NILS HARTMANN
https://nilshartmann.net

One Year

React Hooks

A (Critical) Review

Slides: https://nils.buzz/react-meetup-hooks

NILS HARTMANN

nils@nilshartmann.net

Developer, Architect, Trainer from Hamburg (Freelancer)

JavaScript, TypeScript
React
GraphQL
Java

Trainings, Workshops and Coachings

HTTPS://NILSHARTMANN.NET

Anyone NOT knowing what Hooks are?

REACT HOOKS

Anyone NOT knowing what Hooks are?

Hooks 2 Minute intro

- add State, Lifecycle, Sideffects in functional components (almost no need for class components anylonger)

REACT HOOKS

Anyone NOT knowing what Hooks are?

Hooks 2 Minute intro

- add State, Lifecycle, Sideffects in functional components (almost no need for class components anylonger)

- "Hooks into your components lifecycle"

REACT HOOKS

Anyone NOT knowing what Hooks are?

Hooks 2 Minute intro

- add State, Lifecycle, Sideffects in functional components (almost no need for class components anylonger)

- "Hooks into your components lifecycle"

- Regular JavaScript functions...
  - ...but must start with 'use'
  - ...but must not be used in conditionals, for/loops, Class components
  - ...but behaviour is tied to React

Hooks example

```
import React, { useState } from "react";

export default function SettingsForm(props) {

  const [favColor, setFavColor] = useState("blue");

  return <•••>
    <input value={favColor}
           onChange={e => setFavColor(e.target.value) } />
  <•••>
}
```

- **useState** returns value and setter-function
- When state changes, component re-renders
  - component function will run again

REACT HOOKS

One Year of Hooks...

There are some built-in Hooks, like

- **useState**
- **useReducer** handle state in a Redux-like way but only for one component
- **useEffect** for sideeffects (replaces lifecycle methods in classes)
- **useContext** to receive a Context object
- **useCallback/useMemo/useRef**: solve problems that arise due to using... Hooks

    -

REACT HOOKS

One Year of Hooks...

Libraries ship with Hooks, like

- Redux (useSelector, useDispatch, useStore)
- Router (useHistory, useParams, useLocation)
- Apollo Client (useQuery, useMutation)
- React Intl (useIntl)
- React i18n (useTranslation)

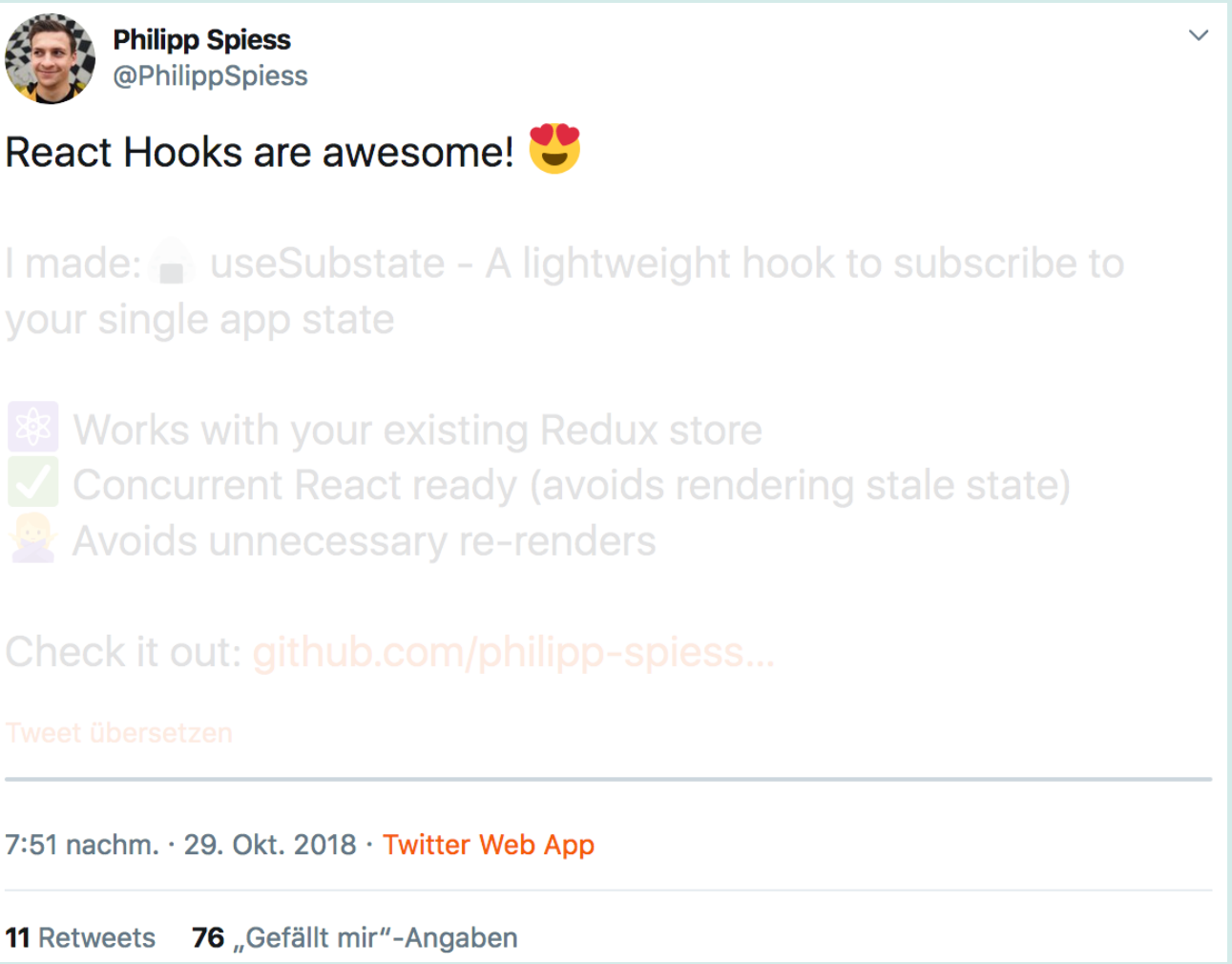
Community has them too,

- https://usehooks.com
- https://nikgraf.github.io/react-hooks/
- https://www.hooks.guide/

React Hooks

Good or Evil?

