Mattermost

Level up your React Hooks

Introduction

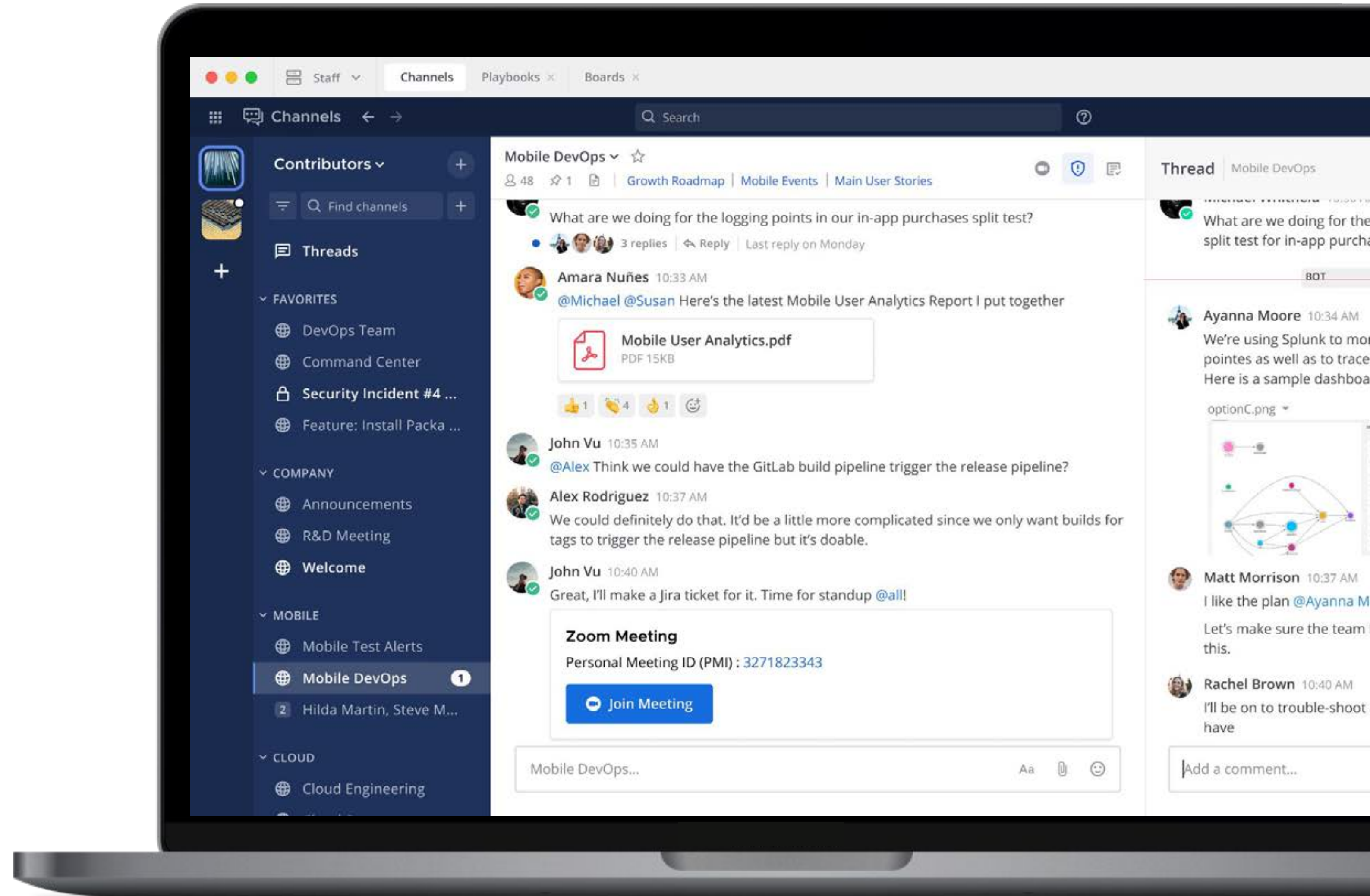
Daniel Espino García

Software Design Engineer at Mattermost

@daniel.espino-garcia at community.mattermost.com

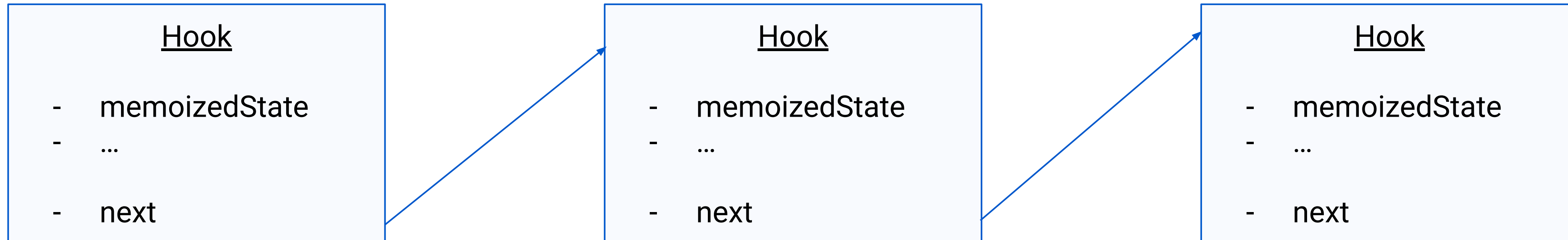
What is Mattermost?

- Collaboration tool.
- Open source alternative to Slack and Microsoft Teams
- Mobile and web: React
- Mobile: functional components
- Webapp: Migrating to functional components
- Open to code contributions



What are hooks?

What are hooks… in the inside?




```
function mountCallback<T>(callback: T, deps:
Array<mixed> | void | null): T {
  const hook = mountWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  hook.memoizedState = [callback, nextDeps];
  return callback;
}
```

```
function updateCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (nextDeps !== null) {
    const prevDeps: Array<mixed> | null = prevState[1];
    if (areHookInputsEqual(nextDeps, prevDeps)) {
      return prevState[0];
    }
  }
  hook.memoizedState = [callback, nextDeps];
