

TECHNOLOGY



ReactJS

React Context API and Redux



Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Differentiate between React Context API and Redux and when to use each
- 🕒 Compare the benefits and drawbacks of using React Context API and Redux
- 🕒 Assess the implementation of React Context API and Redux in a complex React application for efficient state management
- 🕒 Describe the concept of asynchronous data retrieval and its role in modern web development



A Day in the Life of a ReactJS Developer

Sarah's client wants a complex application that requires advanced state management. She knows there are two state management options:

- React Context API
- Redux

After considering the application's requirements, Sarah decides to use Redux, as the application will:

- Manage a large amount of data in a centralized store
- Anticipate the need for asynchronous actions

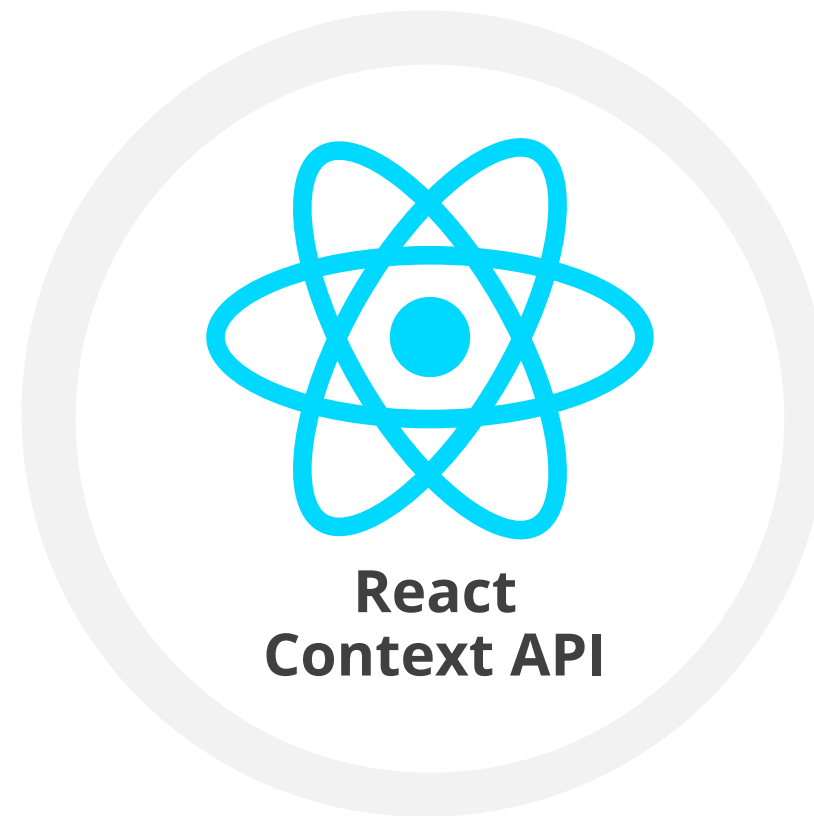
By the end of the day, Sarah will be able to implement the advanced state management options using Redux successfully.



React Context API

What Is React Context API?

React Context API allows centralized data storage and makes it available to all components of an application.



React Context API also enables data sharing across components in a React application without needing props.



Context API: Applications

Context in API can be utilized in the following ways to streamline data sharing and simplify state management within React applications:

Sharing global state

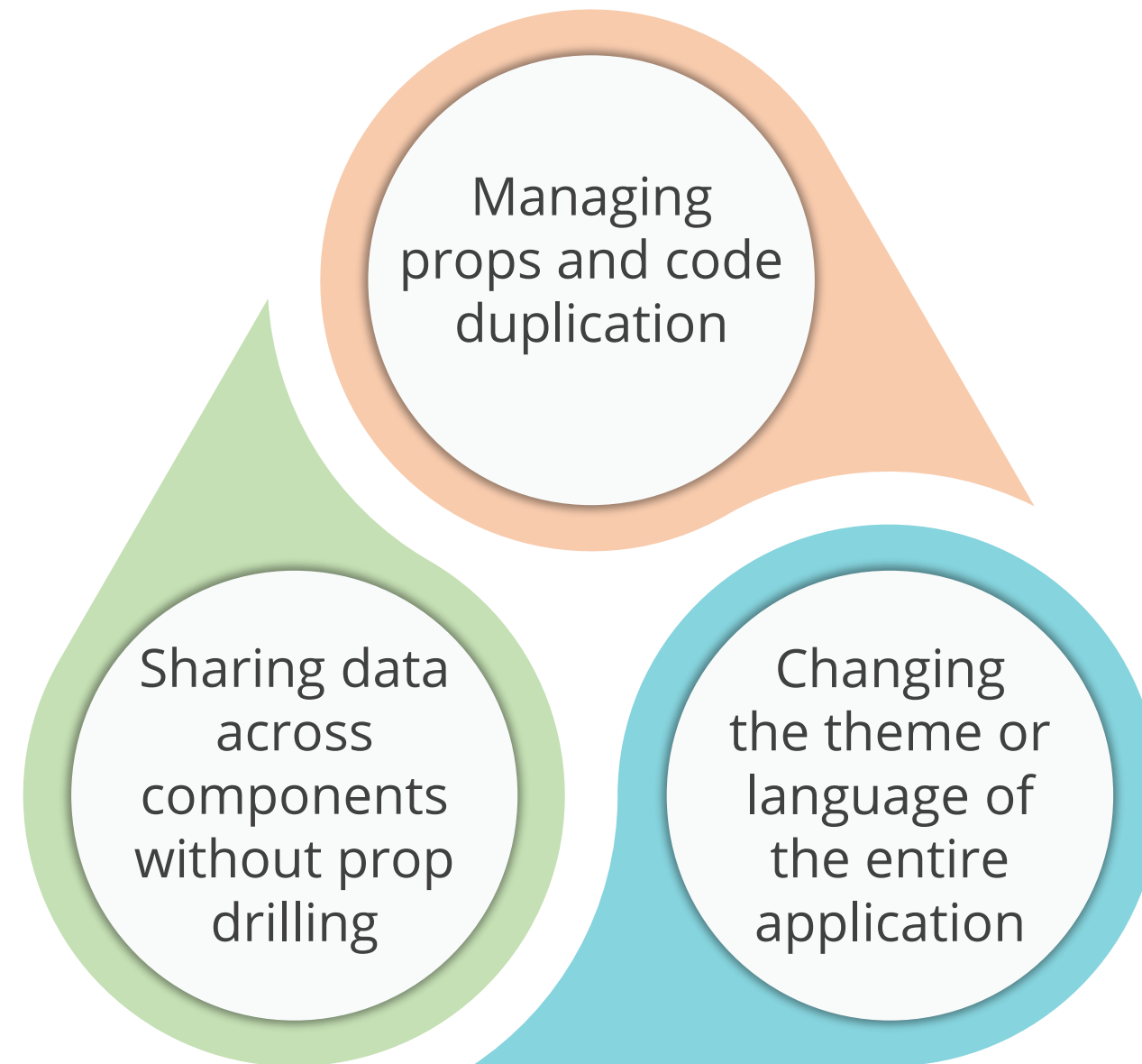
Theming

Localization



When to Use Context API?

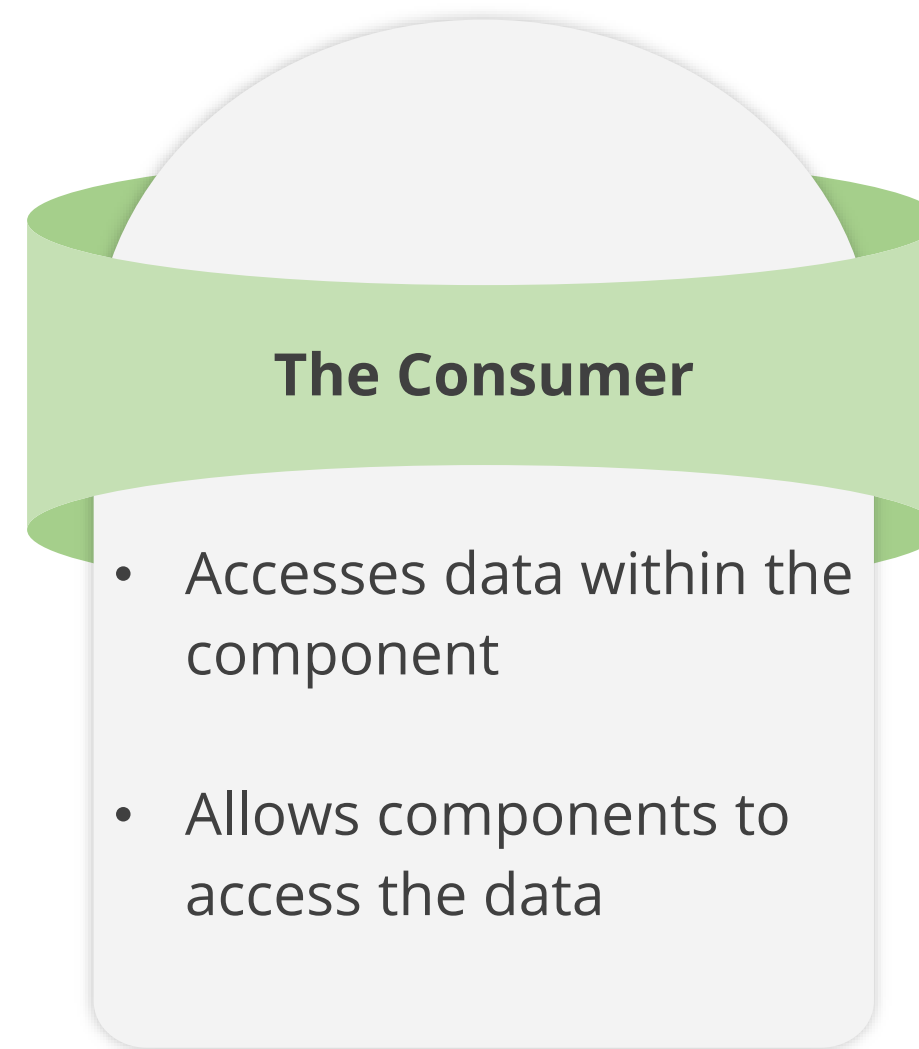
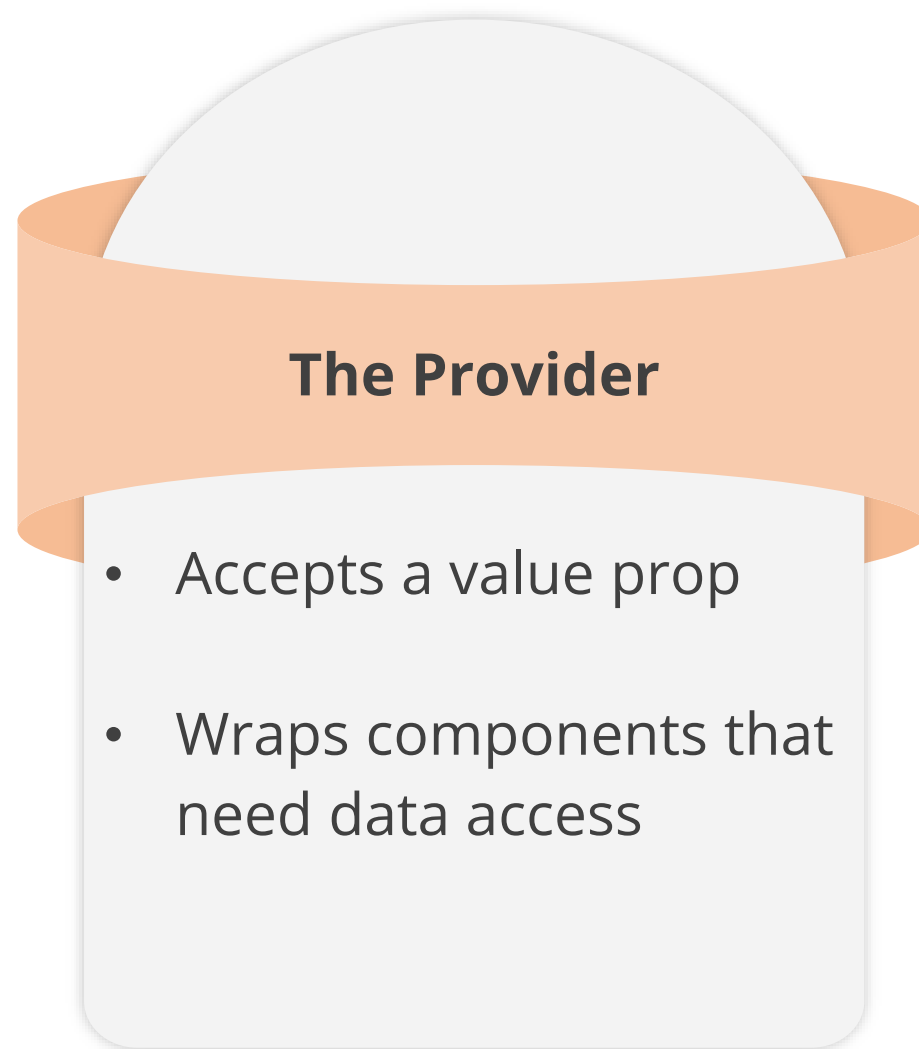
React Context API is a good fit for:



Creating a Context in React

Context is a data store, where data is stored as a key-value pair.

React Context API includes two components:



Creating a Context in React

The following steps are required to create a Context in React:



Using Context with Class Components

The following steps are necessary to use Context API with class components:

- 01 Import the **createContext** method
- 02 Create a context object using the **createContext** method
- 03 Create a provider component that will wrap the child components
- 04 Wrap the child components that need access to the context
- 05 Access the context value in the child components



Example

This code showcases how to use React Context API to share and access data between components.

```
// Create a new context
const ThemeContext = React.createContext('light');
// Wrap the components that need access to the shared data
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

Creating a new context

Example

In this code, the **ThemedButton** function uses the **useContext** Hook to access the current theme from **ThemeContext**.

```
function ThemedButton() {  
  const theme = useContext(ThemeContext);  
  return (  
    <button style={{ background: theme === 'dark' ?  
'black' : 'white' }}>  
      {theme}  
    </button>  
  );  
}
```



Wrapping the component

Example

In this code, the **Toolbar** function renders a **ThemedButton** component, which allows access to the shared data from anywhere within the component tree.

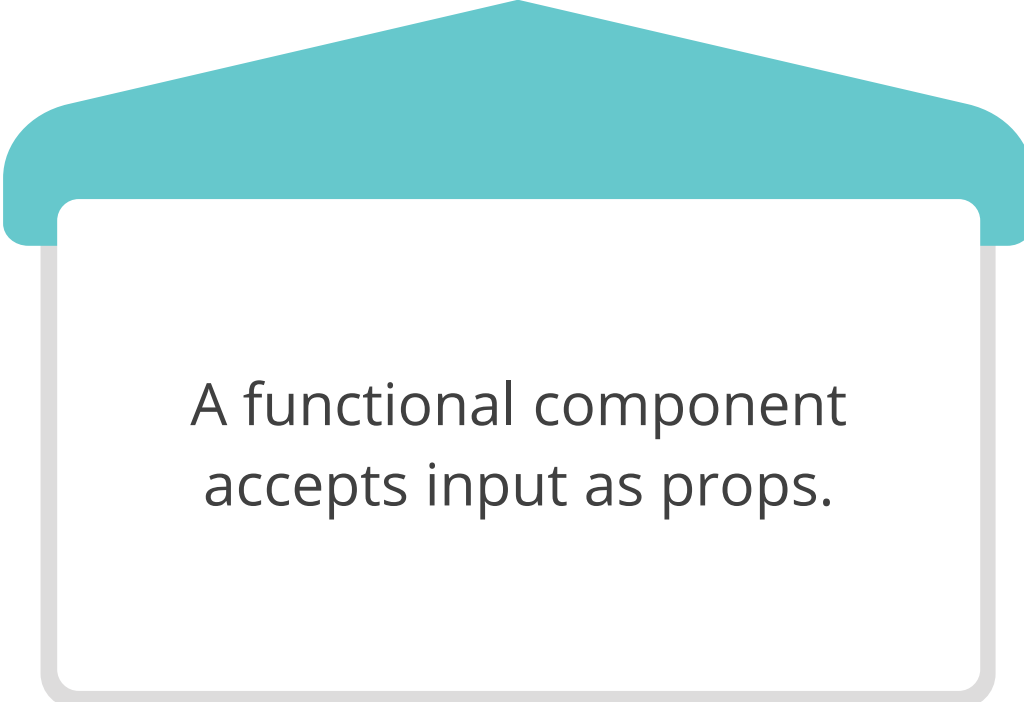
```
// Access the data from anywhere within the component tree
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
```



Accessing the data

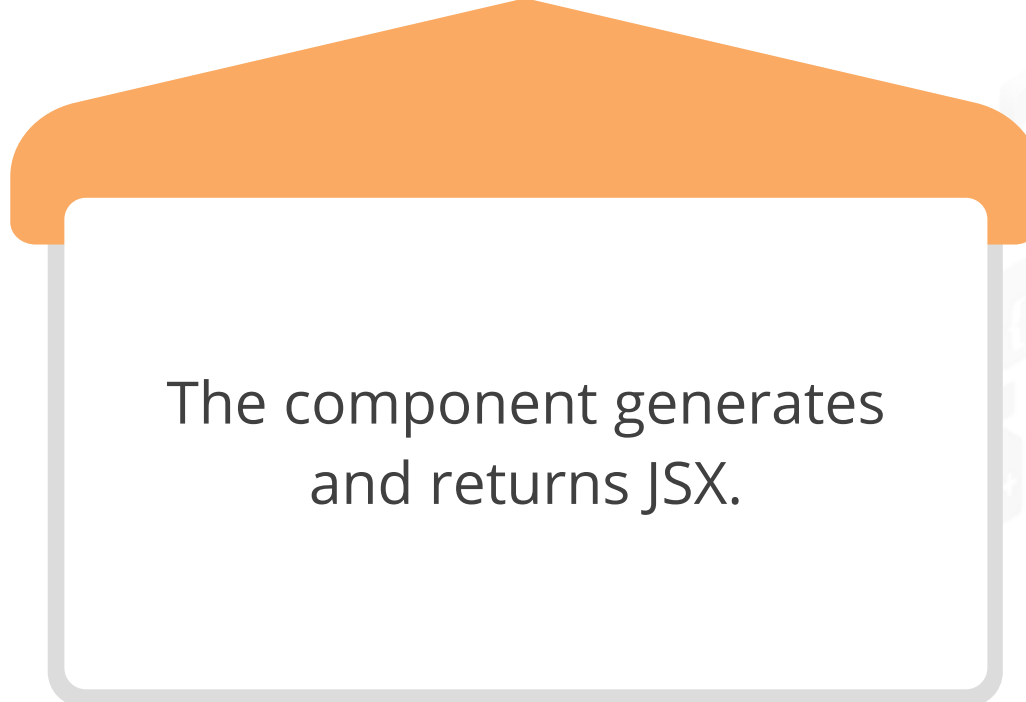
Using Context with Functional Components

Functional components are JavaScript functions that return JSX to render UI elements. The salient functions of the components are as follows:



A functional component accepts input as props.

The diagram shows a light blue trapezoidal shape with a teal roof-like top, representing a component that receives props.



The component generates and returns JSX.

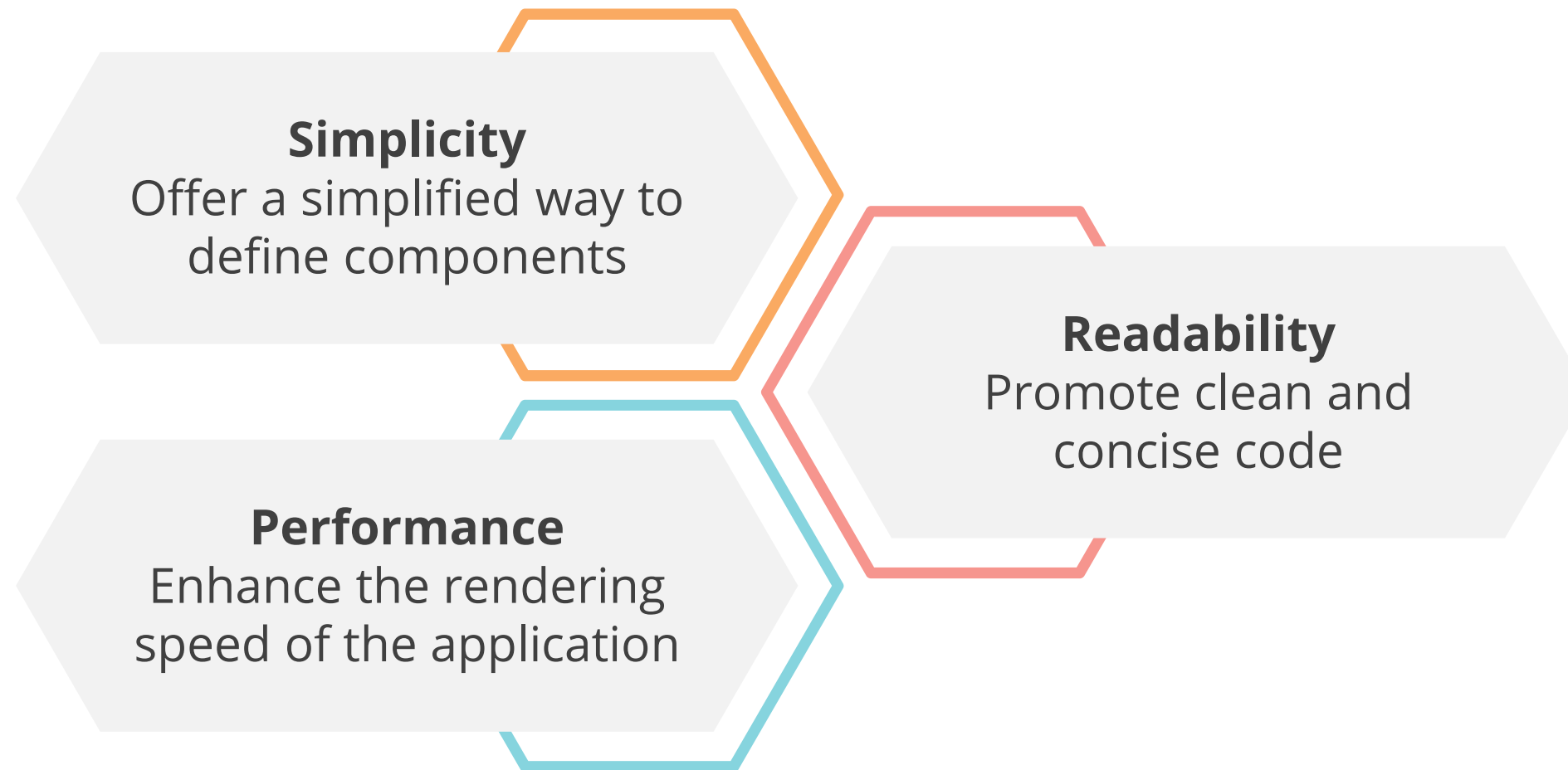
The diagram shows an orange trapezoidal shape with an orange roof-like top, representing a component that returns JSX.

Note

Use the **useContext** Hook to get context data

Functional Components: Benefits

Functional components are a simpler and more concise alternative to class components and they provide the following benefits:



Using React Context and `contextType`: Example

The following code shows how to use React Context and **`contextType`** in functional and class components:

```
import React, { useContext } from 'react';  
  
// Create a context  
const myContext = React.createContext('default value');  
  
//Create a component that consumes the context  
function MyComponent() {  
  const value = useContext(myContext);  
  return <div>{value}</div>;  
}
```



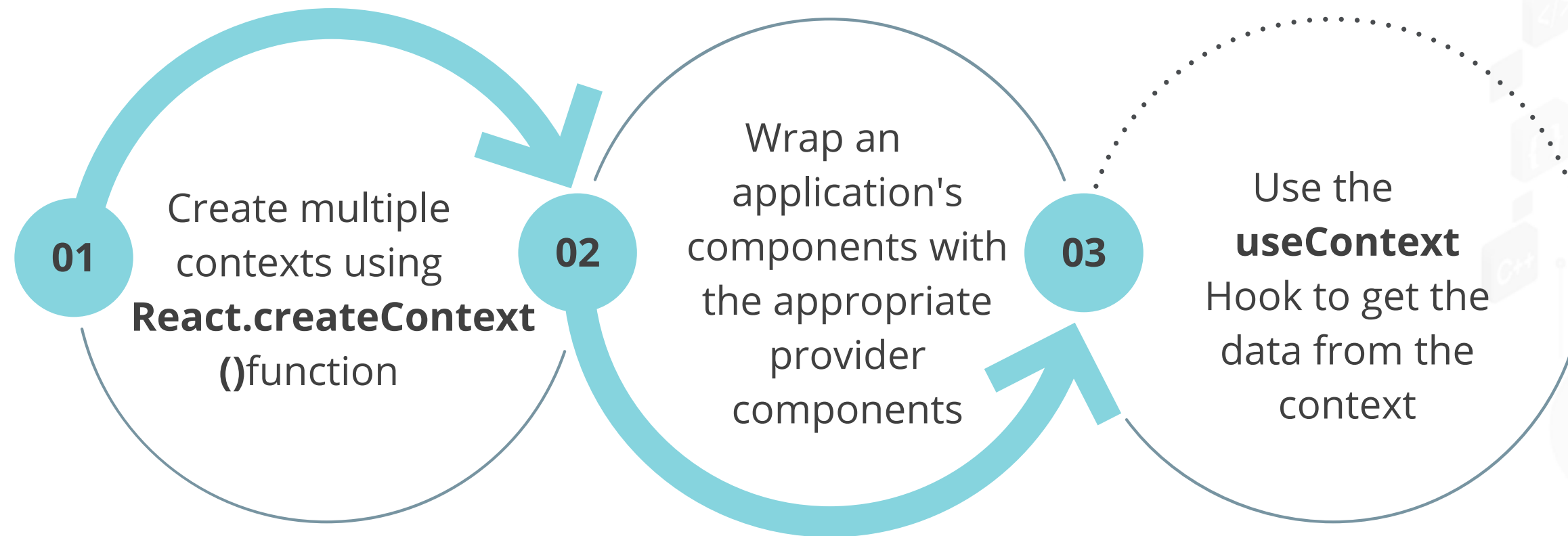
Using React Context and createContext: Example

```
//Create a component that provides the context
function App() {
  return(
    <MyContext.Provider value="Hello World">
      <MyComponent />
    </MyContext.Provider>
  );
}
export default App;
```



Using Multiple Contexts in a React Application

Multiple contexts is possible in React. To create this option, one can follow a certain set of steps:



Example

The following code showcases the usage of multiple contexts in a React application:

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

```
// Create the first context  
const ThemeContext = React.createContext('light');
```

```
// Create the first provider component  
function ThemeProvider(props) {  
  return (  
    <ThemeContext.Provider value="dark">  
      {props.children}  
    </ThemeContext.Provider>  
  );  
}
```



Example

```
// Create the second provider component
function LanguageProvider(props) {
  return (
    <LanguageContext.Provider value="spanish">
      {props.children}
    </LanguageContext.Provider>
  );
}
```



Example

```
// Create a component that uses both contexts
function MyComponent() {
  const theme = useContext(ThemeContext);
  const language = useContext(LanguageContext);

  return (
    <div>
      <p>Theme: {theme}</p>
      <p>Language: {language}</p>
    </div>
  );
}
```



Example

```
// Wrap the components with the provider components
function App() {
  return (
    <ThemeProvider>
      <LanguageProvider>
        <MyComponent />
      </LanguageProvider>
    </ThemeProvider>
  );
}
```



Demo with Context API



Duration: 10 Min.

Problem Statement:

You are given a project to create a **theme button** that toggles the dark mode on and off, using **context API**.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Open the project and create a new file called **ThemeContext.js**
2. Create a new context using **React.createContext()**
3. Create a new toolbar component and render it
4. Create a new file called **ThemeProvider.js** in the src directory
5. Update **App.js** to use the **ThemeProvider** component
6. Run the app by running **npm start** and open **http://localhost:3000**



Redux

Redux Introduction

Redux is a state container that provides predictability in state management. It has the following features:

Centralized State

Supports a single state tree

Predictable State Changes

Provides unidirectional data flow

Immutable State

Prevents direct changes

Middleware Support

Allows adding of custom logic

Redux Introduction

Redux provides a predictable and centralized data flow. It also makes debugging, testing, and maintaining code easy as it:

01

Manages complex states
in an application

02

Maintains a single source
of truth

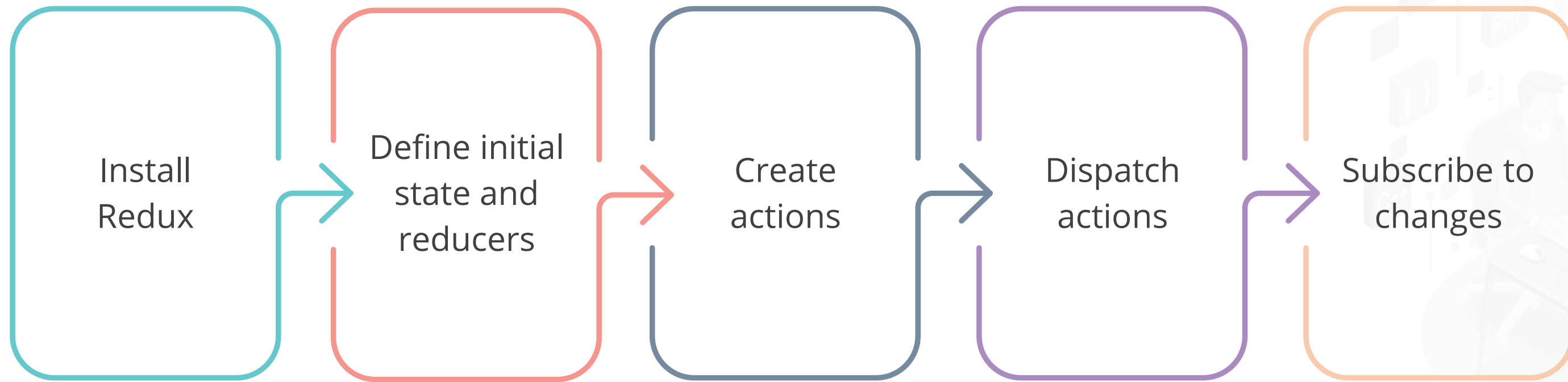
03

Debugs an application
efficiently



Getting Started with Redux

The following steps will help users in getting started with Redux:



Core Concepts

The following are Redux's core concepts:



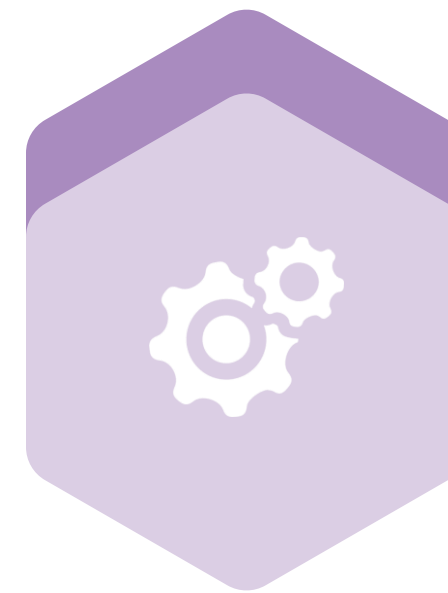
Store

Read-Only State



Actions

Dispatch



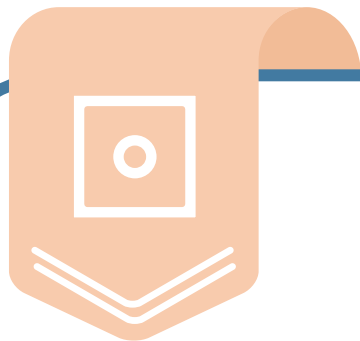
Reducer

Subscription



Redux: Three Principles

The principles that define Redux's core architecture are as follows:



Single Source of Truth

The application's state is stored in a single object tree within a store.



Changes Made with Pure Functions

Reducers specify how actions transform the state tree using pure functions that return new state objects.



Read-Only State

The state can only be changed by emitting actions, preventing direct state modification.

About Actions

In Redux, actions are JavaScript objects that describe an event.

An action object has two properties:



Type

A string that describes the action



Payload

An optional property containing additional data



About Actions

Action creator functions are used for creating actions.

These functions help to:



Actions

- Encapsulate logic for creating the actions
- Promote reusability of code and improve testability



About Reducers

Reducers are pure functions that handle the changes in the application state.

A **reducer function** takes two parameters:

**The
Previous
State**

An object representing the
current application state

**An
Action**

An object indicating the type
of modification required



About Reducers

The following are some features of reducers:

01

Generate identical results when given identical inputs

02

Handle multiple actions

03

Combine multiple reducers into a single root reducer

04

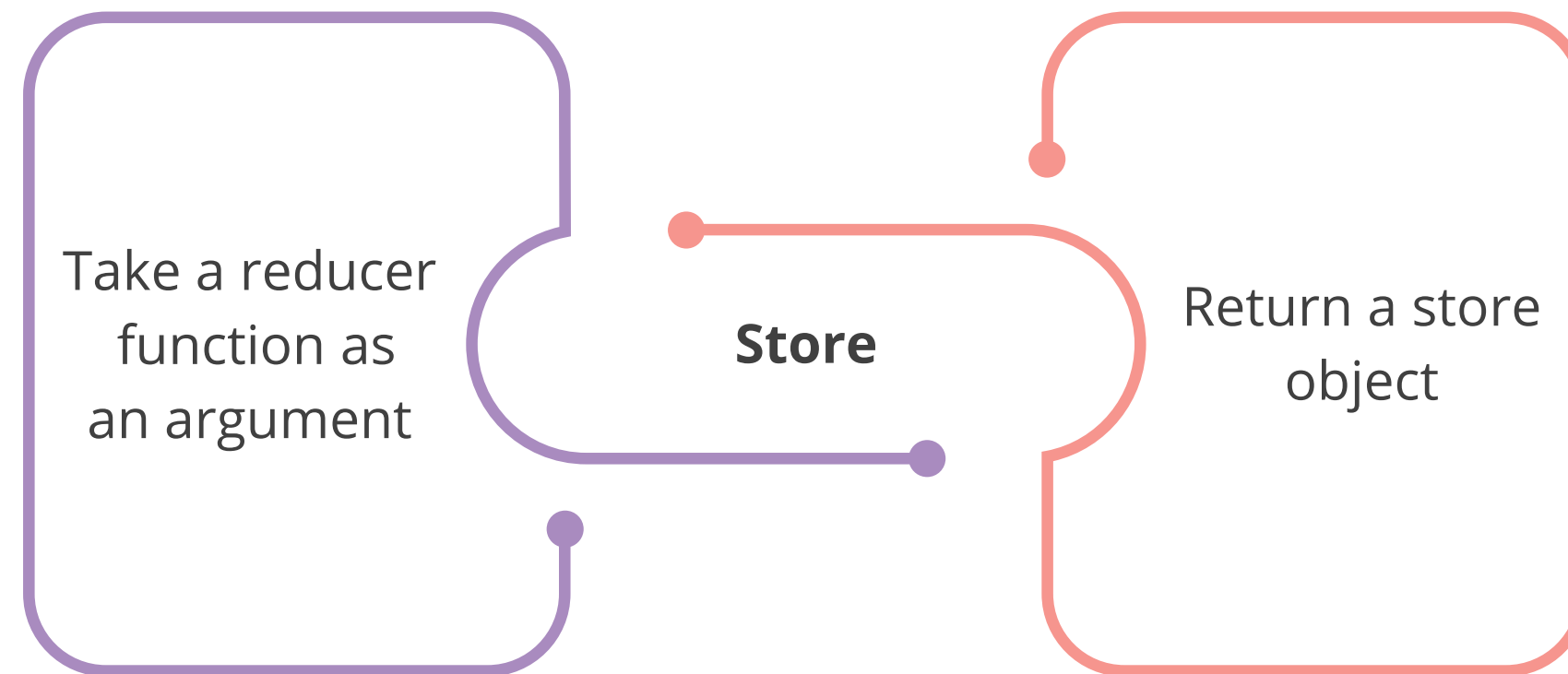
Ensure immutability



About Store

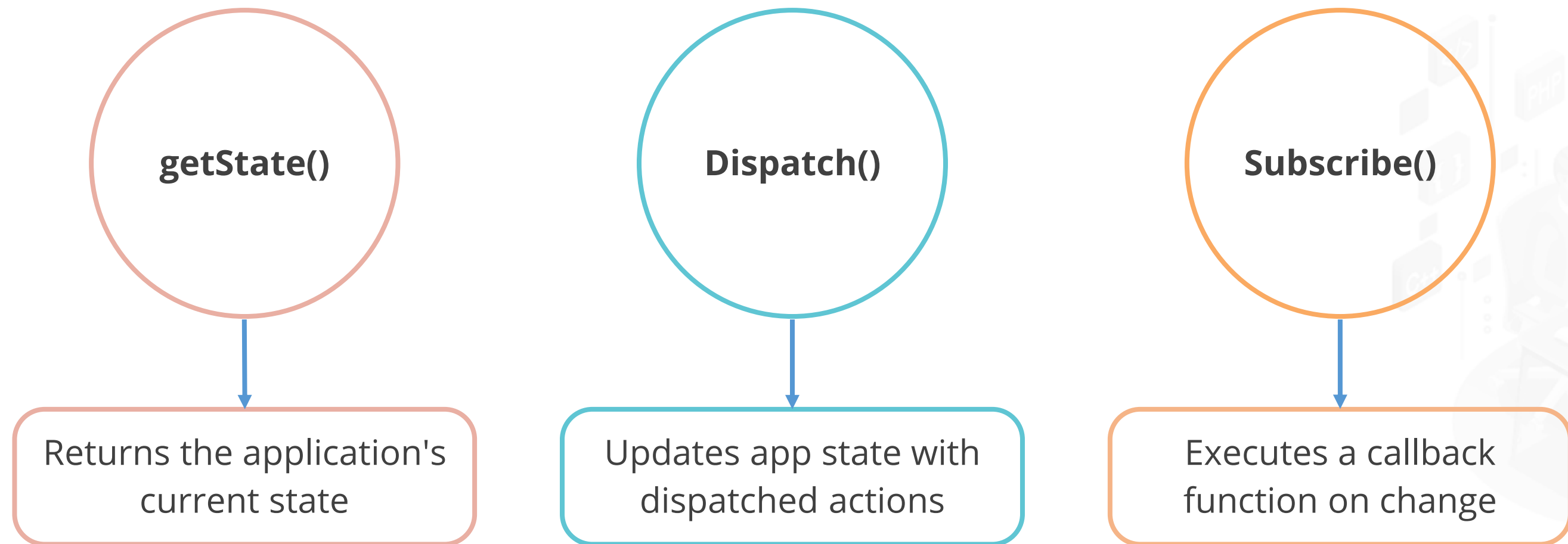
Stores are objects in Redux that hold the application's state.

To create a store, one can use the **createStore()** function, which can:



About Store

The store has three primary responsibilities:



Demo with Redux Principles



Duration: 20 Min.

Problem Statement:

You are given a project to create a React application showcasing **Redux's principle**.

ASSISTED PRACTICE

Assisted Practice: Guidelines

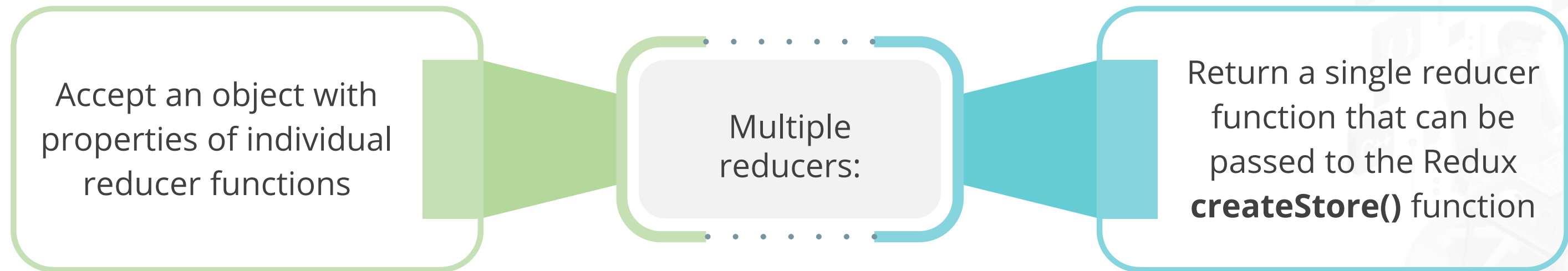
Steps to be followed:

1. Run **npx create-react-app redux-counter** to create a Redux-based counter React app
2. Set up the store in **store.js**
3. Create **Counter.js** and **App.js** components
4. Wrap App with Provider in **index.js**
5. Test the app



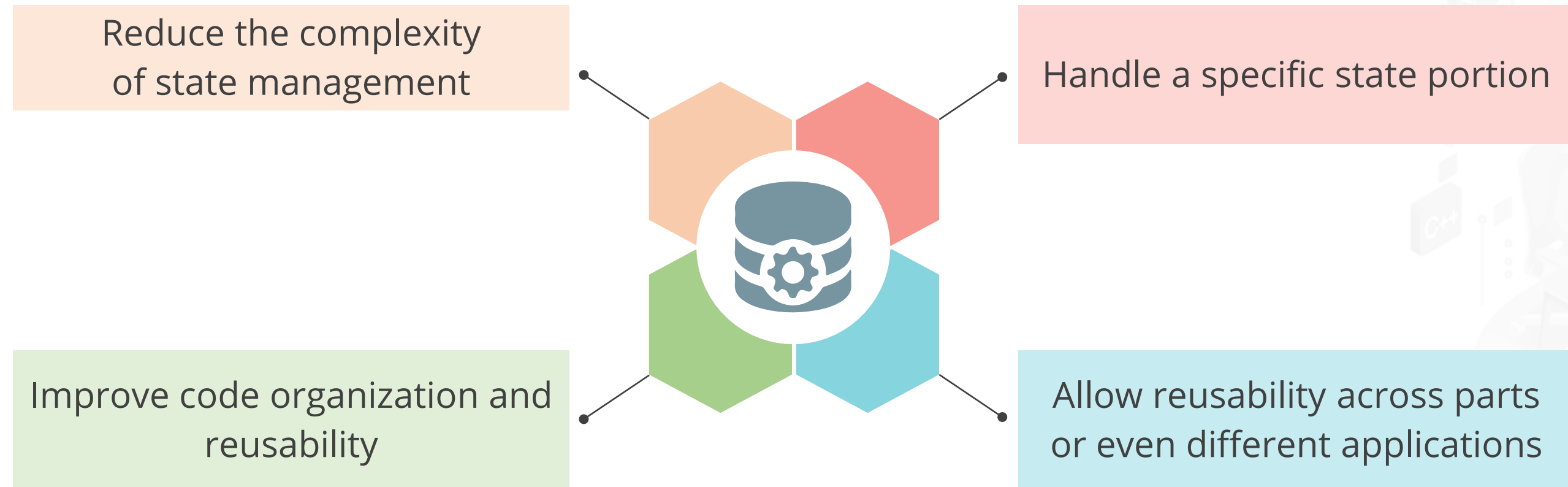
Multiple Reducers

Multiple reducers handle different parts of the state. They form a single root reducer using the **combineReducers()** function. They have the following functions:



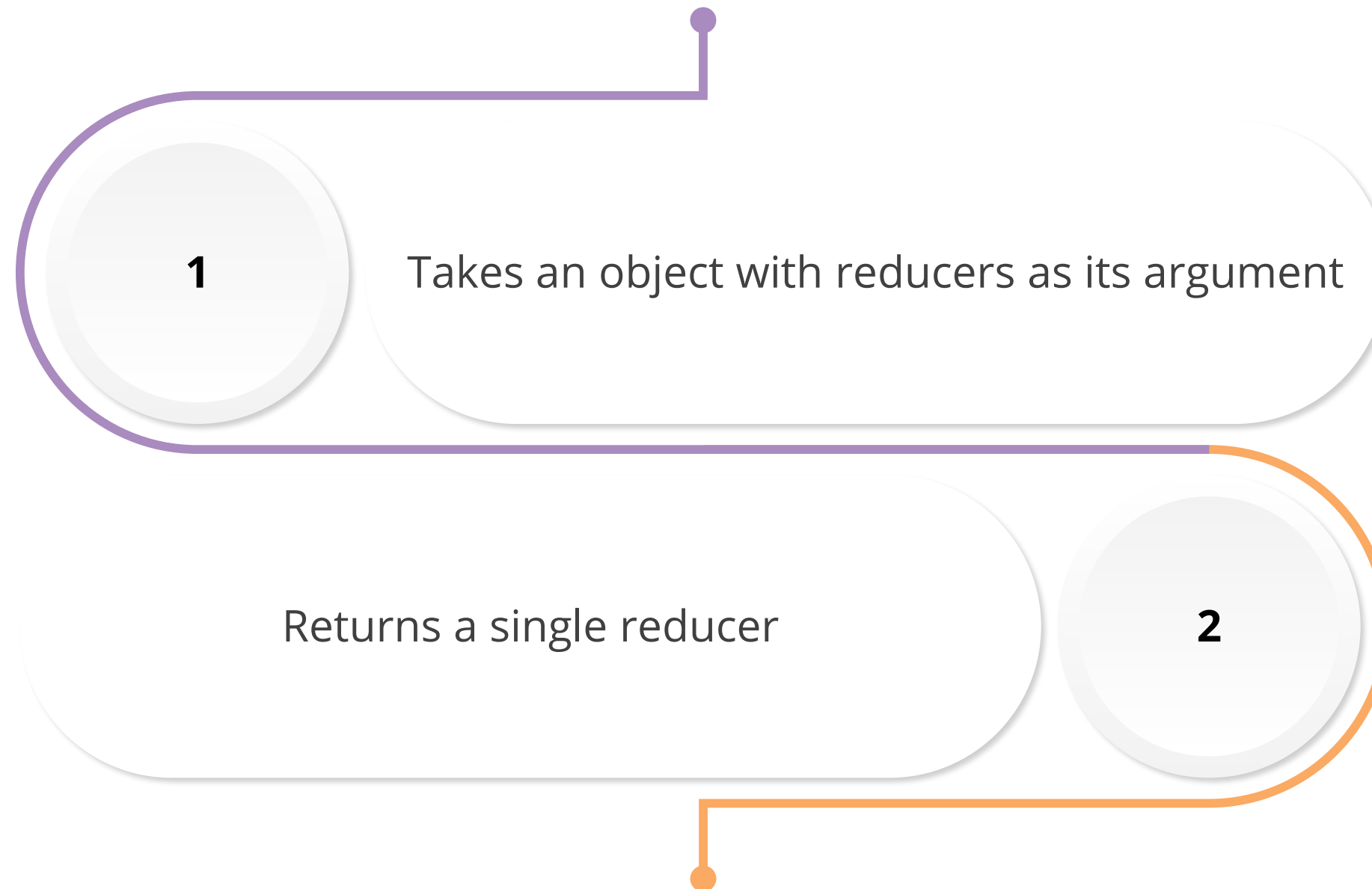
Multiple Reducers: Features

Multiple reducers have the following features:



Combine Reducers

Combine multiple reducers into a single reducer by using the **combineReducers** function. This function:



Combine Reducers: Benefits

Combine reducers are an efficient way to manage state in Redux applications as they:

Manage complex applications with multiple state properties

Handle a specific part of the state tree

Provide a structured approach to state management

Improve code organization and maintainability

Enable modular and reusable reducers

Demo with Combine Reducers



Duration: 15 Min.

Problem Statement:

You are given a project to develop a simple **To-do list** React application to demonstrate the use of **combineReducers**.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new React app with **combined reducers**
2. Install **dependencies**
3. Create the **Todolist.js** and **AddTodo.js components**, and connect them to **Redux**
4. Setup **rootReducer** and create the **Redux store**
5. Run the app

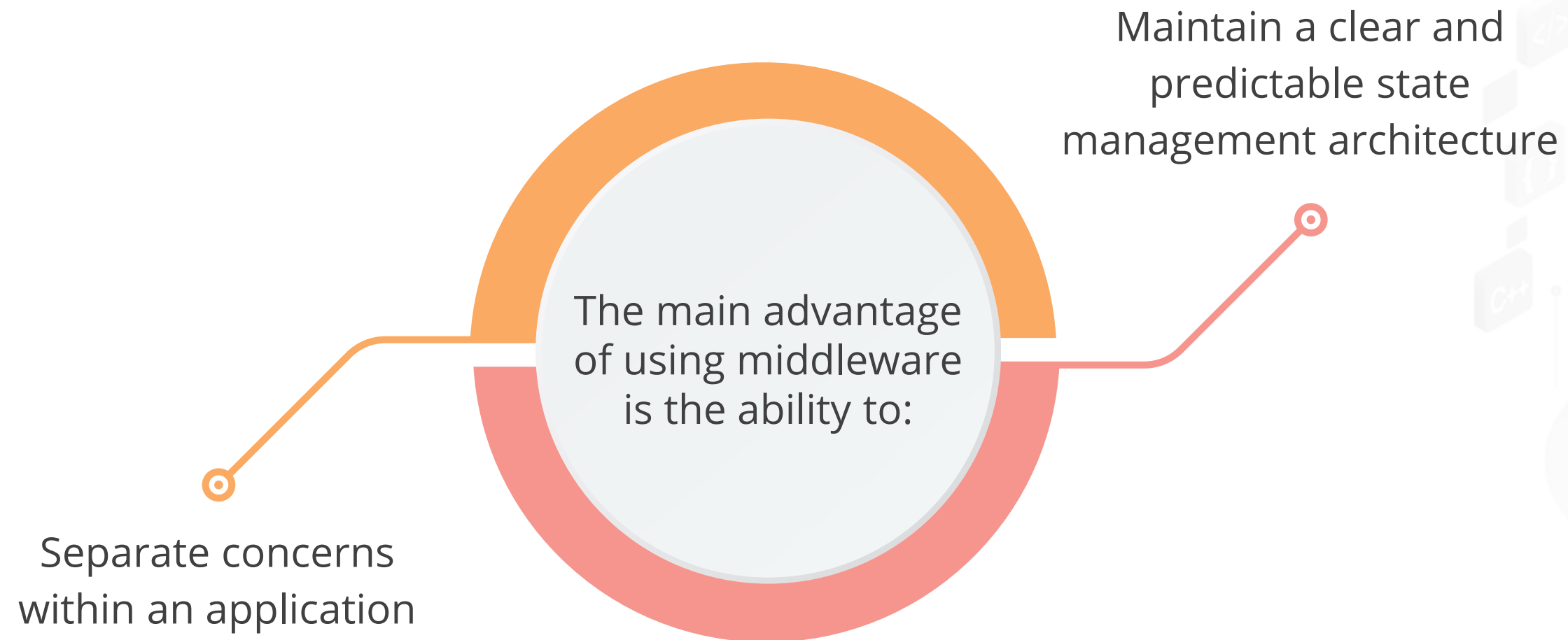


Data Retrieval Using Redux



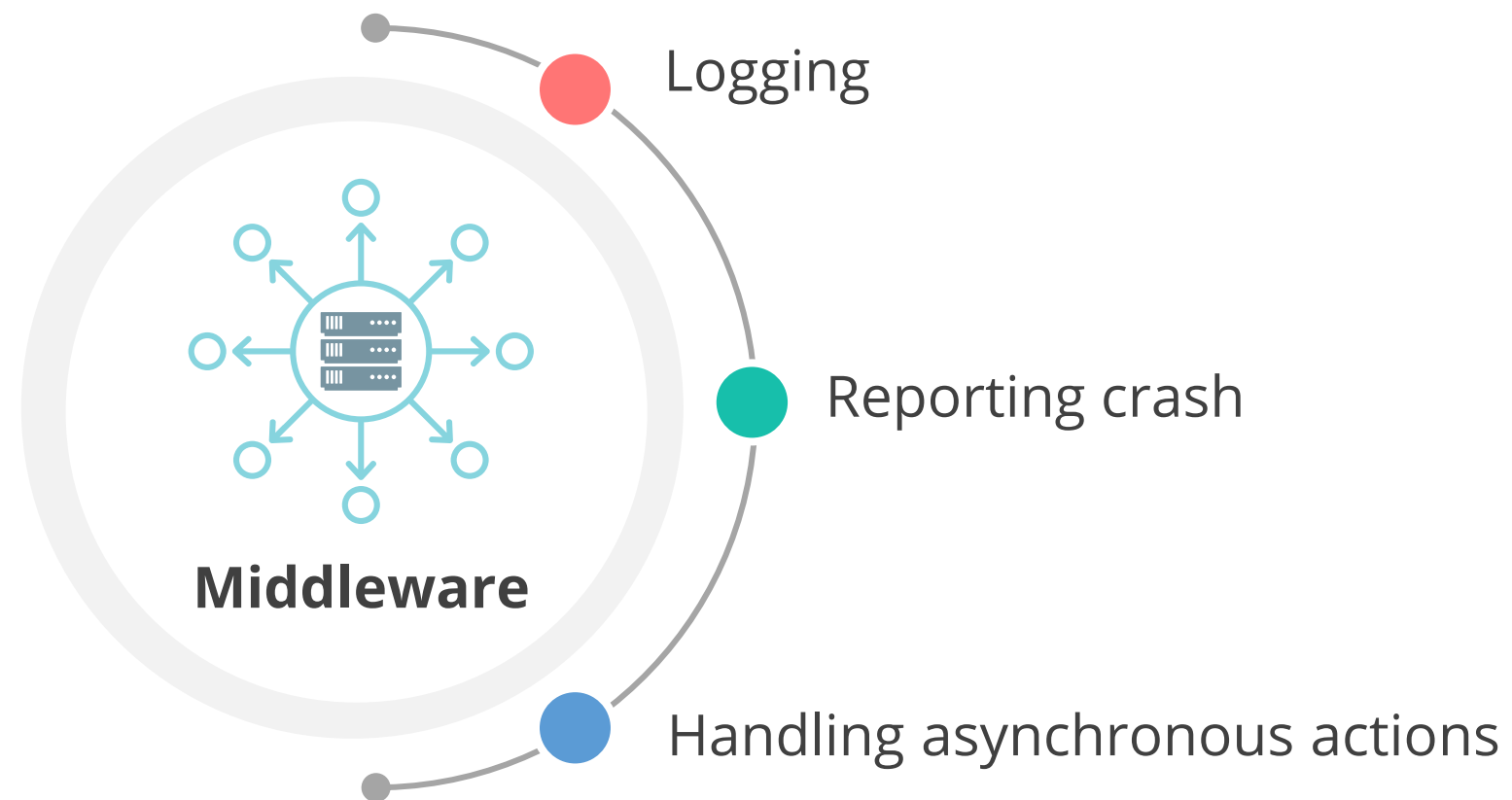
About Middleware

Middleware is software that lies between the action creators and the reducers.



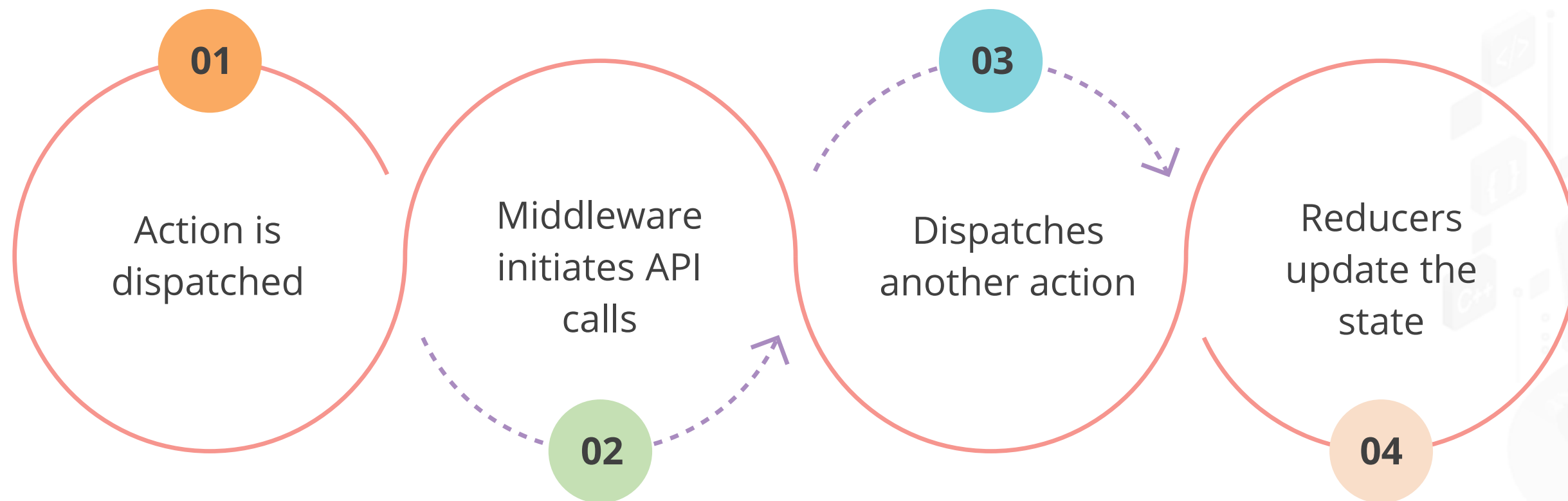
About Middleware

There are several applications for middleware, including:



About Middleware

Middleware can manage asynchronous tasks such as data fetching from an API. The steps in this process are given below:



Keep the application code more modular and maintainable while performing data retrieval in Redux.

About Async Actions

Async actions involve three different action types:

Request

An initial action is performed to signal the start of the process.

Success

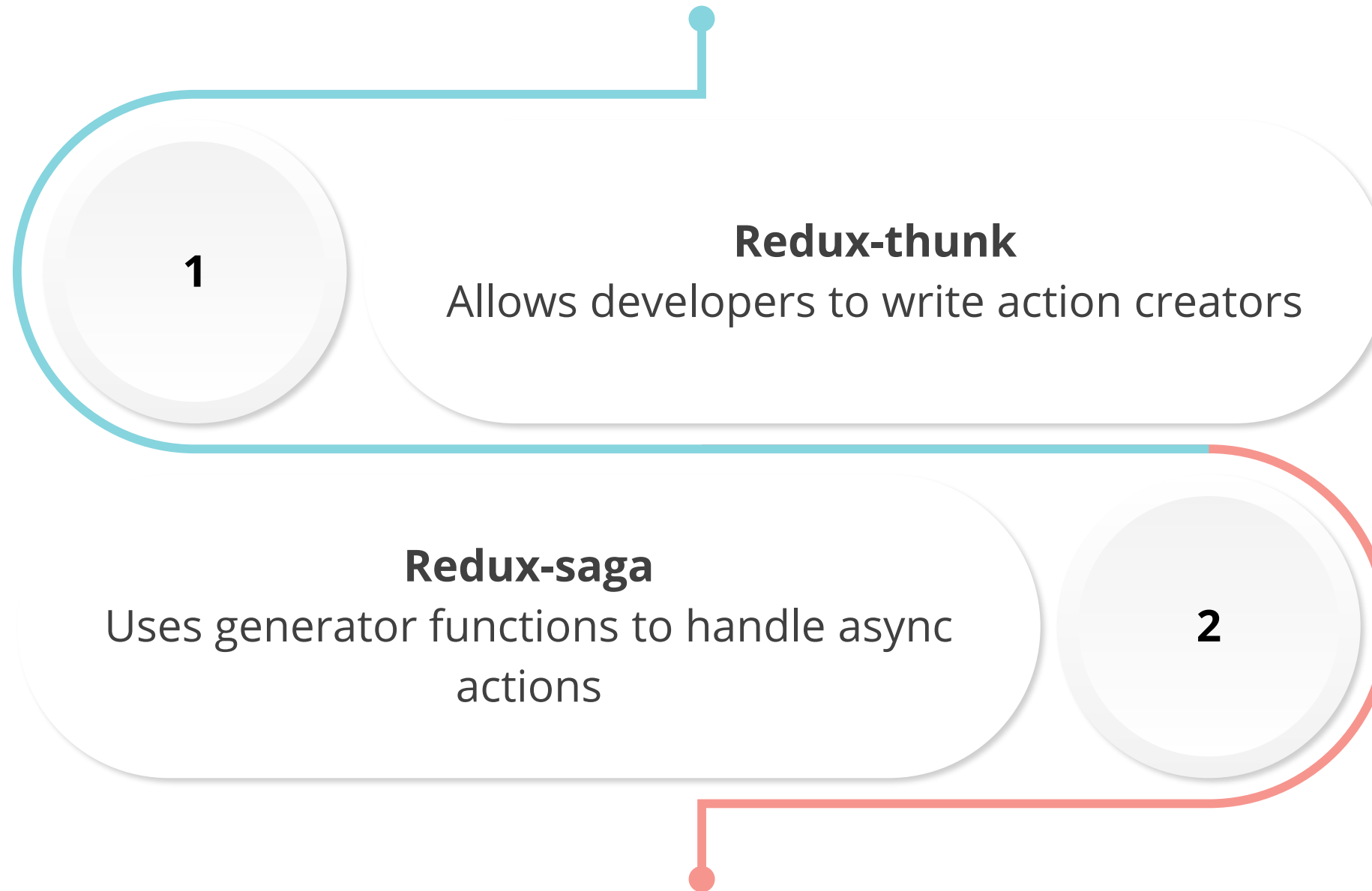
A success action is dispatched with the payload.

Failure

An error action is dispatched with an error message.

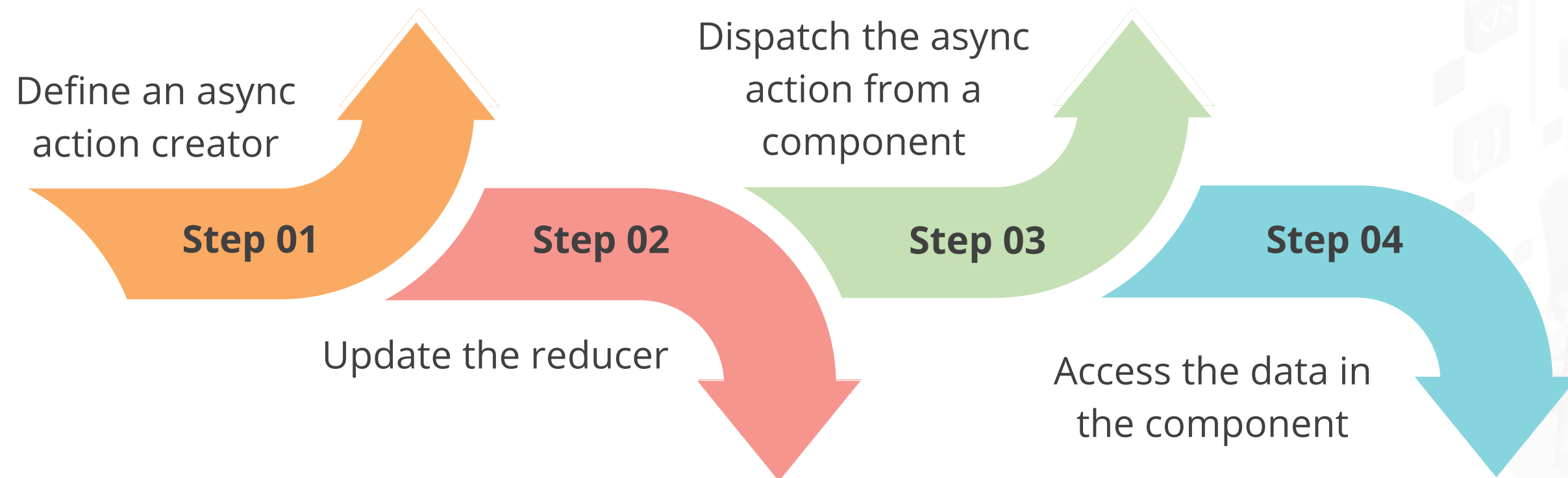
About Async Actions

To handle async actions in Redux, use the following middleware:



Asynchronous Data Retrieval

To retrieve asynchronous data in React using Redux, one can follow these steps:



Asynchronous Data Retrieval

Developers can effectively manage asynchronous data retrieval in React by using:

Asynchronous action
creators and
reducers

useEffect Hook

useSelector Hook

useDispatch Hook

Implementing Asynchronous Data Retrieval with Redux

The following code defines action types and action creators related to data retrieval:

actions.js

```
// actions.js
export const FETCH_DATA_REQUEST =
  'FETCH_DATA_REQUEST';
export const FETCH_DATA_SUCCESS =
  'FETCH_DATA_SUCCESS';
export const FETCH_DATA_FAILURE =
  'FETCH_DATA_FAILURE';

export const fetchDataRequest = () => {
  return {
    type: FETCH_DATA_REQUEST
  }
}
```

```
export const fetchDataSuccess = (data)
=> {
  return {
    type: FETCH_DATA_SUCCESS,
    payload: data
  }
}
export const fetchDataFailure = (error)
=> {
  return {
    type: FETCH_DATA_FAILURE,
    payload: error
  }
}
```

Implementing Asynchronous Data Retrieval with Redux

The following code contains the reducer function responsible for managing the state related to data retrieval:

reducer.js

```
// reducer.js
import { FETCH_DATA_REQUEST, FETCH_DATA_SUCCESS, FETCH_DATA_FAILURE } from '../actions';
const initialState = {
  loading: false,
  data: [],
  error: ''
}
const reducer = (state = initialState, action) => {
  switch(action.type) {
    case FETCH_DATA_REQUEST:
      return {
        ...state,
        loading: true
      }
  }
}
```

Implementing Asynchronous Data Retrieval with Redux

Here is the remaining part of the code:

```
case FETCH_DATA_SUCCESS:
  return {
    loading: false,
    data: action.payload,
    error: ''
  }

case FETCH_DATA_FAILURE:
  return {
    loading: false,
    data: [],
    error: action.payload
  }
default: return state
}}
export default reducer;
```



Implementing Asynchronous Data Retrieval with Redux

The code shows how to use the Redux store and actions in a React component.

component.js

```
// component.js
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { fetchDataRequest, fetchDataSuccess, fetchDataFailure } from './actions';
const Component = () => {
  const dispatch = useDispatch();
  const data = useSelector(state => state.data);
  const error = useSelector(state => state.error);
  const loading = useSelector(state => state.loading);
  useEffect(() => {
    dispatch(fetchDataRequest());
    fetch('url-to-data')
```

Implementing Asynchronous Data Retrieval with Redux


Here is the remaining part of the code:

```
.then(response => response.json())
  .then(data => {
    dispatch(fetchDataSuccess(data));
  })
  .catch(error => {
    dispatch(fetchDataFailure(error.message));
  });
}, []);
return (
  <>
    {loading ? <p>Loading...</p> : null}
    {error ? <p>{error}</p> : null}
    {data.map(item => (
      <p key={item.id}>{item.title}</p>
    ))}
  </>
)
export default Component;
```



Redux Middleware for Async Operations

Middleware in Redux empowers developers to:



Enhance the capabilities of the dispatch function

Prevent actions before they reach reducers

Handle asynchronous operations

Increase state management control and flexibility

Demo with Asynchronous Data Retrieval Using Redux



Duration: 25 Min.

Problem Statement:

You are given a project to build a simple Redux-based weather app that displays the current weather of a city by asynchronously fetching data from an API.

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new React app by running **`npx create-react-app redux-weather-app`**
2. Install dependencies
3. Create the following files: **`reducers.js`, `types.js`, `actions.js`, `weather.js`, `WeatherForm.js`, `App.js`, `index.js`, and `env`**
4. Update the **`axios.get URL`** in `actions.js`
5. Run the app



Key Takeaways

- React Context API allows components to share data without using props.
- Context API should be used when data needs to be shared across multiple levels of a component tree.
- Functional components are JavaScript functions that return JSX to render UI elements.
- Redux is a state container that provides predictability in state management.
- Redux includes actions, reducers, stores, middleware, and async actions.



Demo with Redux Store

Duration: 30 Min.

Project Agenda:

Create a simple React application that manages state using a Redux store

Description:

The specific requirement is to have a counter that can be incremented or decremented through the store.

Perform the following:

1. Create a new React app using create-react-app
2. Open the project and install Redux and React Redux
3. Create the Redux store
4. Create a simple counter that can be incremented or decremented
5. Provide an app access to the Redux store



TECHNOLOGY

Thank You