Design a Dynamic Frontend with React

React Context API and Redux

Engage and Think

Imagine you are managing a large e-commerce site like Amazon. Each department's orders, user profiles, cart, and product listings have their own systems, but they all need to stay in sync. For example, if a user updates their shipping address, the cart, checkout, and orders must all reflect that change instantly without confusion or delay.

How would you manage and share such connected information across different parts of an application without repeating yourself or creating chaos?

Learning Objectives

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

- Explore the differences between React Context API and Redux for state management
- Create a centralized state management structure using React Context API
- Develop a React application implementing Redux for managing complex state
- Demonstrate the flow of data through Context and Redux with component interactions
- Deploy a sample project using Redux Toolkit to streamline Redux setup
- Evaluate the performance and scalability differences between Context API and Redux in real-world scenarios

Introduction to React Context API

What Is React Context API?

It is a centralized method for storing data and makes that data accessible to all components within an application

React Context API

The React Context API also allows components to share data across the application without the need to manually pass properties (props).

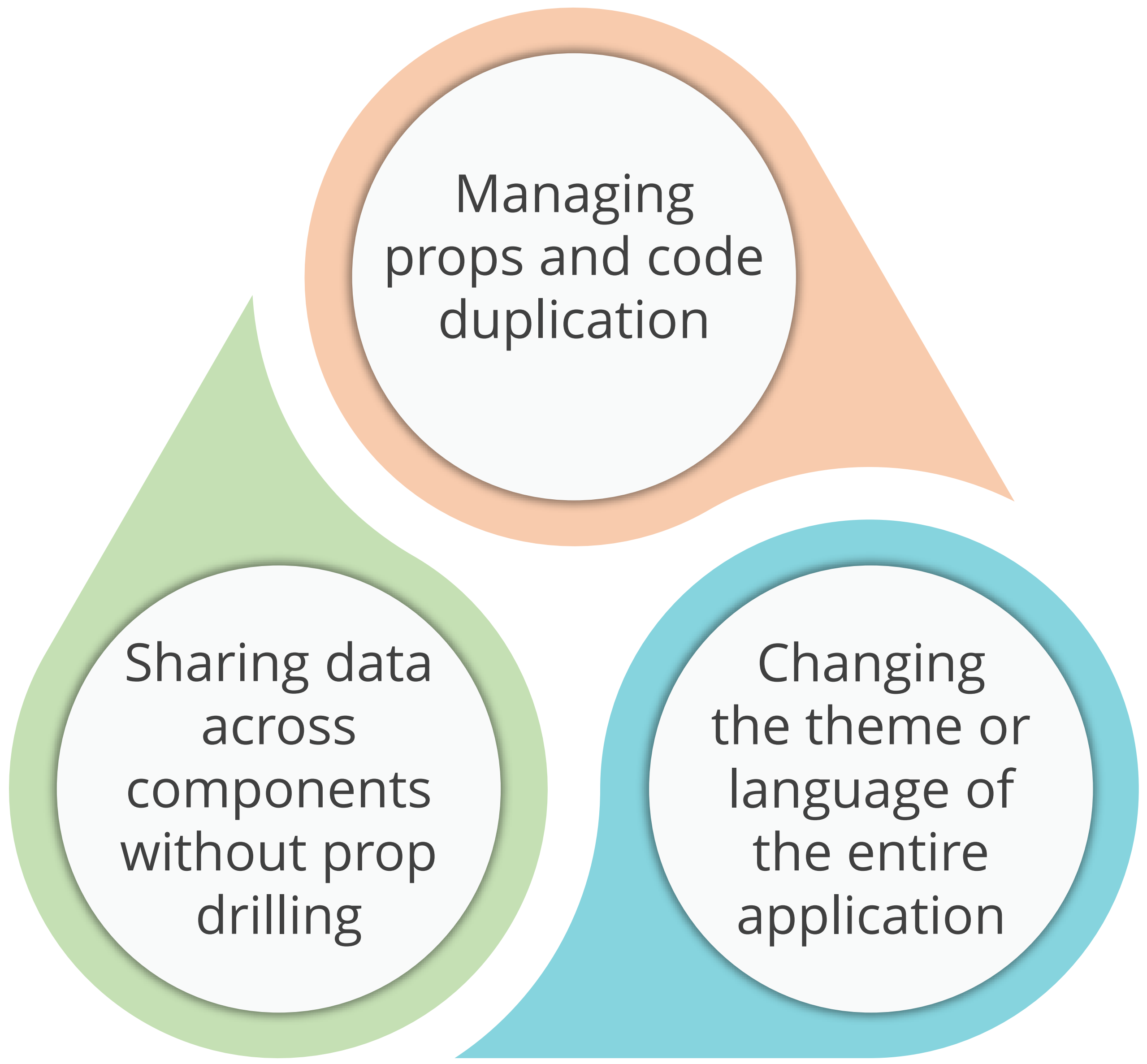
Context API: Applications

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:


Managing props and code duplication
Sharing data across components without prop drilling
Changing the theme or language of the entire application

Creating a Context in React

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

1. Create a context using the **React.create Context()** method
2. Use the **provider** and **consumer** components
3. Access the data from the provider using the **this.context** property

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:

The Provider

- Accepts a value prop
- Wraps components that need data access

The Consumer

- Accesses data within the component
- Allows components to access the data

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

Using Class Components: Example

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

Example

```
// 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

Using Class Components: Example

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

Example

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

Wrapping the component

Using Class Components: Example

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

Example

```
// 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 component generates and 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:

Simplicity
Offer a simplified way to define components

Readability
Promote clean and concise code

Performance
Enhance the rendering speed of the application

Using React Context and contextType: Example

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

Example

```
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 contextType: 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 are possible in React. To create this option, one can follow a certain set of steps:

01 Create multiple contexts using **React.createContext ()**function

02 Wrap an application's components with the appropriate provider components

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

Using Multiple Contexts: Example

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

Example

```
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>
  );
}
```

Using Multiple Contexts: Example

Example

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

Using Multiple Contexts: Example

Example

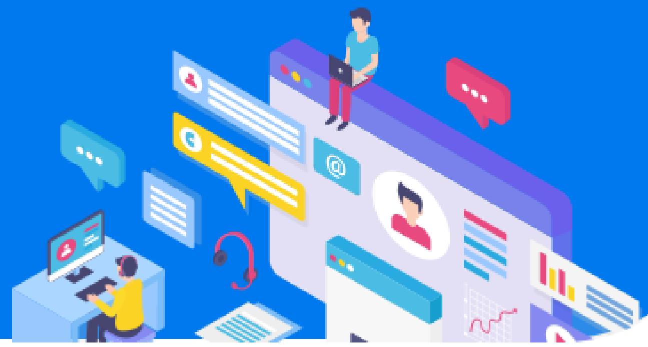
```
// Create a component that uses both contexts
function MyComponent() {
  const theme = useContext(ThemeContext);
  const language = useContext(LanguageContext);

