# Trainer guide
# Examples and exercises (React)
# **React-12 – Context**

## Contents

## **Trainer demo** | To-do list with Context

Link to environment

In this demo, you will show how to convert a to-do list app to use React Context instead of relying on prop drilling.

**src/App.jsx**

- Here we have a simple to-do list app.
- *Click on the different to-do items to demonstrate them toggling between complete and incomplete.*
- In the App component, we store the to-do items in state.
    - Each to-do item has three properties:
        - A unique ID
        - A title
        - A boolean property “isComplete”.
- There is also a `toggleTodo` function, which will toggle a to-do with a given ID between complete and incomplete.
- In the App component, we render a child component <TodoList>. This renders all the to-do items, so we need to pass the array of `todos` as a prop. We also need to make sure that the individual to-do items are able to toggle themselves, so we need to pass the `toggleTodo` function as a prop as well.

**src/TodoList.jsx**

- In the <TodoList> component, we take all of the to-do items that were passed in and render them as individual <TodoItem> components.
- Because each clickable <TodoItem> needs to be able to pass a message to the top to toggle that item as complete, we need to pass the `toggleTodo` function again as a prop.
- We’ve now passed the toggleTodo function down two layers. This is entering prop-drilling territory, which is a bit clunky, especially in larger applications.
- We can eliminate the need for prop drilling by using Context.

**Setting up Context (example)**

- Before we add Context to our to-do list application, let’s understand how to set up Context in the simplest possible case.
- *Link to example environment*
- There are a few steps to setting up Context. And, until you’ve done this a few times, it’s easy to forget or become confused by these steps, so if you ever need to use this example as a reference, you’ve been given a link to it.

- **Step 1** is to create a Context object.
  - The Context object is what other components subscribe to.
  - Creating one is fairly simple - making sure the `createContext` function has been imported from the 'react' package, all you have to do is call `createContext` and assign the result to a variable. In this case, we've called it `MyContext`.
  - `MyContext` is an object. The most important property of this object for us is the `Provider` property, which is a component that gives all of its nested components access to the context. We'll see this being used in the next step.
- **Step 2** is to create a parent component that provides a context value.
  - In this component, we'll create some value that we want to provide to the child components that consume the context.
  - This value can be anything we want, but in this simple example we'll just use a string called `contextValue`.
  - Then, we return the <MyContext.Provider> component. We pass the context value via the `value` prop.
  - Every component that accesses the context will be a child of this <MyContextProvider> component. We use the `children` prop to make sure all the children of <MyContextProvider> are displayed.
- **Step 3**: Now that we've set up the <MyContextProvider>, let's look at our main App component. We need to make sure that any children that need access to the context are wrapped inside the provider that we just made.
  - Notice that the <Child> component is a child of <MyContextProvider>. If this were not the case, then <Child> would not be able to access the context value.
- We've defined the Child component at the bottom of the file. We want to consume the context value from inside this component.
- **Step 4**: To access the context value from within a child component, we use a hook called useContext. When you call useContext, you simply pass in the context you want to access – in this case, MyContext. This will return whatever Context value is being provided by MyContextProvider. We then display that value within the <p> tag.
- The context value is successfully passed to a child component without being passed via props – which is very helpful if our child is deeply nested, because it means we don't have to pass the same prop down multiple layers of components.
- Now we're going to implement Context in the to-do list app to show you how to solve prop drilling in that example.

Back to to-do list app

### src/App.jsx

- Let’s apply those four steps to our to-do list application.
- **Step 1** is to create a Context object. To keep things organised, we’ll do this in a separate file.

### src/context/todo-context.js

- *Create file src/context/todo-context.js*
- First, we need to `import React`, as well as the `createContext` function:

```
import React, { createContext } from 'react'
```

- Now, let’s create our Context object:

```
const TodoContext = createContext()
```

- **Step 2** is to create a Provider component. We’ll do that in the same file.
- *Add the following code to todo-context.js*

```
function TodoProvider({ children }) {
  return (
    <TodoContext.Provider>
      {children}
    </TodoContext.Provider>
  )
}
```

- This provider component needs to provide access to the to-do items, as well as access to the to-do-toggling functionality. Let’s move the to-do items state out of the App component and into our context provider. We’ll also move the `toggleTodo` function.

### src/App.jsx

- *Cut the `useState` call and `toggleTodo` function out of this component.*

**src/context/todo-context**

- *Paste the `useState` call and `toggleTodo` function into TodoProvider:*

```
function TodoProvider({ children }) {
  const [todos, setTodos] = useState([
    { id: 1, title: 'Learn React useState', isComplete: true },
    { id: 2, title: 'Learn React useEffect', isComplete: true },
    { id: 3, title: 'Learn React useContext', isComplete: false },
  ])

  function toggleTodo(id) {
    setTodos((prev) =>
      prev.map((todo) => ({
        ...todo,
        isComplete: todo.id === id ? !todo.isComplete : todo.isComplete
      }))
    )
  }

  return (
    <TodoContext.Provider>
      {children}
    </TodoContext.Provider>
  )
}
```

- Since we're now using the useState hook, we need to import it in this file.

```
import React, { createContext, useState } from 'react'
```

