React Native Integration Test Guide

- What’s an integration test?
- Anatomy of an integration test
- Setup
- Rendering
  - Using the Scoped Component approach
  - Using the Entire Editor approach
- Query elements
  - Use of find queries
  - within queries
- Fire events
- Expect correct element behaviour
- Cleanup
- Helpers
- Common flows
  - Query a block
  - Add a block
  - Open block settings
    - Using the Scoped Component approach
  - FlatList items
  - Sliders
  - Selecting an inner block
- Tools
  - Using the Accessibility Inspector
- Common pitfalls and caveats
  - False positives when omitting await before waitFor function
  - waitFor timeout

- Replace current UI unit tests
- Platform selection

What’s an integration test?

Integration testing is defined as a type of testing where different parts are tested as a group. In our case, the parts that we want to test are the different components that are required to be rendered for a specific block or editor logic. In the end, they are very similar to unit tests as they are run with the same command using the Jest library. The main difference is that for the integration tests, we’re going to use a specific library `react-native-testing-library` for testing how the editor renders the different components.

Anatomy of an integration test

A test can be structured with the following parts:

- Setup
- Rendering
- Query elements
- Fire events
- Expect correct element behaviour
- Cleanup

We also include examples of common tasks as well as tips in the following sections:

- Helpers
- Common flows
- Tools
- Common pitfalls and caveats

Setup

This part usually is covered by using the Jest callbacks `beforeAll` and `beforeEach`, the purpose is to prepare everything that the test might require like registering blocks or mocking parts of the logic.

Here is an example of a common pattern if we expect all core blocks to be available:

```
beforeAll( () => {
    // Register all core blocks
    registerCoreBlocks();
} );
```

Rendering

Before introducing the testing logic, we have to render the components that we want to test on. Depending on if we want to use the scoped component or entire editor approach, this part will be different.

Using the Scoped Component approach

Here is an example of rendering the Cover block (extracted from this code):

```
// This import points to the index file of the block
import { metadata, settings, name } from '../index';

...

const setAttributes = jest.fn();
const attributes = {
    backgroundType: IMAGE_BACKGROUND_TYPE,
    focalPoint: { x: '0.25', y: '0.75' },
    hasParallax: false,
    overlayColor: { color: '#000000' },
    url: 'mock-url',
};

...

// Simplified tree to render Cover edit within slot
const CoverEdit = ( props ) => (
    <SlotFillProvider>
        <BlockEdit isSelected name={ name } clientId={ 0 } { ...props } />
```

```
        <BottomSheetSettings isVisible />
    </SlotFillProvider>
);

const { getByText, findByText } = render(
    <CoverEdit
        attributes={ {
            ...attributes,
            url: undefined,
            backgroundType: undefined,
        } }
        setAttributes={ setAttributes }
    />
);
```

Using the Entire Editor approach

Here is an example of rendering the Buttons block (extracted from this code):

```
const initialHtml = `<!-- wp:buttons -->
<div class="wp-block-buttons"><!-- wp:button {"style":{"border":{"radius":"5px"}}} -->
<div class="wp-block-button"><a class="wp-block-button__link" style="border-radius:5px"
>Hello</a></div>
<!-- /wp:button --></div>
<!-- /wp:buttons -->`;
const { getByLabelText } = initializeEditor( {
    initialHtml,
} );
```

Query elements

Once the components are rendered, it’s time to query them. An important note about this topic is that we should test from the user’s perspective, this means that ideally we should query by elements that the user has access to like texts or accessibility labels.

When querying we should follow this priority order:

1. `getByText`: querying by text is the closest flow we can do from the user's perspective, as text is the visual clue for them to identify elements.
2. `getByLabelText`: in some cases, we want to query elements that don't provide text so in this case we can fallback to the accessibility label.
3. `getByTestId`: if none of the previous options fit and/or we don't have any visual element that we can rely upon, we have to fallback to a specific test id, which can be defined using the `testID` attribute (see here for an example).

Here are some examples:

```
const mediaLibraryButton = getByText( 'WordPress Media Library' );
```

```
const missingBlock = getByLabelText( /Unsupported Block\. Row 1/ );
```

```
const radiusSlider = getByTestId( 'Slider Border Radius' );
```

Note that either a plain string or a regular expression can be passed into these queries. A regular expression is best for querying part of a string (e.g. any element whose accessibility label contains `Unsupported Block. Row 1`). Note that special characters such as `.` need to be escaped.

Use of `find` queries

After rendering the components or firing an event, side effects might happen due to potential state updates so the element we're looking for might not be yet rendered. In this case, we would need to wait for the element to be available and for this purpose, we can use the `find*` versions of query functions, which internally use `waitFor` and periodically check whether the element appeared or not.

Here are some examples:

```
const mediaLibraryButton = await findByText( 'WordPress Media Library' );
```

```
const missingBlock = await findByLabelText( /Unsupported Block\. Row 1/ );
```

```
const radiusSlider = await findByTestId( 'Slider Border Radius' );
```

In most cases we'll use the `find*` functions, but it's important to note that it should be restricted to those queries that actually require waiting for the element to be available.

`within` queries

It’s also possible to query elements contained in other elements via the `within` function, here is an example:

```
const missingBlock = await findByLabelText( /Unsupported Block\. Row 1/ );
const translatedTableTitle = within( missingBlock ).getByText( 'Tabla' );
```

Fire events

As important as querying an element is to trigger events to simulate the user interaction, for this purpose we can use the `fireEvent` function (documentation).

Here is an example of a press event:

Press event:

```
fireEvent.press( settingsButton );
```

We can also trigger any type of event, including custom events, in the following example you can see how we trigger the `onValueChange` event (code reference) for the Slider component:

Custom event – onValueChange:

```
fireEvent( heightSlider, 'valueChange', '50' );
```

Expect correct element behaviour

After querying elements and firing events, we must verify that the logic works as expected. For this purpose we can use the same `expect` function from Jest that we use in unit tests. It is recommended to use the custom `toBeVisible` matcher to ensure the element is defined, is a valid React element, and visible.

Here is an example:

```
const translatedTableTitle = within( missingBlock ).getByText( 'Tabla' );
expect( translatedTableTitle ).toBeVisible();
```

Additionally when rendering the entire editor, we can also verify if the HTML output is what we expect:

```
expect( getEditorHtml() ).toBe(
    '<!-- wp:spacer {"height":50} -->\n<div style="height:50px" aria-hidden="true" class="wp-block-spacer"></div>\n<!-- /wp:spacer -->'
);
```

Cleanup

And finally, we have to clean up any potential modifications we've made that could affect the following tests. Here is an example of a typical cleanup after registering blocks that implies unregistering all blocks:

```
afterAll( () => {
    // Clean up registered blocks
    getBlockTypes().forEach( ( block ) => {
        unregisterBlockType( block.name );
    } );
} );
```

Helpers

In the spirit of making easier writing integration tests for the native version, you can find a list of helper functions in this README.

Common flows

Query a block

A common way to query a block is by its accessibility label, here is an example:

```
const spacerBlock = await waitFor( () =>
    getByLabelText( /Spacer Block\. Row 1/ )
);
```

For further information about the accessibility label of a block, you can check the code of the function `getAccessibleBlockLabel`.

Add a block

Here is an example of how to insert a Paragraph block:

```
// Open the inserter menu
fireEvent.press( await findByLabelText( 'Add block' ) );

const blockList = getByTestId( 'InserterUI-Blocks' );
// onScroll event used to force the FlatList to render all items
fireEvent.scroll( blockList, {
    nativeEvent: {
        contentOffset: { y: 0, x: 0 },
        contentSize: { width: 100, height: 100 },
        layoutMeasurement: { width: 100, height: 100 },
    },
} );

// Insert a Paragraph block
fireEvent.press( await findByText( `Paragraph` ) );
```

Open block settings

The block settings can be accessed by tapping the “Open Settings” button after selecting the block, here is an example:

```
fireEvent.press( block );

const settingsButton = await findByLabelText( 'Open Settings' );
fireEvent.press( settingsButton );
```

Using the Scoped Component approach

When using the scoped component approach, we need first to render the `SlotFillProvider` and the `BottomSheetSettings` (note that we’re passing the `isVisible` prop to force the bottom sheet to be displayed) along with the block:

```
<SlotFillProvider>
    <BlockEdit isSelected name={ name } clientId={ 0 } { ...props } />
    <BottomSheetSettings isVisible />
</SlotFillProvider>
```

See examples:

- Cover block

FlatList items

The `FlatList` component renders its items depending on the scroll position, the view and content size. This means that when rendering this component it might happen that some of the items can't be queried because they haven't been rendered yet. To address this issue we have to explicitly fire an event to make the `FlatList` render all the items.