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

Using Multiple Contexts: Example

Example

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

Assisted Practice



Creating a Theme Button Using Context API **Duration: 10 Min.**

Problem Statement:

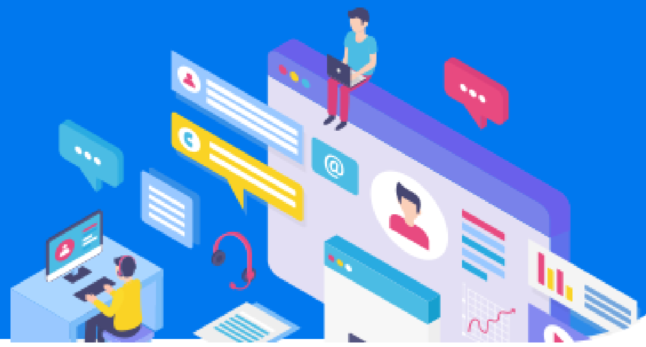
You are tasked to implement a theme button using the Context API in React, providing a centralized theme management system for your application.

Outcome:

By the end of this demo, you will be able to create a functional theme toggle button using React Context API to manage and apply light or dark themes across your application.

Note: Refer to the demo document for detailed steps:
01_Creating_a_Theme_Button_Using_the_Context_API

Assisted Practice: Guidelines



Steps to be followed:

1. Create a new React app
2. Implement the Toolbar.js function
3. Wrap the Toolbar component in a ThemeProvider component
4. Update App.js to use the ThemeProvider component
5. Run the app

Quick Check

You are developing a React application in which multiple components need access to user authentication data (for example, logged-in user info and authentication status). The data should be accessible without prop drilling while ensuring efficient state updates. Which of the following is the best approach?

A. Pass authentication data as props from a parent component to child components

B. Use the React Context API with a provider wrapping the necessary components

C. Store authentication data in a local state within each component that needs it

D. Use the this.context property in all components without wrapping them in a provider

Getting Started with Redux

What Is Redux?

It 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: Benefits

Redux provides a predictable and centralized data flow. It also makes debugging, testing, and maintaining code easy as it is, below are the key benefits:

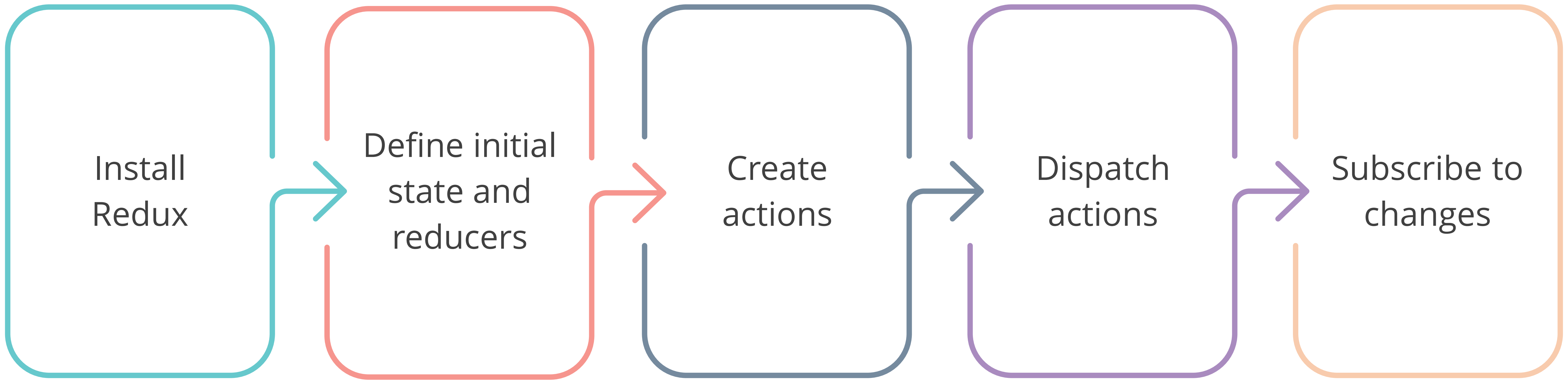
01 Manages complex states in an application

02 Maintains a single source of truth

03 Debugs an application efficiently

Setting up Redux

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



Redux: Three Principles

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

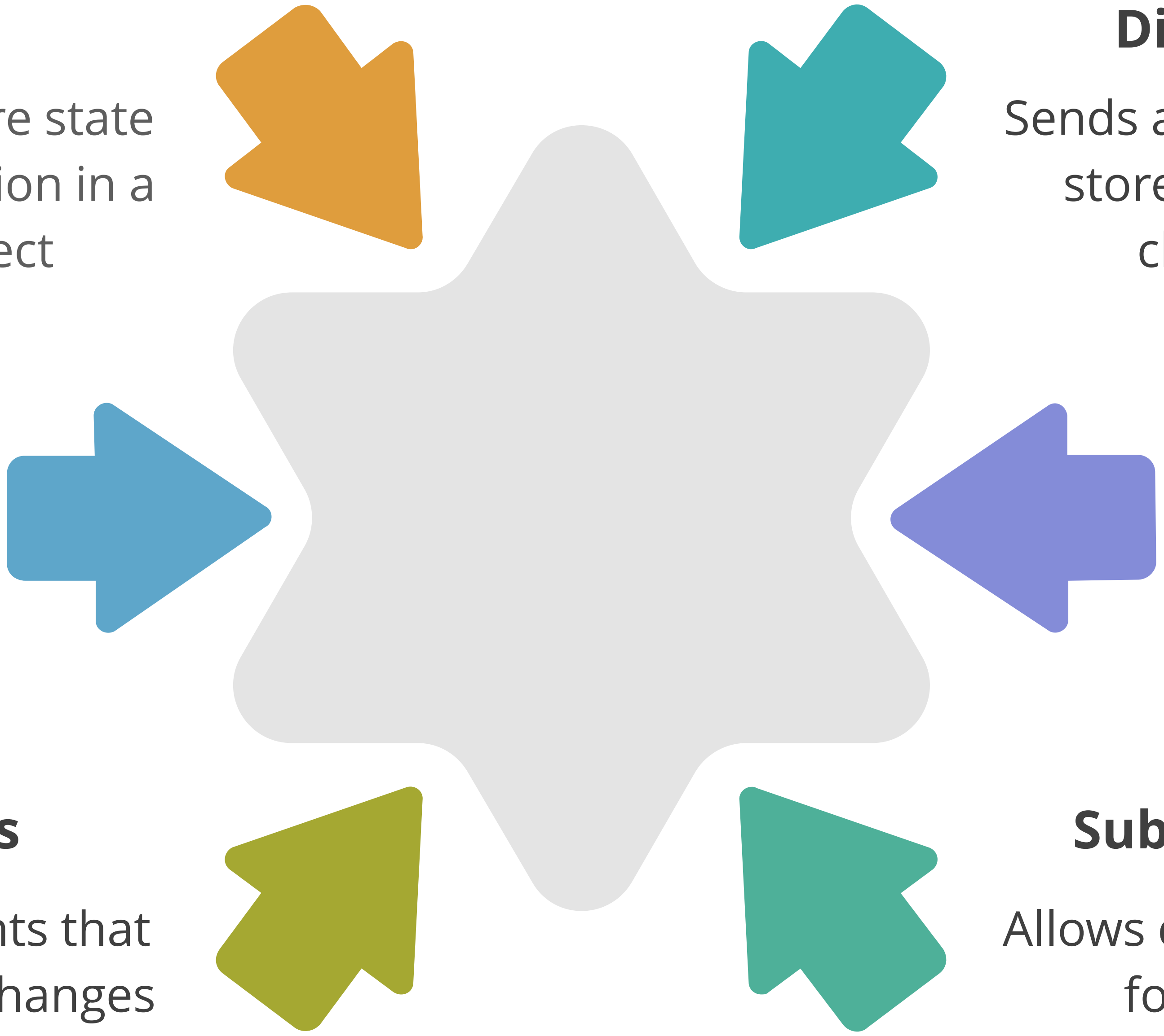
Single source of truth
The application's state is stored in a single object tree within a store.

