



Debug React Native Apps

Because There Will Be Errors

Jack

Table of Contents

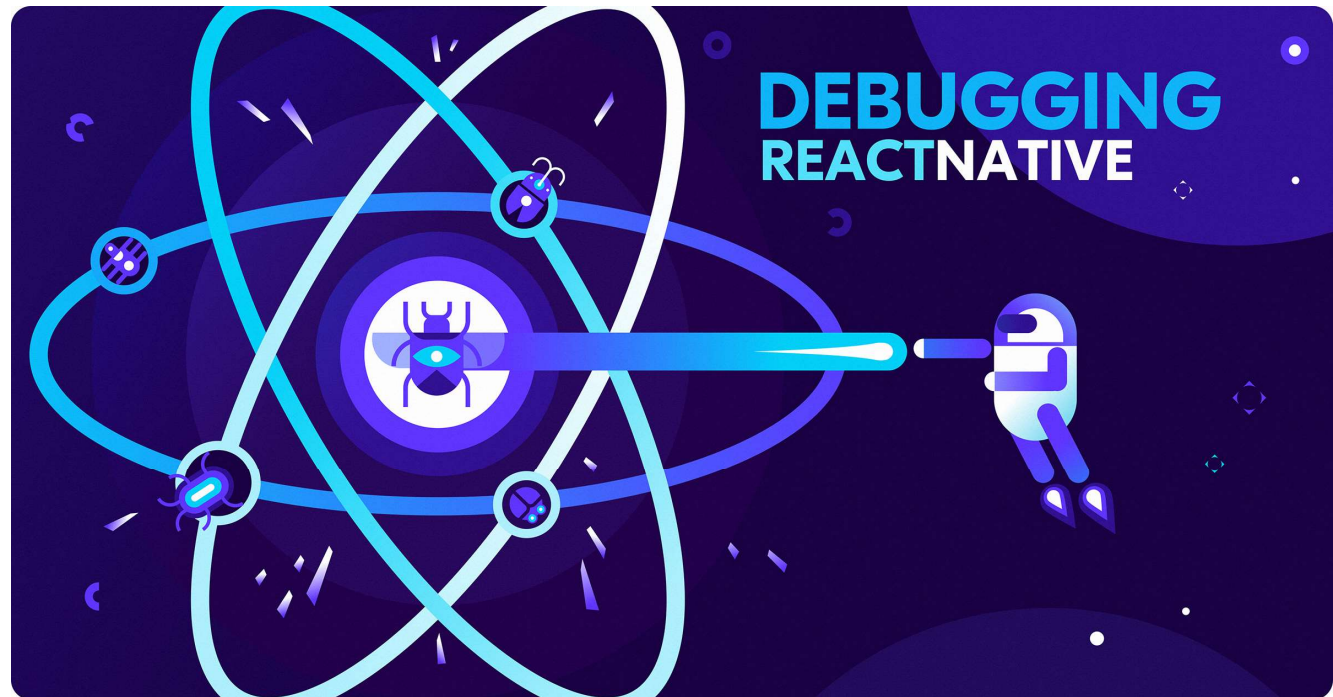
- Introduction
- What to Debug & How to Debug?
- Running the App on a Real Device & Debugging
- Handling Error Messages
- Understanding Code Flow with `console.log()`
- Using the Remote Debugger & Breakpoints
- Working with the Device DevTools Overlay
- Debugging the UI & Using React Native Debugger
- Debugging React Native in VS Code
- Wrap Up
- Useful Resources & Links

Requirements

- A little React knowledge
- A little JavaScript knowledge

Introduction

- We will have a look at what we can do when things go wrong, namely at debugging React Native apps because there always will be some errors and there are different tools you can use to track down your errors, to fix them or to avoid them at runtime



What to Debug?

Error Messages / App Crashes

Logic Errors

Styling. Layout & UX

Syntax errors

Undesired or unexpected app behavior

Unexpected / “wrong” styling or layout

Bugs in your codes. (e.g. using undefined value, wrong types...)

Unexpected / Unhandled user behavior

Inconsistent result on different devices

“Unavoidable errors” (e.g. failing network requests)

Sequence of steps lead to errors

Layout doesn’t “work” on certain devices or orientations

How to Debug?

Read the error messages (seriously)!

Often, you the error messages contains the solution or a (pretty) exact pointer at the problematic code line.

`console.log()`

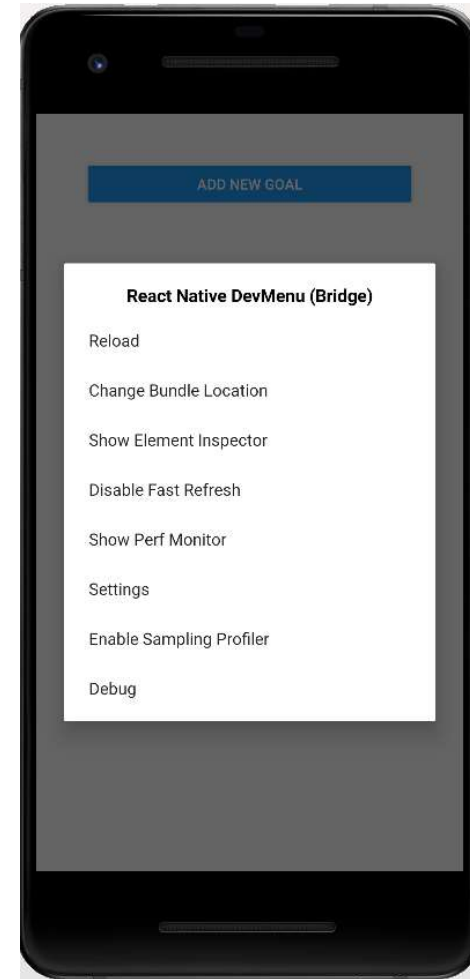
Get a feeling for the “flow” of your code(What happens when? Which value is used where?)

Other debuggers(e.g. Chrome)

Dive into the code in great detail and step by step.