SOME VOICES...



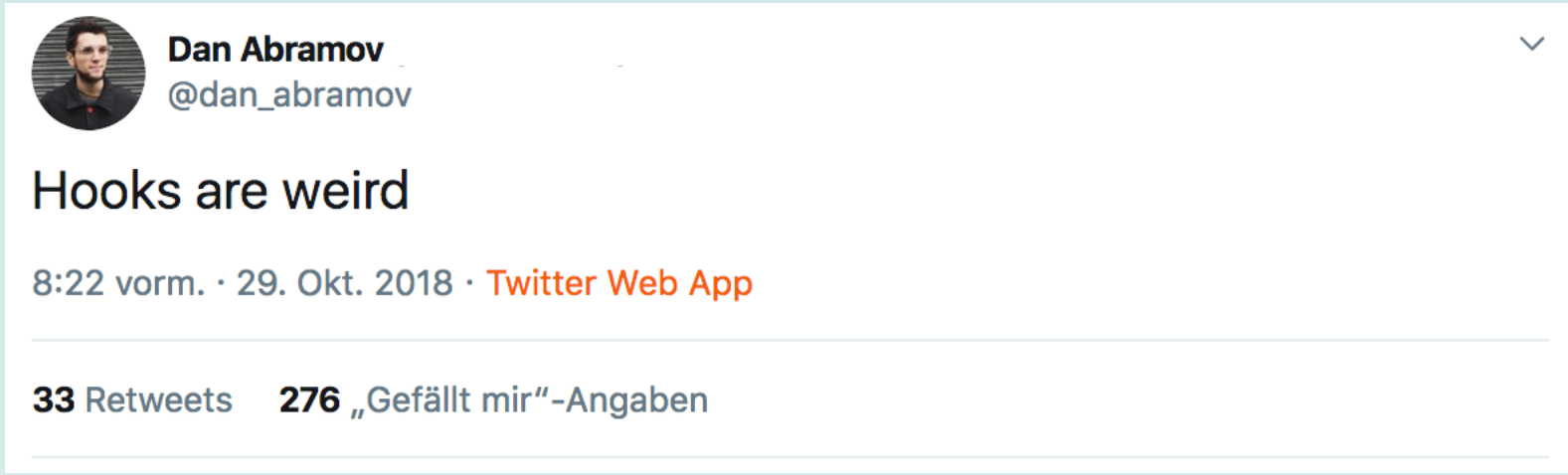
https://twitter.com/philippspiess/status/1056981916489015296

"With Hooks,

React loses its innocence

and becomes Angular"

Attendee of one of my workshops



https://twitter.com/dan_abramov/status/1056808552180793344

https://twitter.com/tomdale/status/1170095532066430977

Tom Dale
@tomdale

Old React was simple and fun and “just JavaScript.” New React is more powerful and more correct and better all around—at the expense of becoming a weird magical meta-language on top of JavaScript.

Tweet übersetzen

12:04 vorm. · 7. Sep. 2019 · Twitter Web App

4 Retweets **73** „Gefällt mir“-Angaben

https://twitter.com/tomdale/status/1170095532922064901

"awesome"

"weird"

"mixed feelings"

"angular"

WHY?

A LOOK AT THE API

A LOOK AT THE API

useContext to access React Context in your functional component

```
export default function SettingsForm(props) {
  const contextValue = React.useContext(ThemeContext);

  return <p>Your context color: {contextValue.color}</p>
}
```

A LOOK AT THE API

useContext to access React Context in your functional component

```
export default function SettingsForm(props) {
  const contextValue = React.useContext(ThemeContext);

  return <p>Your context color: {contextValue.color}</p>
}
```

- **Noteable:** This is probably easy to understand:
  I want to use context "ThemeContext" here in my component

A LOOK AT THE API

useContext to access React Context in your functional component

```
export default function SettingsForm(props) {
  const contextValue = React.useContext(ThemeContext);

  return <p>Your context color: {contextValue.color}</p>
}
```

- **Noteable:** This is probably easy to understand:
  I want to use context "ThemeContext" here in my component

- **But:** if context changes, SettingsForm **will automatically be re-executed**!
  Why? Because it's a ... Hook ("weird magical meta-language")
  "Something" happens ins the background to make that work
  There is no indicator that this will happen. Syntactically "only" JavaScript

A LOOK AT THE API

useState for local State in your functional component

```
export default function SettingsForm(props) {
  const [ favColor, setFavColor ] = useState("red");

  return <input value={favColor}
                onChange={e => setFavColor(e.target.value) />
}
```

A LOOK AT THE API

useState for local State in your functional component

```
export default function SettingsForm(props) {
  const [ favColor, setFavColor ] = useState("red");

  return <input value={favColor}
                onChange={e => setFavColor(e.target.value) />
}
```

- **Noteable:** Return Value
  - what is this? Tuple! (btw: I think Tuples will make it to JavaScript)
  - unsual (yet), but elegant, allows to name my variables as I want them to

A LOOK AT THE API

useState for local State in your functional component

```
export default function SettingsForm(props) {
  const [ favColor, setFavColor ] = useState("red");

  return <input value={favColor}
                onChange={e => setFavColor(e.target.value) />
}
```