Changes made with pure functions
The state is updated by reducers, which are pure functions that return new state objects based on the dispatched actions.

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

Redux: Core Concepts

The following are Redux's core concepts:



Store

Holds the entire state of the application in a single object

Dispatch

Sends actions to the store to trigger changes

Read-only state

Stores data that can only be updated through actions

Reducer

Determines how the state changes in response to actions

Actions

Describe events that trigger state changes

Subscription

Allows components to listen for state changes

Redux 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

Redux Action Creator Function

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

Redux Reducers

In Redux, reducers are pure functions that handle state logic, accepting the initial state and action type to update and return the state. Applications can have multiple reducers.

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

Multiple Reducers

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

- They accept an object with properties of individual reducer functions.
- They return a single reducer function that can be passed to the Redux **createStore()** function.

Multiple Reducers: Features

Multiple reducers have the following features:

- Reduce the complexity of state management
- Handle a specific state portion
- Improve code organization and reusability
- Allow reusability across parts or even different applications

Combine Reducers

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

1. Takes an object with reducers as its argument
2. Returns a single reducer

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

Redux Reducers: Features

The following are some features of reducers:

01 Generate identical results when given identical inputs

02 Handle multiple actions

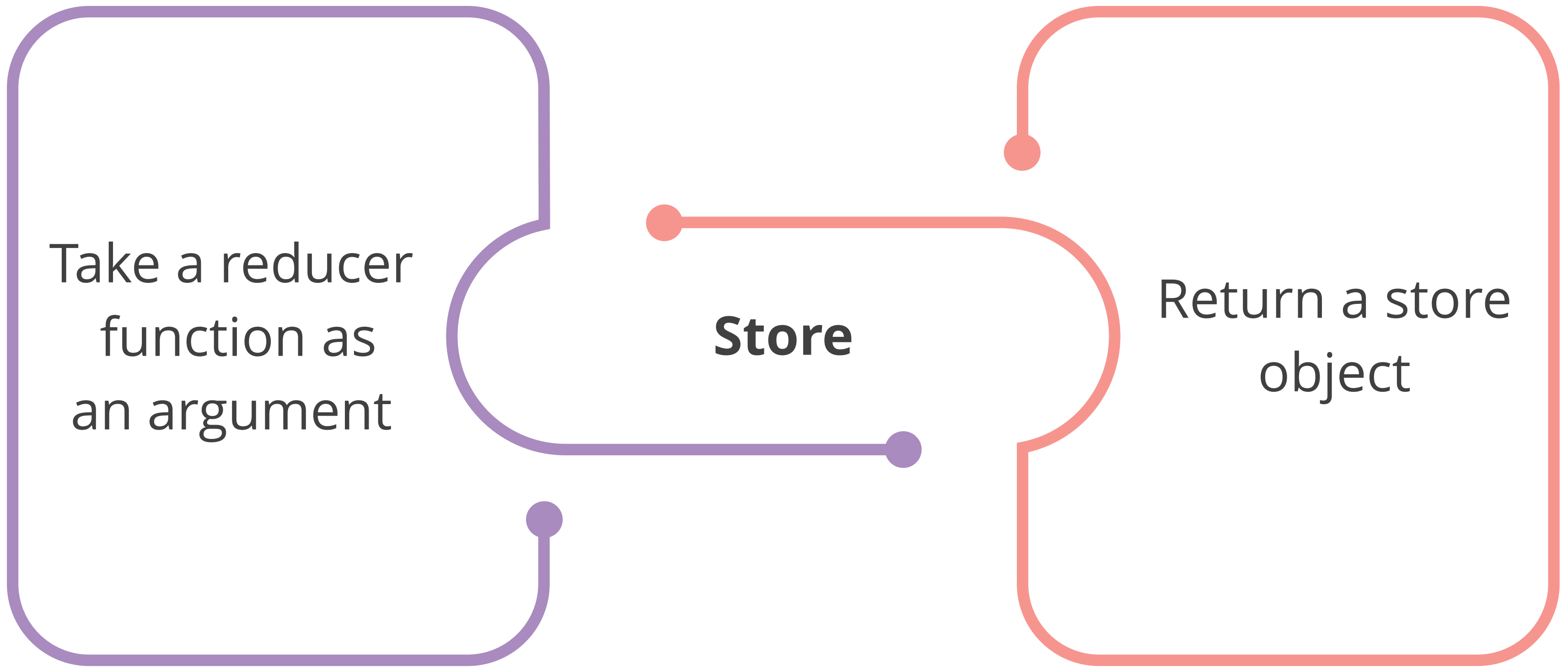
03 Combine multiple reducers into a single root reducer