  return callback;
}
```

/packages/react-reconciler/src/ReactFiberHooks.js

What are hooks… in the inside?

```
function mountCallback<T>(callback: T, deps:
Array<mixed> | void | null): T {
 const hook = mountWorkInProgressHook();
 const nextDeps = deps === undefined ? null : deps;
 hook.memoizedState = [callback, nextDeps];
 return callback;
}
```

Memory overhead

GC Stress

```
function updateCallback<T>(callback: T, deps: Array<mixed> | void | null): T {
 const hook = updateWorkInProgressHook();
 const nextDeps = deps === undefined ? null : deps;
 const prevState = hook.memoizedState;
 if (nextDeps !== null) {
   const prevDeps: Array<mixed> | null = prevState[1];
   if (areHookInputsEqual(nextDeps, prevDeps)) {
      return prevState[0];
   }
 }
 hook.memoizedState = [callback, nextDeps];
 return callback;
}
```

Computation overhead

/packages/react-reconciler/src/ReactFiberHooks.js

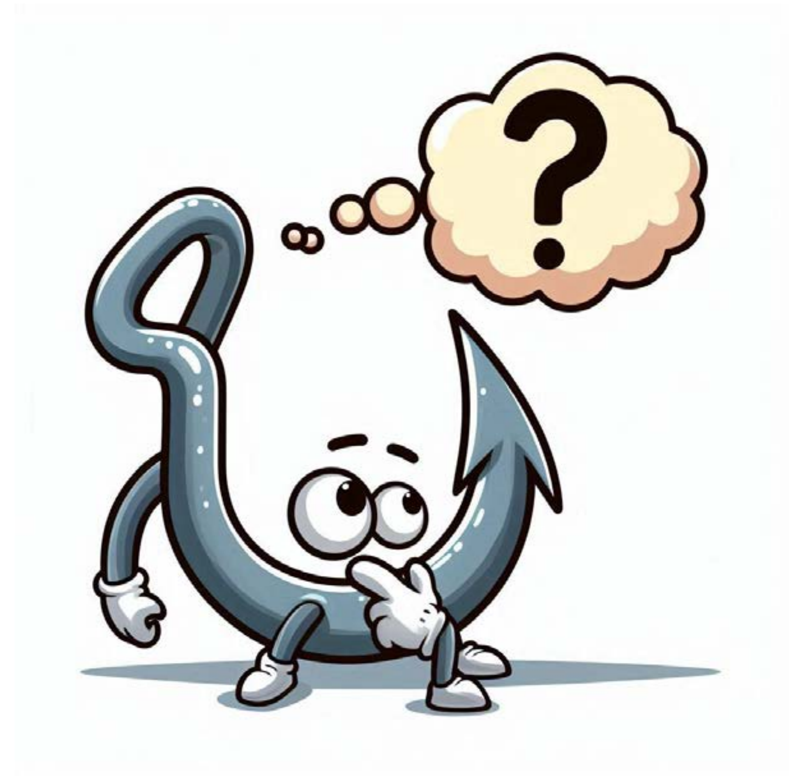
Basis

Make things more readable

- Code is written once, read many times
- And used without reading even more!