- **Noteable:** Return Value
  - what is this? Tuple! (btw: I think Tuples will make it to JavaScript)
  - unsual (yet), but elegant, allows to name my variables as I want them to

- **Noteable:** initial value, **used only once** even if this function is run on each render
  Why? Because it's a ... Hook ("weird magical meta-language")

A LOOK AT THE API

useState for local State in your functional component

```
export default function SettingsForm(props) {
  const [ favColor, setFavColor ] = useState("red");

  return <input value={favColor}
                onChange={e => setFavColor(e.target.value) />
}
```

- **Noteable:** Return Value
  - what is this? Tuple! (btw: I think Tuples will make it to JavaScript)
  - unsual (yet), but elegant, allows to name my variables as I want them to

- **Noteable:** initial value, used only once even if this method is run on each render
  Why? Because it's a ... Hook ("weird magical meta-language")

- **Noteable:** setter-Function leads to re-render
  Why? Because it's a ... Hook ("weird magical meta-language")

A LOOK AT THE API

useState another try

```
export default function SettingsForm(props) {
  const [ easy, setEasy ] = useState(true);

  function handleClick() {
    setEasy(!easy);

    console.log(easy);
      // what is acutally easy here after first button click? 🤔
  }

  return <button onClick={handleClick}>Do it</button>
}
```

- **Noteable:** setting a state is still "defered" or "async"
  Might or might not be obvious

Credits: https://twitter.com/mjackson/status/1102430168839143424

A LOOK AT THE API

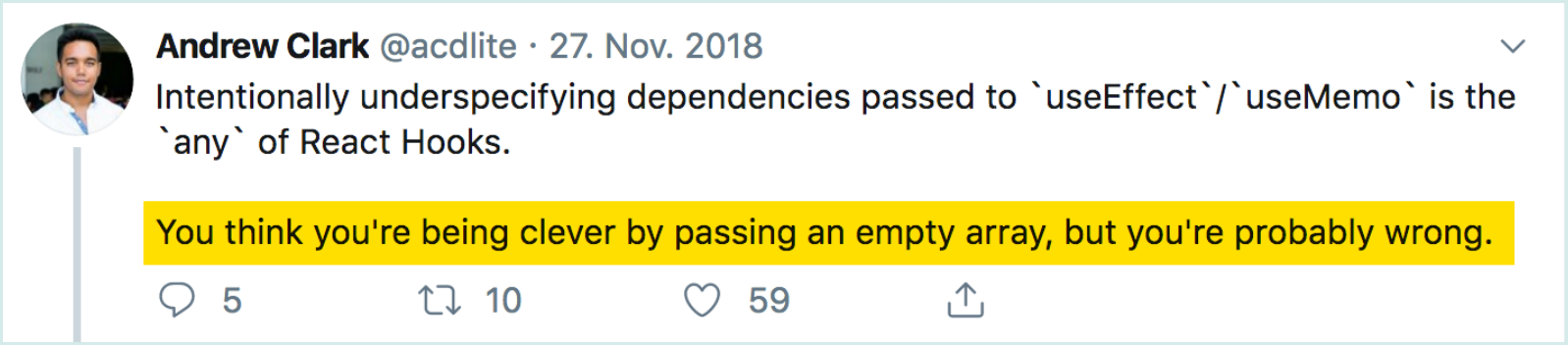
useEffect for sideeffects

```
export default function SettingsForm(props) {
  useEffect( () => { start(); return () => stop() }, [])

  return <...>
}
```

- **Noteable:** Everything
  - what is a sideeffect? API call, DOM API, console.log?
  - what is 2nd argument ("dependency array")
    - unsual way to "limit" call to functions
    - btw: what does [] semantically mean?
      Let's call the experts!