04 Ensure immutability

Redux Store

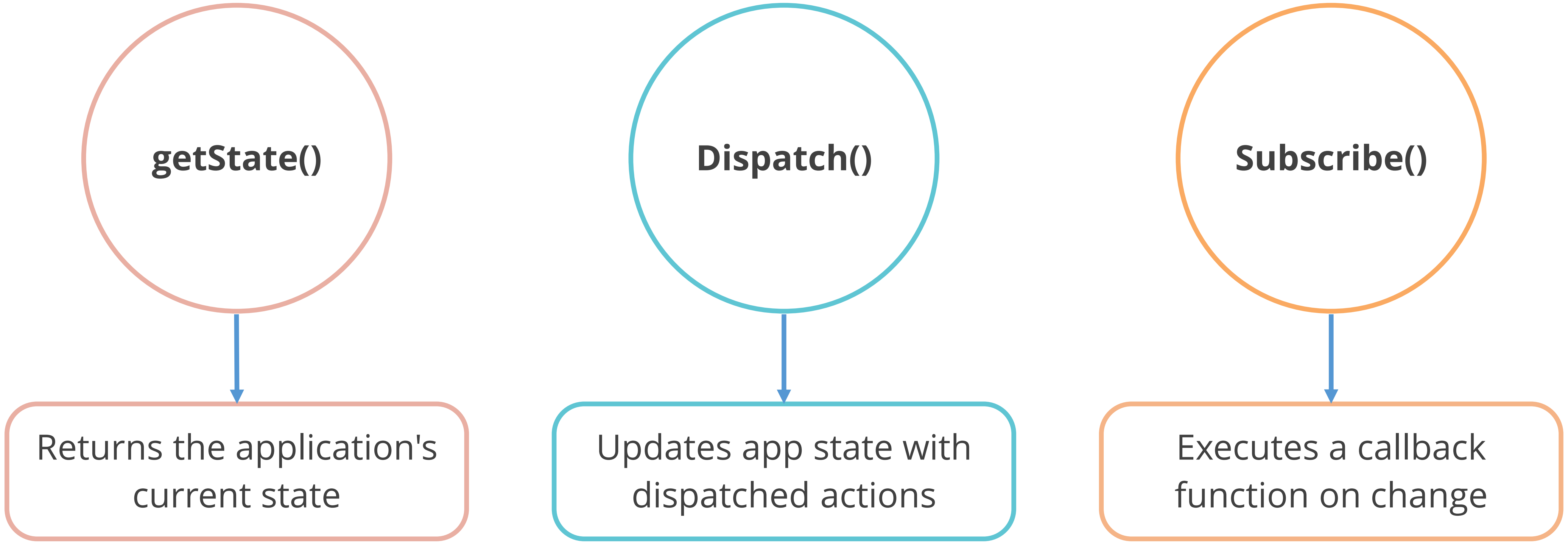
Stores are objects in Redux that hold the application's state.

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

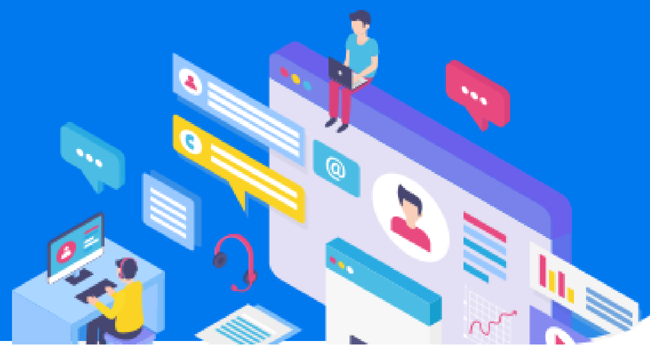


Redux Store

The store has three primary responsibilities:


getState()
Returns the application's current state
Dispatch()
Updates app state with dispatched actions
Subscribe()
Executes a callback function on change

Assisted Practice



Creating a React Application Using the Principles of Redux

Duration: 10 Min.

Problem Statement:

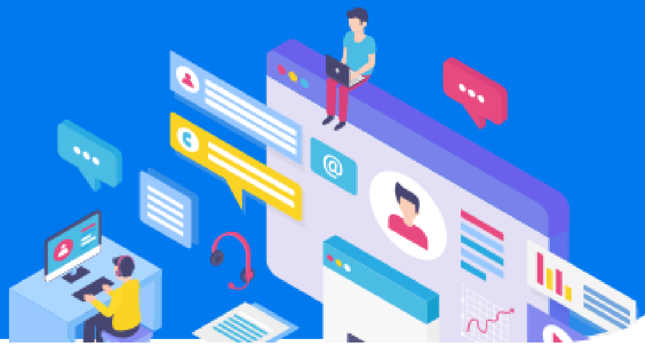
You are given a project to develop a React application that demonstrates the principle of Redux.

Outcome:

By the end of this demo, you will be able to build a React application that uses Redux to manage state efficiently across components.

Note: Refer to the demo document for detailed steps:
02_Creating_a_React_Application_Using_the_Principles_of_Redux

Assisted Practice: Guidelines



Steps to be followed:

1. Create a new React app
2. Create a new file called store.js
3. Open the existing file called App.js
4. Wrap the App component with the Provider component
5. Run the application

Assisted Practice

Creating a React Application Using Combine Reducer

Duration: 10 Min.

Problem Statement:

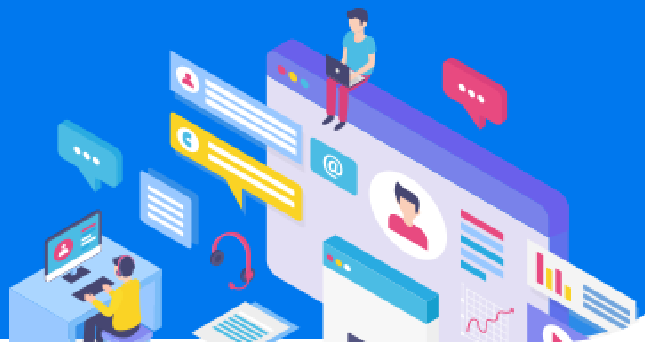
You are given a project to develop a React application that demonstrates the use of combined reducers.

Outcome:

By the end of this demo, you will be able to structure a React application using multiple reducers and integrate them using **combineReducer** for better state management.

Note: Refer to the demo document for detailed steps:
03_Creating_a_React_Application_Using_Combine_Reducers

Assisted Practice: Guidelines



Steps to be followed:

1. Create a new React app
2. Create a new file called reducers.js
3. Create a new file called AddTodo.js
4. Import the rootReducer from reducers.js into the index.js file
5. Run the application and view it in the browser

Quick Check

You are developing a large-scale e-commerce application where multiple components need access to cart data, user authentication status, and theme preferences. To maintain a single source of truth and ensure efficient state management, which approach should you use?

A. Store state locally and pass it as props

B. Use Redux with a centralized store, actions, and reducers

C. Store all state in React's Context API without reducers

D. Modify state directly inside components

Accessing Data via Redux

Data Retrieval Using Redux

It is the process of fetching data from an external source (such as an API or database) and managing it efficiently using Redux, a state management library for JavaScript applications.

When dealing with asynchronous data retrieval, such as fetching data from an API, Redux ensures that the fetched data is stored, managed, and updated in a centralized manner.

