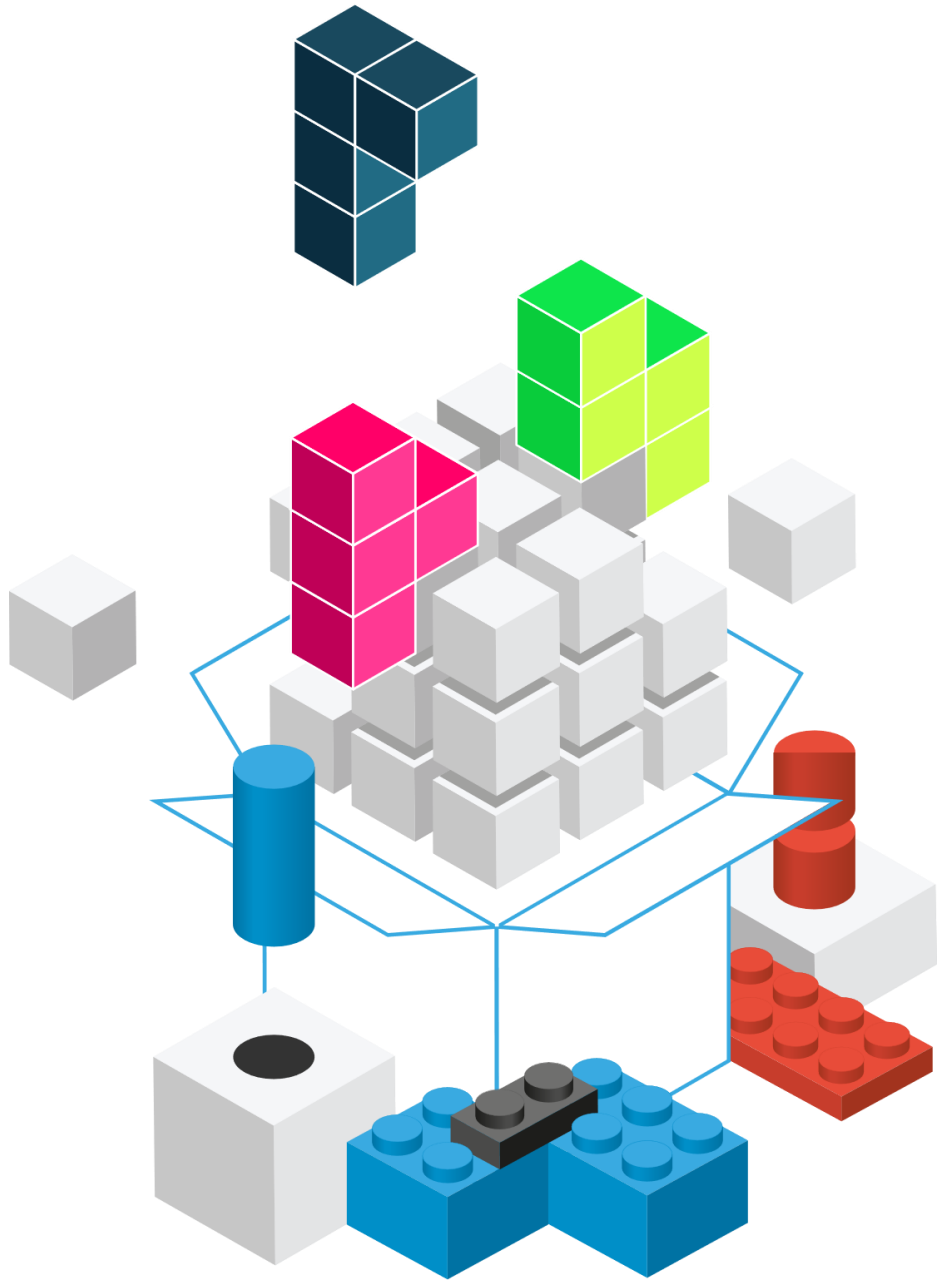
<WA1/>
<AW1/>
2021

Context

The Foundations of React

Fulvio Corno
Luigi De Russis
Enrico Masala



https://reactjs.org/docs/context.html

Full Stack React, Chapter “Advanced Component Configuration with props, state, and children”

React Handbook, Chapter “Context API”

Sort-of Globally Available Props (to avoid props drilling)

CONTEXT, USECONTEXT HOOK

Context

Unidirectional information flow +

Functional components =

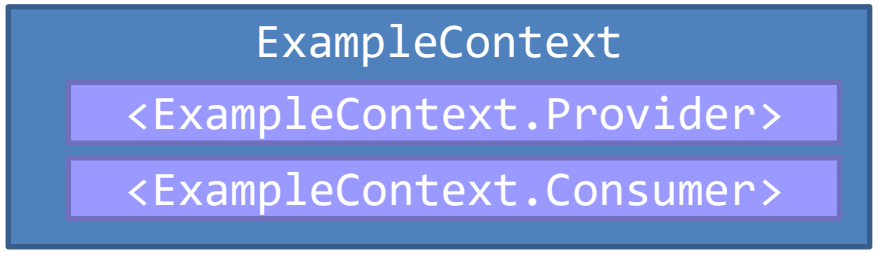
Must pass every prop to the component that needs it, and sometimes it means “drilling through” many components with several props

- Solution: the Context API offers a “global” set of props that are “automatically” available to lower components
  - Without declaring them explicitly at every level
- “Props teleporting”

Examples

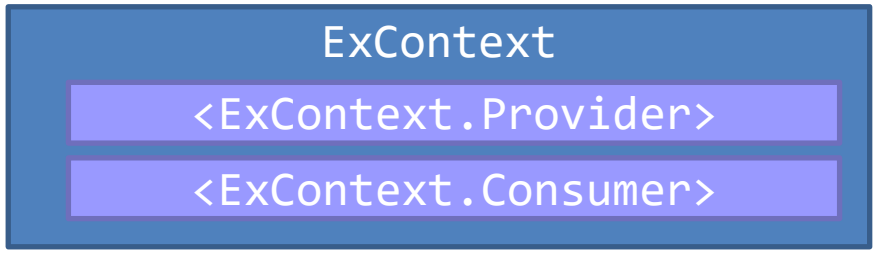
- The current visual theme for the whole page (e.g., dark, light, …)
  - Needed by most visual components (towards the bottom of the tree)
  - Not needed by any container component
- Logged in/logged out status (and basic user information)
  - Needed to enable/disable large portions of the page
  - Needed to provide user info in various parts of the page (e.g., avatar)
  - Needed to call remote APIs with user-related queries
- Shared data cache

Context Ingredients



- Context definition
  - `const ExampleContext = React.createContext()`
  - Defines a context object and stores it into the ExampleContext reference
- Context provider
  - `<ExampleContext.Provider value=...>` component
  - Injects the context `value` into all nested components
- Context consumer *(two possible equivalent techniques)*
  - `<ExampleContext.Consumer>`
    - Renders a function that receives the context current `value` as a parameter
  - `useContext(ExampleContext)`
    - Uses a *hook* to access the context current `value`

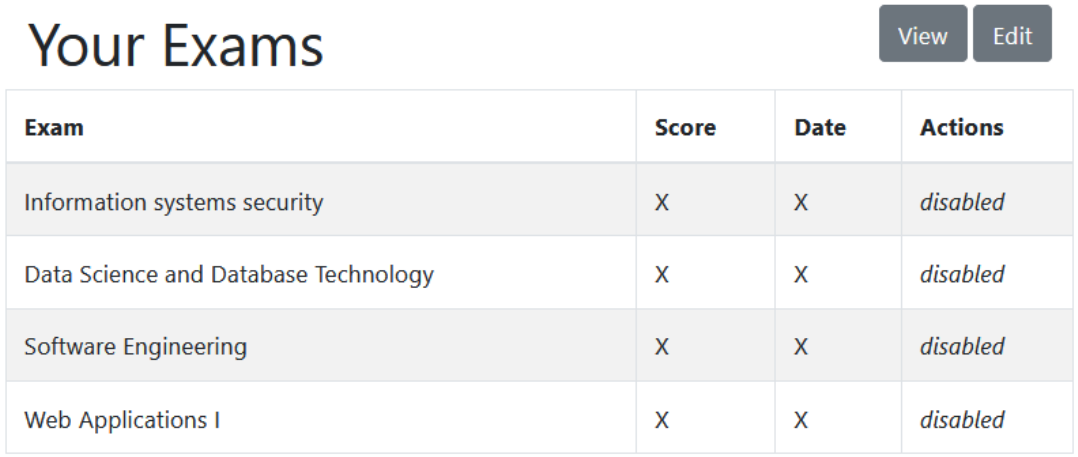
Context Definition



- `const ExContext = React.createContext(defaultValue)`
- Creates a new Context object
  - Contains *ExContext*.`Provider` and *ExContext*.`Consumer`
  - Represents the value of one object
    - May be a complex object with many properties/functions
  - The *ExContext* identifier is used in value propagation
- Components may subscribe (consume) to this context
  - The provided value comes from the closest *Provider* ancestor
    - If no provider is found, the defaultValue is used
    - In all other cases, defaultValue **is ignored**

Example

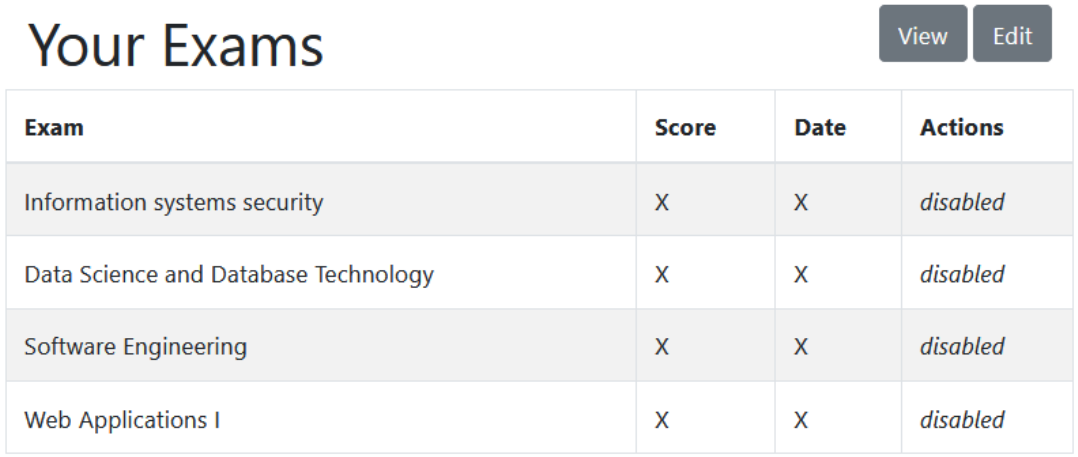
- Add two toggle buttons to the «React Scores» application
  - View/Hide: shows or hides the sensitive data (score and date)
  - Edit/Read: switch between read-only or editable mode



Example

App.js

```
function App() {
  const [privacy, setPrivacy] = useState(true);
  const [editable, setEditable] = useState(false);

  return (
    <Container className="App">
      <Row>
        <Title />
        <Col align='right'>
          <Button variant='secondary'
             onClick={() => setPrivacy(p => !p)}>
                {privacy ? 'View' : 'Hide'}
          </Button> 
          <Button variant='secondary'
             onClick={() => setEditable(p => !p)}>
                {editable ? 'Read' : 'Edit'}
          </Button>
        </Col>
      </Row>
      . . .
```



Example

App.js

```
import { PrivacyMode, EditMode }
    from './createContexts';
```

createContexts.js

```
import React from 'react';

const PrivacyMode = React.createContext();
const EditMode = React.createContext();

export { PrivacyMode, EditMode } ;
```

Context Provider

- A component *`ExContext`*`.Provider` is *automatically created* for each new Context
- The component specifies a `value` prop, that is available to all nested "consumer" components (even if deeply nested)
  - Consumers MUST be nested inside the provider
  - Providers may be anywhere (assuming the context object is visible)
- Providers may be *nested*: each level may override the previous `value`
- When the Provider's `value` changes, all consumers will re-render

Example

App.js

```
function App() {
  . . .
  return (
    <Container className="App">
      . . .
      <Row>
        <PrivacyMode.Provider value={privacy}>
          <EditMode.Provider value={editable}>
            <ExamTable
              courses={fakeCourses}
              exams={fakeExams} />
          </EditMode.Provider>
        </PrivacyMode.Provider>
      </Row>
    </Container>
  );
}
```

createContexts.js

```
import React from 'react';

const PrivacyMode = React.createContext();
const EditMode = React.createContext();

export { PrivacyMode, EditMode } ;
```

Context Consumer (as a component)

- The *automatically created* component <*ExContext*.Consumer> may be used in the render function/method
- You must provide a *callback function* that
  - Receives the context value (from the closest provider, or defaultValue if no provider is found)
  - Returns the React Element to be rendered

```
<ExContext.Consumer>
  {value => /* render something
                based on the context value */}
</ExContext.Consumer>
```

Example

App.js

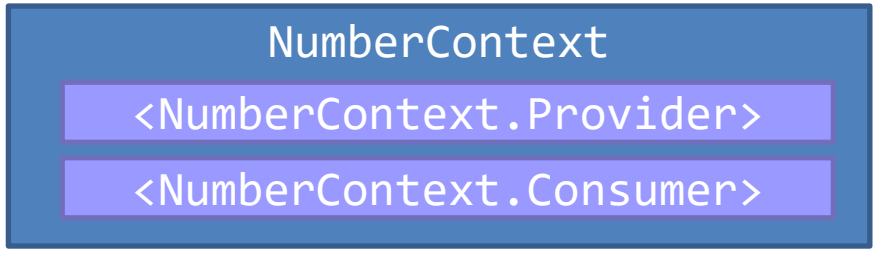
```
function App() {
  . . .
  return (
    <Container className="App">
      . . .
      <Row>
        <PrivacyMode.Provider value={privacy}>
          <EditMode.Provider value={editable}>
            <ExamTable
              courses={fakeCourses}
              exams={fakeExams} />
          </EditMode.Provider>
        </PrivacyMode.Provider>
      </Row>
    </Container>
  );
}
```

examComponents.js

```
import { PrivacyMode, EditMode } from './createCon
texts';

function ExamRow(props) {
    return (<tr>
        <ExamInfo {...props} />
        <EditMode.Consumer>
            {editable => editable ?
                <ExamControls
                    exam={props.exam}
                    deleteExam={props.deleteExam}/>
                : <td><i>disabled</i></td>}
        </EditMode.Consumer>
    </tr>
    );
}
```

Accessing Context With Hooks

- The `useContext` hook allows the current component to *consume* the context
- The argument is a Context object
  - Must have been created by `React.createContext()`
- The value depends on the closest enclosing provider
  - Must be nested inside `<MyContext.Provider>`

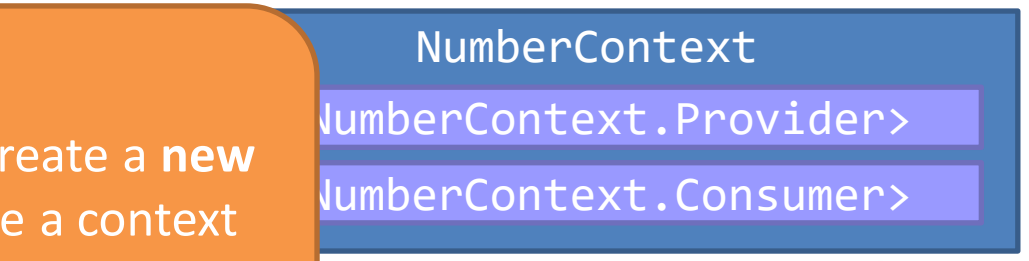



```
function Display() {

  const value = useContext(NumberContext);

  return <div>The answer is {value}.</div>;
}
```

Accessing Context With Hooks

- The `useContext` hook allows the current compo… *consume* the conte…
- The argument is a C…
  - Must have been cre… `React.createContext();`
- The value depends on the closest enclosing provider
  - Must be nested inside `<MyContext.Provider>`

There is no way to create a **new** context, or to create a context **provider**, with Hooks




```
…splay() {

  const value = useContext(NumberContext);

  return <div>The answer is {value}.</div>;
}
```

Example

App.js

```
function App() {
  . . .
  return (
    <Container className="App">
      . . .
      <Row>
        <PrivacyMode.Provider value={privacy}>
          <EditMode.Provider value={editable}>
            <ExamTable
              courses={fakeCourses}
              exams={fakeExams} />
          </EditMode.Provider>
        </PrivacyMode.Provider>
      </Row>
    </Container>
  );
}
```

examComponents.js

```
import { PrivacyMode, EditMode } from './createCon
texts';

function ExamInfo(props) {
  let privacyMode = useContext(PrivacyMode)
  return (<>
    <td>{props.examName}</td>
    <td>{privacyMode? "X" : props.exam.score}</td>
    <td>{privacyMode? "X" : props.exam.date
       .format('DD MMM YYYY')}</td>
  </>)
}
```

Accessing Multiple Contexts

- May call `useContext` more than once
- All the context variables will be available
- No need to nest components

```
function HeaderBar() {
  const user = useContext(CurrentUser);
  const notif = useContext(Notifications);

  return (
    <header>
      Welcome back, {user.name}!
      You have {notif.length} notifications.
    </header>
  );
}
```

https://daveceddia.com/usecontext-hook/

Accessing Multiple Contexts

```
function HeaderBar() {
  return (
    <CurrentUser.Consumer>
      {user =>
        <Notifications.Consumer>
          {notif =>
            <header>
              Welcome back, {user.name}!
              You have {notif.length}
              notifications.
            </header>
          }
        </Notifications.Consumer>
      }
    </CurrentUser.Consumer>
  );
}
```

Consumer Component

```
function HeaderBar() {
  const user = useContext(CurrentUser);
  const notif = useContext(Notifications);

  return (
    <header>
      Welcome back, {user.name}!
      You have {notif.length} notifications.
    </header>
  );
}
```

useContext Hook

Changing Context Values

- When a Consumer child needs to update the context value, the Provider must provide a function callback to perform the update
  - As a prop (by drilling the nesting levels)
  - As part of the context value
- Remember: the **state** is part of the component containing the Provider
  - Not in the provider itself
  - Not in the context object

Caveats

- Don’t put everything into Context
  - Defeats component portability
  - Reduces “purity” of functional components
- Don’t use it for programming laziness
  - Explicit parameter passing is also a good documentation practice
- Don’t use it to correct design errors
  - Often, a refactoring of the component tree (and props/state lifting) may be a cleaner solution

Before You Consider Context…

- Passing a component as a prop (inversion of control)
  - When a nested components needs many props from and upper component
  - The upper component defines JSX of the element (using the available info)
  - The component itself is passed as a prop (just one prop, that will be passed and rendered)
- Use “render props”
  - Callback functions as props (https://reactjs.org/docs/render-props.html)
  - The lower component will call the “render prop” at render time, that has access to the upper component’s props and state
- Use Children Components

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/