A LOOK AT THE API

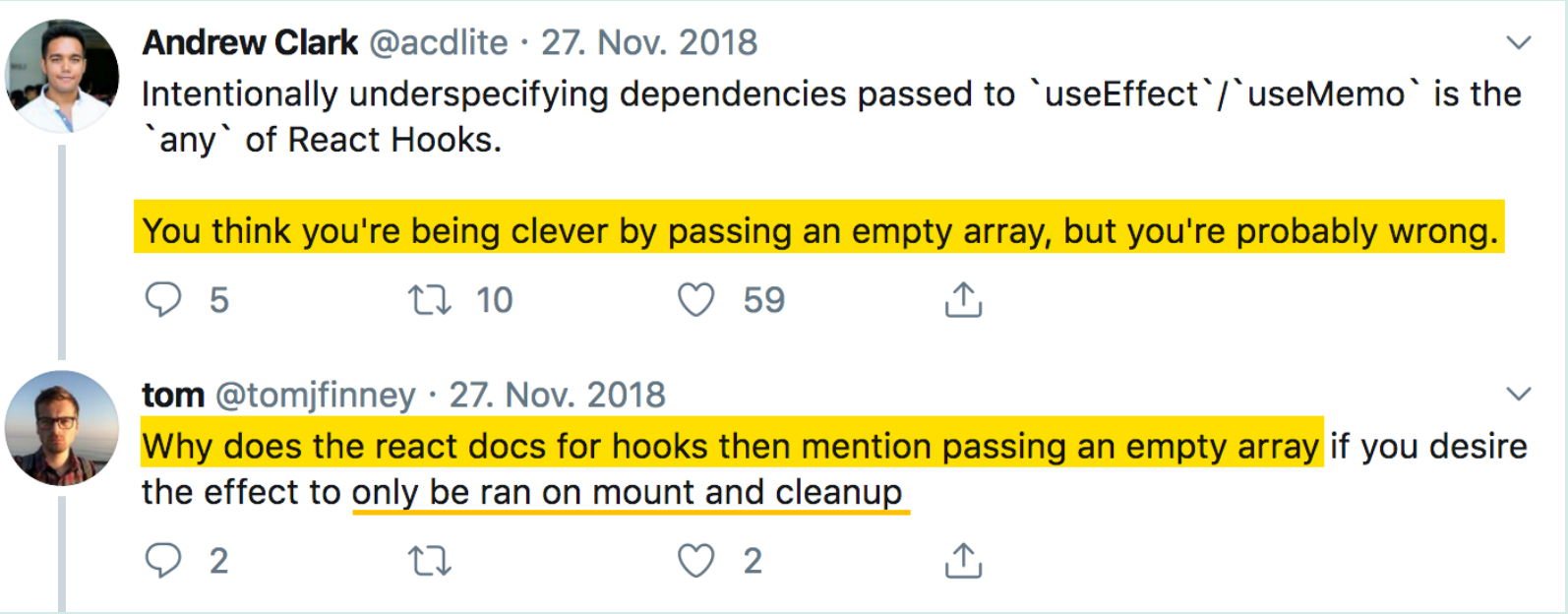
useEffect for sideeffects



https://twitter.com/acdlite/status/1067541310377123840

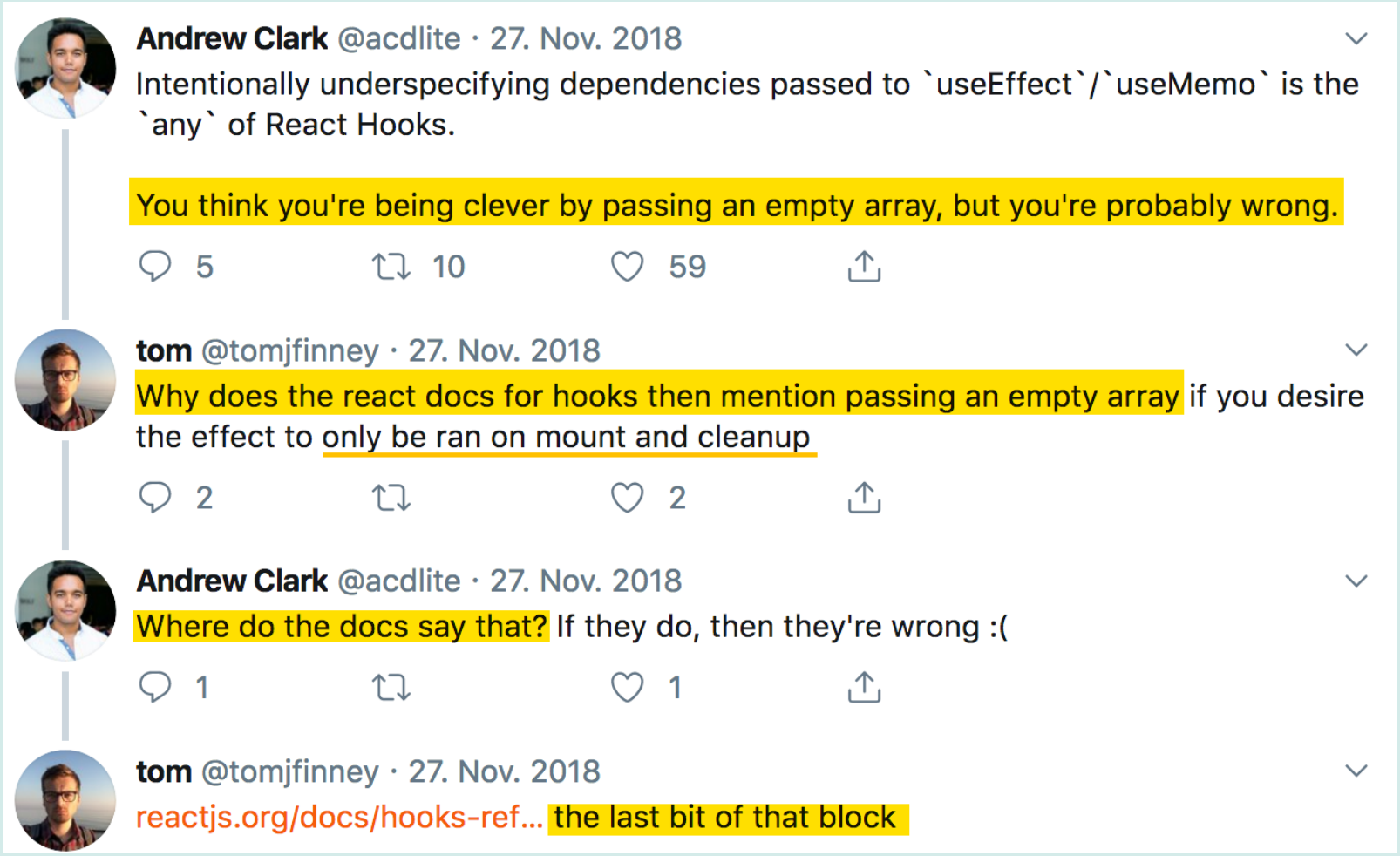
A LOOK AT THE API

useEffect for sideeffects



A LOOK AT THE API

useEffect for sideeffects



A LOOK AT THE API

useEffect for sideeffects

Dan Abramov
@dan_abramov

Antwort an @albertgao @mpocock1 und 2 weitere

Sorry, I don't think we have a broad agreement on the team about it either, and that's why our messaging is inconsistent. There is also some temptation to see Hooks+Suspense as one package, and ignore shortcomings of "fetch in effect" scenario that's common today. We'll fix.