Data Retrieval Using Redux: Components

Redux centrally manages the state to streamline data retrieval and updates across components. The key components are:

Middleware
Async actions

Middleware

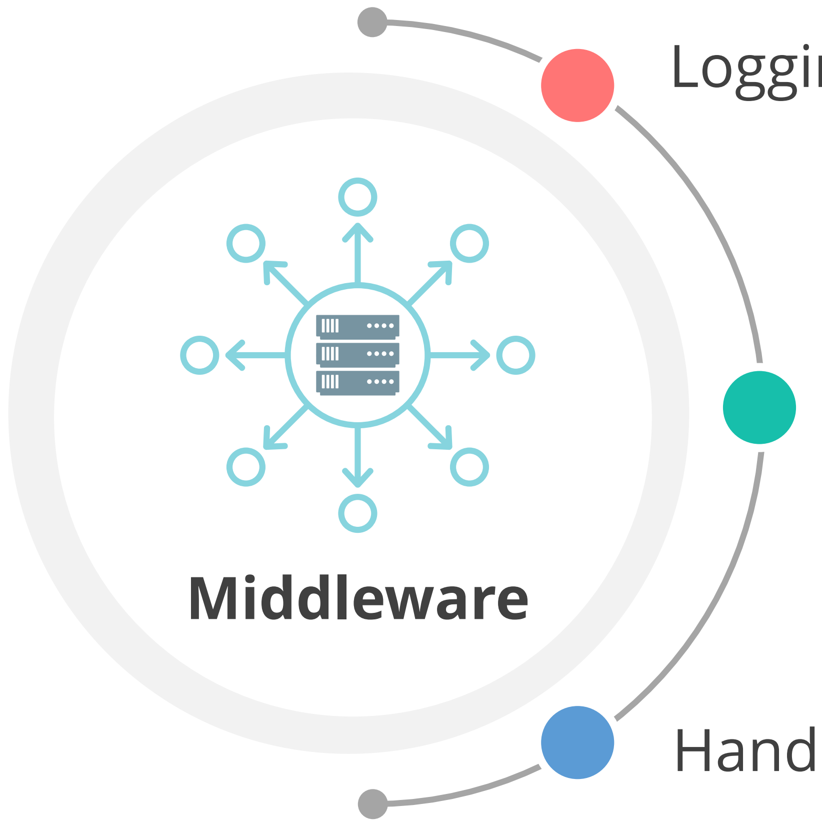
It is a software that lies between the action creators and the reducers.

The main advantage of using middleware is to:

- Maintain a clear and predictable state management architecture
- Separate concerns within an application

Middleware: Applications

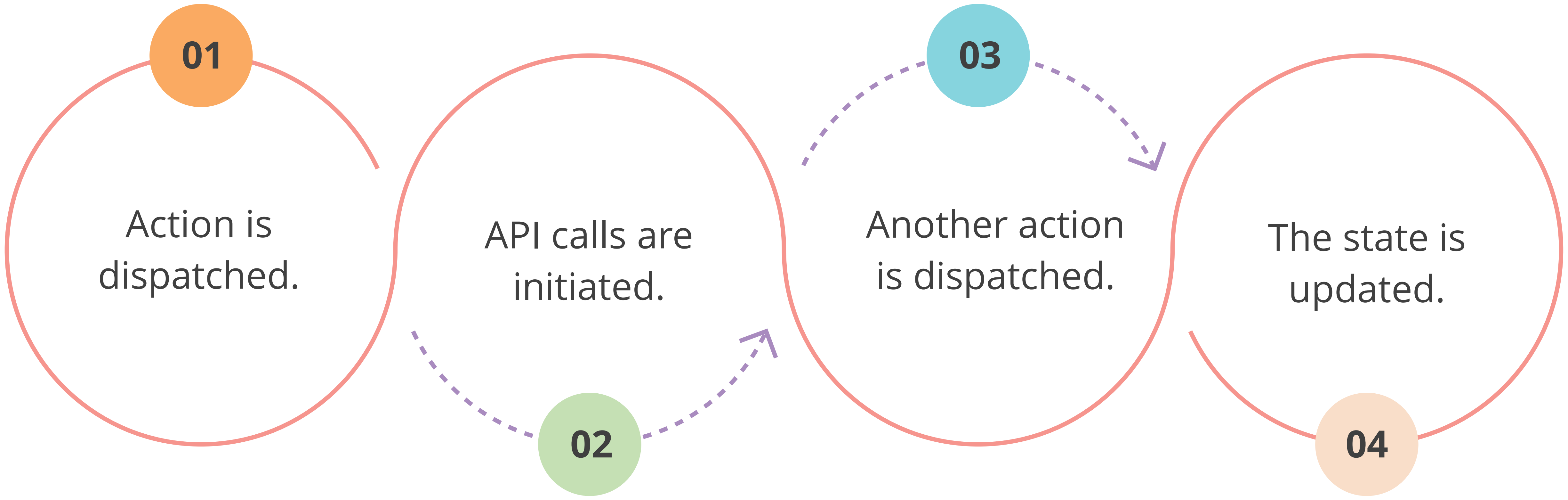
There are several applications for middleware, including:



- Logging action
- Reporting crash
- Handling asynchronous actions

Middleware: Steps

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



Note:

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

Async Actions

Redux uses asynchronous actions to retrieve data from external APIs, databases, or any other external data source. These actions involve an asynchronous operation that takes time to complete and return data.

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.

Async Actions

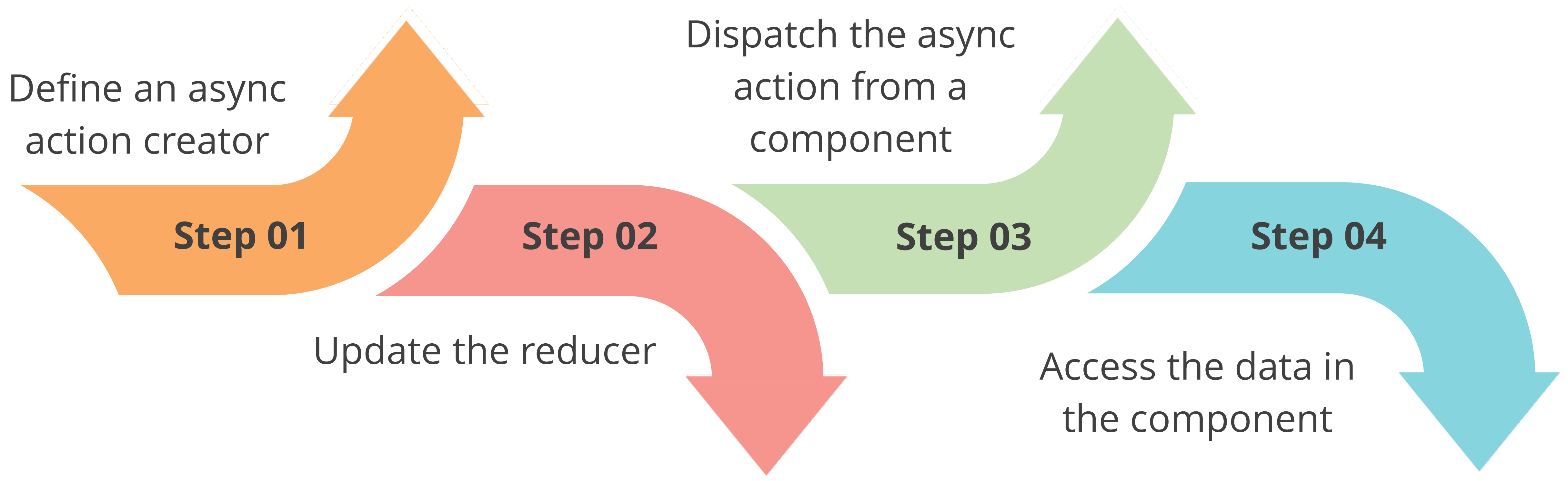
To handle async actions in Redux, use the following middleware:

Redux-thunk
Allows developers to write action creators

Redux-saga
Uses generator functions to handle async actions

Asynchronous Data Retrieval

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


Define an async action creator
Step 01
Step 02
Update the reducer
Dispatch the async action from a component
Step 03
Step 04
Access the data in the component

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:

Example

```
// 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:

Example

```
// 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.

Example

```
// 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

While traditional Redux requires middleware like Redux Thunk for handling asynchronous operations, Redux Toolkit (RTK) simplifies this process with built-in support for async actions using createAsyncThunk.

Understanding createAsyncThunk

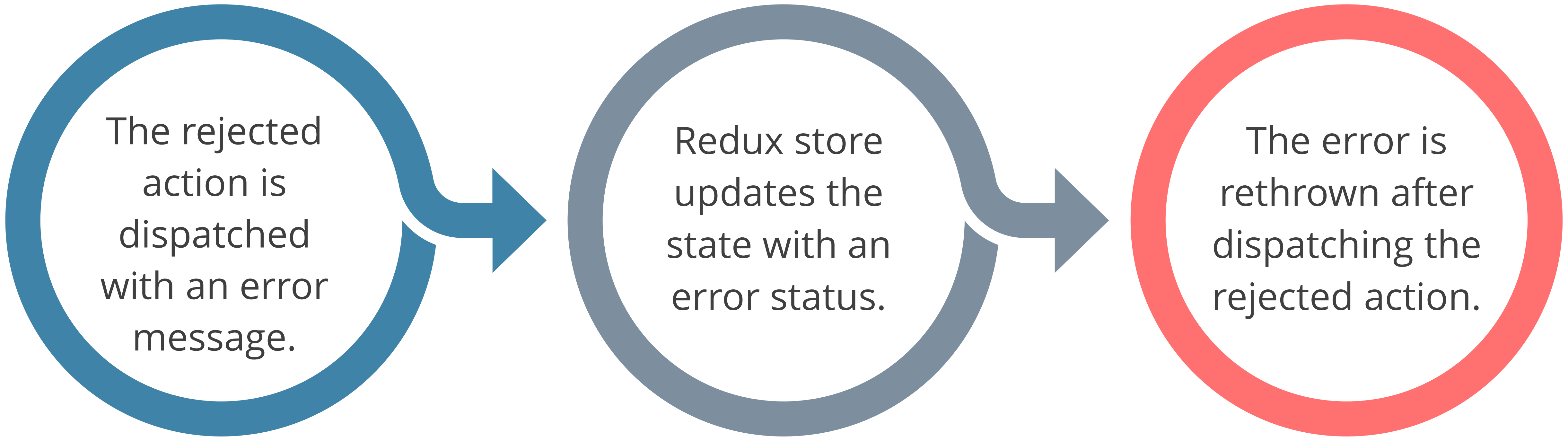
createAsyncThunk is a function that streamlines the process of managing asynchronous logic in Redux applications.

A **thunk** function performs asynchronous operations and dispatches actions based on the results.

Error Handling and Dispatching Actions from Thunks

createAsyncThunk provides a convenient way to handle errors within the thunk and dispatch actions accordingly.

When an error occurs during the API calls:



The rejected action is dispatched with an error message.

Redux store updates the state with an error status.

The error is rethrown after dispatching the rejected action.

In some cases, users might want to catch and handle the error within the thunk without rethrowing it.

Assisted Practice

Creating a React Application Demonstrating Retrieval of Data

Duration: 10 Min.

Problem Statement:

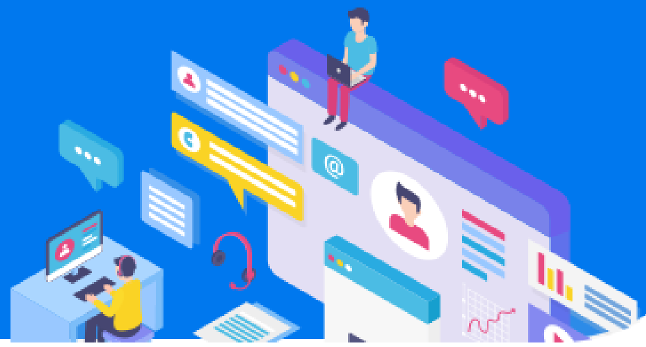
You are given a project to develop a React application that demonstrates retrieval of data asynchronously.

Outcome:

By the end of this demo, you will be able to create a React application that retrieves and displays data from an external source using asynchronous operations.

Note: Refer to the demo document for detailed steps:
04_Creating_a_React_Application_Demonstrating_Retrieval_of_Data

Assisted Practice: Guidelines



Steps to be followed:

1. Create a new React app