Over-optimization is not good

- Amdahl's Law
- Moore's Law



Dependencies

Extract constants from components

```
function Component() {
    const foo = 'foo';
    ...
}
```

VS.

```
const foo = 'foo'
function Component() {
    ...
}
```

- Less dependencies
- Less stress for the GC
- Easier to understand that is always the same value

Extract constants from components

```
import {foo} from 'foo';

function Component() {
   const myCallback = (id) => {
       foo(id);
   }
   ...
}
```

VS.

```
function Component() {
  const myCallback = useCallback((id) => {
      foo(id);
  }, []);
  ...
}
```

```
function Component() {
   ...
}
```

- Less GC stress
- More readable
- Less dependencies
- Harder to add unexpected dependencies

Only direct dependencies are dependencies

```
const blur = useCallback(() => {
 input.current?.blur();
}, []);
const handleAndroidKeyboard = () => {
 blur();
};
useEffect(() => {
 ...
 handleAndroidKeyboard();
 ...
}, [blur]);
```

VS.

```
const blur = useCallback(() => {
 input.current?.blur();
}, []);
const handleAndroidKeyboard = useCallback(() => {
 blur();
}, [blur]);
useEffect(() => {
 ...
 handleAndroidKeyboard();
 ...
}, [handleAndroidKeyboard]);
```

- Clearer dependencies
- Easier to read
- Future proof!

Persisting values

Intermediate variables

```
function Component({isArchived, hasPermissions}) {
 const callback = useCallback(() => {
     if (!isArchived && hasPermissions) {
         ...
     }
 }, [isArchived, hasPermissions])
 ...
}
```

VS.

```
function Component({isArchived, hasPermissions}) {
 const canPost = !isArchived && hasPermission;
 const callback = useCallback(() => {
     if (canPost) {
         ...
     }
 }, [canPost])
 ...
}
```

- Clearer dependencies
- Easier to read
- Minor performance improvement!

useState: initialize with function

```
function Dropdown(options, defaultValue) {
    const currentOption = useState(getOptionFromValue(options,
defaultValue));
    ...
}
```

vs.

```
function Dropdown(options, defaultValue) {
    const currentOption = useState(() =>
getOptionFromValue(options, defaultValue));
    ...
}
```

- Performance optimization

useState: Don't

```
function InfiniteScroll() {
 const [page, setPage] = useState(0);
 const onEndReached = useCallback(async () => {
   await fetchPage(page);
   setPage(page + 1);
 }, [page]);
 ...
}
```

VS.

```
function InfiniteScroll() {
 const page = useRef(0);
 const onEndReached = useCallback(async () => {
   await fetchPage(page.current);
   page.current += + 1;
 }, []);
 ...
}
```

- Less dependencies
- Minor performance improvement!

useState: function argument

```
function InfiniteScroll() {
 const page = useRef(0);
 const [elements, setElements] = useState([]);
 const onEndReached = useCallback(async () => {
   const newElements = await fetchPage(page.current);
   setElements([...elements, ...newElements]);
   page.current += + 1;
 }, [elements]);
 ...
}
```

VS.

```
function InfiniteScroll() {
 const page = useRef(0);
 const [elements, setElements] = useState([]);
 const onEndReached = useCallback(async () => {
   const newElements = await fetchPage(page.current);
   setElements((prev) => [...prev, ...newElements]);
   page.current += + 1;
 }, []);
 ...
}
```

- Less dependencies
- Less re-renders

useState: effect smell

```
function Component({width, height}) {
 const [aspectRatio, setAspectRatio] = useState(width / height);
 useEffect(() => {
   setAspectRatio(width / height);
 }, [width, height])
 ...
}
```

VS.

```
function Component({width, height}) {
 const aspectRatio = width / height;
 ...
}
```

- Less re-renders
- Less unneeded boilerplate

useState summary

- Values that affect the render
- Should persist between renders regardless of the props / other states

useRef summary

- Component references
- Values that do not affect the render
- Should persist between renders regardless of the props / other states