- Now, in order to provide anything to our children components, we need to create a context value. Let's make an object that contains both `todos` and `toggleTodo`, and provide it via the `value` prop:

```
function TodoProvider({ children }) {
  const [todos, setTodos] = useState([
    { id: 1, title: 'Learn React useState', isComplete: true },
    { id: 2, title: 'Learn React useEffect', isComplete: true },
    { id: 3, title: 'Learn React useContext', isComplete: false },
  ])

  function toggleTodo(id) {
    setTodos((prev) =>
      prev.map((todo) => ({
        ...todo,
        isComplete: todo.id === id ? !todo.isComplete : todo.isComplete,
      }))
    )
  }

  const contextValue = { todos, toggleTodo }

  return (
    <TodoContext.Provider value={contextValue}>
      {children}
    </TodoContext.Provider>
  )
}
```

- Now any child of TodoProvider will be able to access this value.
- **Step 3** is to make sure that any nested components that require access to the context are wrapped inside of TodoProvider. We'll first need to make sure that TodoProvider is exported from this file.
    - *Add the **export** keyword before the TodoProvider definition.*

**src/App.jsx**

- Let's now import the context provider within App.jsx.

```
import { TodoProvider } from './context/todo-context'
```

- Now, let's wrap <TodoList> inside our provider.

```
export default function App() {
  return (
    <main>
      <h1>☑ Todo List</h1>
      <TodoProvider>
        <TodoList todos={todos} toggleTodo={toggleTodo} />
      </TodoProvider>
    </main>
  )
}
```

- Because the context provider provides `todos` and `toggleTodo`, we should no longer be passing those as props. Let's remove them.
- *Remove the `todos` and `toggleTodo` props from the <TodoList> call:*

```
export default function App() {
  return (
    <main>
      <h1>☑ Todo List</h1>
      <TodoProvider>
        <TodoList />
      </TodoProvider>
    </main>
  )
}
```

- Now that we're providing the context values, the final step is to consume the context from within the child components. Let's update <TodoList> first.

**src/components/TodoList.jsx**

- First, we're no longer accessing `todos` and `toggleTodo` as props, so we can remove them from the props list.
- *Remove the `todos` and `toggleTodo` props from the TodoList definition:*

```
export default function TodoList() { ... }
```

- We also won't need to pass `toggleTodo` to <TodoItem>, because that should be provided through the context, so we can remove that too.
- *Remove the `toggleTodo` prop from the <TodoItem> call.*
- But we still need to consume the to-do items via context in this component.
- Before we do that, we need to make sure that the context object is imported into this file, because we need to subscribe to it.

**src/context/todo-context.js**

- *Make sure TodoContext is exported from this file.*

```
export const TodoContext = createContext()
```

**src/components/TodoList.jsx**

- Now let's import the context object that we just exported.

```
import { TodoContext } from '../context/todo-context'
```

- We’ll also need to import the useContext hook from React.

```
import React, { useContext } from 'react'
```

- We can now consume the context.
- *Write the following in the body of TodoList():*

```
const contextValue = useContext(TodoContext)
```

- Since we made the context value an object, we can access the to-dos via dot notation:

```
const todos = contextValue.todos
```

- Or we could destructure the context value to extract the to-dos on a single line.

```
const { todos } = useContext(TodoContext)
```

- Now the TodoList component accesses the to-dos via Context.
- *The TodoList component should now look like this:*

```
export default function TodoList() {
  const { todos } = useContext(TodoContext)

  return (
    <ul>
      {todos.map((todo) => (
        <TodoItem key={todo.id} todo={todo} />
      ))}
    </ul>
  )
}
```

**src/components/TodoItem.jsx**

- Let’s finish up by updating the TodoItem component.
- First, since the `toggleTodo` function is provided via Context, we should remove it from the props list.
- *Remove the `toggleTodo` prop from the TodoItem definition.*
- Instead, we’ll access `toggleTodo` via Context. Let’s add the necessary imports:

```
import React, { useContext } from 'react'
import { TodoContext } from '../context/todo-context'
```

- Finally, we'll consume the `toggleTodo` function from `TodoContext`:

```
export default function TodoItem({ todo }) {
  const { toggleTodo } = useContext(TodoContext)

  return (
    <li
      onClick={() => toggleTodo(todo.id)}
      className={todo.isComplete ? 'complete' : ''}
    >
      {todo.title}
    </li>
  )
}
```

**src/App.jsx**

- Now our app functions as intended, but with no prop drilling in sight!
- Notice how clean and readable our App function looks now.
- You've now seen how to implement Context in a React application.

- One thing to note is that Context is one of the more advanced React features, so don't worry if you don't feel you understand it completely.

Solution reference

## **React-12c** | Exercise: Using Context

[Link to environment]

This exercise gives learners practice using the useContext hook in an React application that implements Context.

**src/App.jsx**

- The fruit market now has a cart feature - all the fruits in your cart are now displayed at the top of the page.
- [This page] demonstrates how the app is supposed to function. *Click on the "Add to cart" buttons and see the items being added to the cart.*
- However, in this exercise, the "Add to cart" buttons are currently not working.

**Exercise**

- Your task is to make the "Add to cart" buttons function as intended.
- This app has a context set up already. You only need to update the ItemCard component.
- Explore the files to understand the existing context setup before attempting the exercise.

*Solve the exercise live before moving on.*

**Solution (src/components/ItemCard.jsx)**

- *Import the useContext hook:*

```
import React, { useContext } from 'react'
```

- *At the top of the body of the ItemCard component, access* ***`addToCart`*** *via* *`CartContext`:*

```
const { addToCart } = useContext(CartContext)
```

[Solution reference]