Running the App on a Real Device & Debugging

- With the app running on a real device, you can debug it from there, too.
Shake the device a little to bring up the developer menu.
- There, you can enable the debugger and the other features covered on this slide.



Handling Error Messages: Syntax Error

- Missing bracket

App.js

```
35 <Button title="Add New Goal" onPress={() => setIsAddMode(true)} />
```

Failed to compile

Syntax Error
Unexpected token, expected ","

Source

```
30 | return (
31 |   <View style={styles.screen}>
32 |     <Button title="Add New Goal" onPress={() => setIsAddMode(true)} />
33 |     <GoalInput
34 |       visible={isAddMode}
35 |       onAddGoal={addGoalHandler}
  
```

C:\Users\jack\debugDemo\test1\ddl\App.js (32:68)

Device log

```
error: SyntaxError: C:\Users\jack\debugDemo\test1\ddl\App.js: Unexpected token, expected ",", (32:67)
30 |   return (
31 |     <View style={styles.screen}>
32 |       <Button title="Add New Goal" onPress={() => setIsAddMode(true)} />
33 |       <GoalInput
34 |         visible={isAddMode}
35 |         onAddGoal={addGoalHandler}
  
```

ERROR [Error: TransformError SyntaxError: C:\Users\jack\debugDemo\test1\ddl\App.js: Unexpected token, expected ",", (32:67)]

Metro log

It's still pointing us at the wrong solution, expecting a comma but it does point us at the right code

Handling Error Messages: Bugs in Your Codes

- Undefined value

App.js

```
11 const addGoalHandler = goalTitle => {
12   if (goalTitle.length === 0) {
13     return;
14   }
15   console.log('addGoalHanler', goalTitle);
16   setCourseGoals(currentGoals => [
17     ...currentGoals,
18     {id: Math.random().toString(), value: goalTitle},
19   ]);
20   setIsAddMode(false);
21 };
```

Do length
check

GoalInput.js

```
11 const addGoalHandler = () => {
12   props.onAddGoal();
13   setEnteredGoal('');
14 };
```

Forgot to
return value



Log 1 of 1

Uncaught Error

undefined is not an object (evaluating 'goalTitle.length')

Source

```
10 |
11 |   const addGoalHandler = goalTitle => {
> 12 |     if (goalTitle.length === 0) {
    |           ^
13 |       return;
14 |     }
15 |     console.log('addGoalHanler', goalTitle);
```

C:\Users\jack\debugDemo\test1\dd1\App.js (12:18)

Call Stack

```
addGoalHandler
C:\Users\jack\debugDemo\test1\dd1\App.js:12:18
```

Device log

It informs us that the problem has something to do with something being undefined and that it is related to this goal title length check.

Understanding Code Flow with console.log()

App.js

```
25 const removeGoalHandler = goalId => {  
26   console.log('TO BE DELETED:' + goalId);  
27   console.log(courseGoals);  
28   setCourseGoals(currentGoals => {  
29     return currentGoals.filter(goal => goal.id !== goalId);  
30   });  
31 };
```

Add some info text
like to be deleted
in front of this

The entire
component will be
re-rendered when
we update our
course goals with
this line

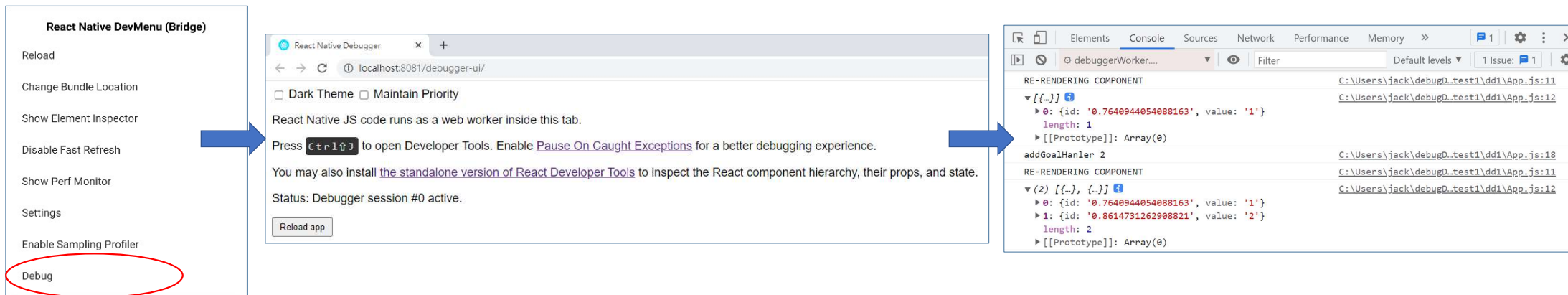
```
7 export default function App() {  
8   const [courseGoals, setCourseGoals] = useState([]);  
9   const [isAddMode, setIsAddMode] = useState(false);  
10  console.log('RE-RENDERING COMPONENT');  
11  console.log(courseGoals);  
}
```

So we can maybe just also add a
little console log, re-rendering
component, so that we know that
the course goal list we're printing
after this one is the list after every
re-render cycle

We of course see that that's behaving correctly here but simply imagine you have a more complex flow in your code you want to debug, then such console log statements can really help you understand how your code is running, how often it's running and if the correct values are getting used.

Using the Remote Debugger & Breakpoints: Launch

- Now sometimes, console log alone doesn't get you that far, you need more help and in such cases, you can debug your code remotely.
- On Android simulator, you press **control m** and it will open debug menu
- If you press **Debug**, a new tab should open up in the browser which automatically navigated to localhost <http://localhost:8081/debugger-ui/>
- You can open the Chrome developer tools now with the shortcut you should be seeing here



React Native DevMenu (Bridge)

- Reload
- Change Bundle Location
- Show Element Inspector
- Disable Fast Refresh
- Show Perf Monitor
- Settings
- Enable Sampling Profiler
- Debug**

React Native Debugger

localhost:8081/debugger-ui/

☐ Dark Theme ☐ Maintain Priority

React Native JS code runs as a web worker inside this tab.

Press **Ctrl+J** to open Developer Tools. Enable [Pause On Caught Exceptions](#) for a better debugging experience.

You may also install [the standalone version of React Developer Tools](#) to inspect the React component hierarchy, their props, and state.

Status: Debugger session #0 active.

Reload app

Chrome DevTools Console

debuggerWorker...

RE-RENDERING COMPONENT

```

{
  "id": "0.7640944054088163",
  "value": "1",
  "length": 1,
  "[[Prototype]]": Array(0)
}

```

addGoalHandler 2

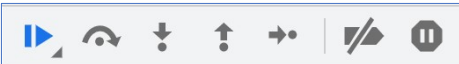
RE-RENDERING COMPONENT

```

(2) [
  {
    "id": "0.7640944054088163",
    "value": "1",
    "length": 2,
    "[[Prototype]]": Array(0)
  },
  {
    "id": "0.8614731262908821",
    "value": "2",
    "length": 2,
    "[[Prototype]]": Array(0)
  }
]

```

Using the Remote Debugger & Breakpoints: Set Breakpoints

- You can dive into sources for example to dive into your source code and set breakpoints.
- Now the cool thing is you can walk through your code step-by-step with these controls  and you can also hover over things to look into your code, for example to see the current value in goal title.



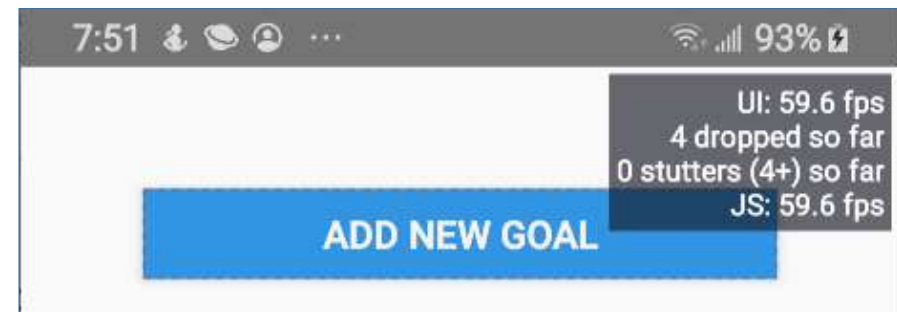
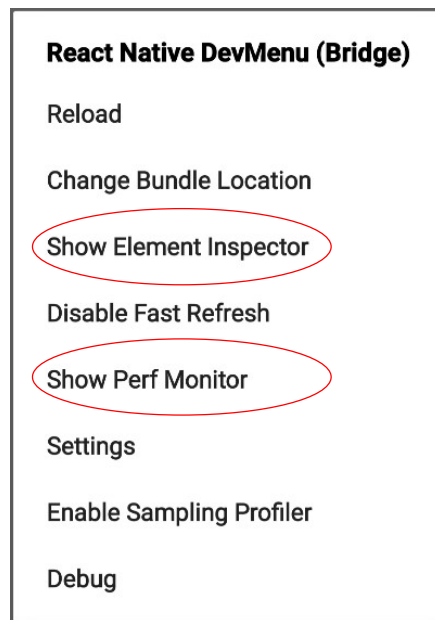
The screenshot displays the Chrome DevTools Remote Debugger interface. The Sources panel on the left shows the file structure, with 'App.js' selected. The main editor shows the source code of 'App.js' with a breakpoint set at line 18. The console on the right shows the current value of 'goalTitle' as '3'. The Breakpoints panel on the right shows the active breakpoint at line 18.

```
15 if (goalTitle.length === 0) {
16   return;
17 }
18 console.log('addGoalHanler', goalTitle);
19 setCourseGoals(currentGoals => [
```

Line 18, Column 5 (source mapped from [index.bundle?platform=andro](#))

Working with the Device DevTools Overlay

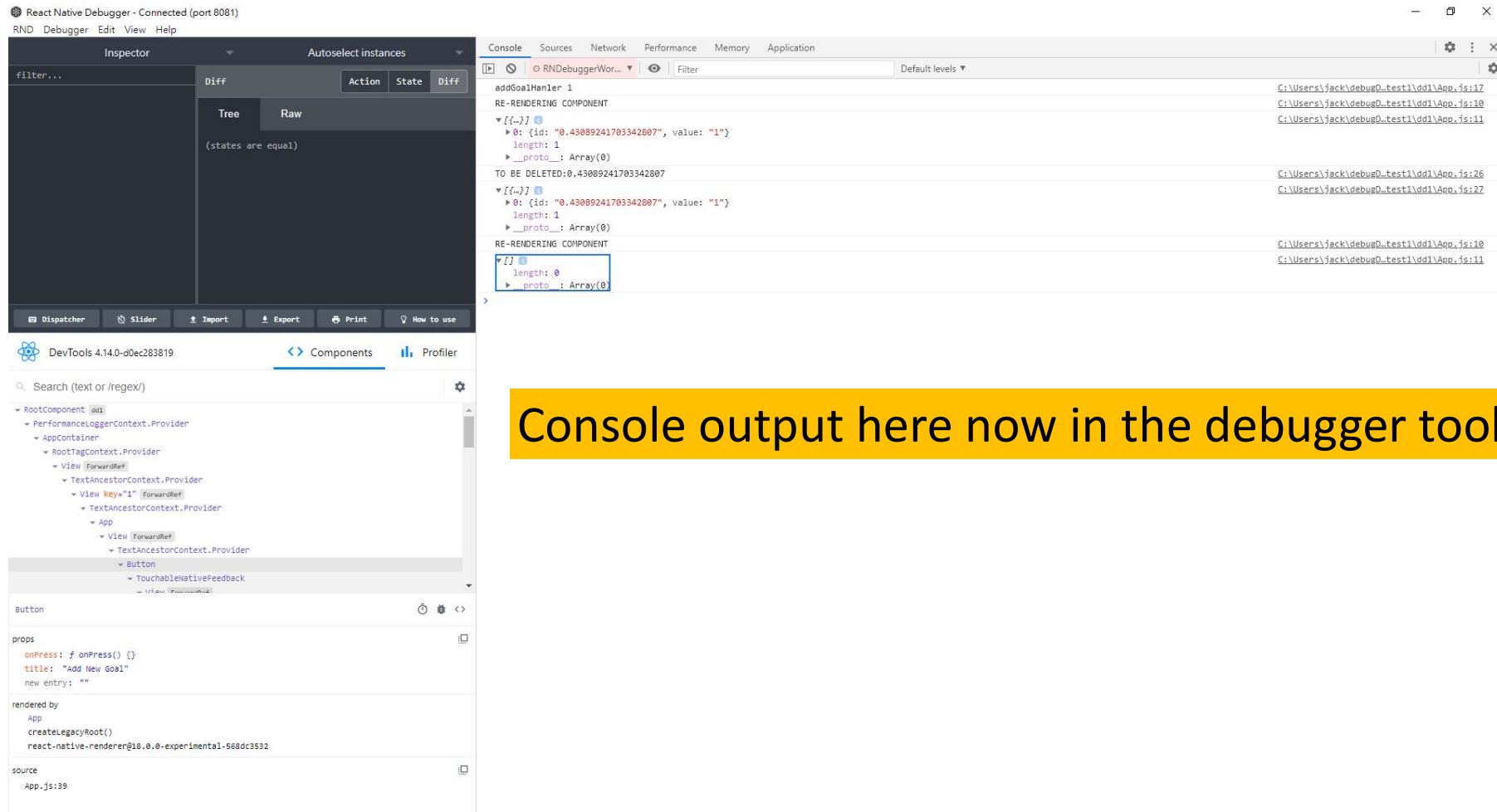
- If you enable **Perf Mointor**, you get this overlay which basically informs you about the performance you are having and there you can see at how many frames your app is running, how many frames were dropped and so on.
- If you enable **Element Inspector**, you can click onto items in your user interface to get information about them.



Debugging the UI & Using React Native Debugger

- There are several even better tools for inspecting the user interface, like **React Native Debugger**
- For this to work, enable remote Javascript debugging on the devices, just what we did before to debug this in Chrome.
- Now with this opened up, press command t on Mac or **control t** on Windows or Linux in here to open a new tab and open and confirm that React Native debugger port which the Chrome tab also used before and confirm this and now it's trying to connect there and to make this succeed, and you'll see your console output here now in the debugger tools.

React Native Debugger: Console Output

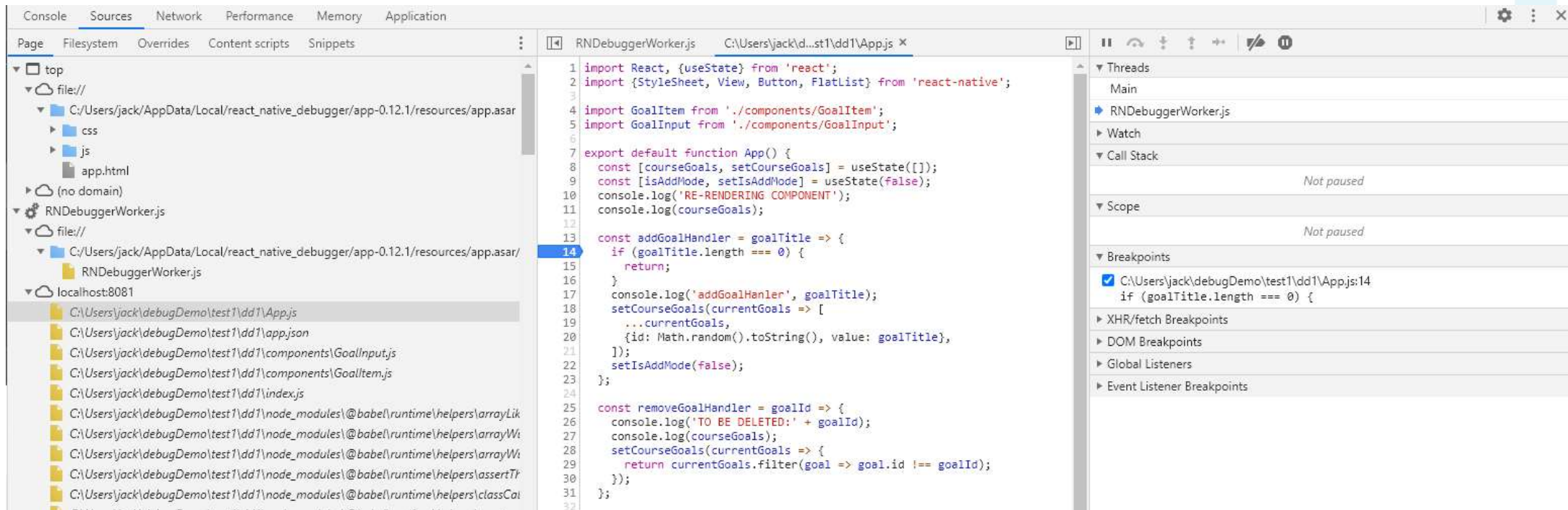


The screenshot displays the React Native Debugger interface. The top bar indicates the application is connected to port 8081. The main window is divided into several panels:

- Inspector:** Shows the component tree with the selected component being a `Button` inside a `TouchableNativeFeedback` component. The state is shown as `(states are equal)`.
- Console:** Displays the log output. The most recent log entry is a warning: `Warning: Can only perform a diff on the props object, not the second parameter. This is likely a bug in the component library you are using, or you have provided a function as a prop. See https://fb.me/react-warning-can-only-diff-props for more details. Please report this error.` The log entry is expanded, showing the props object: `{id: "0.43089241703342807", value: "1", length: 1, __proto__: Array(0)}`.
- Components:** Shows the component tree with the selected component being a `Button` inside a `TouchableNativeFeedback` component.
- Profiler:** Shows the performance profile of the application.

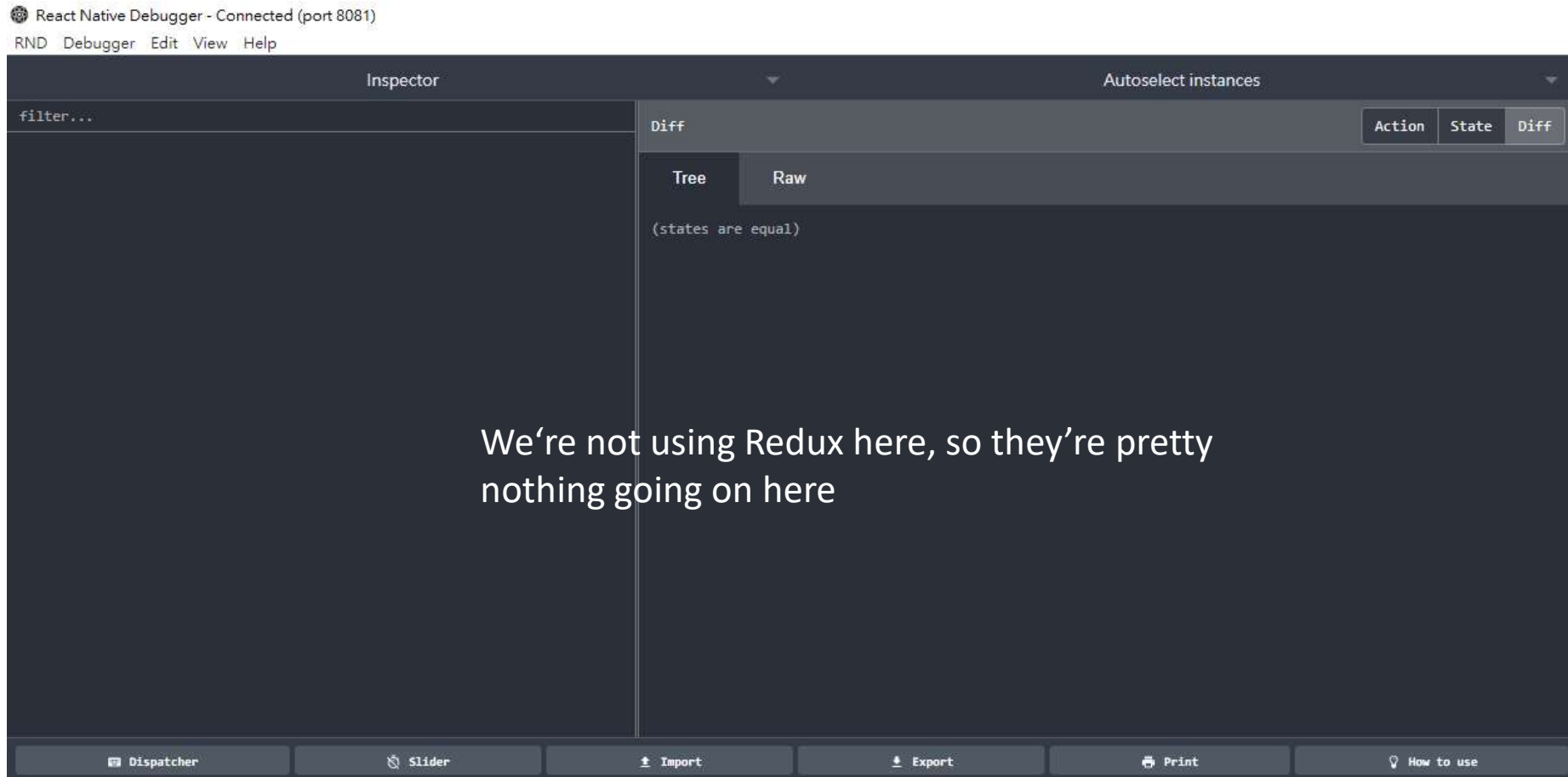
The console output is highlighted in a yellow box with the text: **Console output here now in the debugger tools**

React Native Debugger: Sources

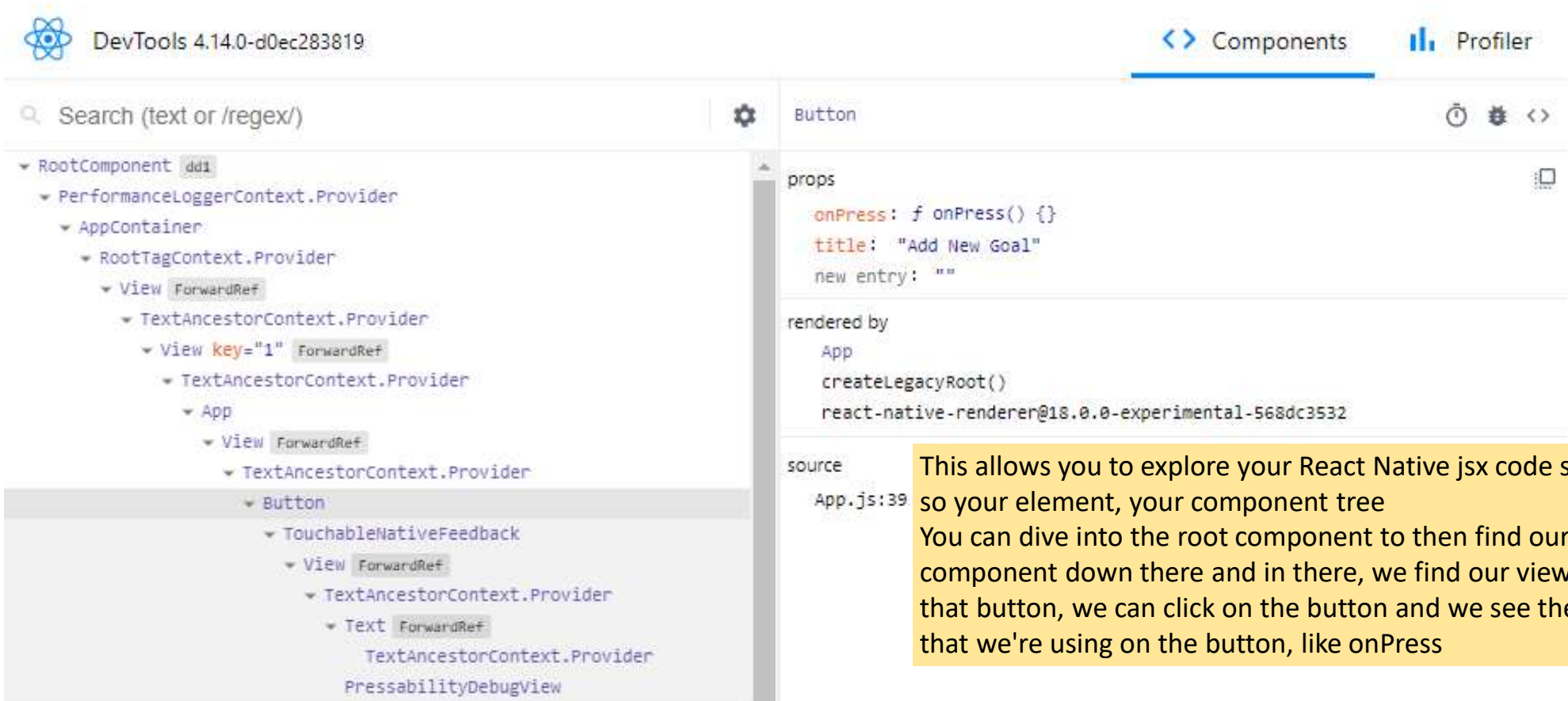


You'll also see, if I expand this, that in sources, you can again dive into your code here if you want to.

React Native Debugger: Redux



React Native Debugger: Elements



This allows you to explore your React Native jsx code so to say, so your element, your component tree

You can dive into the root component to then find our app component down there and in there, we find our view with that button, we can click on the button and we see the props that we're using on the button, like onPress

Elements debugging tool

React Native Debugger: Props and States of Elements

The screenshot shows the React Native Debugger interface. On the left, the component tree is displayed with 'GoalInput' selected. The right panel shows the props for 'GoalInput':

```
props
  onAddGoal: f addGoalHandler() {}
  onCancel: f cancelGoalAdditionHandler() {}
  visible: true
  new entry: ""
```

Below the props, the 'hooks' section shows 'State: "1234"'. A small inset window displays a text input field containing '1234' and two buttons labeled 'CANCEL' and 'ADD'. The 'rendered by' section shows the component is rendered by 'App' using 'createLegacyRoot()' and 'react-native-renderer@18.0.0-experimental-568dc3532'. The 'source' section shows 'App.js:40'.

We can even change the visible prop to toggle this modal like this if we want to

if I start typing here, like 1234, you'll see that this also updates here

React Native Debugger: Styles

DevTools 4.14.0-d0ec283819

Components Profiler

Search (text or /regex/)

View

ForwardRef

props

- children: [<Button />, <GoalInput />, <FlatList />]
- style: [{...}, {...}]
- new entry: ""

style {

- padding: 50;
- attribute: value;

}

You can even inspect the style and not just see the style but also change it, for example to increase the padding or reduce it and this gives you an easy way of testing different looks and layouts on the screen to find out what's looking good for you and what you want to change.

ADD NEW GOAL

1234

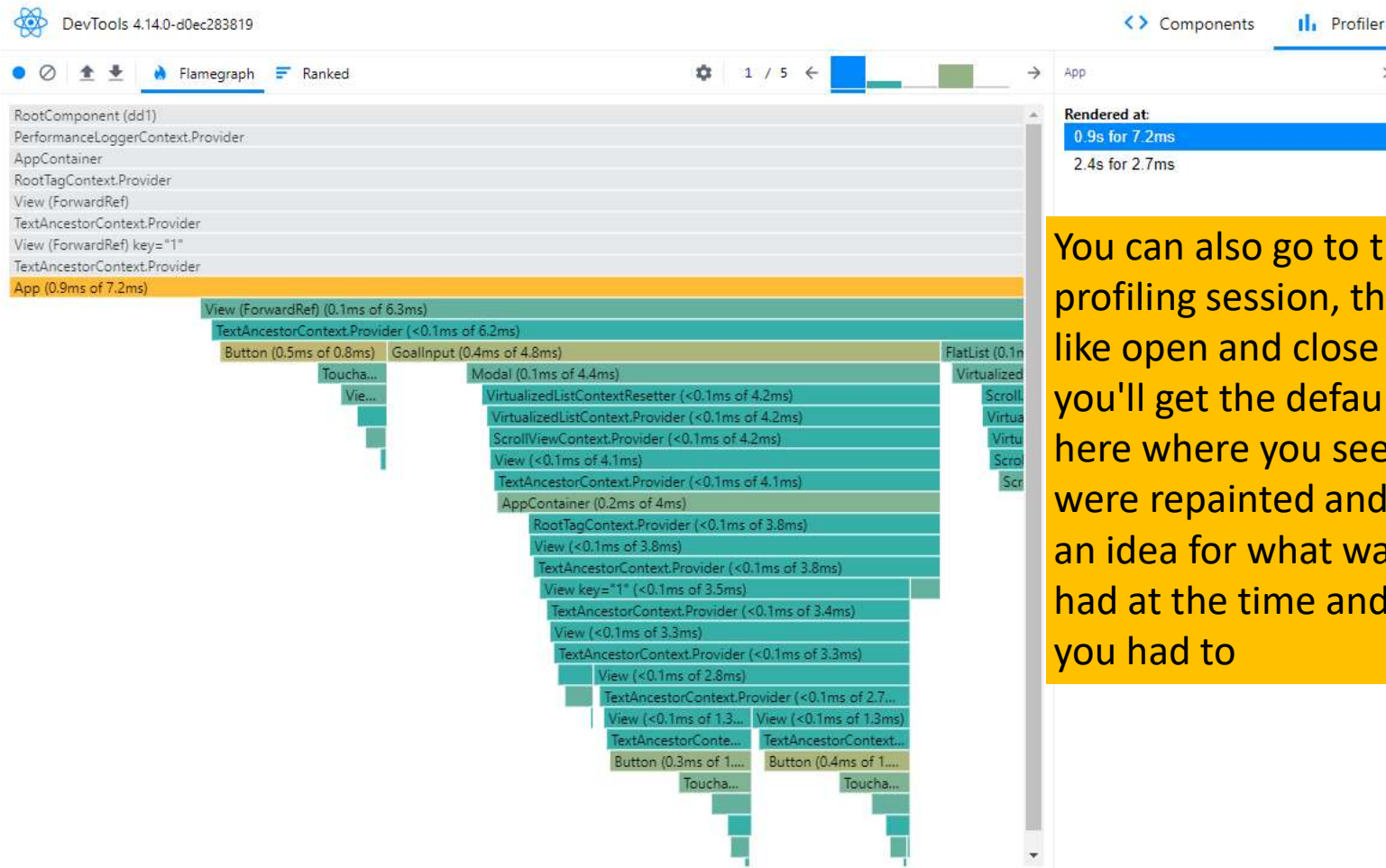
ADD NEW GOAL

1234

ADD NEW GOAL

1234

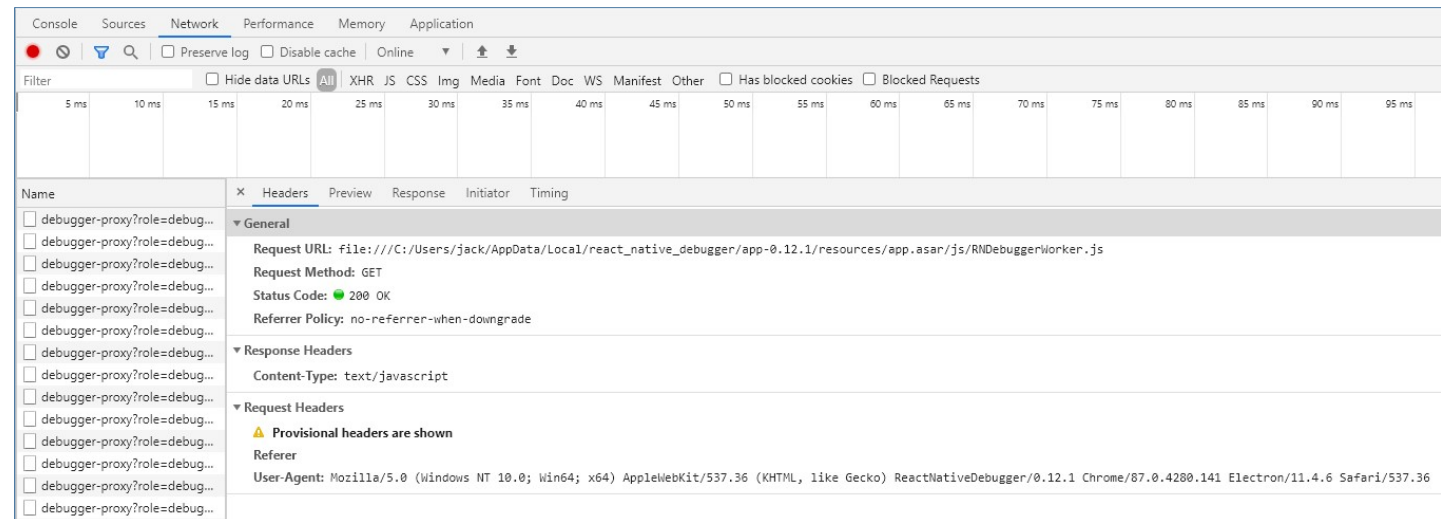
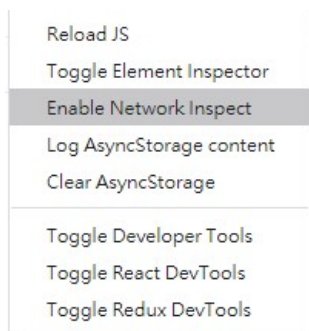
React Native Debugger: Profiler



You can also go to the profiler tab here and start a profiling session, then do something on the screen like open and close the modal and stop this and you'll get the default React dev tools experience here where you see which of your components were repainted and you can dive into that to get an idea for what was repainted, which props it had at the time and how many re-render cycles you had to

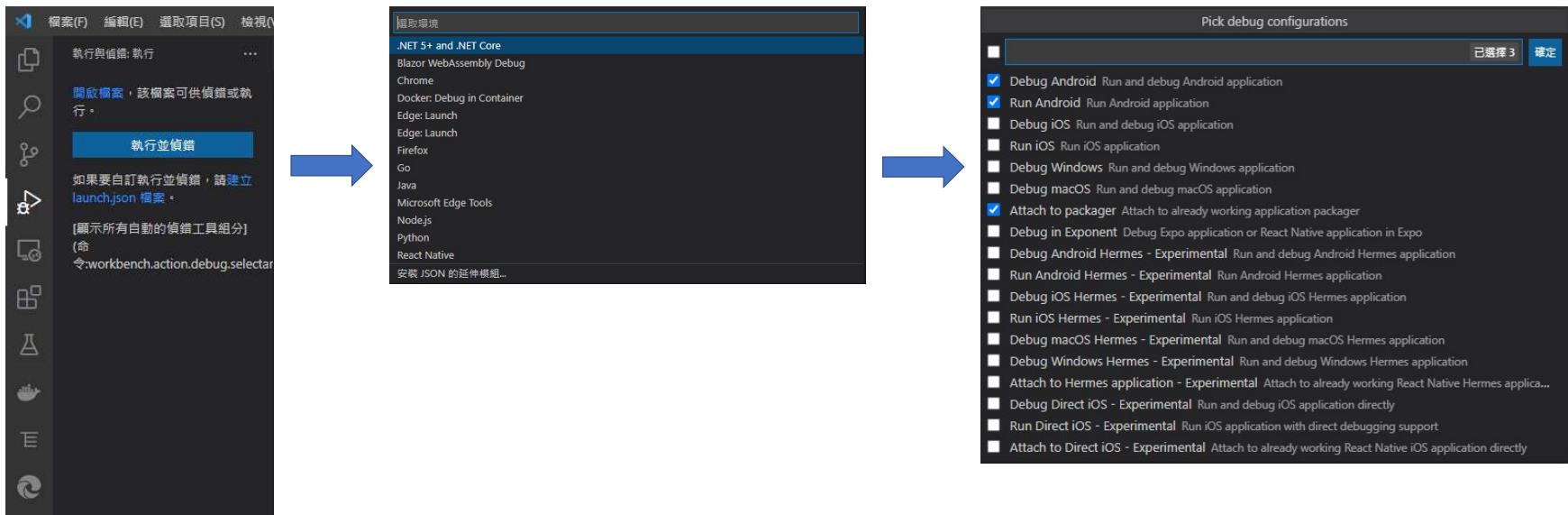
React Native Debugger: Network

- You can right click anywhere here, , then you can go to the network tab and you'll see outgoing network requests.
- Now these are all just debugging related requests but later in the course when we'll add our own network requests, where we send requests to our own web server, we can even inspect those here and look into them and see if we're sending and receiving the right data, something which is otherwise pretty hard to do with.

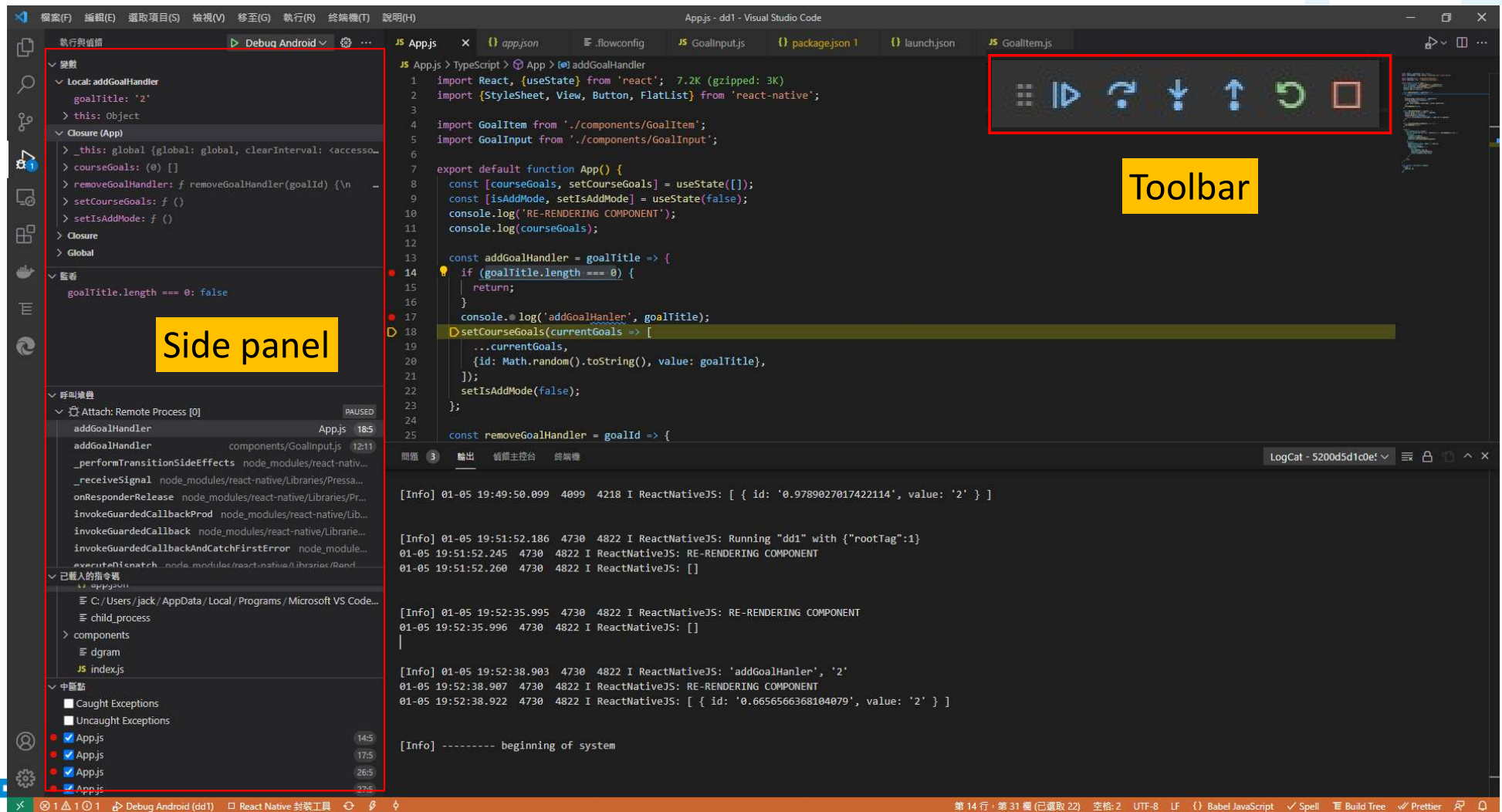


Debugging React Native in VS Code: Prerequisites

- VS Code installed
- React Native environment set up: <https://reactnative.dev/docs/environment-setup>
- Install Microsoft's React Native Tools extension for VS Code
- Open launch.json from the .vscode directory and add debug configurations



Debugging React Native in VS Code: Enjoy Debugging!



Top 8 Tools for Debugging React Native Applications



Wrap Up

- So now we had a detailed look at the React Native debugging experience, at the React Native debugger and the different tools you have for finding and fixing errors, for analyzing your user interface, your component tree
- React Native debugger really is a cool tool for looking into your app, for setting breakpoints, viewing the console, viewing your component tree, viewing the styles you're using there and so much more. It really allows you to dive deeply into your application code, into your UI, into your logic and find out if everything is working the way it should work and you can even go in here and change certain things like the styling as you saw, to experiment with different settings and find out where you need to tweak your app for it to work correctly.
- Just simply debugging React Native in VS Code

Maybe the Best Debugging Tool For You Is Yet to Come



Useful Resources & Links

- Debugging
<https://reactnative.dev/docs/debugging>
- Chrome Dev Tools
<https://developers.google.com/web/tools/chrome-devtools/>
- React Native Debugger
<https://github.com/jhen0409/react-native-debugger>