Tweet übersetzen

11:55 nachm. · 28. Nov. 2018 · Twitter Web Client

2 „Gefällt mir"-Angaben

https://twitter.com/dan_abramov/status/1067914987753091074

A LOOK AT THE API

useEffect for sideeffects

Dan Abramov @dan_abramov · 29. Nov. 2018

Working on it.

> **Dan Abramov** @dan_abramov · 28. Nov. 2018
>
> Antwort an @albertgao @mpocock1 und 2 weitere
>
> Sorry, I don't think we have a broad agreement on the team about it either, and that's why our messaging is inconsistent. There is also some temptation to see Hooks+Suspense as one package, and ignore shortcomings of "fetch in effect" scenario that's common today. We'll fix.

1

https://twitter.com/dan_abramov/status/1067917403709943808?s=20

A LOOK AT THE API

useEffect for sideeffects

Dan Abramov @dan_abramov · 29. Nov. 2018

Working on it.

> **Dan Abramov** @dan_abramov · 28. Nov. 2018
>
> Antwort an @albertgao @mpocock1 und 2 weitere
>
> Sorry, I don't think we have a broad agreement on the team about it either, and that's why our messaging is inconsistent. There is also some temptation to see Hooks+Suspense as one package, and ignore shortcomings of "fetch in effect" scenario that's common today. We'll fix.

1

https://twitter.com/dan_abramov/status/1067917403709943808?s=20

- end of thread -

USING HOOKS

Using Hooks: this is simple...

```
import React, { useState } from "react";

export default function SettingsForm(props) {
  const [ favColor, setFavColor ] = useState("blue");
  return <input value={favColor} onChange={...} />
}
```

Using Hooks: let's add context...

```
import React, { useState, useContext } from "react";

export default function SettingsForm(props) {

  const login = useContext(LoginContext);




  const [ favColor, setFavColor ] = useState("blue");
  return <input value={favColor} onChange={...} />
}
```

Using Hooks: and now... boom!

```
import React, { useState, useContext } from "react";

export default function SettingsForm(props) {

  const login = useContext(LoginContext);

  if (login.loggedIn) {
    return <Redirect to="/login" />
  }

  const [ favColor, setFavColor ] = useState("blue");
  return <input value={favColor} onChange={...} />
}
```

USING HOOKS

Using Hooks: and now... boom!

```
import React, { useState, useContext } from "react";

export default function SettingsForm(props) {

  const login = useContext(LoginContext);

  if (login.loggedIn) {
    return <Redirect to="/login" />
  }

  const [ favColor, setFavColor ] = useState("blue");
  return <input value={favColor} onChange={...} />
}
```

Why? Because it's a ... Hook ("weird magical meta-language")
Hooks must always be called in the same order

USING HOOKS

Linter: saves us!

- eslint-plugin-react-hooks is really good and helpful

USING HOOKS

...and another one: useHistory from React Router

```
import { useHistory } from "react-router-dom";

export default function SettingsForm(props) {

  function saveAndRedirect() {
    saveSettings().then(
     () => useHistory().push("/home")
    );
  }

  return <•••><button onClick={saveAndRedirect}>Save</button><•••>
}
```

USING HOOKS

...this works

```
import { useHistory } from "react-router-dom";

export default function SettingsForm(props) {
  const history = useHistory();
  function saveAndRedirect() {
    saveSettings().then(
     () => history.push("/home")
    );
  }

  return <•••><button onClick={saveAndRedirect}>Save</button><•••>
}
```

Might not be big difference, but...

USING HOOKS

Might not be a big difference, but...

- we may have lots of unneccessary useHistory calls (or other Hooks)
- you have to know where you can use Hooks
- forces you to structure your code in exactly this way
- it's not "standard javascript"

Do you remember why React doesn't add a template language?

- To allow us the use of our "favorite" language: JavaScript
  - no need to learn a new language...

Does that mean Hooks (or React) are evil?

- No, but... they have their "price" (as classes have)
- It's "rethinking" again

Consequences

OF USING HOOKS

CONSEQUENCES

Can Custom Hooks replace existing patterns?

- Custom Hooks are another way for reusable logic
- HOCs?
- Render Properties?

- Would be good imho
  - esp. HOCs hard to understand

- **But**...

CONSEQUENCES

Example: React Router **withRouter** HOC

```
import { withRouter } from "react-router-dom";

function SettingsForm( { history } ) {

  function saveAndRedirect() {
    saveSettings().then( () => history.push("/home") );
  }

  return <•••><button onClick={saveAndRedirect}>Save</button><•••>

}

export default withRouter(SettingsForm);
```

A LOOK AT THE API

Example: React Router **useHistory** Custom Hook

```
import { useHistory } from "react-router-dom";

export default function SettingsForm( props ) {
  const history = useHistory();
  function saveAndRedirect() {
    saveSettings().then( () => history.push("/home") );
  }

  return <•••><button onClick={saveAndRedirect}>Save</button><•••>

}
```

Difference:

- Only one component (vs SettingsForm and HOC'd SettingsForm)
- SettingsForm "knows" about Router, it's not "just a prop" anymore

A LOOK AT THE API

Another one: React Router (with render prop)

```
// App.js
<Route path="/settings/:id"
       render={({match}) => <SettingsForm settingsId={match.params.id} />
```

```
// SettingsForm.js
export default function SettingsForm( {settingsId} ) {

  // do something with settingsId
  return •••;
}
```

Noteable:

- SettingsForm does not know anything about Router
- Routing "Logic" (Params, Routes, ...) are at *one* place (good imho)

A LOOK AT THE API

Another one: React Router with new Route API and useParams

```
// App.js
<Route path="/settings/:id"><SettingsForm /></Route>
```

new Router
5.2 API
no render prop anymore!

A LOOK AT THE API

Another one: React Router with new Route API and useParams

```
// App.js
<Route path="/settings/:id"><SettingsForm /></Route>
```

new Router
5.2 API
no render-Prop anymore

```
// SettingsForm.js
import { useParams } from "react-router-dom";

export default function SettingsForm( ) {
  const { settingsId } = useParams();

  // do something with settingsId
  return •••;
}
```

Noteable:

- SettingsForm knows about Router API and Routing "Logic" (which Params)
- What about "Colocation"?

(credits: https://twitter.com/andrewgreenh/status/1177213442710745091?s=20)

A LOOK AT THE API

What about this one? (from: https://twitter.com/Wolverineks/status/1177818104048472065)

```
function RouterContext({ children }) {
  return children({
    history: useHistory(),
    params: useParams(),
    ...
  });
}
```

A LOOK AT THE API

What about this one? (from: https://twitter.com/Wolverineks/status/1177818104048472065)

```
function RouterContext({ children }) {
  return children({
    history: useHistory(),
    params: useParams(),
    ...
  });
}

<Route path="/settings/:id">
  <RouterContext>
    {({ params }) => <SettingsForm settingsId={params.id} />
  </RouterContext>
</Router>
```

A LOOK AT THE API

What about this one? (from: https://twitter.com/Wolverineks/status/1177818104048472065)

```
function RouterContext({ children }) {
  return children({
    history: useHistory(),
    params: useParams(),
    ...
  });
}

<Route path="/settings/:id">
  <RouterContext>
    {({ params }) => <SettingsForm settingsId={params.id} />
  </RouterContext>
</Router>
```

Noteable: welcome back, render properties! 🤯

CONSEQUENCES

What does that mean?

- Hooks provide new ways for structuring code, architecture etc
- But they come with a price
  - nothing comes for free, nothing special here

- We probably we will see new patterns, best-practices evolving
  - React development won't become boring

A LOOK AT THE API

Example: Redux useDispatch and useSelector instead of connect

```
import { useDispatch, useSelector } from "react-redux";

export default function SettingsForm(props) {
  const favColor = useSelector(state => state.theme.favColor);
  const dispatch = useDispatch();

  const setNewColor = (r,g,b) => dispatch(actions.setNewColor(r,g,b));

  return <•••><ColorPicker onSet={setNewColor}/><•••>
}
```

A LOOK AT THE API

Example: Redux useDispatch and useSelector instead of connect

```
import { useDispatch, useSelector } from "react-redux";

export default function SettingsForm(props) {
  const favColor = useSelector(state => state.theme.favColor);
  const dispatch = useDispatch();

  const setNewColor = (r,g,b) => dispatch(actions.setNewColor(r,g,b));

  return <•••><ColorPicker onSet={setNewColor}/><•••>
}
```

- Consequences:
  - We now have only one component, have seen that already
  - the component is bound to Redux, have seen that already

A LOOK AT THE API

Example: Redux useDispatch and useSelector instead of connect

```
import { useDispatch, useSelector } from "react-redux";

export default function SettingsForm(props) {
  const favColor = useSelector(state => state.theme.favColor);
  const dispatch = useDispatch();
                                        "forces" re-rendering  of the ColorPicker component
  const setNewColor = (r,g,b) => dispatch(actions.setNewColor(r,g,b));

  return <•••><ColorPicker onSet={setNewColor}/><•••>
}
```

- Consequences:
  - We now have only one component, have seen that already
  - the component is bound to Redux, have seen that already
  - It has different rendering behaviour (not related to redux)

A LOOK AT THE API

We can fix this:

```
import { useDispatch, useSelector } from "react-redux";

export default function SettingsForm(props) {
  const favColor = useSelector(state => state.theme.favColor);
  const dispatch = useDispatch();

  const setNewColor = React.useCallback(
    (r,g,b) => dispatch(actions.setNewColor(r,g,b)),
    [ dispatch ]
  );

  return <•••><ColorPicker onSet={setNewColor}/><•••>
}
```

A LOOK AT THE API

We can fix this:

```
import { useDispatch, useSelector } from "react-redux";

export default function SettingsForm(props) {
  const favColor = useSelector(state => state.theme.favColor);
  const dispatch = useDispatch();

  const setNewColor = React.useCallback(
    (r,g,b) => dispatch(actions.setNewColor(r,g,b)),
    [ dispatch ]
  );

  return <•••><ColorPicker onSet={setNewColor}/><•••>
}
```

remember the dependency array? 👋

- "Nice!" (Fortunately we only have *one* callback function here...)

A LOOK AT THE API

Again: is this really a problem?

- In most cases not as performance might be good enough to re-render all the time, so useCallback (and useMemo) is not a must

- I wonder how many CPU engery is wasted due to billions of unneccessary re-renderings in React Apps world wide 🤓

- But this is – esp. for Beginners – not easy to understand (call me a Beginner)

CONSEQUENCES

Let's have a look at the component lifecycle

- Want to start and stop a timeout (automatically after render)

```
export default function App() {
  const [running, setRunning] = useState(false);

  useEffect(() => {
    const id = setTimeout(() => {

      setRunning(false);
    }, 2000);

    setRunning(true);
    return () => clearTimeout(id);
  }, []);

  return <>Running: {running.toString()}</>;
}
```

CONSEQUENCES

Let's have a look at the component lifecycle

- Want to start and stop a timeout (automatically after render)

```
export default function App() {
  const [running, setRunning] = useState(false);

  useEffect(() => {
    const id = setTimeout(() => {
      console.log(running);
      setRunning(false);
    }, 2000);

    setRunning(true);
    return () => clearTimeout(id);
  }, []);

  return <>Running: {running.toString()}</>;
}
```

1. what will be dumped to console?