Intermediate state summary

- Can be computed from the props / states

Memoization

What to memo?

```
const myString = useMemo(() => 'hello ' + props.name,
[]);

const someProps = useMemo(() => ({prop1: 1, prop2: 2,
prop3: props.other}, [props.other]));
return <MyComponent {...someProps}>;

const getComplexString = useCallback(() => {
   ...
   return complexString
}, [...]);
return <MyComponent complexString={getComplexString()}>;
```

```
const myObjectProp = useMemo(
   () => ({name: props.name, surname: props.surname}),
   [props.name, props.surname],
);

const myListProp = useMemo(() => [prop1, prop2],
[prop1, prop2]);

const myComplexString = useMemo(() => {
   ...
   return complexString;
}, [...]);
```

useMemo summary

- Reference types (lists and objects) used as props
- Reference types used as dependencies
- Heavy calculations

useCallback summary

- Functions used as props (not called)
- Functions used as dependencies

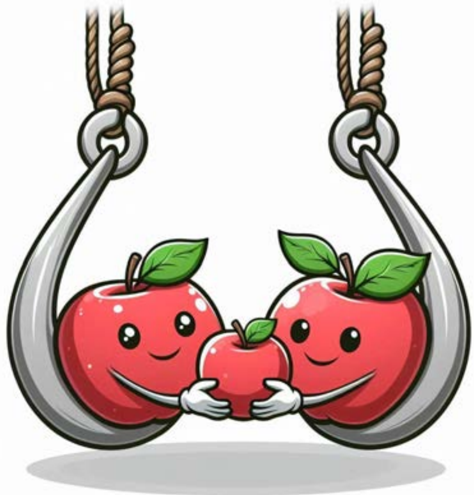
Custom hooks

Custom hooks

```
useEffect(() => {
  const backListener =
Navigation.events().registerNavigationButtonPressedListener(({bu
ttonId}) => {
    if (buttonId === BACK_BUTTON) {
      callback();
    }
  });
  return () => backListener.remove();
}, [callback]);
```

VS.

```
useBackNavigation(onPressClose);
```



- Less code repetition
- Easier to maintain
- Easier to read

Custom hooks: be stable!

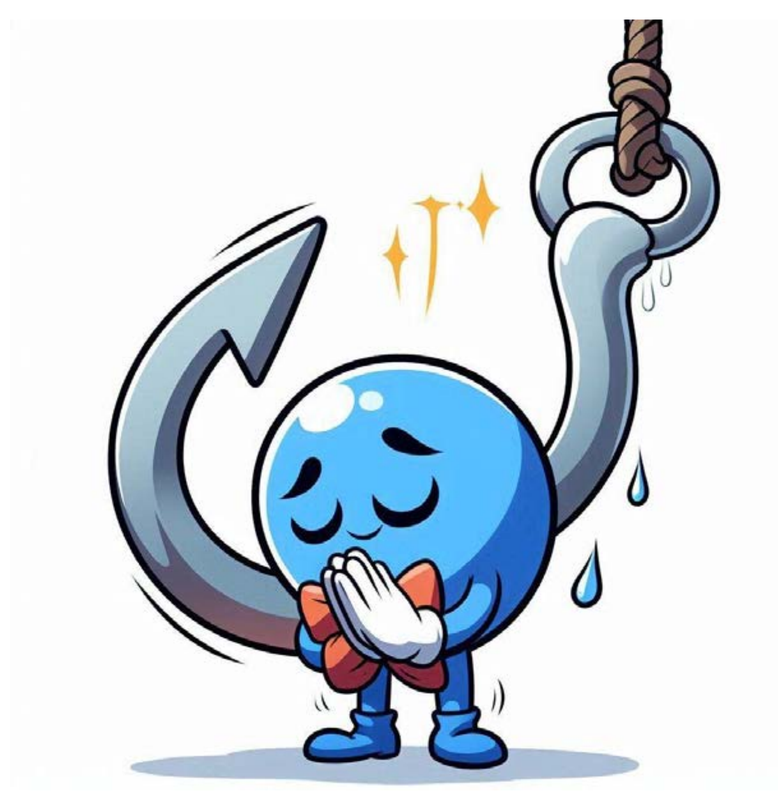
```
function useWonderfulFunction() {
    return () => 'This is wonderful!'
}
```

VS.

```
function useWonderfulFunction() {
    return useCallback(() => 'This is wonderful!', []);
}
```

- Future proof
- More intuitive

Closing



Mattermost community server:

THANK YOU!

Slides: