

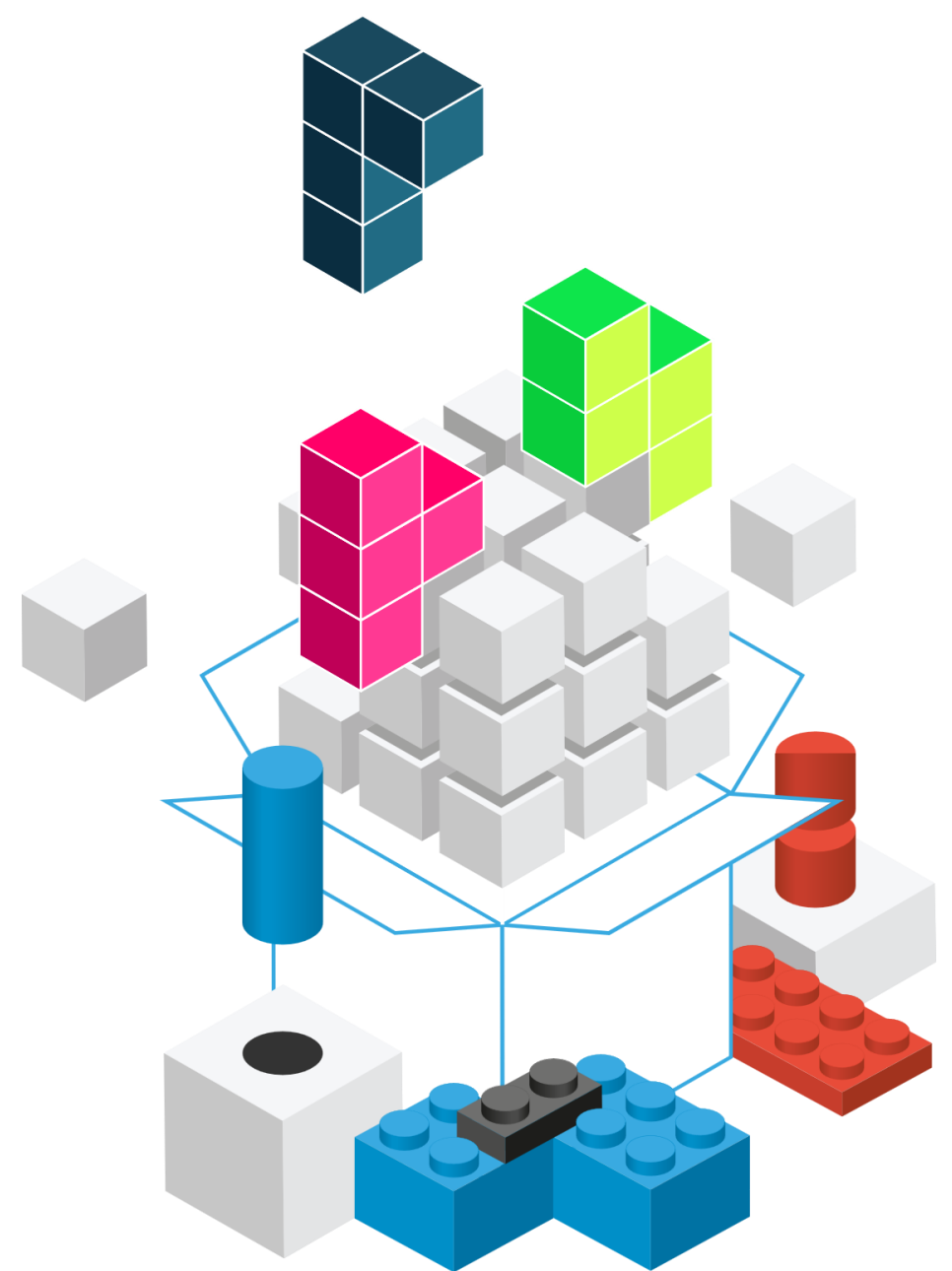
<WA1/>  
<AW1/>  
2024

# Context

## The Foundations of React

Fulvio Corno

Luigi De Russis





<https://react.dev/learn/passing-data-deeply-with-context>

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 =


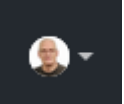

---

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

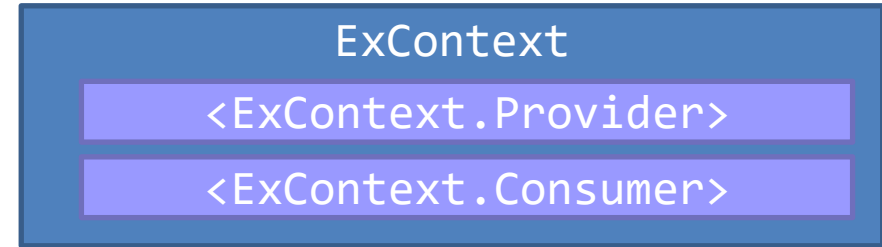
- Solution: the Context API offers a “global” set of props that are “automatically” available to lower components
  - Without declaring them explicitly at every level
- “Props teleporting”



# Examples

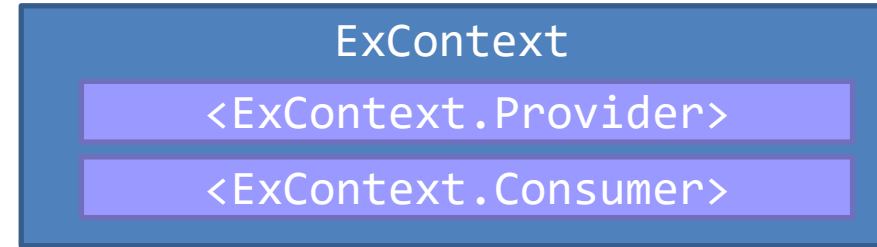
- The current visual theme for the whole page (e.g., dark, light, ...) 
  - Needed by most visual components (towards the bottom of the tree)
  - Not needed by any container component
- Logged in/logged out status (and basic user information) 
  - Needed to enable/disable large portions of the page
  - 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
- Multi-language support 

