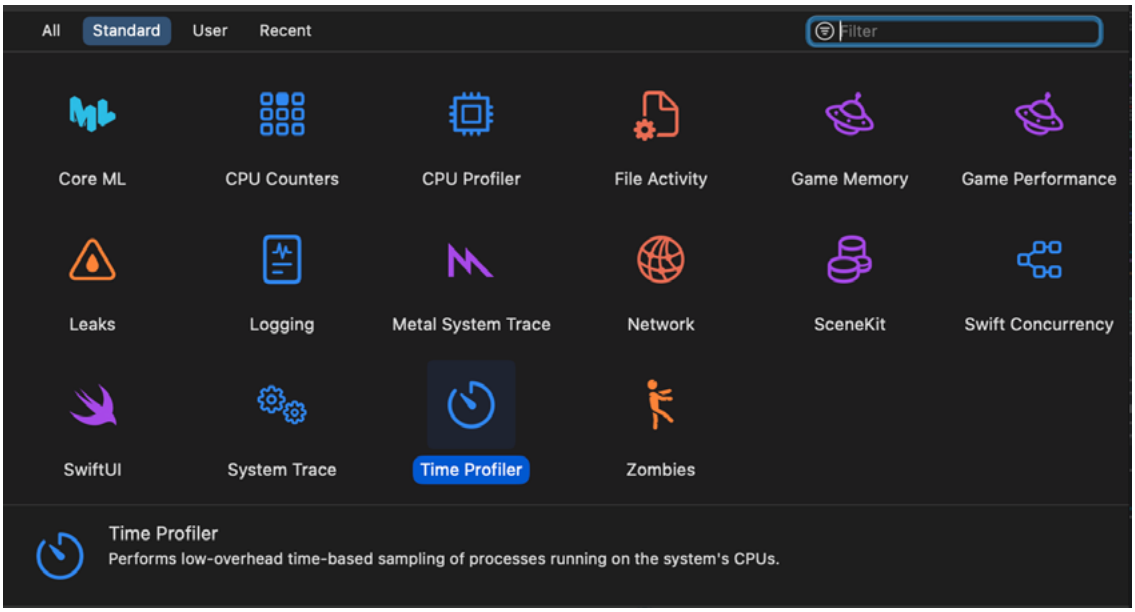
This will help you to detect overlappings (you can't see a component) or if you want to flatten your component tree. Even though RN does a view flattening it sometimes can't do it with all of them, so here we can do some optimization focusing on specific items.



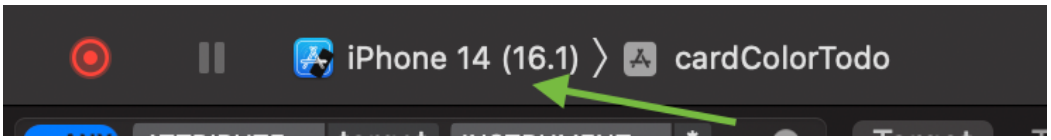
Let's say we have a TODO list app, and when the Add button is pressed, it adds the new item to the list. However, it takes a couple of seconds to show up on the list because there is some logic before the item is added. Let's go to our dev toolbox and pick up our first instrument so we can confirm and measure the delay.

iOS Instruments

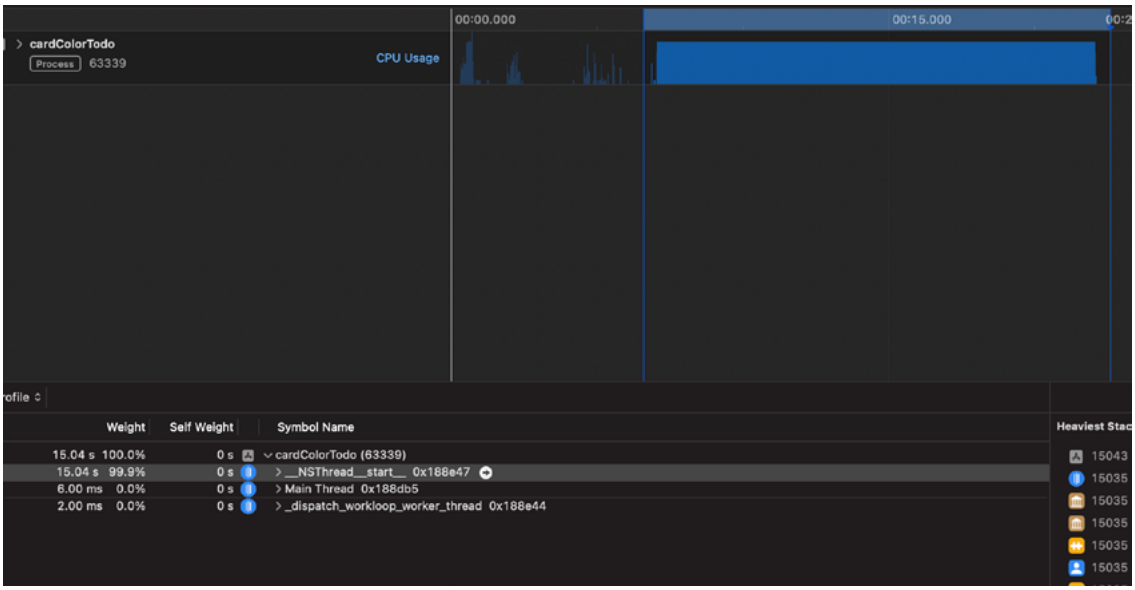
Instruments is a debugging and profiling tool that comes prepackaged with xCode, and is literally a box of tools, each of them serving a different purpose. You choose from a list of templates, and you choose any of them depending on your goal: improving performance or battery life or fixing a memory problem.
We are going to use Time Profiler. Let's dive into it. With xCode open, we go to Open Developer Tool -> Instruments. Then, scroll down to find the Time Profiler tool.



