

The Ultimate Guide to React Native Optimization



Improve user experience, performance,
and stability of your apps.

Created by **{callstack}**

2020

Table of Contents

Organizational part

Introduction to React Native Optimization

First Group

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

Second group

1. Always run the latest React Native version to access the new features
2. How to debug faster and better with Flipper
3. Automate your dependency management with `autolinking`
4. Optimize your Android application startup time with Hermes
5. Optimize your Android application's size with these Gradle settings

Third Group

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) when in an emergency

Thank you

Authors

Callstack

Organizational part

Optimizing React Native apps with a limited development budget can be difficult but is not impossible. In such a case, you need to focus on the essentials of your app and squeeze as much as possible out of them to maintain your business continuity.

That's why we prepared this guide.

In the following chapters, we will show you how to optimize the performance and stability of your apps. Thanks to the practices described in the guide, you will improve the user experience and speed up the time-to-market of your apps.

The guide contains best practices for optimizing the following aspects:

- 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 impact on better user experience. With faster time-to-market, you can stay ahead of your competitors, whereas an easier and faster maintenance process will help you to reduce your spendings on that particular process.

What the guide will look like and what topics it will cover.

The guide is divided into three groups:

The first group is about improving performance by understanding React Native implementation details and knowing how to make maximum out of it. Here are the topics we will discuss:

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

The second group is focused on 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 new features
2. How to debug faster and better with Flipper
3. Automate your dependency management with `autolinking`
4. Optimize your Android application startup time with Hermes
5. Optimize your Android application's size with these Gradle settings

The third group says about improving the stability of the application by investing in testing and continuous deployment. This part says about:

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) when in an emergency

The structure of each article is simple:

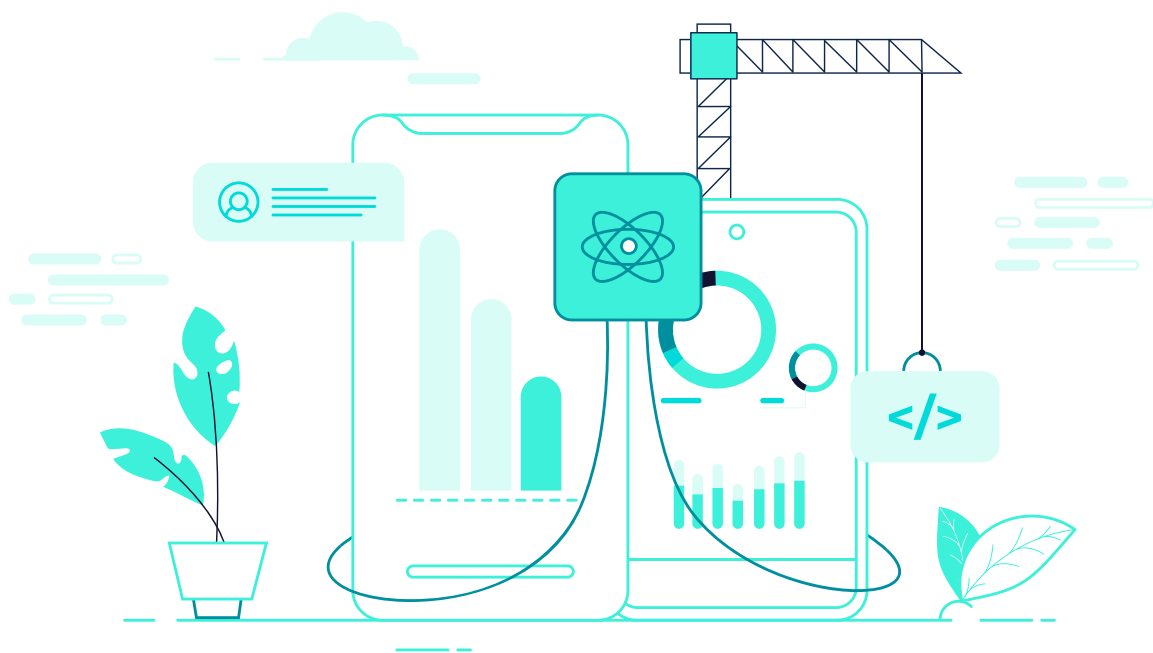
Issue: The first part describes the main problem and what you may be doing wrong.

Solution: The second part says about how that problem may affect your business and what are the best practices to solve it.

Benefits: The third part is focused on the business benefits of our proposed solution.

OK, the informational and organizational part is already covered. Now, let's move on to the best practices for optimizing the performance of your app.

Let's go!



Introduction to React Native Optimization

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

With React Native, you create components that describe how your interface should look like. During 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 applications done with React Native are equally fast and offer same level of user experience.

Every declarative approach (incl. 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 analyse every callsite to the external APIs. For example, when working in a multi-threaded environment, you write your code in a thread safe way, 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 underlying iOS and Android APIs calls, how you compose the components together can make all the difference. All your components will offer the same level of performance and responsiveness.

But is *the same* a synonym of *the best*? It's not.

That's when our checklist come into play. Use React Native to its potential.

As discussed before, React Native is a declarative framework and takes care of rendering the application for you. In other words, it's not you that 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 it interacts with the underlying platform APIs.

If you need help with performance, stability, user experience or other complex issues - contact us! As the React Native Core Contributors and leaders of the community, we will be happy to help.



First Group

Improve performance by understanding React Native implementation details.

Introduction

In this group, we will dive deeper into most popular performance bottlenecks and 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 also will let you significantly improve the stability and performance of your application by performing the small tweaks and changes.

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

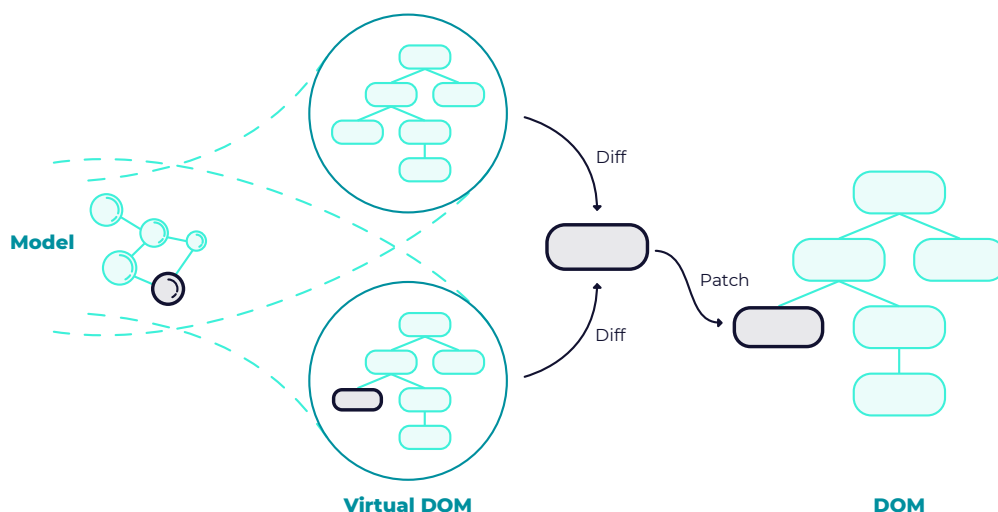
1. Pay attention to UI re-renders

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

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. Your job is 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, performs only the required and smallest number of actual updates.



The rule here is simple - 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 (previous and current props) would take longer.

Negative impact on the performance, UI flicker, and FPS decrease

While the above heuristics is 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 being performed) or frames dropping (while there's an animation happening and an update is coming along).

Note: You should never perform any premature optimisations. Doing so may have a counter-positive effect. Try looking into this 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 out for extraneous operations that you would not expect to happen.

Solution: Optimize the number of state operations and remember to use pure and 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. In this section, 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, { Component } from 'react';
import { TextInput } from 'react-native';

export default function UselessTextInput() {
  const [value, onChangeText] = React.useState('Text');

  return (
    <TextInput
      style={{ height: 40, borderColor: 'gray', borderWidth: 1 }}
      onChangeText={text => onChangeText(text)}
      value={value}
    />
  );
}
```

Read more: <https://snack.expo.io/q75wcVYnE>

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

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

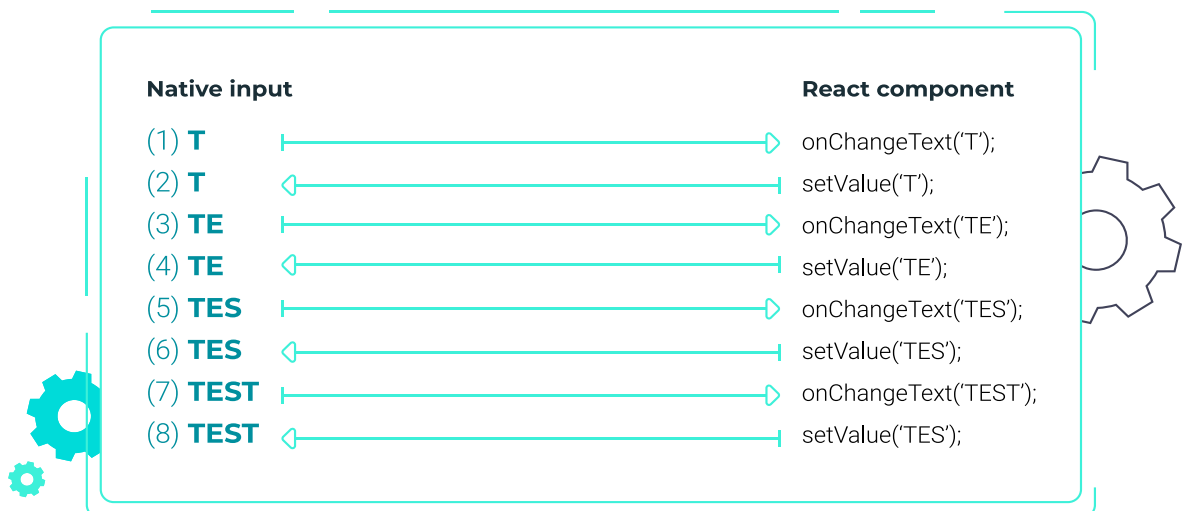


Diagram that shows what happens while typing TEST

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

The benefit of such approach is simple. 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 validation, masking it or completely modifying.

Unfortunately, the above approach, while being ultimately cleaner and more compliant with the way React works, has one downside. It is most noticeable when there is limited resources available and / or user is typing at a very high rate.

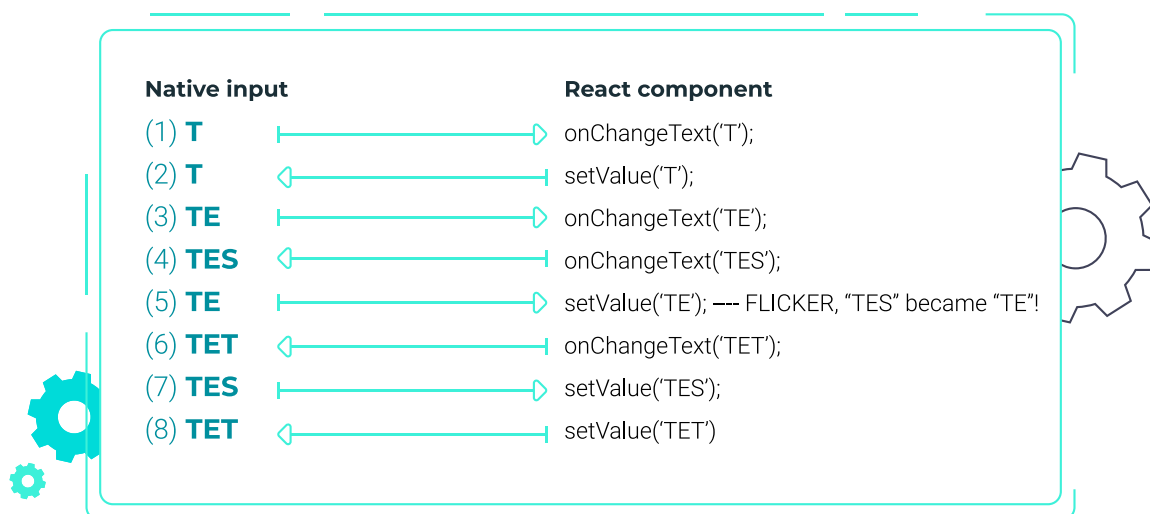


Diagram that shows what happens while typing TEST too fast

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

Next, operation 3 arrives, followed by another update (operation 4). The user typed *E* & *S* while React Native was busy doing something else, delaying the synchronisation of the *E* letter (operation 5). As a result, the native input will change its value temporarily 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 value of *TET*. This wasn't intentional - 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 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* - final value of the input.

The root cause of this situation lies in the order of operations. If the 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 *value* prop from *TextInput* entirely. As a result, the data will flow only one way, from native to the JavaScript side, eliminating the synchronization issues that were described earlier.

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

export default function PizzaTranslator() {
  const [text, setText] = useState('');
  return (
    <View style={{padding: 10}}>
      <TextInput
        style={{height: 40}}
        placeholder="Type here to translate!"
        onChangeText={text => setText(text)}
        defaultValue={text}
      />

      <Text style={{padding: 10, fontSize: 42}}>
```

```
    {text.split(' ').map((word) => word && '🍕').join(' ')}  
  </Text>  
</View>  
);  
}
```

Read more: <https://snack.expo.io/DYMECpVPQ>

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 input validation or masking, you still need to control the data that user is typing and alter what ends up being displayed within the `TextInput`. React Native team is well aware of this limitation and is currently working on the new re-architecture that is going to resolve this problem as well.

Global state

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

We recommend using specialized libraries like *Redux* or *Overmind.js* to handle your state management in more optimized way.

First, your state management library should take care of updating component only when defined subset of data had changed - this is the default behavior of `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 derivation until the set of passed dependencies will change.

```
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)
    }
  }
)

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

const VisibleTodoList = connect(
  mapStateToProps,
)(TodoList)

export default VisibleTodoList
```

A typical example of selectors with redux state management library

Common bad performance practice is a belief that state management library can be replaced with usage of custom implementation that is based on *React Context*. It may be handy at the beginning because it reduces boilerplate code that state management libraries introduce. But using this mechanism 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 is easier than implementation of custom selectors mechanism to your current solution.

You can also optimize your application on single component level. Simply using *Pure Component* instead of regular *Component* and using memo wrapper for function components will save you a lot of re-renders. It may not have an impact at the first glance, but you will see the difference when non-memoized components are used in list that shows big set of data. It is usually enough as for components optimizations.

Do not try to implement these techniques in advance, because such an optimization is used rarely and in very specific cases.

Benefits: Less resources needed, faster application

You should always keep the performance of your app in the back of your head, but do not try to optimize everything in advance, because it is usually not needed. You will end up wasting time on solving nonexistent problems.

Most of hard-to-solve performance issues are caused by bad architectural decisions around state management, so make sure it is well designed. Particular components should not introduce issues as long as you use *Pure Component* or *memo* wrapper.

After all, 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 overall, more satisfaction from interacting with the interface.
o lower battery usage and overall, more satisfaction from interacting with the interface.

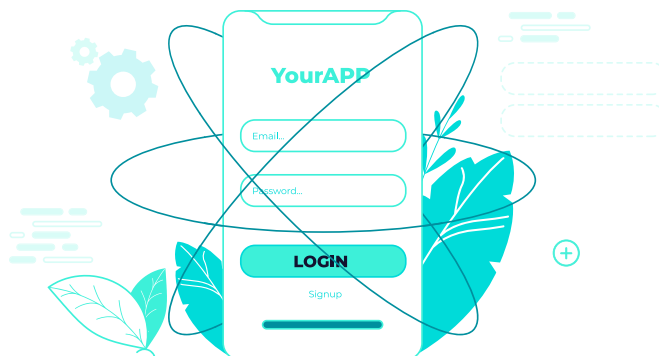
2. Use dedicated components for certain layouts

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

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

In 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 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 the final stage of its development.

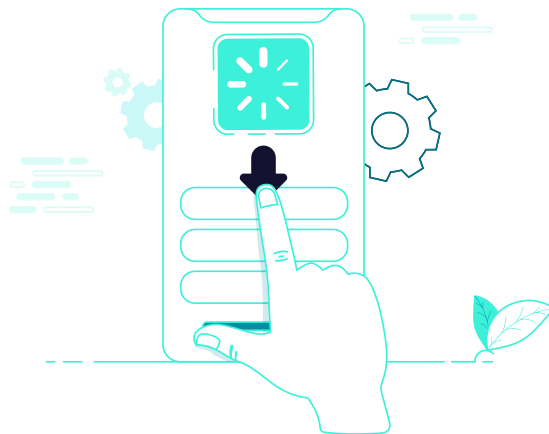


On top of *primitive components*, React Native ships with a set of *higher-order components* that are designed and optimized to serve a certain purpose.

Being unaware of them or not using them in all the places can potentially affect your application performance, especially as you populate your state with real production data. 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 from performance improvements and risking 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 broader API to cover the vast majority of mobile scenarios.

Solution: Always use 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 get into trouble when the data grows. Dealing with the 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 performance of adding new list element based on *ScrollView*,

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

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

const getRandomItem = () => {
  const item = objects[Math.floor(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());

export default function List() {
  const [items, setItems] = useState(_items);

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

```

    }, [items]));

    return (
      <View style={{marginTop: 20}}>
        <Button title="add item" onPress={addItem} />
        <ScrollView>
          {items.map(({name, icon}) => (
            <View
              style={{
                borderWidth: 1,
                margin: 3,
                padding: 5,
                flexDirection: 'row',
              }}>
              <Text style={{fontSize: 20, width: 150}}>{name}</Text>
              <Text style={{fontSize: 20}}>{icon}</Text>
            </View>
          ))}
        </ScrollView>
      </View>
    );
  }
}

```

Read more: <https://snack.expo.io/qjtEVHrdV>

to list based on *FlatList*.

```

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

const objects = [
  ['avocado', '🥑'],
  ['apple', '🍏'],
  ['orange', '🍊'],
  ['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());

  export default function List() {
    const [items, setItems] = useState(_items);

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

    return (
      <View style={{ marginTop: 20 }}>
        <Button title="add item" onPress={addItem} />
        <FlatList
          data={items}
          keyExtractor={({ id }) => id}
          renderItem={({ item: { name, icon } }) => (
            <View
              style={{
                borderWidth: 1,
                margin: 3,
                padding: 5,

```

```

        flexDirection: 'row',
      }}>
      <Text style={{ fontSize: 20, width: 150 }}>{item[0]}</Text>
      <Text style={{ fontSize: 20 }}>{item[1]}</Text>
    </View>
  )}
/>
</View>
);
}

```

Read more: <https://snack.expo.io/1muB1wKya>

The difference is significant, isn't it? In provided example of 5000 list items, *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 60 FPS.

Just using *FlatList* may not be enough in some cases. *FlatList* performance optimizations relay 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 scroll area should be reserved for upcoming elements.

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



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

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 automatically opt-in to all the performance optimisations done by the React Native so far and subscribe for further updates to come.

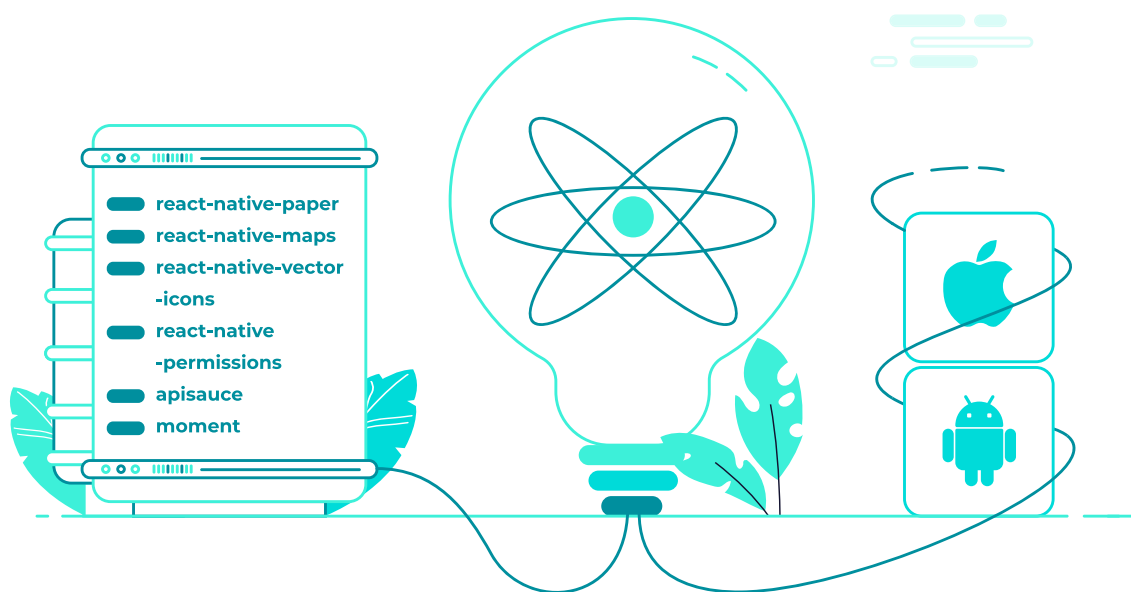
At the same time, you also save yourself a lot of time reimplementing most common UI patterns from the ground up. Sticky section headers, pull to refresh - you name it. These are already supported by default, if you choose to go with `FlatList`.

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

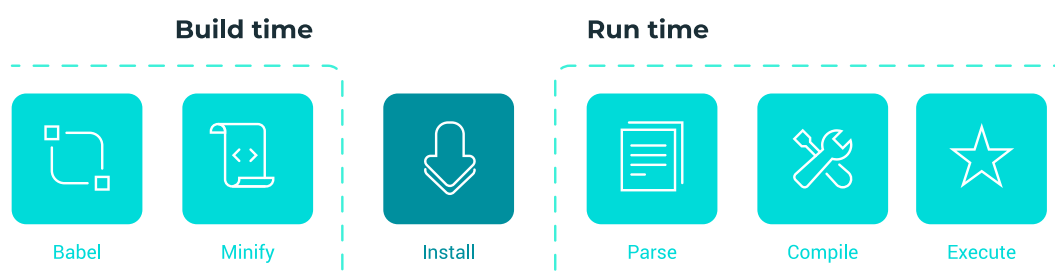
The key difference lies in the performance of the mobile devices and the tooling used for bundling and compiling the application.

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 an important factor.

Interpretation with convetional engine



Read more: <https://snack.expo.io/7H5S504j3>

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) milliseconds between when the icon gets selected from the application drawer and when it becomes fully interactive.

Unfortunately, Metro – the default React Native bundler – [doesn't support tree shaking as of now](#). If you're not familiar with this notion, [read this article](#).

It means that all the code that you pull from `npm` and import to your project will be present in your production bundle, loaded into the memory and parsed.

That can have a negative impact on the total startup time of your application.

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 the dates. Let's imagine you are about to calculate the 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 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 date with day.js

If there are no alternatives, the 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.

Benefits: your app loads faster which can make the difference

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

According to [Akamai's report](#) on the online retail performance, just one-second delay in mobile load times can **cut the conversion rates by up to 20%**.

That's why 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.

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.

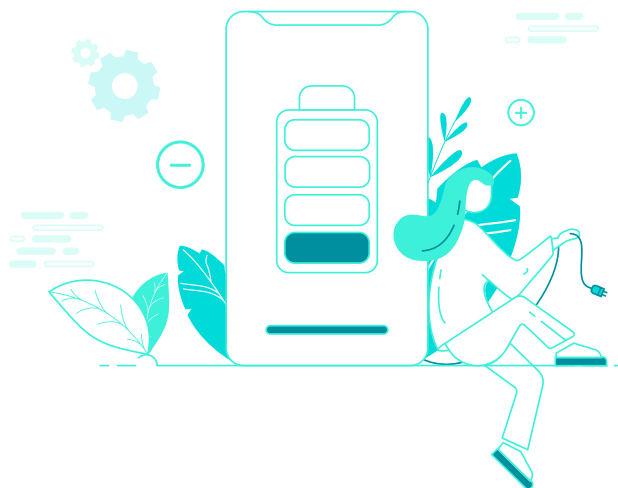
Issue: You use web libraries that are not optimized for mobile

As discussed earlier, one of the best things about React Native is that you can write the mobile application with JavaScript, reusing some of your React components and doing the 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 to the network or have large batteries.

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



In other words – how do 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 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 an ability to render advanced graphics (3D structures, diagrams), it is very likely that you're better going with the dedicated mobile library.

The reason is simple – these libraries were developed within the web environment in the first place, assuming 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 OSes, 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:

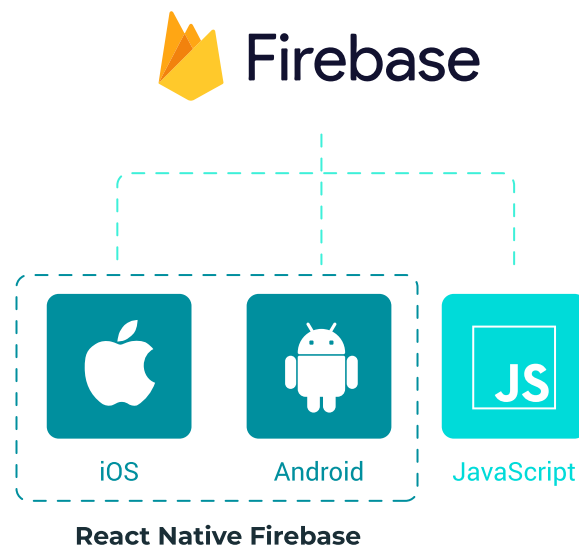
```
import database from 'firebase/database';

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

An example reading from Firebase Realtime Database in RN

As you can see, the difference is minimal and boils down to a different import statement. 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 simple things and maximum reusability, you can choose to go with the web version of the library. That will give you access to the same features as in the browser at 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 escape hatch is what makes React Native extremely versatile and enterprise-ready.

It enables you to build features faster on many platforms at once, without compromising on the performance and user experience – something other hybrid frameworks made a standard tradeoff.

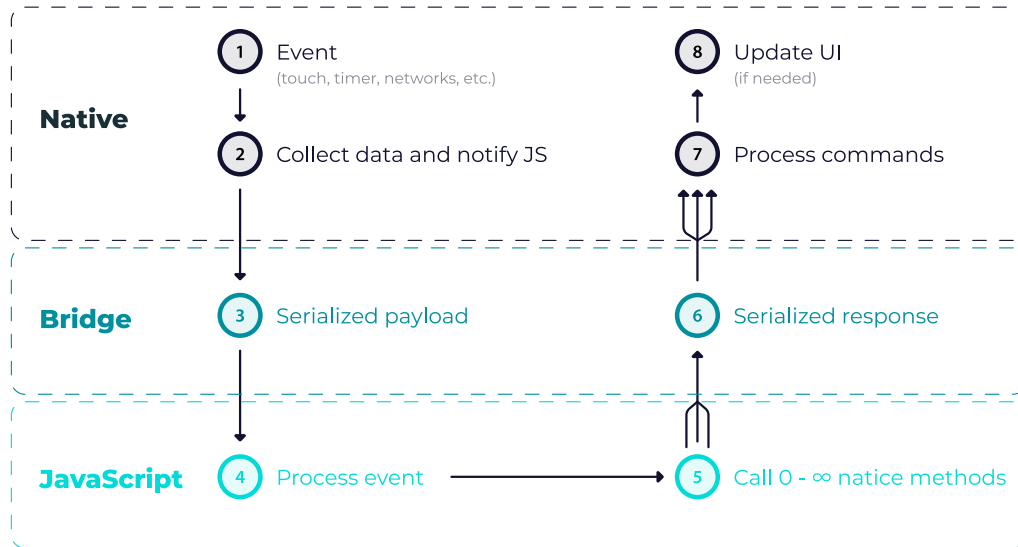
5. Find the balance between native and JavaScript

Find the harmony between native and JavaScript to build fast-working and easy-to-maintain apps.

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

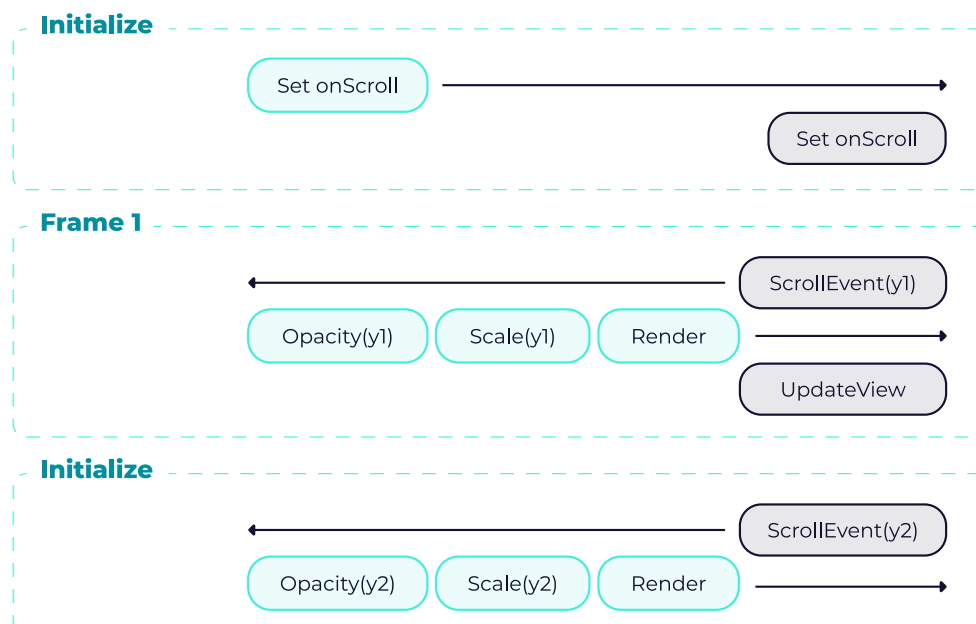
When working with React Native, you're going to be developing JavaScript most of the time. However, there are situations when you need to write a bit of native code. For example, you're working with a 3rd party SDK that doesn't have an official React Native support yet. In that case, you need to create 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 bridge, which provides bidirectional communication between JavaScript and native worlds. As a result, JavaScript can execute native APIs and pass the necessary context to receive the desired return value. That 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 be up for another task already.



The number of JavaScript calls that arrive over to 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 your 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 certain 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 another kind of 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 it should be.

Solution: Use the right amount of abstraction on the JS side – validate and check 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 simple JavaScript module that does nothing more but 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 native module

In 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 in the next tick when the native side processes the call and receives an exception from the native module.

In such scenario, you have lost a bit of 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 native module with arguments validation

The above is not only tied to the native modules itself. 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 – just like you execute your native method with the JavaScript arguments.

To put this into better perspective, let's take a closer look at styling within React Native apps.

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

export default class App extends React.Component {
  render() {
    return (
      <View style={{flex: 1, justifyContent: 'center', alignItems: 'center'}}>
        <View style={{
          backgroundColor: 'coral',
          width: 200,
          height: 200
        }}/>
      </View>
    );
  }
}
```

Read more: <https://snack.expo.io/7H5S504j3>

The easiest way to style a component is to pass it an object with styles. While it works, you will not see it happening too much. It is generally considered an anti-pattern, unless you're dealing with dynamic values, such as changing the style of the component based on the state.

```
import * as React from 'react';
import { View, StyleSheet } from 'react-native';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>
        <View style={styles.box} />
      </View>
    );
  }
}
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center'
  },
  box: {
    backgroundColor: 'coral',
    width: 200,
    height: 200,
  },
});
```

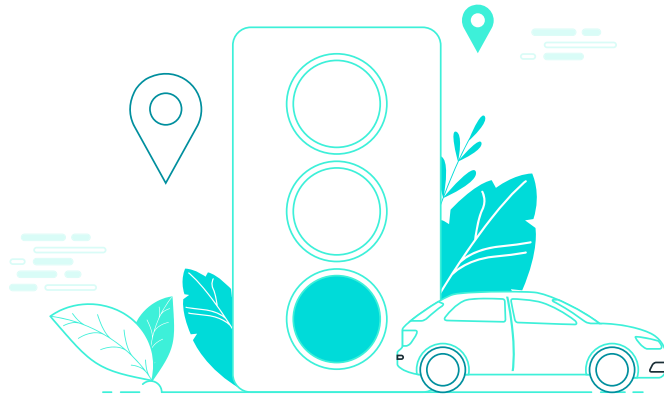
Read more: <https://snack.expo.io/GUFPWl8BD>

React Native uses StyleSheet API to pass styles over the bridge most of the time. That API processes your styles and makes sure they're passed only once over the bridge. During runtime, it substitutes the value of style prop with a numeric unique identifier that corresponds to the cached style on the native side.

As a result, rather than sending a large array of objects every time React Native is to re-render its UI, the bridge has to now deal with an array of numbers, which is much easier to process and transfer.

Benefits: The codebase is faster and easier to maintain

Whether you're facing any performance challenges right now, it is a good practice 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 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 straight copy-paste from the documentation – very simple and easy to understand.

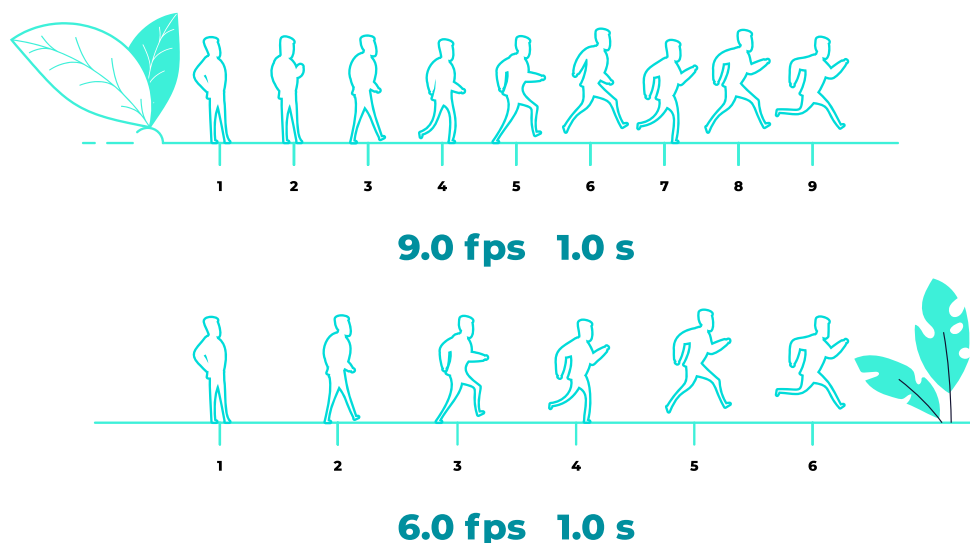
Mastering this approach to the development of native modules is why a lot of JavaScript developers can easily extend their applications with additional functionality without getting specialized in Objective-C or Java.

6. Animate at 60FPS no matter what

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

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

Mobile users are used to smooth and well-designed interfaces that quickly respond to their interactions and provide prompt visual feedback. It is one of these things that is not really the case when it comes to web development but makes the difference on mobile. 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 through the bridge is limited. There's no built-in priority queue as of now. 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 perform 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 much about ensuring they're running at 60 FPS.

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 the best way 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 world of a sea of applications, providing smooth and interactive UI might be one of your only ways to win over the customers that 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 it may affect the 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 ensuring the animations are always running smoothly is a fundamental part that builds such experience.

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

One-off animations

Enabling usage of native driver is the easiest way of quickly improving your animations performance. However the subset of style props that can be used together with 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 native driver for opacity animation

For more complex use cases, you can use React Native Reanimated library. Its API is compatible with basic Animated library and introduce a set of low- level functions to control your animations. What's more important, it introduces the possibility to animate all possible style props with native driver. So animating height or color will no longer be an issue. However, transform and opacity animations still will be slightly faster since they are GPU accelerated. But regular users should not see any difference.

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 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 simple swipeable element by combining it with Animated. However it will still require JS callbacks, but there is a remedy for that!

The most powerful pair of tools for gesture-driven animations is the usage of Gesture Handler combined with Reanimated. Those were designed to work together and give the possibility to build complex gesture-driven animations that are fully calculated on the native side. The only limitation here is your imagination (and coding skills, since Reanimated low-level API is not so straightforward).

```
import React, { Component } from 'react';
import { StyleSheet, View } from 'react-native';
import { PanGestureHandler, State } from 'react-native-gesture-handler';
import Animated from 'react-native-reanimated';
import runSpring from './runSpring';

const {
  set,
  cond,
  eq,
  add,
  multiply,
  lessThan,
  spring,
  startClock,
  stopClock,
  clockRunning,
  sub,
  defined,
  Value,
  Clock,
  event,
  SpringUtils,
} = Animated;
```

```

class Snappable extends Component {
  constructor(props) {
    super(props);

    const TOSS_SEC = 0.2;

    const dragX = new Value(0);
    const state = new Value(-1);
    const dragVX = new Value(0);

    this._onGestureEvent = event([
      { nativeEvent: { translationX: dragX, velocityX: dragVX, state: state } },
    ]);

    const transX = new Value();
    const prevDragX = new Value(0);

    const clock = new Clock();

    const snapPoint = cond(
      lessThan(add(transX, multiply(TOSS_SEC, dragVX)), 0),
      -100,
      100
    );

    this._transX = cond(
      eq(state, State.ACTIVE),
      [
        stopClock(clock),
        set(transX, add(transX, sub(dragX, prevDragX))),
        set(prevDragX, dragX),
        transX,
      ],
      [
        set(prevDragX, 0),

```

```

        set(
          transX,
          cond(defined(transX), runSpring(clock, transX, dragVX, snapPoint), 0)
        ),
      ]
    );
  }

  render() {
    const { children, ...rest } = this.props;
    return (
      <PanGestureHandler
        {...rest}
        maxPointers={1}
        minDist={10}
        onGestureEvent={this._onGestureEvent}
        onHandlerStateChange={this._onGestureEvent}>
        <Animated.View style={{ transform: [{ translateX: this._transX }] }}>
          {children}
        </Animated.View>
      </PanGestureHandler>
    );
  }
}

export default class Example extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Snappable>
          <View style={styles.box} />
        </Snappable>
      </View>
    );
  }
}

```

```

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.io/EM0KZfwJd>

Low-level handling of gestures might not be a piece of cake, but fortunately, there are already 3rd party libraries that utilize mentioned tools and expose *CallbackNodes*. *CallbackNode* is nothing more than an *Animated.Value*, but it is 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 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 { block, set, greaterThan, lessThan, Value, cond, sub, interpolate } =

```



```

Animated;

export default class Example extends React.Component {
  gestureCallbackNode = new Value(0);

  contentPos = this.gestureCallbackNode;

  renderHeader = name => (
    <View
      style={{
        width: '100%',
        backgroundColor: 'lightgrey',
        height: 40,
        borderWidth: 2,
      }}>
      <Text style={{textAlign: 'center', fontSize: 20, padding: 5}}>Drag me</Text>
    </View>
  );

  renderInner = () => (
    <View style={{backgroundColor: 'lightblue'}}>
      <Animated.View
        style={{
          opacity: interpolate(this.contentPos, {inputRange:[0,1],
outputRange:[1,0]}),
          transform: [{
            translateY : interpolate(this.contentPos, {inputRange:[0,1],
outputRange:[0,100]}),
          }]
        }}>

        <Lorem />
        <Lorem />
      </Animated.View>
    </View>
  );

```

```

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

const IMAGE_SIZE = 200;

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
});

```

Read more: <https://snack.expo.io/KFpkVKYB9>

Giving your JS operations 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*.

```

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',
};

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

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

```

```

    setExpensiveTaskState(ExpensiveTaskStates.done);
  });
};
return (
  <View style={styles.container}>
    {Platform.OS === 'web' ? (
      <Text style={{ textAlign: 'center' }}>
        !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: { ' ' }
          <Text style={{ fontWeight: 'bold' }}>{expensiveTaskState}</Text>
        </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,  
},  
});
```

Read more: <https://snack.expo.io/Wv8u!mKwJ>

This handy React Native module allows to execute any code after all running animations were finished. In practice, you can show a placeholder, wait for the animation to finish and then render actual UI. It would make your JavaScript animations run smoothly and avoid interruptions by other operations. Usually smooth enough to provide a great experience.

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

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.

If you need help with performance, stability, user experience or other complex issues - contact us! As the React Native Core Contributors and leaders of the community, we will be happy to help.



Second Group

Improve performance by using the latest React Native features.

Introduction

React Native is growing fast, so does the number of its features

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

All that proves that React Native is developing at a really healthy pace. Contributions made by both community and Facebook enable more and more advanced use cases of the framework. A great example of that is Hermes – a whole 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 – it improves your Android application's performance, start-up time, and reduces its overall size.

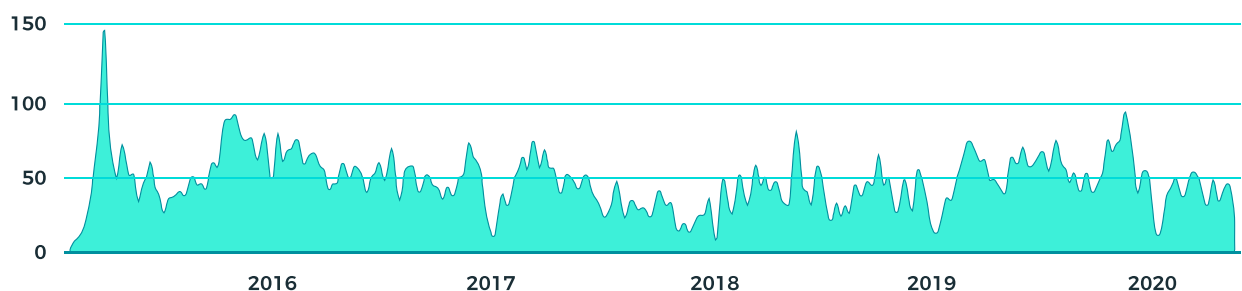
In this section, we will show you some of the features you can turn on right now to start off with your optimization process. We also encourage you keep track of all the new React Native features to make sure you use the framework to its full potential.

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

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

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

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



React Native is one of these libraries that is growing in a very rapid way.

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. Most popular changes among the community include features such as [Fast Refresh](#) or

Autolinking - both of them we describe in the following sections.

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

@react-native-community/cli	react-native
^5.0.0 (next)	master
^4.0.0 (master)	^0.62.0
^3.0.0	^0.61.0
^2.0.0	^0.60.0
^1.0.0	^0.59.0

That's why, keeping up with the newest React Native upgrades may seem like 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 the part of the react-native core.

It would be because of the changes introduced by Facebook during a process called *the lean core*. The goals of the effort were to:

- Make react-native package smaller, more flexible and easier to maintain by extracting some parts of the core and moving them to 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 effect on a react-native-upgrade. Now, you have to install the extracted packages as an additional dependency and until you do it, your app will not compile or crash at runtime. However, from a developer's perspective, migration to community packages is usually nothing more than introducing a new dependency and rewriting imports.

Another important issue is the third-parties support. Your code usually relies on external libraries and there's a risk that they also might 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 necessary adjustments before you upgrade,
- Look for the alternatives or patch the modules yourself – by using a handy utility called patch-package or creating a temporary fork with 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 the catch-up game, you opt out from 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 (developer experience) enhancements.

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

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

Upgrading React Native might be not the easiest thing in the world. But there are tools that make this process much simpler 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

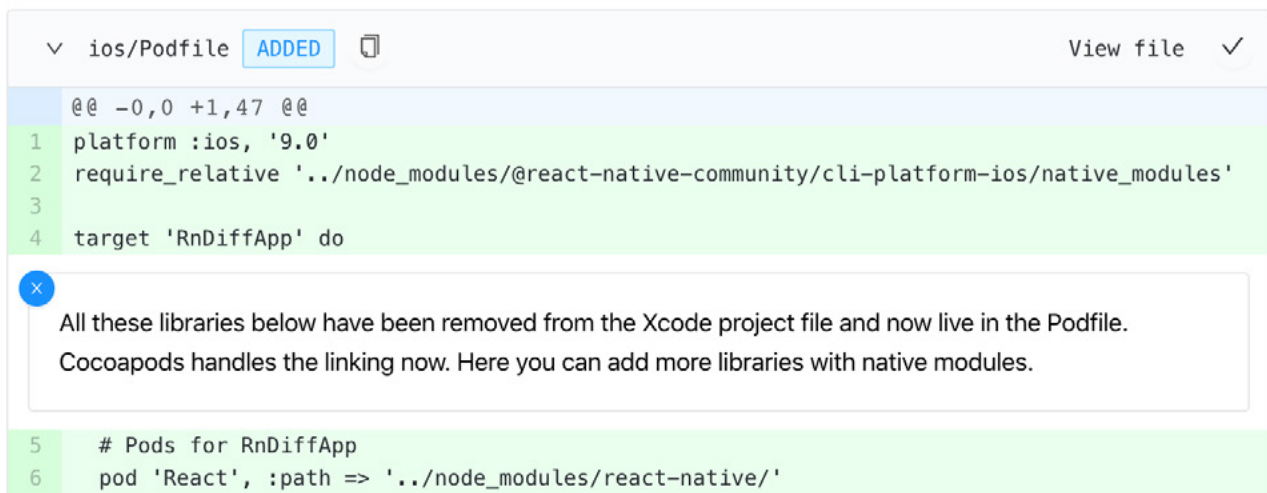
[React Native Upgrade Helper](#) is a good place to start. On a high level, it gives you an overview of the changes that happened to React Native since the last time you have upgraded your local version.

The screenshot shows the React Native Upgrade Helper web interface. At the top, it asks "What's your current React Native version?" with a dropdown set to "0.59.10" and "To which version would you like to upgrade?" with a dropdown set to "0.60.0". A blue button says "Show me how to upgrade!". Below this is a diff view of the `package.json` file, comparing the state before and after the upgrade. The left side shows the original state, and the right side shows the updated state. Changes are highlighted in green (additions) and red (deletions). The diff shows updates to the `scripts`, `dependencies`, and `devDependencies` sections. For example, the `start` script is updated from `node node_modules/react-native/local-cli/cli.js start` to `react-native start`. The `dependencies` section shows updates for `react` from `16.8.3` to `16.8.6` and `react-native` from `0.59.10` to `0.60.0`. The `devDependencies` section shows updates for `@babel/core` from `^7.4.5` to `^7.5.5`, `@babel/runtime` from `^7.4.5` to `^7.5.5`, `@react-native-community/eslint-config` from `^0.0.5` to `^0.5.0`, `eslint` from `^6.1.0` to `^6.1.0`, and `metro-react-native-babel-preset` from `^0.54.1` to `^0.55.0`.

```
@@ -3,20 +3,23 @@
3  "version": "0.0.1",
4  "private": true,
5  "scripts": {
6    "start": "node node_modules/react-native/local-cli/cli.js start",
7    "test": "jest"
8  },
9  "dependencies": {
10   "react": "16.8.3",
11   "react-native": "0.59.10"
12 },
13 "devDependencies": {
14   "@babel/core": "^7.4.5",
15   "@babel/runtime": "^7.4.5",
16   "babel-jest": "^24.8.0",
17   "jest": "^24.8.0",
18   "metro-react-native-babel-preset": "^0.54.1",
19   "react-test-renderer": "16.8.3"
20 },
21 "jest": {
22   "preset": "react-native"
23 },
24 "scripts": {
25   "start": "react-native start",
26   "test": "jest",
27   "lint": "eslint ."
28 },
29 "dependencies": {
30   "react": "16.8.6",
31   "react-native": "0.60.0"
32 },
33 "devDependencies": {
34   "@babel/core": "^7.5.5",
35   "@babel/runtime": "^7.5.5",
36   "@react-native-community/eslint-config": "^0.5.0",
37   "babel-jest": "^24.8.0",
38   "eslint": "^6.1.0",
39   "jest": "^24.8.0",
40   "metro-react-native-babel-preset": "^0.55.0",
41   "react-test-renderer": "16.8.6"
42 },
43 "jest": {
44   "preset": "react-native"
45 }
```

Differences in `package.json` between React Native 0.59 and React Native 0.60

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 a special information that will give more context on why something has happened.



```

  ios/Podfile ADDED
  @@ -0,0 +1,47 @@
  1 platform :ios, '9.0'
  2 require_relative '../node_modules/@react-native-community/cli-platform-ios/native_modules'
  3
  4 target 'RnDiffApp' do
  5   # Pods for RnDiffApp
  6   pod 'React', :path => '../node_modules/react-native/'

```

All these libraries below have been removed from the Xcode project file and now live in the Podfile. Cocoapods handles the linking now. Here you can add more libraries with native modules.

Additional explanation of 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. That in most cases includes a dedicated blog post published on React Native blog as well as the raw changelog.



Useful content for upgrading



React Native 0.60 includes Cocoapods integration by default, AndroidX support, auto-linking libraries, a brand new Start screen and more.

1. [Official blog post about the major changes on React Native 0.60](#)
2. [\[External\] Tutorial on upgrading to React Native 0.60](#)
3. [React Native 0.60 changelog](#)

Check out [Upgrade Support](#) if you are experiencing issues related to React Native during the upgrading process.

Keep in mind that `RnDiffApp` and `rndiffapp` are placeholders. When upgrading, you should replace them with your actual project's name. You can also provide your app name by clicking the settings icon on the top right.

Useful content to read while upgrading to React Native 0.60

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. 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 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, a bit 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 and 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 more stable API.

If the error occurs during the build time, simply bumping the dependency to its latest version should make it work.

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. This step is easy

to overlook and is often a result of runtime exceptions. Using [Flow](#) or [TypeScript](#) should guarantee 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 a 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.

Benefit: You're running 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 addresses the latest underlying changes to the iOS and Android. That includes the breaking changes that apply to mobile phones, such as when certain APIs get deprecated.

Here is an example: Last year, [Google has announced that all Android applications submitted to Google Play after August 1, 2019 have 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 necessary adjustments.

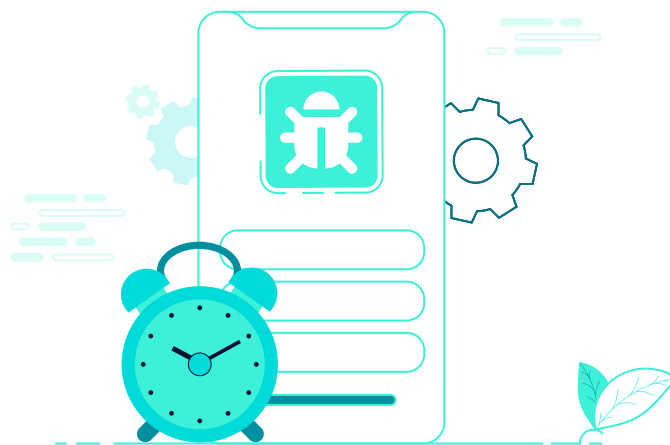
Upgrades like this are really critical to keeping your users satisfied. After all, they would be disappointed if the app started to crash with the newer version of the operating system or disappeared from the App Store. There might be some additional workload associated with every release, but staying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

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.

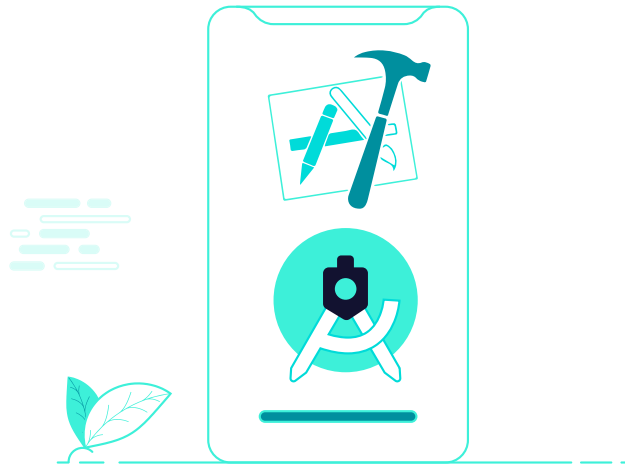
Issue: You're using Chrome Debugger or some other hacky way to debug and profile your React Native application

Debugging is one of the more challenging parts of every developer's daily work. It is relatively easy to introduce a new feature when everything seems to work, but finding 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:

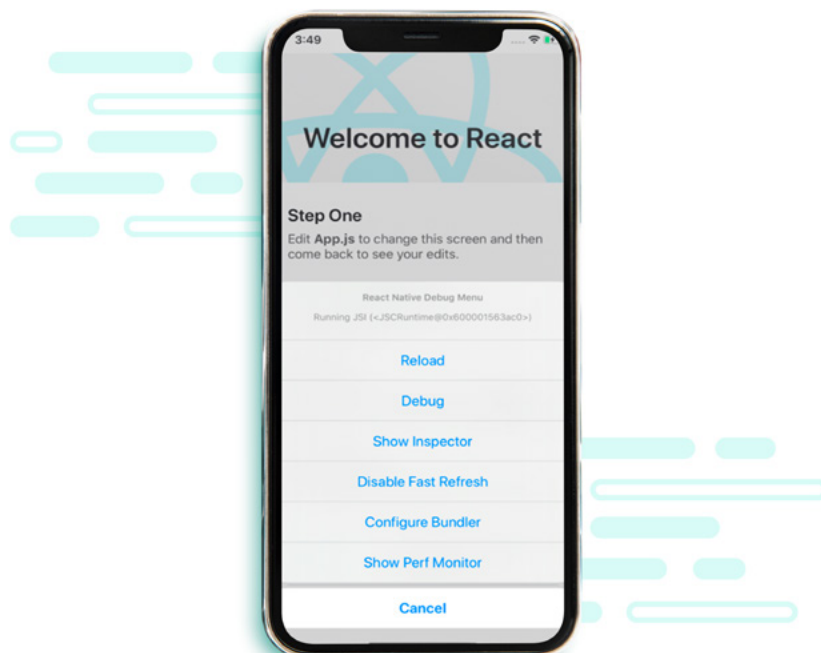
- JavaScript: your application's code or React Native, or
- Native code: third-party libraries or React Native itself.



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 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.

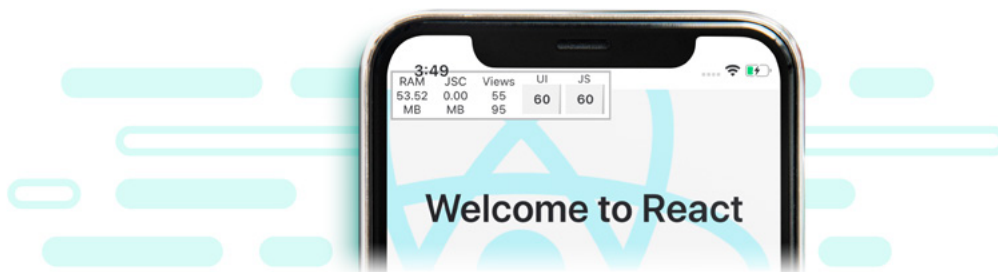
By default, React Native ships with built-in debugging utilities.



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 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.

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 case due to the overhead of WebSocket message passing.

Another inconvenience is a fact, that you cannot easily debug network requests with 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 unhandy 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 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 vast majority of JavaScript developers.

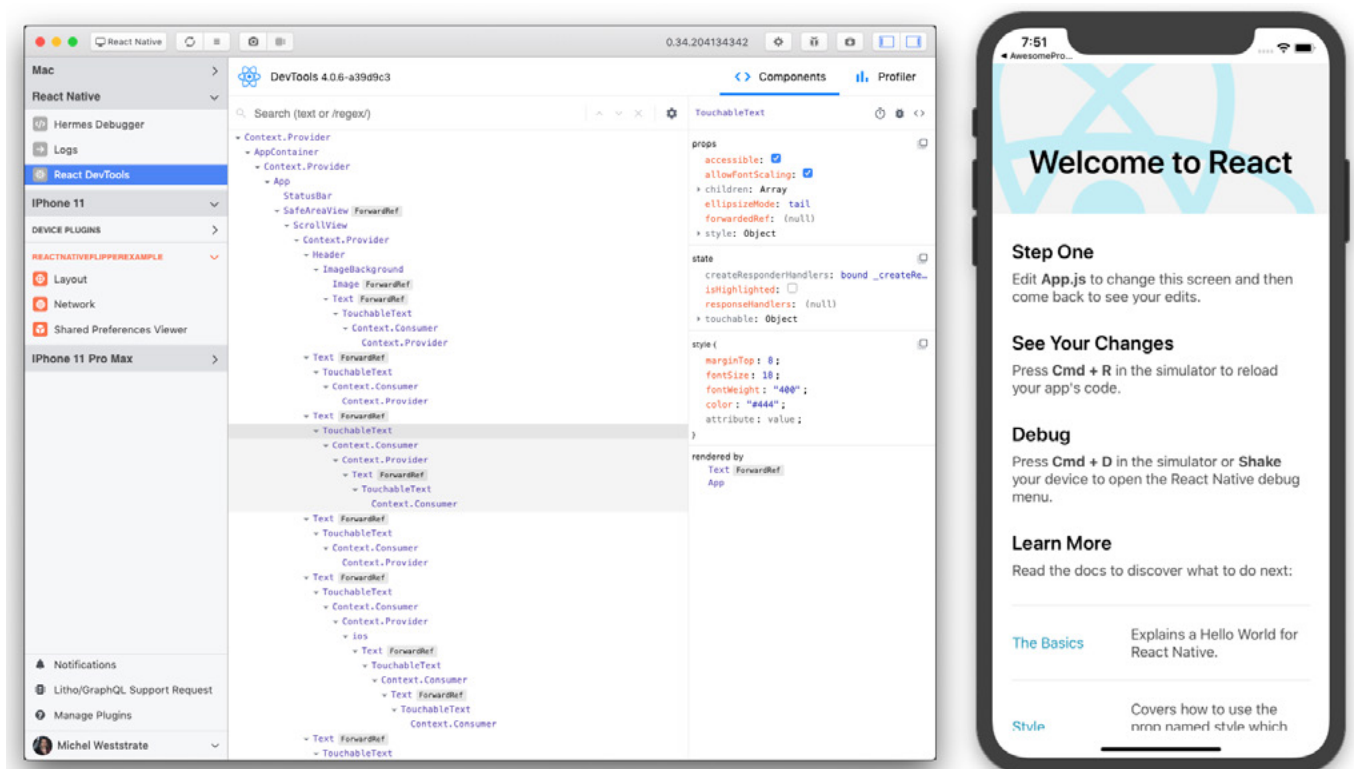
An inconvenient tooling usually decreases the velocity of the team and frustrates its members. As a result, they are not as effective as they could. That affects the quality of the app and makes the releases less frequent.

Solution: Turn on Flipper and start debugging with it

Wouldn't it be good to have one comprehensive tool to handle all of the above use-cases? Of course! 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. It is shipped by default with React Native since version 0.62 and was launched in September 2019.



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 performance caveats of using Chrome Debugger. It comes with a network inspector, React DevTools and even native view hierarchy tool.

What's more, Flipper allows for previewing logs from native code and tracking 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 in 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 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. That minimizes context switches. Without Flipper, a developer debugging an issue related to displaying the data fetched from backend had to use Chrome debugger (to preview logs), in-emulator network requests debugger and probably in-emulator layout inspector or 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.

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

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

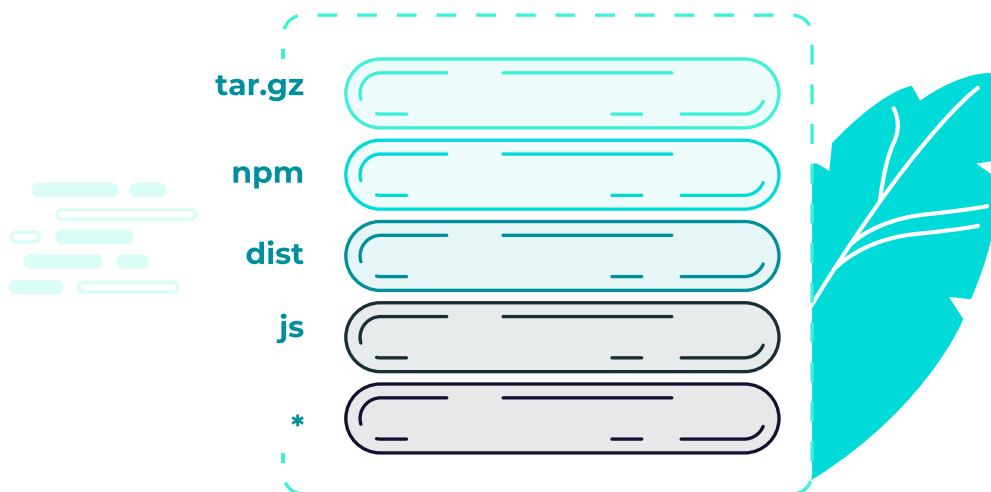
Having all debugging utilities in one interface is definitely ergonomic and does not disrupt any interactions with a emulator or device. The process will be less burdensome for your team and that will positively impact the velocity of the product development and bug fixing.

3. Automate your dependency management with `autolinking`

Switch to `autolinking` to add new packages quickly with no worries about the native code.

Issue: You're adding libraries manually or using the deprecated `react-native link`

Unlike most of the packages available on `npm`, React Native libraries usually consist of more than just a JavaScript code. Depending on the type of the functionality they provide, they may contain additional native code for Android and iOS platforms.



For example, a [react-native-fbads](#) is a React Native module used to interact with underlying Facebook SDK and - as the name suggests - to display ads within the application. To do so, the module ships with JavaScript code that allows to call the SDK from the React Native realm. On top of that, it provides Objective-C (for iOS) and Java (for Android) native modules that proxy the JavaScript calls to the proper Facebook SDK parts. It also requires Facebook SDK to be present in your app. In other words, it leaves the installation of it up to you.

Historically, React Native didn't provide an out-of-the-box solution for cases like that. Developers were encouraged to follow the best practices for a given platform.

On Android, the recommendation was to use [Gradle](#), which was already a platform of choice within the Android community. React Native used Gradle for building its source code and pulling its own dependencies, which naturally enforced all community packages to follow the same strategy.

On iOS, on the other hand, the situation was a bit more complex. By default, React Native projects didn't use any sophisticated tooling for managing dependencies - pulling them down was on you. Some community modules have started using [CocoaPods](#), which was similar to Gradle, because of the way it structured the project and provided proper dependency management. Unfortunately, CocoaPods wasn't compatible with how all React Native projects used to manage the dependencies by default.

React Native tried to partially solve this problem by introducing `react-native link` - a CLI command that once run, tries to perform all the necessary steps for you. It performed a naive *find & replace* within your configuration files and tried to add the required packages.



Unfortunately, there had always been a risk of hitting a dependency that is not compatible with the way you manage your dependencies. In that case, the only solution was to migrate to a system that works. That task alone wasn't easy - it required a lot of native-related knowledge and an understanding of build systems. If you have ever upgraded to a React Native version that introduced certain native changes, you will perfectly know what we are talking about.

Over time, CocoaPods have started to become more and more popular within the community. Eventually, React Native decided to switch to CocoaPods and make it a default way of managing the external dependencies on iOS.

As a result, both iOS and Android now have a fully-featured solution for dependency management. Thanks to that, developers can use a *npm-like* tool to pull down the dependencies, instead of downloading the files manually and putting them somewhere on the disk.

While this has helped with the confusion around adding external native dependencies, the situation still called for additional steps to run after simple `yarn add`.



3. Install The Javascript Package

Add the package to your project using your favorite package manager

```
$ yarn add react-native-fbads
```

Link the native dependencies

```
$ react-native link react-native-fbads
```

For RN < 0.60

If you have more than one Targets on your Xcode project, you might link some of them manually by dragging

`Libraries/ReactNativeAdsFacebook.xcodeproj/Products/libReactNativeAdsFacebook.a` to 'Build Phases' -> 'Link Binary With Libraries'.

For RN >= 0.60

If you are working with RN > 0.60 kindly add the following line in your Podfile

```
pod 'ReactNativeAdsFacebook', :path => '../node_modules/react-native-fbads'
```


Thanks to the fact that both Gradle and CocoaPods have a public API that can be used to manipulate the project, React Native team quickly shipped a feature called *autolinking*. It automates all the mentioned steps and removes the differences between a React Native and JavaScript package.

```
$ yarn add react-native-fbads
```

Installing React Native package should be no different from a regular JavaScript library

Long story short - if you're performing additional steps after installing React Native modules, you should keep on reading!

Codebase is harder to upgrade and maintain and you spend more time on adding additional packages

If you are still managing your dependencies "*the legacy way*" as described above, you're missing out on the build improvements and automation. As a result, experimenting with new dependencies becomes more challenging and it takes longer to set them up. Some libraries may even cease to work as developers migrate them to the new build system.

Also, you need to spend more time on upgrading to newer React Native versions as there is a bunch of native dependencies and native code that has to be revised and upgraded.

The new system is based on dedicated native build tools, such as CocoaPods and Gradle. Because of that, it is able to handle a lot of those meticulous steps for you.

Solution: Switch to `autolinking` (CocoaPods/Gradle based)

Autolinking is a new way of managing your native dependencies that, by design, is fully transparent and does not require any additional effort on your side. It is very easy to integrate and it hooks in places that you had to handle yourself.

It works the same for both iOS and Android. For the purpose of this section, let's focus on Android.

Before

```

app/settings.gradle
...

rootProject.name = 'HelloWorld'

include ':react-native-fs'
project(':react-native-fs').
projectDir = new File(rootProject.
projectDir, '../node_modules/react-
native-fs/android')

include ':app'
...

```

```

app/build.gradle
...

dependencies {
    compile project(':react-native-
fs')

    compile fileTree(dir: "libs",
include: ["*.jar"])
    compile "com.android.
support:appcompat-v7:23.0.1"
    compile "com.facebook.
react:react-native:+" // From node_
modules
}
...

```

After

```

app/settings.gradle
...

rootProject.name = 'HelloWorld'

apply from: file("../node_modules/@
react-native-community/cli-platform-
android/native_modules.gradle");
applyNativeModulesSettingsGradle
(settings);

include ':app'
...

```

```

app/build.gradle
...

dependencies {
    compile fileTree(dir: "libs",
include: ["*.jar"])
    compile "com.android.
support:appcompat-v7:23.0.1"
    compile "com.facebook.
react:react-native:+" // From node_
modules
}

apply from: file("../..node_modules/
@react-native-community/cli-platform-
android/native_modules.gradle");
applyNativeModulesAppBuildGradle
(project)
...

```

```

MainApplication.java
...

import com.rnfs.RNFSPackage;

...

@Override
protected List<ReactPackage>
getPackages() {
    return Arrays.<ReactPackage>asList(
        new MainReactPackage(),
        new RNFSPackage()
    );
}
...

```

```

MainApplication.java
...

import com.facebook.react.
PackageList;

...

@Override
protected List<ReactPackage>
getPackages() {
    return new PackageList(this).
getPackages();
}
...

```

Rather than letting Gradle know the details of every package you're using, you replace the list of packages with a single line that calls into the React Native CLI. This little helper checks your `package.json` for the possible React Native packages and automatically performs the necessary actions.

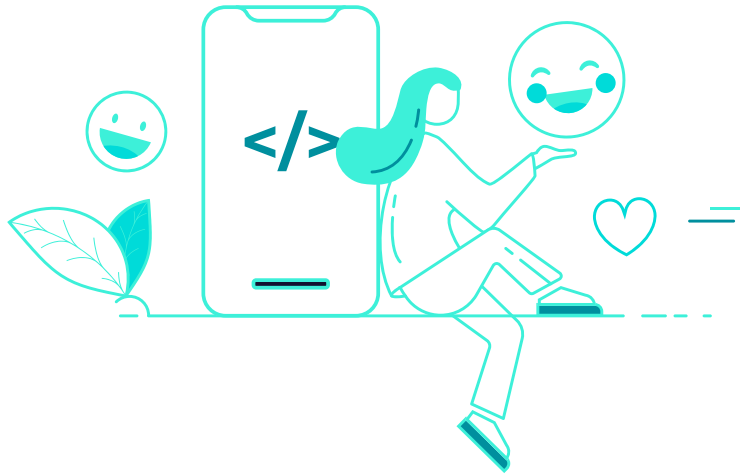
Here is an example: In *build.gradle*, the call into the React Native CLI results in an array of packages that are then registered in the pipeline. It is worth noting that paths are calculated dynamically, based on the location of your source files. Consequently, all different non-standard architectures, including popular *monorepo*'s, are now supported by default.

Another great trait of *autolinking* is that it generates the list of packages for you on Android. Thanks to that, all the packages defined by your external dependencies are automatically registered, without the need to open Android Studio and learn how to import packages in Java.

The principle here is simple - you don't have to be aware of what the library you are downloading consists of. The possibility of exploring those details should be left as an option to the most curious developers.

Benefit: You can quickly add new packages and don't worry about native code

Thanks to using *autolinking*, you can forget about all the differences between regular JavaScript and React Native packages and focus on building your application.



You no longer have to worry about external dependencies or additional build steps, including pulling SDK or linking assets.

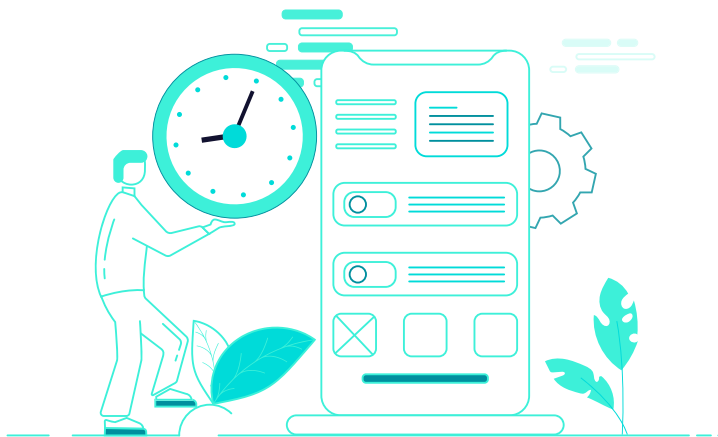
In the long run, you will appreciate this approach for its ease of maintenance and the speed of upgrading. The *CocoaPods*, *Gradle* and [React Native CLI](#) helpers ensure that the knowledge needed to both set up *autolinking* and use it within the application is as basic as possible and that it is easy to grasp for JavaScript developers.

4. Optimize your Android application startup time with Hermes

Achieve 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 Android.

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



It is not easy to describe the startup time with a single metric. 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 Lighthouse report, [there are six performance metrics used to profile your web application](#). One of them is *Time to Interactive* (*TTI* in short), which measures the time until the application is ready for the first interaction.

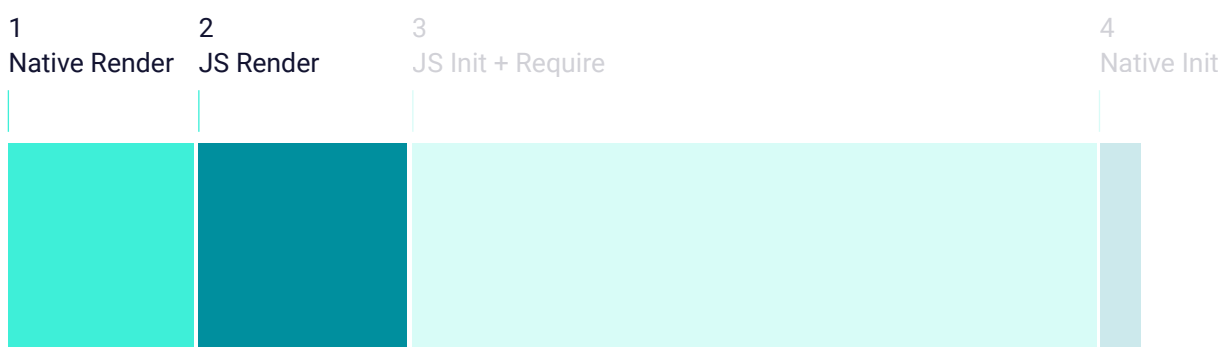
There are quite a few things that happen from the moment you press the application icon from the drawer for the first time.



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

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

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



The first one involves operations 1 and 2 from the diagram 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 and is associated with the business logic that you have created for your application. The length of this group is highly dependant 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 on Android can be one of the reasons your app gets 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 easy to understand and 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

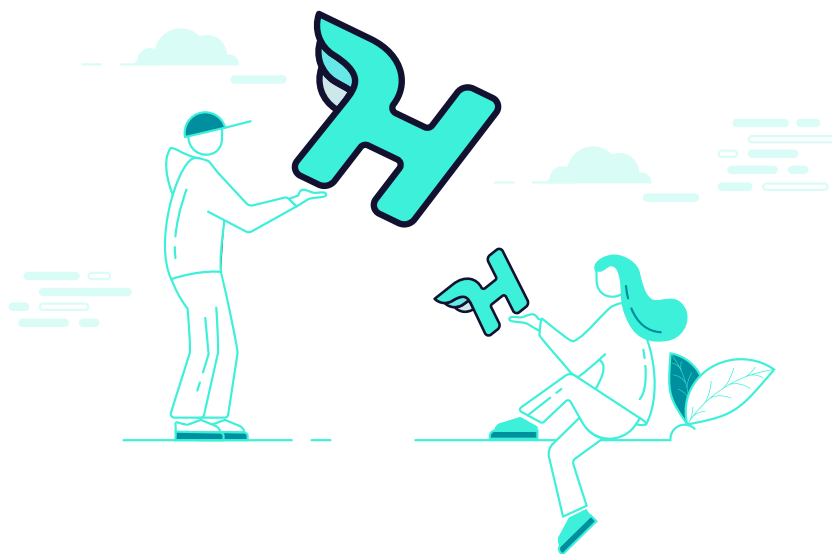
While React Native application takes care of a native interface, it still requires JavaScript logic to be running at a runtime. To do so, it spins off its own JavaScript virtual machine. By default, it uses [JavaScriptCore](#). This engine is a part of *WebKit* and by default is only available on iOS. Now it's also a preferred choice for compatibility purposes on Android. 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 don't have an easy life. They constantly ship new heuristics to improve the overall performance, including the time needed to load the code and then to execute it. To do so, they benchmark common JavaScript operations and challenge the CPU and memory needed to complete this process.

The V8 team has recently published a blog post on [improving the regular expressions' performance](#). Be sure to check it out.

Most of the work of developers handling the JavaScript engines is being tested against major and 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 be most likely already 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, 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 Facebook team ended up doing.



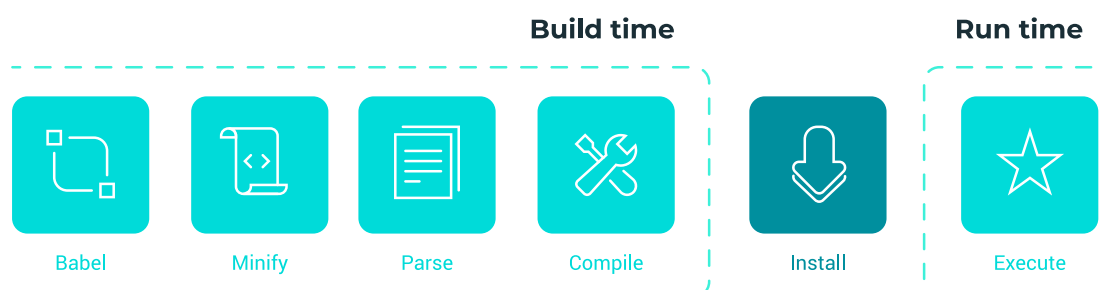
Meet Hermes - a JavaScript engine made specifically with React Native in mind. It is optimized for mobile and focuses on relatively CPU-insensitive metrics, such as application's size or the *Time to Interactive*. Right now, it is only available on Android, with potential future support for iOS.

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.

Bytecode precompilation with Hermes



It can spend more time on 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 the eager loading of 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, 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, make sure that you are running at least React Native 0.60.4 and turn the following in your `android/app/build.gradle`:

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

`enableHermes` is set to `false` at the time of writing this content. Be sure to swap to `true`.

Thanks to that, your project should clean and rebuild successfully. If that happens, congratulations - your application is now using Hermes.

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

Making your application load fast is not an easy task. It's 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 your application's performance.

Apart from that, you can also look into other significant improvements shipped by the Facebook team. To do so, get familiar with [their write-up on React Native performance](#). It is often a game of tiny and simple improvements that make all the difference when applied at once.

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.

5. Optimize your Android application's size with these Gradle settings

Improve TTI, reduce memory usage and 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 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 do early in the process. But it takes only a few additional dependencies for the application to grow from standard 5 MB to 10, 20 or even 50, 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 bundle size grow so rapidly? We will answer those questions in the next few paragraphs. But first, let's have a look on what a typical React Native bundle is made of.

By default, React Native application on Android consists of:

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

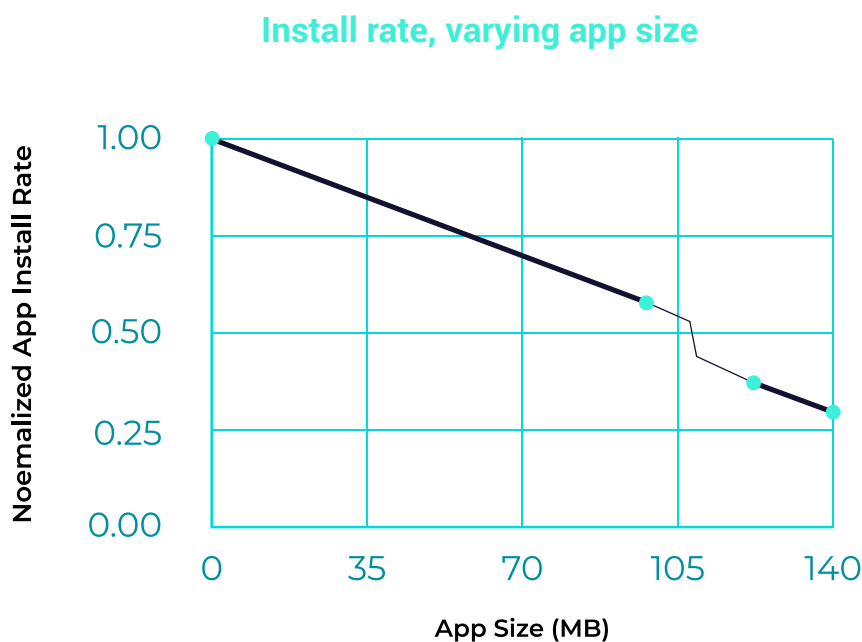
React Native offers some optimizations that allow you to improve the structure of the bundle and its overall size. But they are disabled by default.

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.

Bigger APK size means more time needed to download from 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 the network accessibility and reliability are far from perfect. Projects such as [Loon](#) promise to improve that situation, but that will take time.

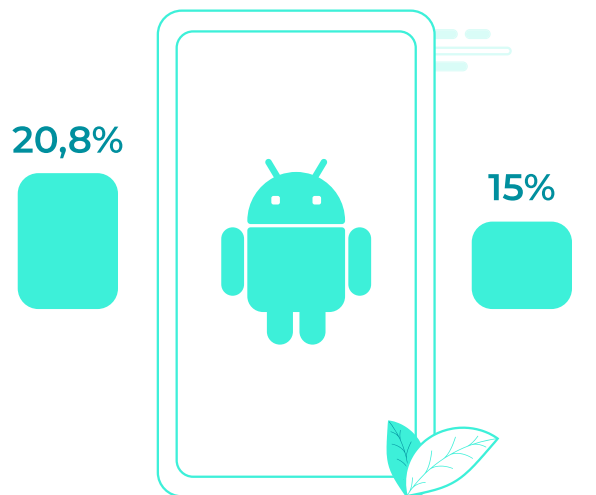
Right now, there are still markets where every megabyte of traffic has its price. In those regions, the application's size directly impacts the conversion and the installation/cancellation ratio increases along with the app size.



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 the device fragmentation. Android market is very diverse in that respect. There is a relatively big share of mid- to low-end devices that may face issues when dealing with bigger APKs.



In March 2019 ONLY 20.8% Android smartphones where high-end, up from 15% in March 2018

Source: <https://newzoo.com/insights/infographics/10-key-facts-about-the-android-smartphone-market/>

As we have stressed out 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 the last factor worth mentioning in this context – the device storage.

Apps usually end up taking more space after the installation. Sometimes they even may not fit into the device memory. In such 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 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 device. In other words, your `apk` (the file outputted from the build process) is actually four separate applications packaged into a single file. 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.

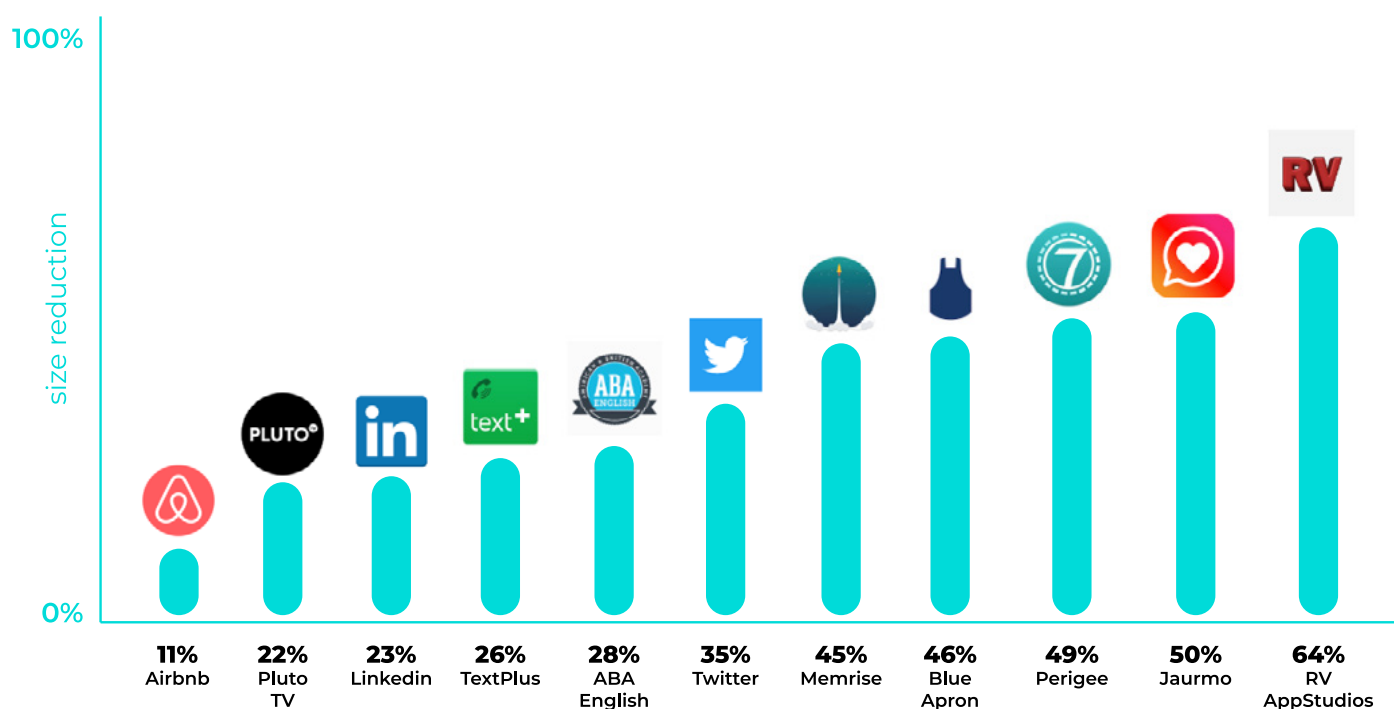
[App Bundle](#) is a publishing format that allows you to contain all compiled code and resources. It's all due to the fact that *Google Play Store Dynamic Delivery* will later build tailored APKs depending on end users' devices.

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:

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

Building a React Native app as App Bundle

Main advantage of Android App Bundle over builds for multiple architectures per CPU is the ease of delivery. After all, you have to ship only one artifact and *Dynamic Delivery* will do the whole 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 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 additional setup to achieve optimal results. In this example, we were able to reduce size of the mentioned 28 MB build by 700Kb. 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 a lot of them.

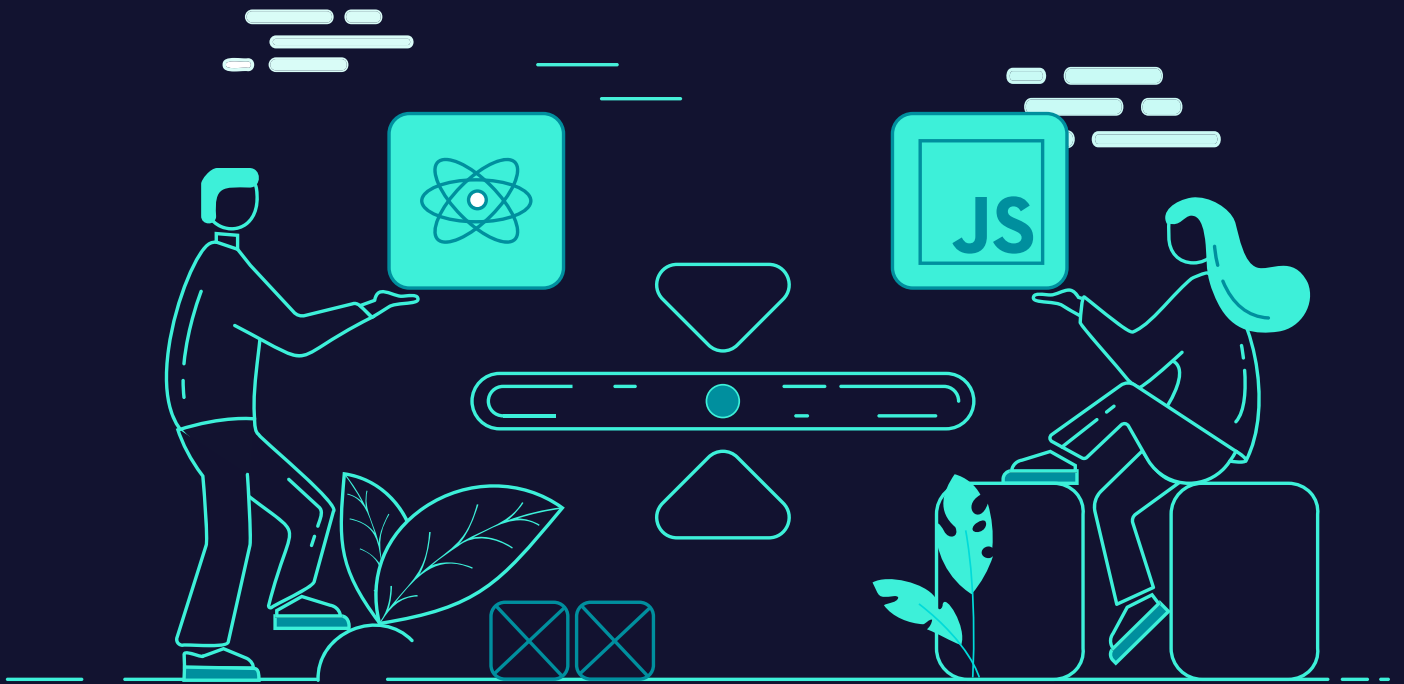
Benefit: Smaller APK, slightly faster TTI, slightly less memory used

All the mentioned steps are relatively easy to introduce and 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 interaction 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 connection. The bigger your platform gets, the more important it is to support those minor groups, as every percent of users translate into hundreds of thousands of actual users.

If you need help with performance, stability, user experience or other complex issues - contact us! As the React Native Core Contributors and leaders of the community, we will be happy to help.



Third Group

How to ship quicker with stable development environment.

Introduction

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 environment. 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. It's called Over-The-Air updates or OTA in short.

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 Over The Air updates, as most of the articles suggest. It's about building a steady and healthy development environment where React Native shines and accelerates the innovation.

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

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 rely only 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 a 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 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. It's easy to end up with a test that doesn't have a good coverage of your

features. Or only to test a 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 of 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, app can be removed from app store 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 to 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 lowering of your rating 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 regressions and crashes that it will lose its whole velocity and confidence.

Solution: Don't aim at 100% coverage, focus on key pieces of the app. Use unit tests (Snapshots), integration tests (Detox)

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 the full coverage because of the government regulations they must abide by. But in such case you're probably already aware of the problem.

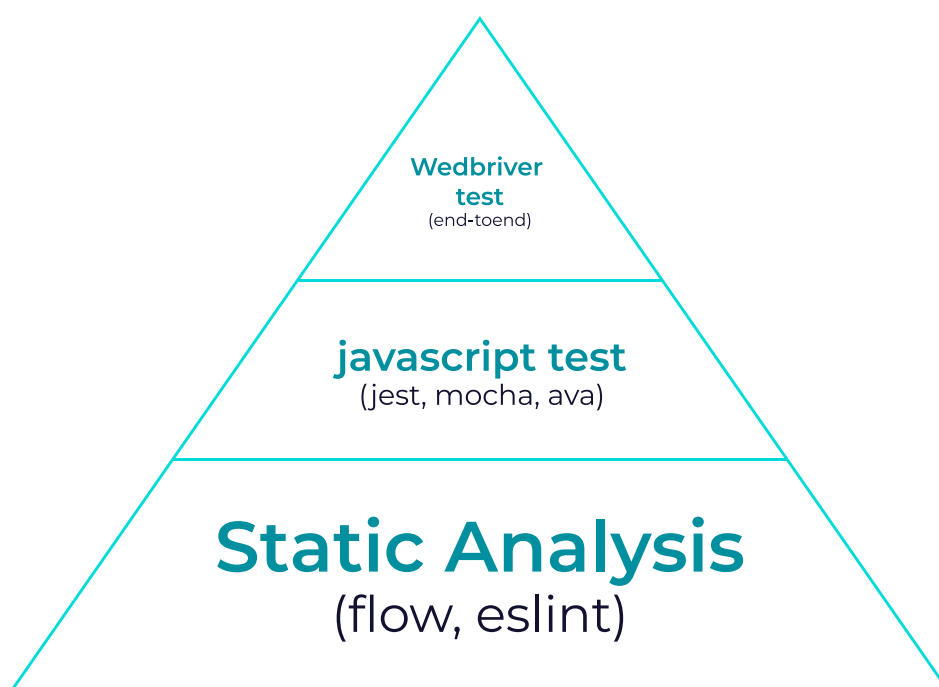
It's crucial for you to focus your time on testing the right thing. Learning to identify business-critical features and capabilities is usually more important than writing a test itself. After all, you want to boost confidence in your code, not a write test for the sake of it. Once you do that, all you 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 React Native core.

Therefore, for practical reasons, we can distinguish between:

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

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 be referring to it also in the following sections.

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

```
function QuestionsBoard({ questions, onSubmit }) {
  const [data, setData] = React.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>
  );
}
```

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 - and 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 pretty 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 implementation details of your components, making it more resilient to internal refactors.

A test for the **QuestionsBoard** component would look as follows:

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

test('form submits two answers', () => {
  const allQuestions = ['q1', 'q2'];
  const mockFn = jest.fn();
  const { getAllByA11yLabel, getByText } = render(
    <QuestionsBoard questions={allQuestions} onSubmit={mockFn} />
  );

  const answerInputs = getAllByA11yLabel('answer input');

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



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

Test suite taken from the official RNTL documentation

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

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

Note: You may have also heard of a technique called “snapshot testing” for JS. It can help you in some of the testing scenarios, when repetitive data is being asserted from the test. The technique is widely adopted in React ecosystem, because of a built-in support from Jest. But it’s a low-level API and should be avoided, unless you have a firm experience in testing.

If you’re into learning more about the snapshots, check one of the Rogelio’s (one of the Jest contributors) [talks about snapshot testing](#) and the [project repository](#) which can help you with testing differences between various states of data, including React components

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 a 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, 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 and a part of the React Native testing pipeline. Using it you will be able to ensure that your framework of choice is supported by the latest React Native versions. That is especially important in the context of the future changes that may happen at a framework level.

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

This quick snippet showed 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 are [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 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 on innovating and pleasing users with flashy new functionalities, and not squashing bugs and regressions over and over again.

2. Have a working Continuous Integration (CI) in place

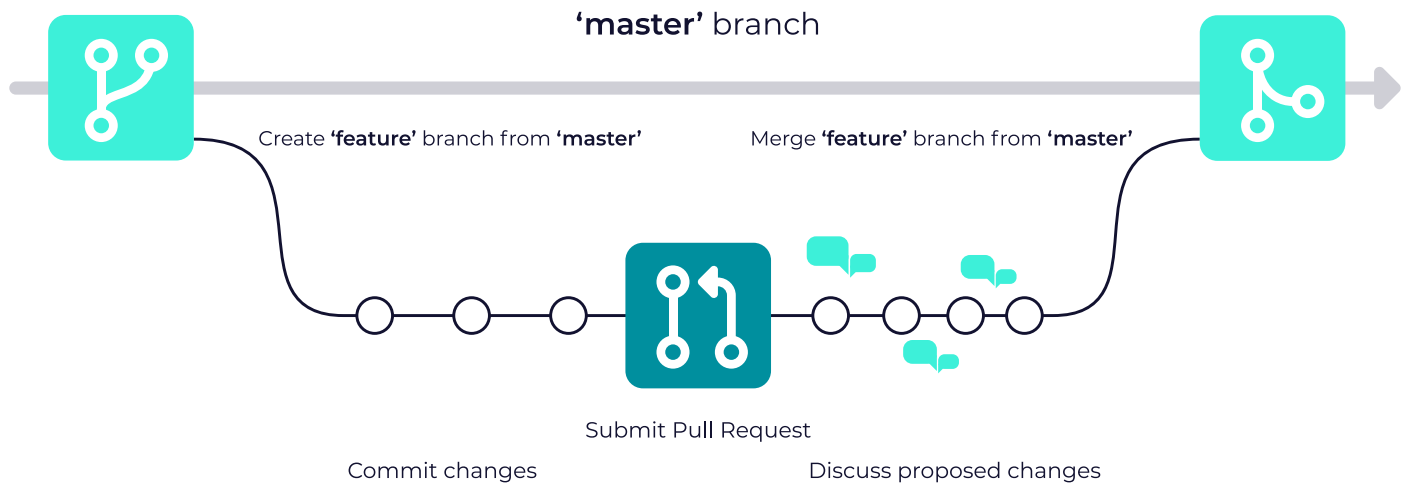
Use CI provider to improve building, testing, and distribution of your apps.

Issue: Lack of CI or having an unstable one means 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*.

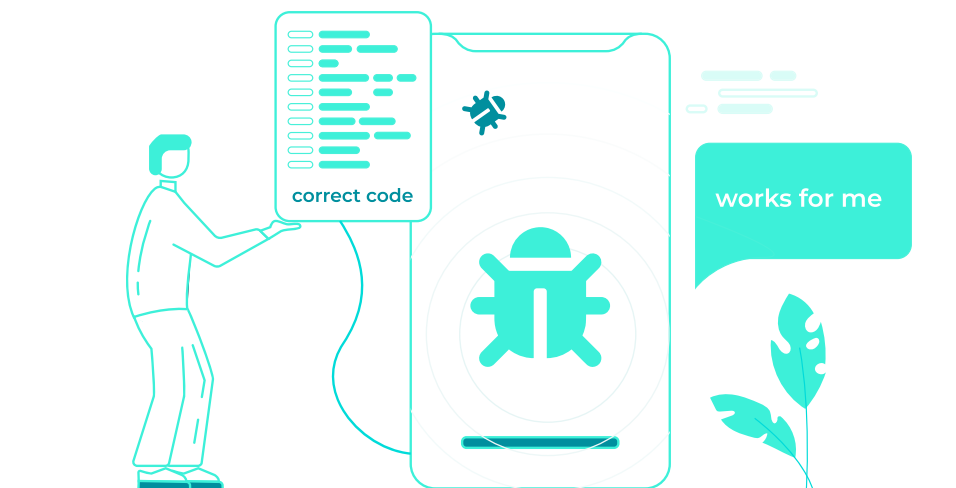
For better context, let's take a look at the early days of the development process. When you're starting out, your focus is on shipping the first iteration (MVP) as fast as possible. Because of that you may overlook the importance of the architecture itself. When you're done with the changes, you submit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



An example of workflow on Github, where changes are proposed in a 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 reason for that is simple - we all have different configurations, environments and ways of working.



While relying on the same configuration, similar principles of the development and attention to details is helps move faster at the early stages, it may result in shipping something that breaks the tests.

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 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 with 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 an 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.

Using a CI service, you can not only test your code, but also build a new version of 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 CircleCI) 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](#), [Travis](#) and the recently released [Github Actions](#). For the purposes of this section, we have selected the CircleCI.

It is the default CI provider for React Native and all projects created by its Community. In fact, there is actually an example project demonstrating the use of CI with React Native. You can learn more about it [here](#). We employ it later in this section to present different CI concepts.

Note: The good rule of the thumb is to take advantage of what React Native / 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.

With most of the 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
jobs:
  android:
    working_directory: ~/repo
    docker:
      - image: reactnativecommunity/react-native-android
    steps:
      - checkout
      - run: npm i -g envinfo && envinfo
      - run: yarn install
      - run: cd android && chmod +x gradlew && ./gradlew assembleRelease

workflows:
  version: 2
  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. Github and its Github Actions is an example of such provider. You may find many similarities between those services.

There are three most important building blocks of a CircleCI configuration: *commands*, *jobs* and *workflows*.

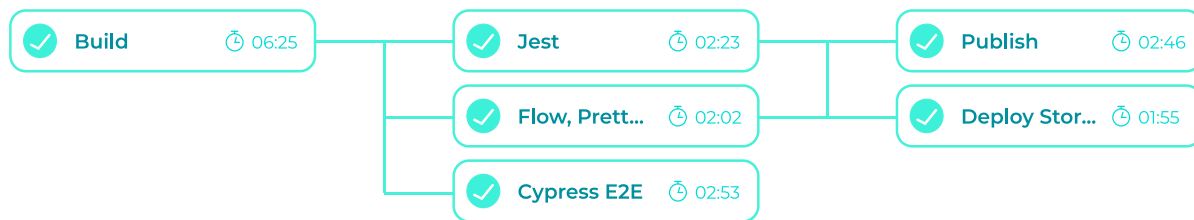
Command is nothing more but a shell script. It is executed within the specified environment. Also, it is what performs the actual job in the cloud. It can be anything, from a simple command to install your dependencies, such as `yarn install` (if you're using *Yarn*) to a bit more complex one `./gradlew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

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 [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 workflow, but this can be changed by specifying requirements for a *Job*.



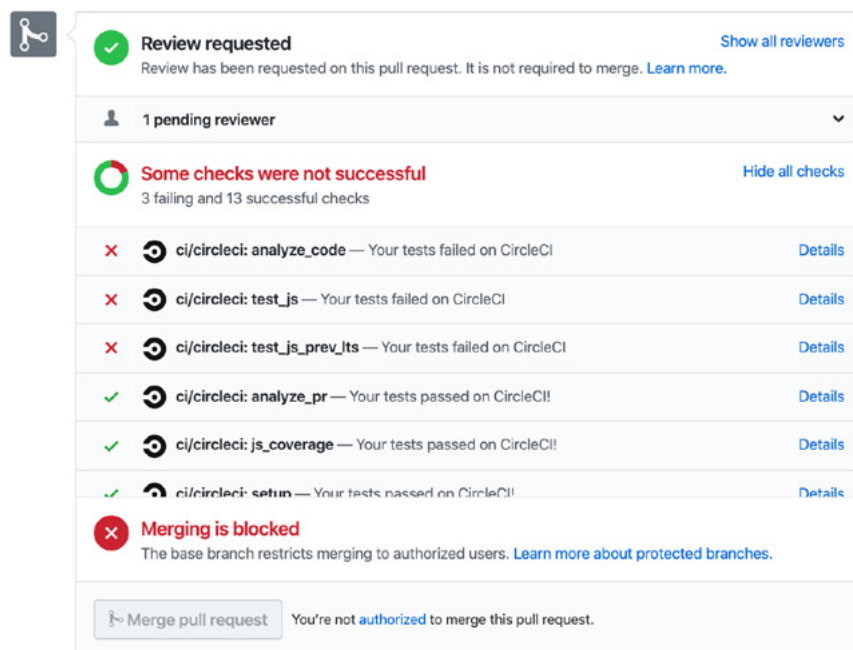
Workflow contains jobs that can be grouped to run in a sequence or in parallel

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

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

Properly configured and working CI provider can save you a lot of time when shipping 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.

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.

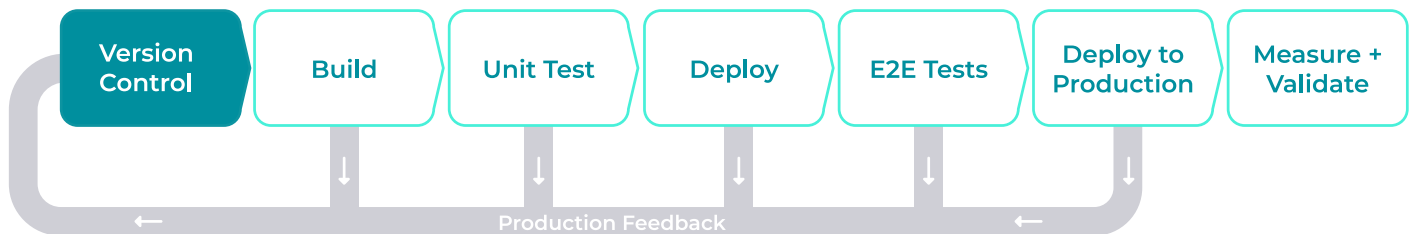
Issue: Building and distributing your apps manually is a complex and time consuming process

As you have learned in the previous section, automation of the critical pieces of the development lifecycle can help you improve the overall development speed and security. The shorter 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 reason for that is simple - 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 better developer experience for someone who's working on a native application. Developers that are used to such 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, it is very easy to ensure that all releases are done the same way and enroll company-wide standards.

Solution: Establish a continuous deployment setup (based on `fastlane`) that makes the build and generates the changelog. Ship to your users instantly

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

First 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 by doing its magic.



Second 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 it to its destination. Again, you can either upload the files to a desired service (e.g. *App Store*) manually or using 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 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.

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.

Note: Describing the ins and outs of the setup would make this article 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.

Setting up Fastlane

Before going into the details for Android and iOS, you have to make sure that the [Fastlane have been installed and configured](#) on our devices.

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 *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 our 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 *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 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 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 the 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 [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 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 *master*. 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 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.

```
deploy_ios:
  macos:
    xcode: '11.3.1'
  working_directory: ~/CI-CD
  steps:
    - checkout
    - attach_workspace:
        at: ~/CI-CD
    - run: HOMEBREW_NO_AUTO_UPDATE=1 brew install watchman
    - run: bundle install
    - run: cd ios && bundle exec fastlane deploy

workflows:
  version: 2
  deploy:
    jobs:
      - deploy_ios
```

Part of CircleCI configuration that executes Fastlane build lane

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

Note: This is just 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 in 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.

Benefit: Short feedback loop along with nightly or weekly builds let you verify features faster and ship critical bugs more often.

With automated deployment you no longer waste your time for 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.

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 the 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 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 *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 for Apple and Google to review your app.

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 review your product and allows it to be distributed.

Even though the review times got much better over the course of years, it is still a good escape hatch to be able to immediately recover from an error that slipped through the testing pipeline and got into production.

Solution: Implement OTA updates with App Center / CodePush

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.

The most popular and widely used tool for OTA updates is [CodePush](#), a service that is now a part of Microsoft's [App Center](#) suite.

Note: You have to create an account in the App Center in order to continue. If you were reading the previous section, you should already have one. The OTA option will be visible under the application you have created. It is generally a good practice to use both OTA and release capabilities from App Center for easier configuration.

Configuring the native side

To integrate CodePush to 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 native side, you can use the JavaScript API to enable the updates and define when they should happen. The simplest way that enables fetching updates on the app startup is to use the `codePush` wrapper and wrap your main component.

```
import codePush from "react-native-code-push";

class MyApp extends Component {}

MyApp = 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 solution:

```
class MyApp extends Component { }

MyApp = 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 the 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.

Benefits: Ship critical fixes and some content instantly to the users

With OTA updates integrated to 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 the startup. It may be a mishandled error - you never had a backend failure during the development and forgot to handle such edge cases.

A potential fix for this issue is simple - it may be enough to just display a fallback message and inform 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 that bad UX will affect the majority of users.

If you need help with performance, stability, user experience or other complex issues - contact us! As the React Native Core Contributors and leaders of the community, we will be happy to help.

Thank you

We hope that you will find the aforementioned best practices for React Native optimization useful and they will make your work easier. We did our best to make this guide comprehensive and describe both 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



Mike Grabowski

Co-Founder & CTO of [Callstack](#). Mike is a React Native core contributor and the author of libraries like RNPM and Haul. Mike makes sure our infrastructure performs well to its limits.



Jakub Mazurek

TypeScript enthusiast experienced with React and Node. He feels best in creating solutions from scratch, making proof of concept projects and experimenting with new technologies. In his free time, he enjoys various outdoor activities like hiking and riding a bike.

e-book

The Ultimate Guide to React Native Optimization

Improve user experience, performance, and stability of your apps.

{callstack}

We are [the official Facebook partners](#) on React Native. We've been working on [React Native projects](#) for over 5 years, delivering high-quality solutions for our clients and contributing greatly to the React Native ecosystem. [Our Open Source projects](#) help thousands of developers to cope with their challenges and make their work easier every day.

{callstack.com}