CONSEQUENCES

Let's have a look at the component lifecycle

- Want to start and stop a timeout (automatically after render)

```
export default function App() {
  const [running, setRunning] = useState(false);

  useEffect(() => {
    const id = setTimeout(() => {
      console.log(running);
      setRunning(false);
    }, 2000);

    setRunning(true);
    return () => clearTimeout(id);
  }, []);

  return <>Running: {running.toString()}</>;
}
```

1. what will be dumped to console?
2. what will the linter say?

CONSEQUENCES

Let's have a look at the component lifecycle

- Want to start and stop a timeout (automatically after render)

```
export default function App() {
  const [running, setRunning] = useState(false);

  useEffect(() => {
    const id = setTimeout(() => {
      console.log(running);
      setRunning(false);
    }, 2000);

    setRunning(true);
    return () => clearTimeout(id);
  }, [running]);

  return <>Running: {running.toString()}</>;
}
```

1. what will be dumped to console?
2. what will the linter say?
3. what happens if we add 'running'?

CONSEQUENCES

Let's have a look at the component lifecycle

- This behaviour makes totally sense
- Hard to get maybe, but hard due to the nature of the problem
  - async code
  - values/behaviour dependending on each other
- Is declarative programming style apropriate here?

CONSEQUENCES

Let's have a look at the component lifecycle

- We want to cancel the running timeout
- Somehow need to get access to the cleanup function or the id

```
export default function App() {
  const [running, setRunning] = useState(false);

  function cancel() { 🤔 }

  useEffect(() => {
    const id = setTimeout(() => setRunning(false), 2000);

    setRunning(true);
    return () => clearTimeout(id);
  }, []);

  return <button onClick={cancel}>Running: {running.toString()}</button>;
}
```

CONSEQUENCES

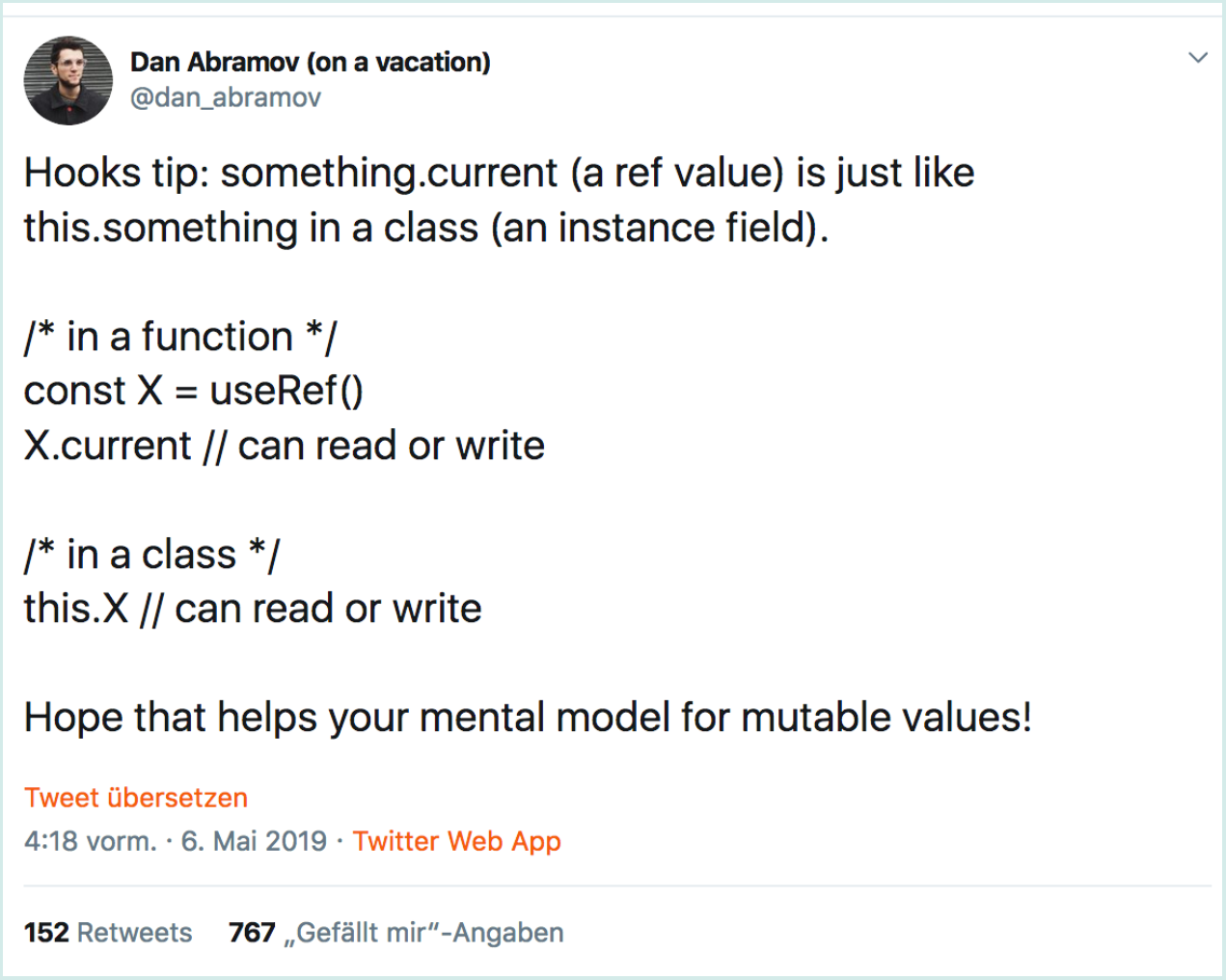
Let's have a look at the component lifecycle

- We want to cancel the running timeout
- Somehow need to get access to the cleanup function or the id

```
export default function App() {
  const [running, setRunning] = useState(false);
  const timerRef = useRef();
  function cancel() { clearTimeout(timerRef.current); }

  useEffect(() => {
    const id = setTimeout(() => setRunning(false), 2000);
    timerRef.current = id;
    setRunning(true);
    return () => clearTimeout(id);
  }, []);

  return <button onClick={cancel}>Running: {running.toString()}</button>;
}
```

A LOOK AT THE API

useRef



https://twitter.com/dan_abramov/status/1125223181701263360

CONSEQUENCES

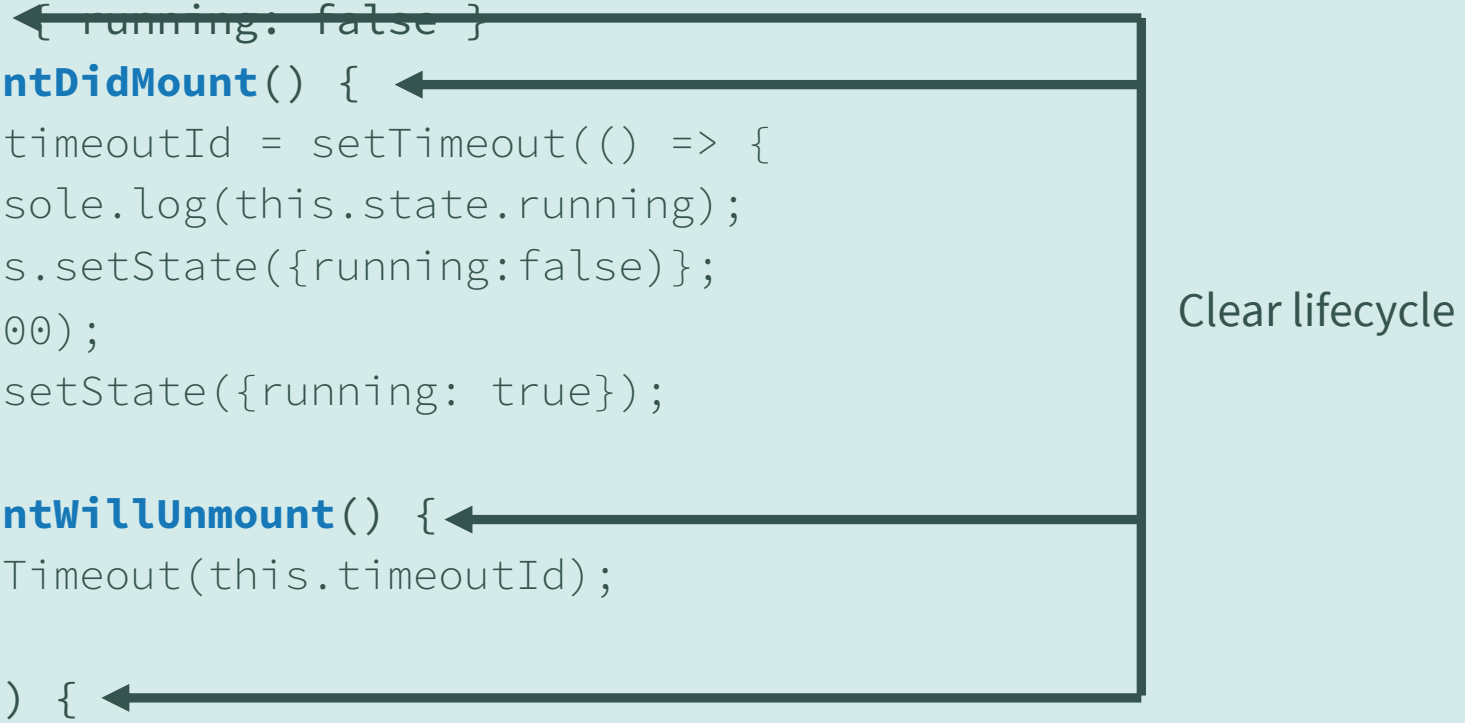
A look back to...

```
export default class App extends React.Component {
  state = { running: false }
  componentDidMount() {
    this.timeoutId = setTimeout(() => {
      console.log(this.state.running);
      this.setState({running:false)};
    ), 2000);
    this.setState({running: true});
  }
  componentWillUnmount() {
    clearTimeout(this.timeoutId);

  render() {
    return <button onClick={() =>
      clearInterval(this.timeoutId)}>...</button>
  }
}
```

CONSEQUENCES

A look back to...

```
export default class App extends React.Component {
  state = { running: false }
  componentDidMount() {
    this.timeoutId = setTimeout(() => {
      console.log(this.state.running);
      this.setState({running:false)};
    ), 2000);
    this.setState({running: true});
  }
  componentWillUnmount() {
    clearTimeout(this.timeoutId);

  render() {
    return <button onClick={() =>
      clearInterval(this.timeoutId)}>...</button>
  }
}
```



SUMMARY

Summary

- Hooks are kind of a new (meta-)language
  - they solve problems
  - but they raise new ones
- Beside learning the new API we get we even have more ways to structure our code
  - When to use custom hooks ? What to put in Custom Hooks?
  - Forget about smart/container and dumb components?
  - React was always about flexibility... some like, some don't

- I don't like hypes and that's probably the real reason I'm/I was sceptical

NILS HARTMANN
https://nilshartmann.net

Thanks a lot!

Slides: https://nils.buzz/react-meetup-hooks

NILS@NILSHARTMANN.NET