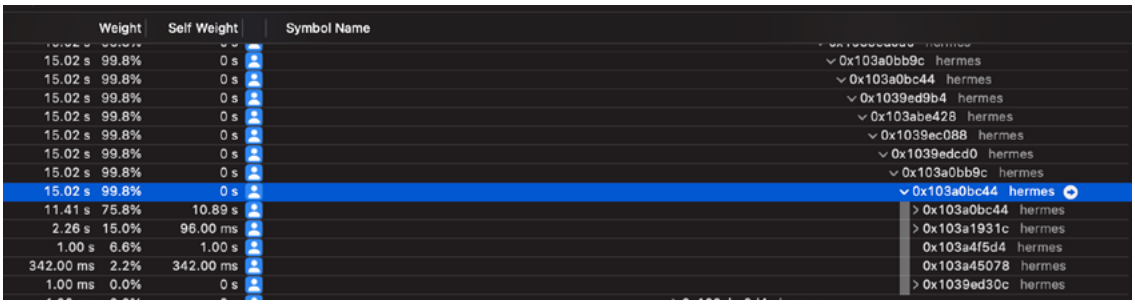
It will open a new window. To start profiling your app, click on the drop-down menu and select your device and the app.



When the app opens, start using it normally, or in this case, add a new TODO item.



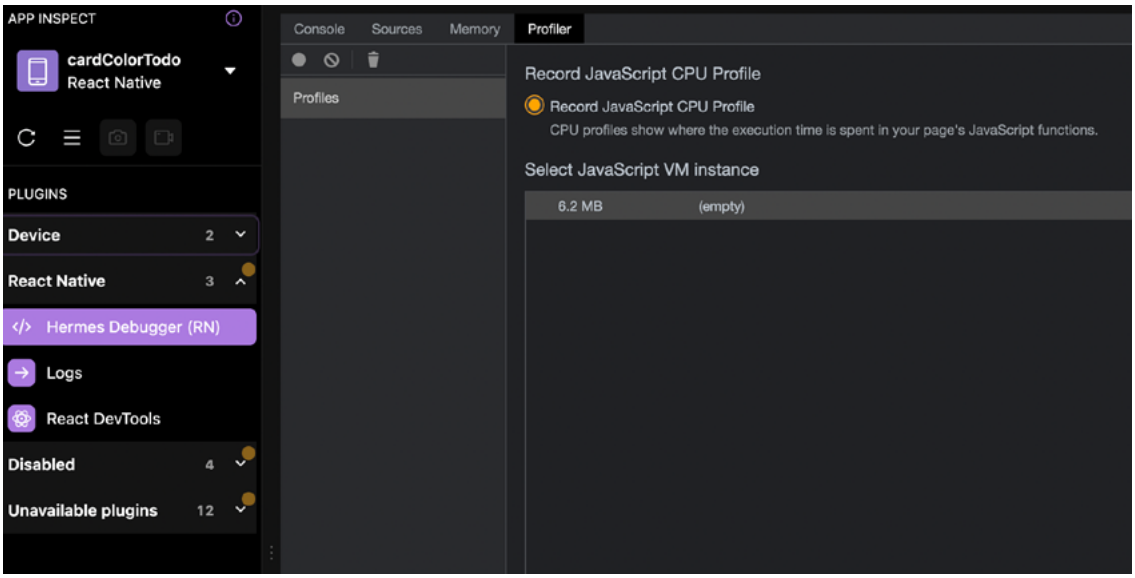
After playing around and adding the new TODO item, we can see there is a big blue rectangle, which means there is something that is taking a lot of time to finish. Let's take a look at the threads.



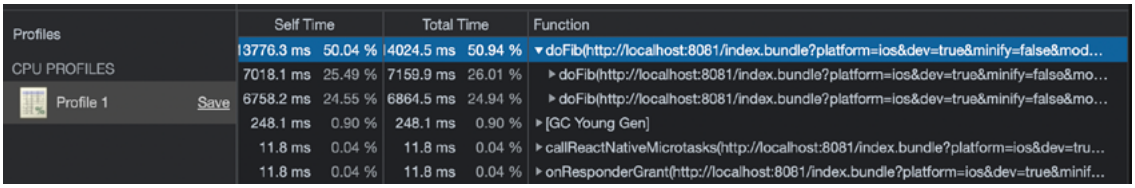
You can expand by pressing option+click over the chevron, which will expand to display useful information. At least for now it is showing the memory address, but we will need to find another way to find where the problem is.

Solution: Combining with a tool specific for JS Context tracking.

Let's use Flipper, the same one that we used in Pay Attention to UI re-renders, but we are going to use another monitor called Hermes Debugger (RN). With the app open and running, we go to Flipper, select the running app if not selected already, and go to Hermes Debugger (RN) -> Profiler

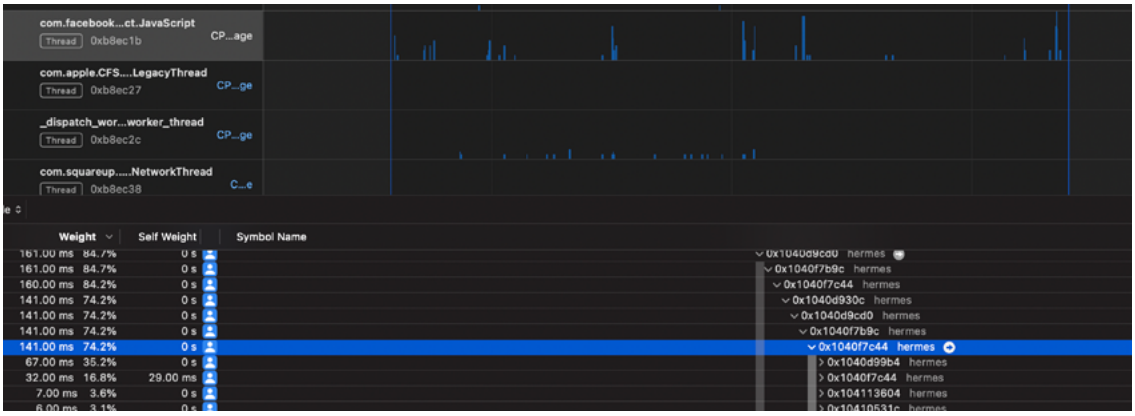


We click Start so the profiler begins. We do the same flow and actions as before when profiling with Time Profiler. When we Stop, we will see all the data collected.



By default the data will be sorted bottom-up with the heavy tasks at the top. We can see that a function called doFib is taking ~14 sec to complete, it is a good start, let's go into that function and see what we can do. The fixes will vary depending on your code.

After applying a possible fix, we first check Time Profiler again. We click on the record button and start using the app, in our case let's add a new TODO item.



As we can see, the fix we applied did work, we aren't seeing the big blue rectangle like before. This is a good sign. Let's continue with our profiling path to check how it looks in Flipper.
Start profiling the app one more time using Hermes Debugger (RN) -> Profiler.

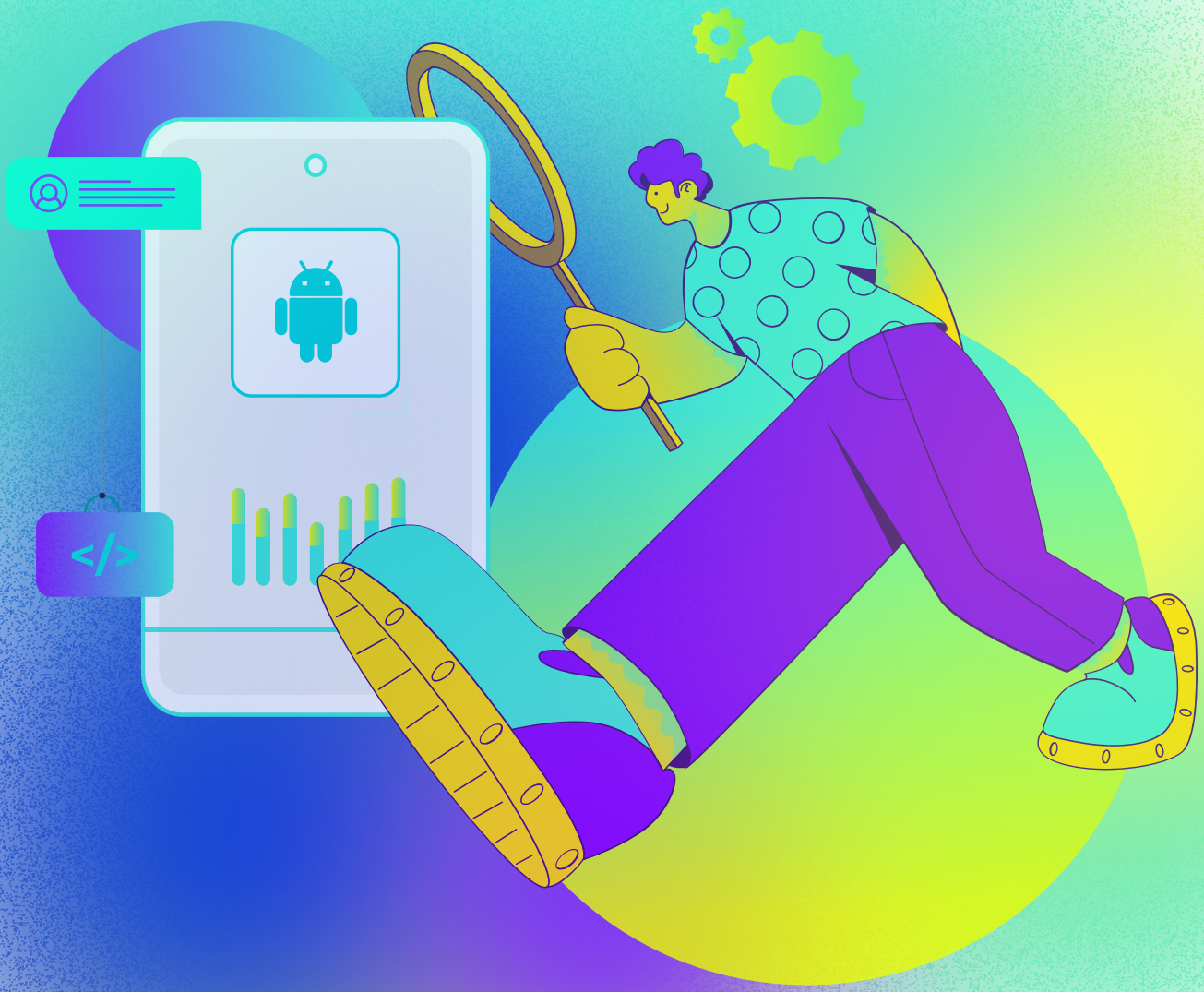
We don't see the doFib function anymore, only other expected RN tasks.

Benefits: Having a faster and more responsive app.

70% of the users will leave the app if the response to a given action takes too long. Profiling our apps has become one of the main steps in our development life cycle. Using specific tools like Time Profiler will help us understand if our app is responding fast or where we could find areas of improvement. Remember, users are becoming more sensitive to speed and delays, even a 0.1 sec of improvement can increase a conversion rate by 10.1%.

PART 3 | CHAPTER 7

Know how to profile Android



Get real-time metrics to better your app understanding

Issue: You encounter a performance issue that comes directly from Android runtime.

In the event of any performance issues, we mostly use React Profiler to troubleshoot and resolve our problems. Since most of the performance problems originate from the JS realm, we don't usually need to do anything beyond that. But sometimes we'll encounter a bug or performance issue that comes directly from the Android runtime. In such a case, we'll need a fine tool to help us gather the following metrics from the device:

- CPU
- memory
- network
- battery usage

Based on that data, we can check whether our app consumes more energy than usual or in some cases, uses more CPU power than it should. It is useful especially to check the executed code on lower-end (LE) Android devices. Some algorithms can run faster on some devices and the end user will not spot any glitches, but we have to remember, some customers can use LE devices and the algorithm or function can be too heavy for their phones. High-end devices will handle it because their hardware is powerful.

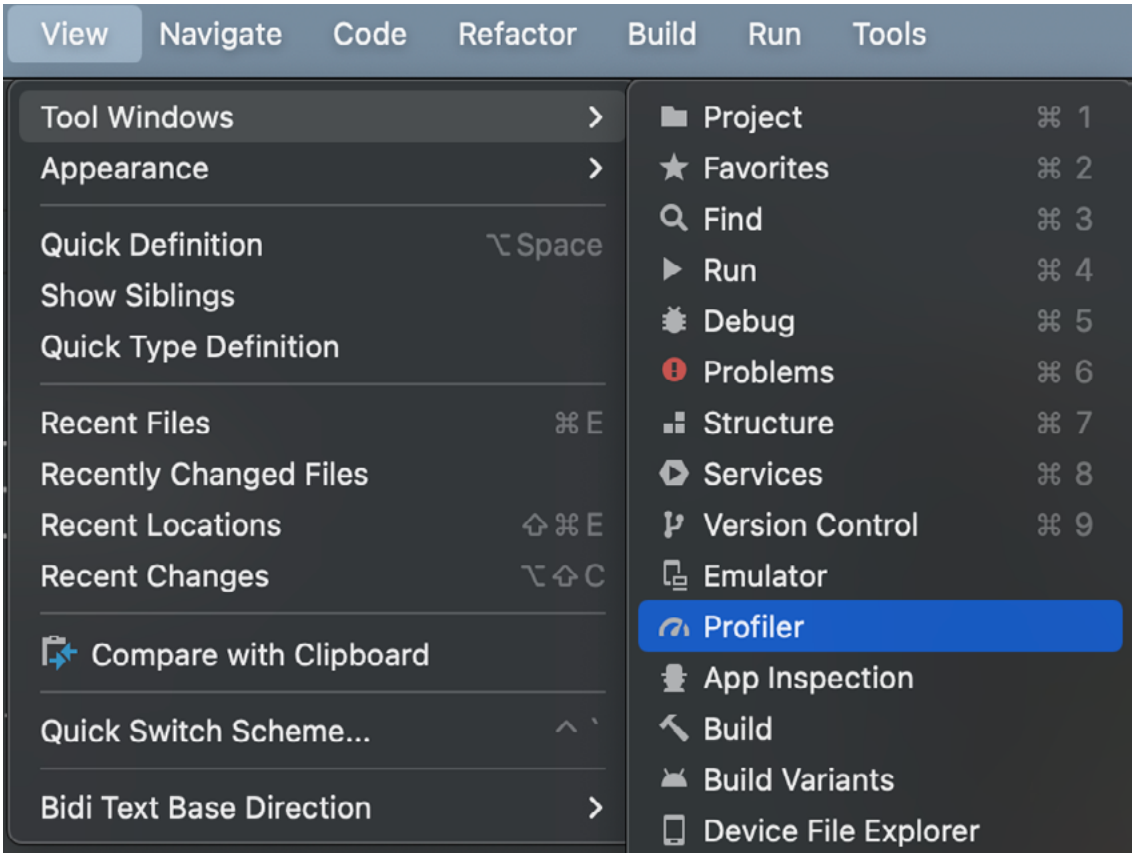
Solution: Profile your app with Android Profiler in Android Studio

Android Profiler in Android Studio

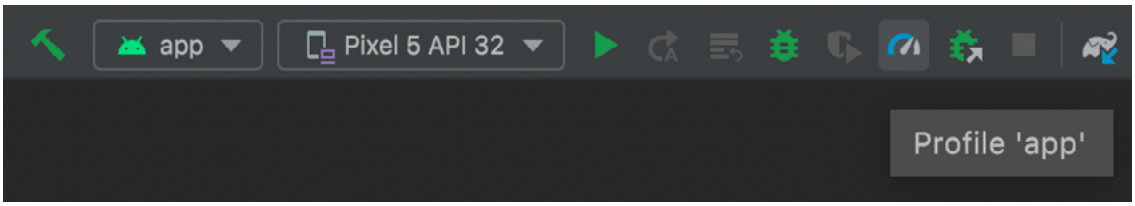
Android Studio is the IDE developed by JetBrains. It is officially supported by Google and the official IDE, which can be used to develop any Android app. It is very powerful and contains lots of functionalities in one place. One of those tools is Android Profiler.

If you have not installed Android Studio yet, you can install it using this link.

To open the Profiler, choose View > Tool Windows > Profiler from the Android Studio menu bar:

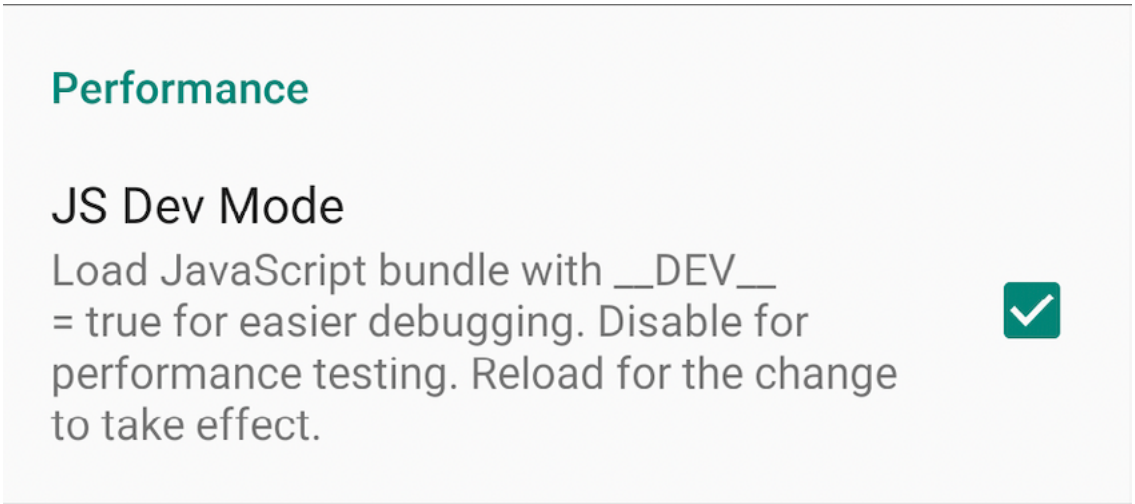


Or click Profile in the toolbar.

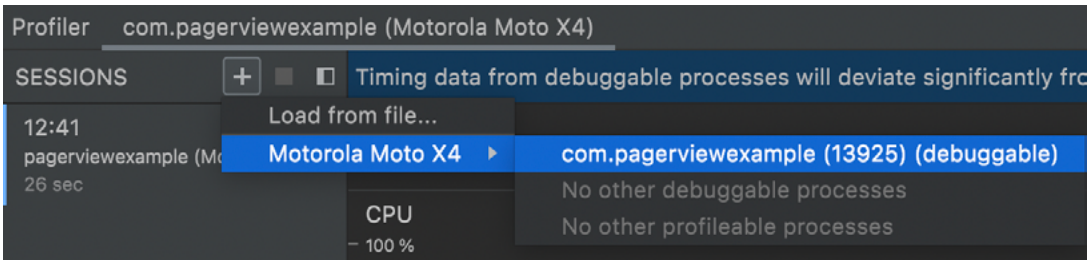


Before you start profiling the app, please remember:

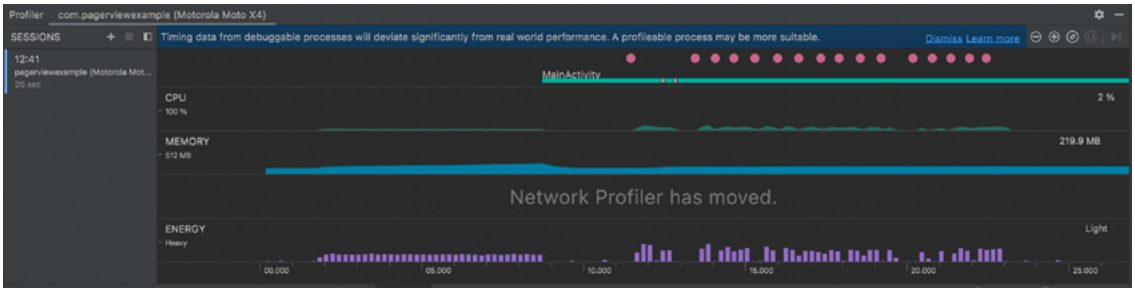
- Run the app on a real Android device that is affected, preferably a lower-end phone or emulator if you don't have one. If your app has runtime monitoring set up, use a model that is either the most used by users or the one that's affected by a particular issue.
- Turn off development mode. You must be sure that the app uses a JS bundle instead of the metro server, which provides that bundle. To turn it off, please share your device, click on Settings and find JS Dev Mode:



After that, go to the Profiler tab and add a new profiler session:



Wait for the session to attach to your app and start performing actions that could cause some performance issues, like swiping, scrolling, navigating, etc. Once you're done, you should see some metrics like these:



Each greenfield React Native app has only one Android Activity. If your app has more than one, it's most likely a brownfield one. Read more about the brownfield approach here. In the above example, we don't see anything interesting. Everything works fine without any glitches. Let's check each metric:

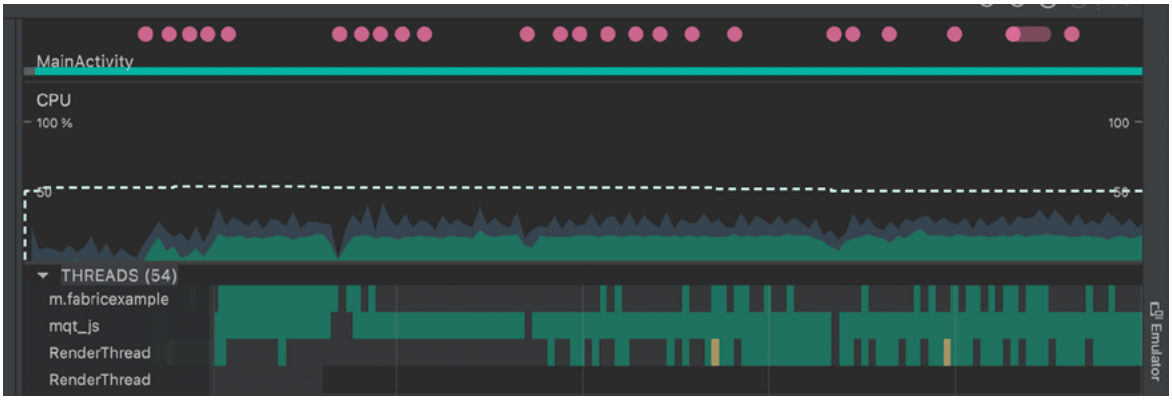
- The CPU metric is strictly correlated to energy consumption because the CPU needs more energy to do some computations.
- The memory metric is not changing while using the app, which is expected. Memory usage can grow, e.g. when opening new screens, and drop when the garbage collector (GC) releases free memory, e.g. when navigating out of a screen. When memory increases unexpectedly and keeps on growing, it may indicate a memory leak, which we want to avoid, as it can crash the app with out of memory (OOM) errors.
- The network section has been moved to a separate tool called the Network Tab. In most cases, this metric is not needed, because it is mostly related to the backend infrastructure. If you would like to profile a network connection, you can find more information here.
- The energy section gives hints on when our app's energy usage is low, medium, or high, impacting the daily experience of using the app.

Use Android Profiler in action

In the previous example, we could see some relations between each metric:

To see a more detailed view, we have to double-click on the tab. Now we can see more details. When the user started to do some touch action (swiping in the above example), we could see more CPU work. Each app will have its own signature of CPU spikes and lows. It's important to build an intuition about it, by interacting with it and pairing certain activities, like touch events, with the increased usage. In other words, some spikes are expected, because the work needs to be done. The problem starts when CPU usage is very high for extended periods of time or in unexpected places.

Let's imagine you would like to pick the best list or scroll view component for your React Native app, which has the best performance on a lower-end device. You noticed the current solutions could be revamped or improved and you started working on this. In your experiment, you would like to check how your solution works for LE devices using the above-described solution. When you double-clicked on CPU, you could spot the below data:



Here you can see the `mqt_js` thread is used almost all the time and does some heavy computation because your computations are done on the JS side. You can start thinking about how to improve it. There are multiple options to check:

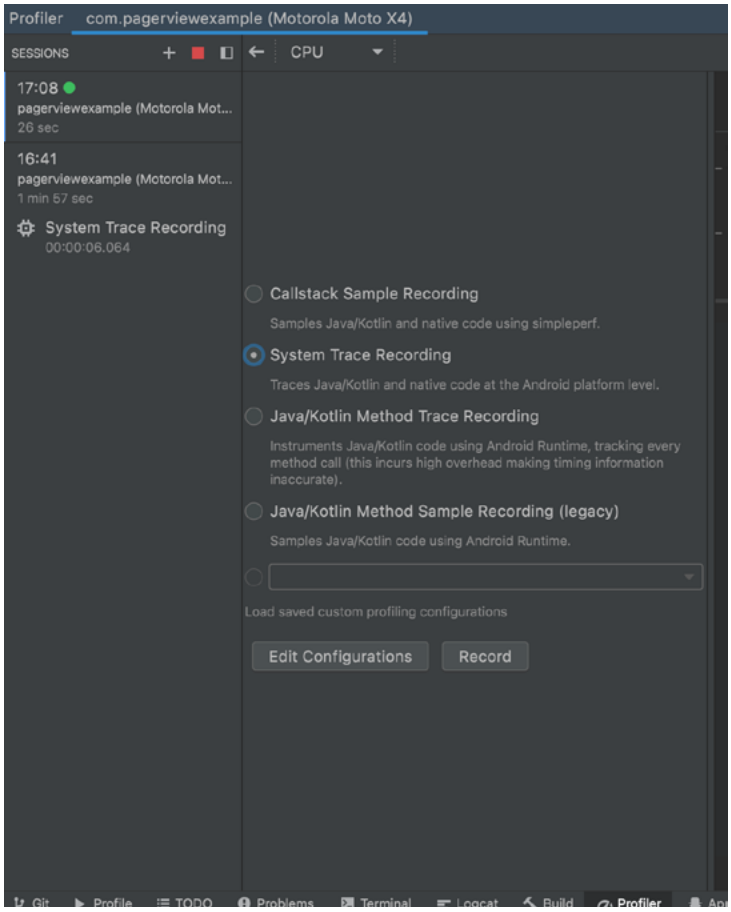
- Replace the bridge with JSI in terms of communication – do tests if JSI is faster than the bridge.
- Move some part of the code to the native side – on the native side you have more control over threads execution and can schedule some work to not block the JS or UI thread.
- Use a different native component – replace the native scroll view with your custom solution.
- Use shadow nodes – do some expensive calculation with C++ and pass it to the native side.

You can try out all of those solutions and compare the effect between each other. The profiler will provide you with a metric and based on that you can make a decision about which approach fits best to your particular problem.

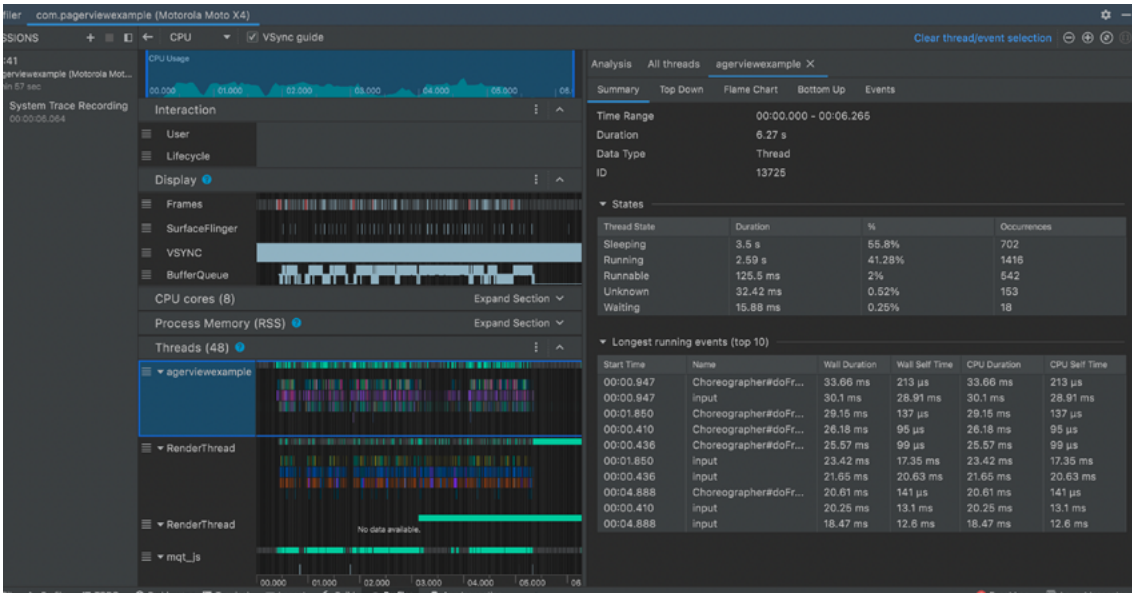
There's more info about the Android Profiler here.

System Tracing

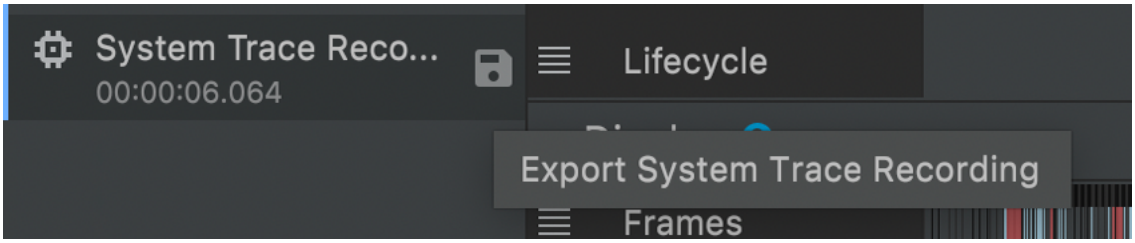
Using the Android Studio CPU Profiler, we can also make a system tracing. We can check when the appropriate function has been called. We can triage all threads and see which function is the costliest which affects the UX. To enable system tracing, click on the CPU section and select System Trace Recording



After some interaction, you should be able to see all the threads with details:



You can also save your data by clicking the Save Button:



And use the data in a different tool, e.g. Perfetto:

You'll also want to check the official Android Profiling guide by the React Native core team. They use different tools, but the outcome will be the same. The guide provides case studies and how to spot an issue on different threads:

- UI thread
- JS thread
- Native module thread
- Render Thread (only Android)

You can find more about threading models in the New Architecture chapter.

Flipper performance plugin for Android

We already know Flipper can be quite handy in hunting performance issues. One of the most interesting plugins to help us out on Android is android-performance-profiler. It can be used as a standalone tool or on a CI. It can generate beautiful reports, so this tool can be used to make some sophisticated experiments.

Here is a picture of an example experiment:

	Results FABRIC List scrolling	Results FABRIC Showing 100 Tweets	Results NO_FABRIC List scrolling	Results NO_FABRIC Showing 100 Tweets
Score	67	54	83	60
Average Test Runtime	11187ms	6307ms	12150ms	5535ms
Average FPS	47.9	36.5	48.2	40.7
Average CPU usage	99.7%	106.7%	95.5%	111.2%
Average RAM usage	239.9MB	237.3MB	229.1MB	226.6MB
Processes with high CPU usage detected	Total: 3.8s fabric_bg for 3.7s UI Thread for 0.1s	Total: 2.5s mqt_js for 1.1s FrescoDecodeExe for 0.95s UI Thread for 0.4s fabric_bg for 0.05s	None	Total: 2s mqt_js for 1s FrescoDecodeExe for 0.65s mqt_native_modu for 0.25s FrescoloBoundEx for 0.1s
Framework Detection	React Native	React Native	React Native	React Native

Comparison of the new vs old architecture of React Native by Almouro. Source.

You can also automate your experiments with e2e tests and generate reports locally or on a CI. Those reports can be used to compare solutions with each other.

Benefits: Real-time metrics will improve your app understanding

As stated above, users will abandon your app if the response time is too long. Using specific tools will help you understand the root cause of the app's performance issue.

Thank you

We hope that you found the aforementioned best practices for React Native optimization useful and that they will make your work easier. We did our best to make this guide comprehensive and describe both the technical and business aspects of the optimization process.

If you enjoyed it, don't hesitate to share it with your friends who also use React Native in their projects.

If you have more questions or need help with cross-platform or React Native development, we will be happy to provide a free consultation.

Just contact us!

Authors

Michał Pierzchała

As Head of Technology at Callstack, he is passionate about building mobile and web experiences, high-quality JS tooling, and Open Source. Core Jest and React Native community contributor. Space exploration enthusiast.
twitter.com/thymikee
github.com/thymikee

Jakub Bujko

With multiple years of delving deep into react.js development in his pocket, Kuba went on to master mobile development. Passionate about edge technologies, clean and minimalistic code, and charting the paths for the future of React and React Native development.
twitter.com/f3ng__liu
github.com/Xiltyn

Maciej Jastrzębski

React & React Native developer with multiple years of experience building native iOS and Android apps. Passionate about building robust and delightful apps along with writing well-architected and readable code. Loves learning new things. He likes to travel in his free time, hike in the mountains, and take photographs.
twitter.com/mdj_dev
github.com/mdjastrzebski

Piotr Trocki

Software developer who started his journey from mobile apps. Now Piotr is focused on mastering both Native (Android, iOS) and React Native technologies in brownfield applications. When not coding, he

spends his free time on the dance floor.
twitter.com/Tr0zZe
github.com/troZee

Jakub Binda

A dedicated software developer who pays a lot of attention to the details in every task he does. Always committed and eager to learn, Kuba likes to create things and dive into how they work. A father of two and a husband to the woman of his life. Those two roles motivate him the most and give him the strength to move mountains.
github.com/jbinda

Szymon Rybczak

Szymon is a 16-year-old React Native Developer with two years of experience and currently doing mobile app development at Callstack. In his free time, he likes to discover new and interesting technologies.
github.com/szymonrybczak
twitter.com/SzymonRybczak

Hur Ali

TypeScript enthusiast mastering the React-Native and Native realm. He feels best in diving deep with mobile tech, making proof-of-concept projects, and experimenting with new technologies. In his free time, he enjoys playing FIFA and contribution to OSS.
twitter.com/hurali97
github.com/hurali97

Oskar Kwaśniewski

React Native Developer at Callstack. Currently, he's strengthening his knowledge of native development and making some OSS contribu-

tions. During his free time, he enjoys riding a bike, going to the gym, and playing video games.

github.com/okwasniewski
twitter.com/o_kwasniewski

Tomasz Misiukiewicz

React Native Developer at Callstack with a strong background in web development. Big fan of keeping the code clean and simple. Loves to learn new stuff and enhance his programming skillset every day.
github.com/TMisiukiewicz

Eduardo Graciano

Senior mobile developer at Callstack. Hacking almost all kinds of mobile tech and always looking forward to making readable and maintainable code without messing up everything.
github.com/gedu
twitter.com/teddydroid07

Andrew Andilevko

React Native developer with a background in Android development. He likes complex tasks to constantly expand his expertise and knowledge. He spends his free time with his wife and pug.
github.com/andrewworld

James Ide

I work on Expo, which I co-founded with Charlie Cheever when we wanted to make it easier to make and use universal mobile apps that run everywhere..
https://github.com/ide
https://twitter.com/JI

About Callstack

We are a team of React and React Native experts and consultants focused on delivering cutting-edge technology. We love to share our knowledge and expertise with others.

That's why we contribute to the React Native community by creating and maintaining many open-source libraries and organizing the biggest conference focused on React Native - the React Native EU conference. We're official Meta and Vercel partners.