Here is an example of the FlatList used for rendering the block list in the inserter menu:

```
const blockList = getByTestId( 'InserterUI-Blocks' );
// onScroll event used to force the FlatList to render all items
fireEvent.scroll( blockList, {
    nativeEvent: {
        contentOffset: { y: 0, x: 0 },
        contentSize: { width: 100, height: 100 },
        layoutMeasurement: { width: 100, height: 100 },
    },
} );
```

Sliders

Sliders found in bottom sheets should be queried using their `testID`:

```
const radiusSlider = await findByTestId( 'Slider Border Radius' );
fireEvent( radiusSlider, 'valueChange', '30' );
```

Note that a slider's `testID` is "Slider " + label. So for a slider with a label of "Border Radius", `testID` is "Slider Border Radius".

Selecting an inner block

One caveat when adding blocks is that if they contain inner blocks, these inner blocks are not rendered. The following example shows how we can make a Buttons block render its inner Button blocks (assumes we've already obtained a reference to the Buttons block as `buttonsBlock`):

```
const innerBlockListWrapper = await within( buttonsBlock ).findByTestId(
    'block-list-wrapper'
);
fireEvent( innerBlockListWrapper, 'layout', {
    nativeEvent: {
        layout: {
            width: 100,
        },
    },
} );

const buttonInnerBlock = await within( buttonsBlock ).findByLabelText(
    /Button Block\. Row 1/
);
fireEvent.press( buttonInnerBlock );
```

Tools

Using the Accessibility Inspector

If you have trouble locating an element's identifier, you may wish to use Xcode's Accessibility Inspector. Most identifiers are cross-platform, so even though the tests are run on Android by default, the Accessibility Inspector can be used to find the right identifier.

Screenshot of the Xcode Accessibility Inspector app. The screenshot shows how to choose the corre

Common pitfalls and caveats

False positives when omitting `await` before `waitFor` function

Omitting the `await` before a `waitFor` can lead to scenarios where tests pass but are not testing the intended behaviour. For example, if `toBeDefined` is used to assert the result of a call to `waitFor`, the assertion will pass because `waitFor` returns a value, even though it is not the `ReactTestInstance` we meant to check for. For this reason, it is recommended to use the custom matcher `toBeVisible` which guards against this type of false positive.

`waitFor` timeout

The default timeout for the `waitFor` function is set to 1000 ms, so far this value is enough for all the render logic we're testing, however, if while testing we notice that an element requires more time to be rendered we should increase it.

Replace current UI unit tests

Some components already have unit tests that cover component rendering, although it's not mandatory, in these cases, it would be nice to analyze the potential migration to an integration test.

In case we want to keep both, we'll add the word "integration" to the integration test file to avoid naming conflicts, here is an example: packages/block-library/src/missing/test/edit-integration.native.js.

Platform selection

By default, all tests run in Jest use the Android platform, so in case we need to test a specific behaviour related to a different platform, we would need to support platform test files.

In case we only need to test logic controlled by the Platform object, we can mock the module with the following code (in this case it's changing the platform to iOS):

```
jest.mock( 'Platform', () => {
    const Platform = jest.requireActual( 'Platform' );
    Platform.OS = 'ios';
    Platform.select = jest.fn().mockImplementation( ( select ) => {
        const value = select[ Platform.OS ];
        return ! value ? select.default : value;
    } );
    return Platform;
} );
```

□□□ #2
□ □□□ □□□ 2026-10-31 18:27:01 UTC

□ □□□ □□□ 2026-10-31 18:27:03 UTC