# Three Context Ingredients



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

# Context Definition



```
const ExContext = React.createContext(defaultValue)
```

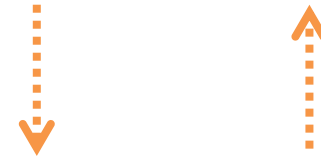
- Creates a new Context object
  - Contains 2 properties: `ExContext.Provider` and `ExContext.Consumer`
  - Represents the **value** of one state object
    - May be a complex object with many properties/functions
  - The `ExContext` identifier is used in value propagation
- Components may subscribe (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

- Create a (very) simple multi-language application
  - Italian and English
  - with a toggle button to change the entire application language

Welcome to a simple multilanguage app!

Translate to Italian



Benvenuti in una semplice applicazione multi-lingua!

Traduci in inglese

# Example

App.jsx

```
...  
function App() {  
  const [language, setLanguage] = useState('english');  
  
  function toggleLanguage() {  
    setLanguage((language) =>  
      (language === 'english' ? 'italian' : 'english'));  
  }  
  
  return (  
    <div className="App">  
      <Welcome />  
      <Button toggleLanguage={toggleLanguage} />  
    </div>  
  )  
  ...  
}
```

Welcome to a simple multilanguage app!

Translate to Italian



# Example

App.jsx

```
import LanguageContext  
  from './languageContext';
```

languageContext.js

```
import React from 'react';  
  
const LanguageContext = React.createContext();  
  
export default LanguageContext;
```

# 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.jsx

```
import LanguageContext from './languageContext';
...

function App() {
  ...
  return (
    <div className="App">
      <LanguageContext.Provider value={language}>
        <Welcome />
        <Button toggleLanguage={toggleLanguage} />
      </LanguageContext.Provider>
    </div>
  );
}
...
```

languageContext.js

```
import React from 'react';

const LanguageContext = React.createContext();

export default LanguageContext;
```

# 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.jsx

```
import LanguageContext from './languageContext';
...

function App() {
  ...
  return (
    <div className="App">
      <LanguageContext.Provider value={language}>
        <Welcome />
        <Button toggleLanguage={toggleLanguage} />
      </LanguageContext.Provider>
    </div>
  );
}
...
```

## Components.jsx

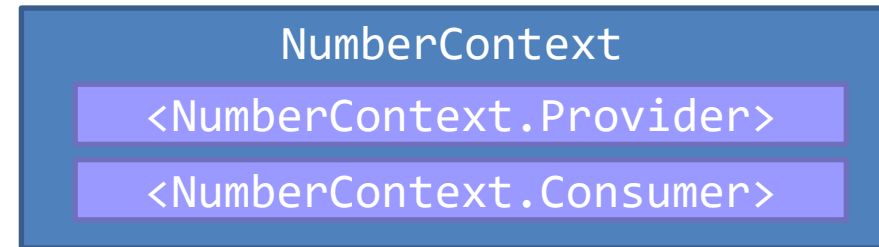
```
import LanguageContext from './languageContext';
import translations from './translations';

function Button(props) {
  return (
    <LanguageContext.Consumer>
      {language =>
        <button
          onClick={props.toggleLanguage}>
          {translations[language]['button']}
        </button>
      }
    </LanguageContext.Consumer>
  );
}

function Welcome() {
  return (
    <LanguageContext.Consumer>
      {language =>
        <p> {translations[language]['welcome']} </p>
      }
    </LanguageContext.Consumer>
  );
}
```

# 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>;  
}
```

An arrow points from the `NumberContext` box in the diagram above to the `useContext(NumberContext)` call in the code block below.

# 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 with `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 object, or to create a context provider, with Hooks

NumberContext

`NumberContext.Provider`

`NumberContext.Consumer`

```
display() {  
  
  const value = useContext(NumberContext);  
  
  return <div>The answer is {value}</div>;  
}
```

# Example

App.jsx

```
import LanguageContext from './languageContext';
...

function App() {
  ...
  return (
    <div className="App">
      <LanguageContext.Provider value={language}>
        <Welcome />
        <Button toggleLanguage={toggleLanguage} />
      </LanguageContext.Provider>
    </div>
  );
}
...
```

Components.jsx

```
import { useContext } from 'react';
import LanguageContext from './languageContext';
import translations from './translations';

function Button(props) {
  const language = useContext(LanguageContext);

  return (
    <button onClick={props.toggleLanguage}>
      {translations[language]['button']}
    </button>
  );
}

function Welcome() {
  const language = useContext(LanguageContext);

  return (
    <p> {translations[language]['welcome']} </p>
  );
}
```



# Accessing Multiple Contexts

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

- 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>
  );
}
```

# Accessing Multiple Contexts: Component vs. Hook

```
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>  
  );  
}
```

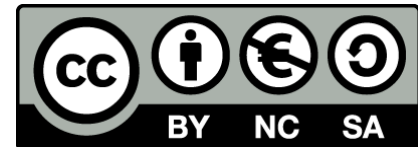
useContext Hook

# 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**
    - Example: `{ language: 'English', toggleLanguage : toggleLanguage }`
- Remember: the **state** is part of the **component containing the Provider**
  - Not in the provider itself
  - Not in the context object

# Caveats

- Do not 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



# 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/>