2. Create a new file called reducers.js
3. Create a new file called types.js
4. Create a new file called actions.js
5. Get the API key
6. Create a new file called Weather.js
7. Create a new file called WeatherForm.js
8. Open the existing file App.js in the src folder
9. Create a new file called index.js
10. Create a new file called .env
11. Run the app and view it in the browser

Quick Check

You are building a React-Redux application that fetches product data from an external API. Since the API call is asynchronous, you want to ensure that the request, success, and failure states are handled correctly. Which approach should you use?

A. Call the API directly inside the React component and update the state manually

B. Use Redux Thunk middleware to dispatch async actions that handle request, success, and failure

C. Store the API response in a local variable inside the reducer and update it synchronously

D. Dispatch only a single action to update the store once the API call completes

Introduction to Redux Toolkit (RTK)

What Is Redux Toolkit?

It is an advanced, opinionated, and practical toolkit designed to simplify the process of writing Redux logic.

Redux Toolkit

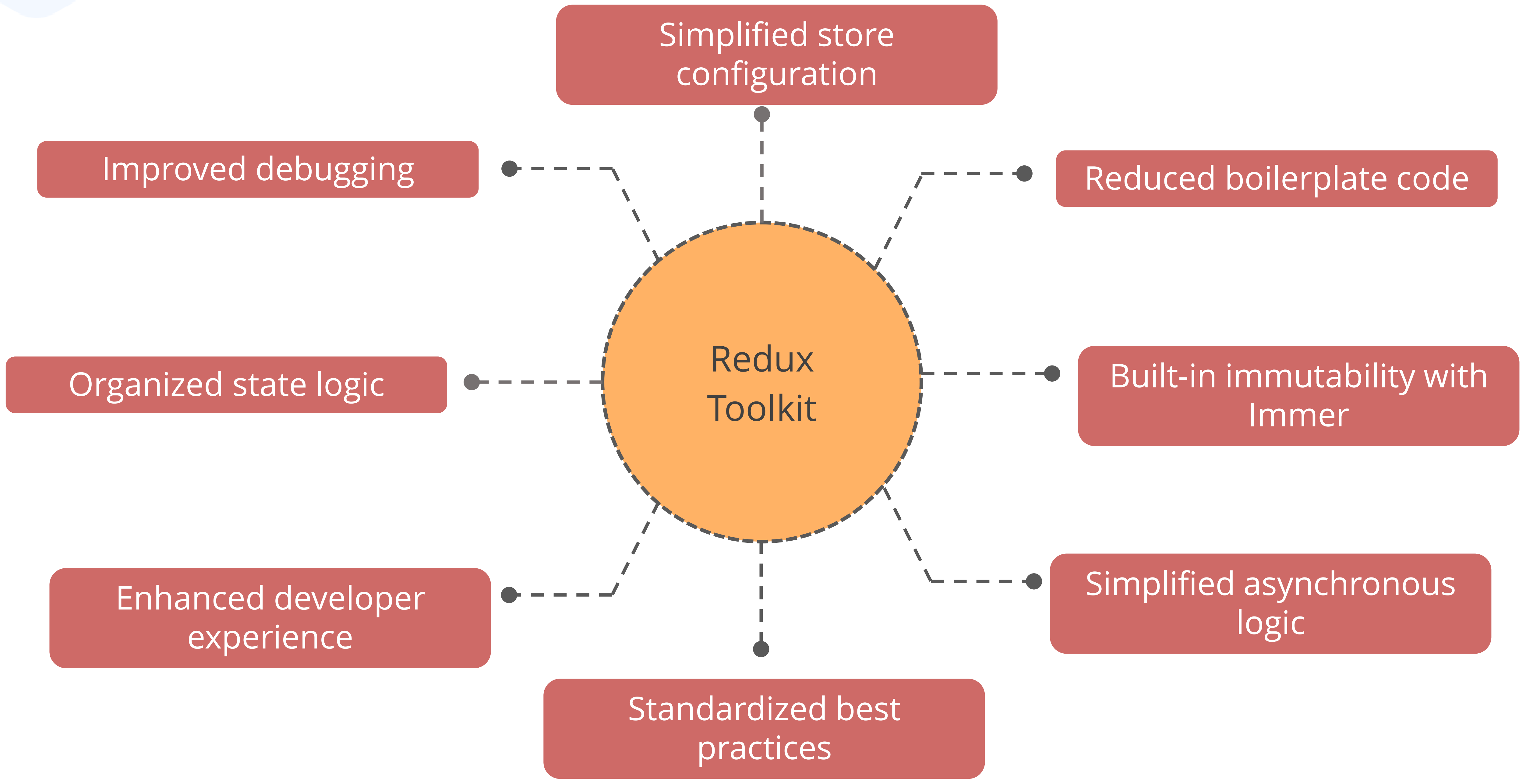
It aims to address the common complaints about Redux being too verbose and complicated by providing a set of tools that streamline Redux development.

Redux Toolkit: Core Concepts

Some key concepts and functions that enhance the Redux experience are:

configureStore	Simplifies the process of store setup
createSlice	Abstracts the process of writing action creators and reducers
Immer	Reduces the boilerplate code (computer language text that users can reuse) and chances of making immutability-related mistakes
createAsyncThunk	Abstracts the process of dispatching actions related to an asynchronous request
Redux logic	Encourages writing more consistent, scalable, and maintainable Redux code

Advantages of RTK


Redux Toolkit
Simplified store configuration
Reduced boilerplate code
Built-in immutability with Immer
Simplified asynchronous logic
Standardized best practices
Enhanced developer experience
Organized state logic
Improved debugging

Setting Up Redux Toolkit in a Project

The process is as follows:

1. Create a React app

```
npm create vite@latest redux-app
-- --template react
```

2. Move to the directory

```
cd redux-app
```

Setting Up Redux Toolkit in a Project

The process is as follows:

3. Install preprovided modules

```
npm install
```

4. Install React Redux and Reduxjs Toolkit

```
npm install react-redux
@reduxjs/toolkit
```

Simplifying Redux Store Setup with Redux Toolkit

configureStore Function

This function replaces the traditional process of manually combining reducers, applying middleware, and creating the store.

The function takes an object as its argument, allowing users to define various aspects of the store.

Integrating Slices Using configureStore

Slices are small reducers that can be combined to form the root reducer.

Slices encapsulate the logic for a specific piece of the state, making the codebase more modular and maintainable.

Once the slices are defined, integrating them into the Redux store is straightforward using **configureStore**.

Integrating Middleware

Redux middleware provides a way to extend the store's capabilities.

Common use cases for middleware include:

- Logging data
- Performing asynchronous actions
- Handling side effects

RTK allows the inclusion of middleware directly in the configureStore function.

Simplified Store Configuration

The example showcases a comprehensive store setup with multiple slices, asynchronous logic handling, and custom middleware.

Example

```
import { configureStore, getDefaultMiddleware } from '@reduxjs/toolkit';
import thunk from 'redux-thunk';
import logger from 'redux-logger';
import rootReducer from './rootReducer';

const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(thunk, logger),
  devTools: process.env.NODE_ENV !== 'production',
});

export default store;
```

Quick Check

You are developing a React-Redux application and need to simplify state management using Redux Toolkit. You want to define multiple state slices and integrate them efficiently into the Redux store. Which approach should you use?

A. Manually combine reducers and create the store with createStore()

B. Use configureStore() from Redux Toolkit to automatically set up the store with slices

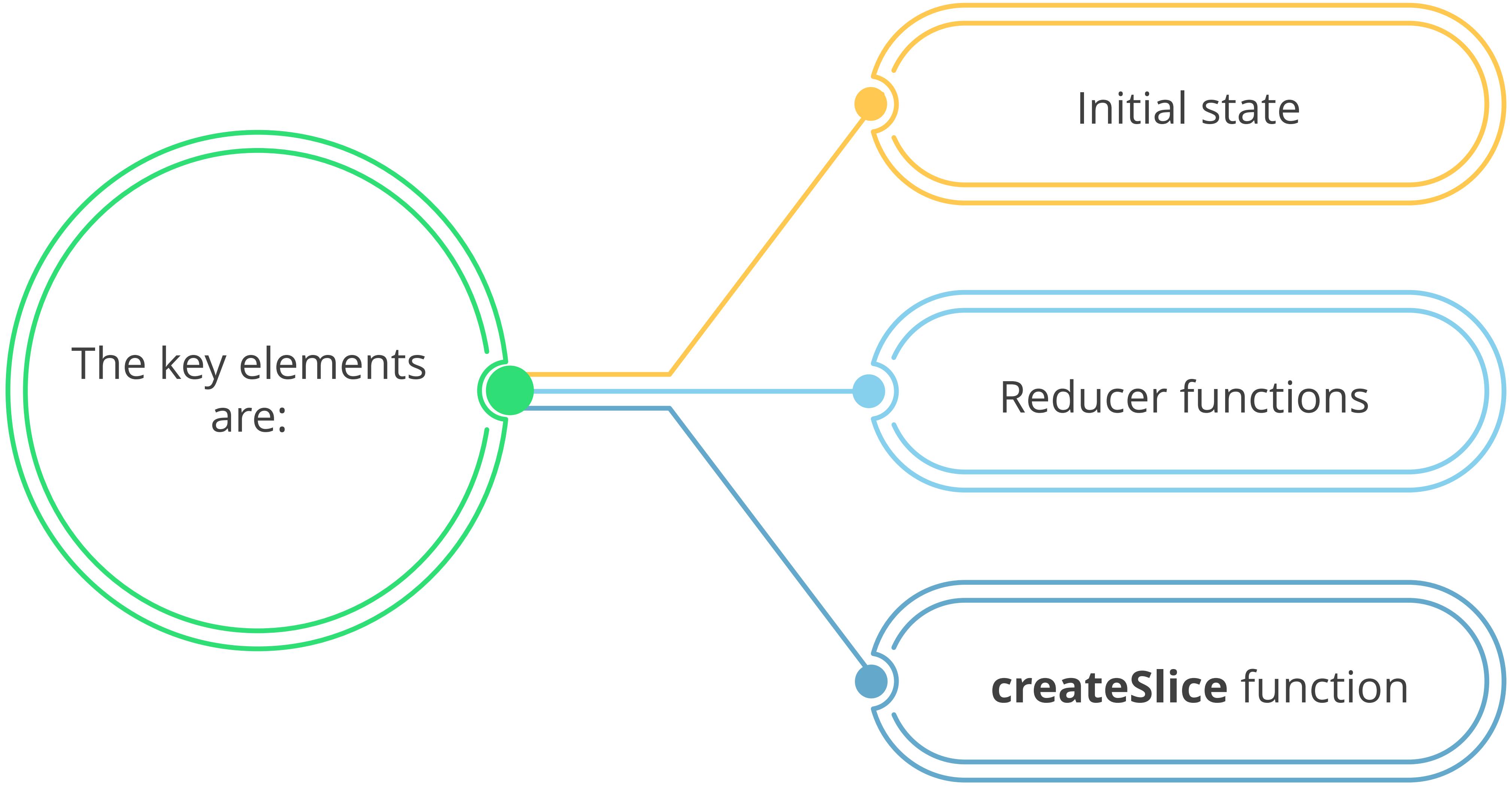
C. Store state inside individual components and avoid using Redux

D. Define all application states in a single slice to reduce complexity

Reducing Boilerplate with createSlice

Fundamentals of createSlice

It is a utility function designed to minimize boilerplate and streamline the process of creating Redux actions and reducers.



Fundamentals of createSlice

Initial state:

This represents the starting point of the state tree.

Example

```
// Example: Initial state for a counter slice
const initialState = {
  value: 0,
};
```

Fundamentals of createSlice

Reducer functions:

Each key-value pair defines a reducer function that handles a specific action and updates the state accordingly.

Example

```
// Example: Reducer functions for a counter
slice
const reducers = {
  increment: (state) => {
    state.value += 1;
  },
  decrement: (state) => {
    state.value -= 1;
  },
};
```

Fundamentals of createSlice

createSlice function:

The **createSlice**, also known as **counterSlice**, object contains automatically generated action creators and a reducer based on the specified name, initial state, and reducer functions.

Example

```
// Example: Creating a counter slice with
createSlice
import { createSlice } from '@reduxjs/toolkit';


const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers,
});
```

Automating Action Creators and Reducers

createSlice automates the generation of action creators and reducers, reducing the need for manual coding and minimizing boilerplate.

Action creators

Example

```
// Example: Automatically generated action creators
const { increment, decrement } = counterSlice.actions;
```

Increment and decrement are action creators that users can use directly in their components.

Automating Action Creators and Reducers

Reducers

Example

```
// Example: Automatically generated reducer
const counterReducer = counterSlice.reducer;
```

counterReducer is a reducer function that can be included in the Redux store configuration.

Managing Asynchronous Actions in Redux Toolkit

About Async Actions

Async actions involve three different action types:

Request

Dispatched to initiate the asynchronous operation

Success

Dispatched to deliver the retrieved data

Failure

Dispatched to report the error details

Handling Loading States

Asynchronous operations often involve loading states to inform of the happenings in the background.

The **createAsyncThunk** function automatically generates action types for pending, fulfilled, and rejected states, making it easy to handle loading states in the reducers.

Accessing Loading States in Components

Users can check the status in the Redux store to conditionally render content based on the loading state.

This ensures a smooth user experience while displaying loading indicators or error messages as needed.

Quick Check

You are developing a React-Redux application that fetches user data from an API. Sometimes, the request fails due to network issues. How should you handle errors while dispatching actions using Redux thunk?

A. Dispatch a success action even if the API fails

B. Modify the state directly inside the thunk without dispatching actions

C. Handle errors only in the component without updating the Redux store

D. Use a try-catch block in the thunk and dispatch an error action on failure

Creating a React Application with Redux Store

Duration: 25 Mins.

Project agenda: To develop a React application using Vite that demonstrates the use of Redux Toolkit for managing application state, Context API for theme toggling, and asynchronous data fetching using createAsyncThunk. The project features a sports dashboard that retrieves and displays live match data from an external API, handles user login state, and allows theme switching between light and dark modes.

Description: You are tasked with building a React application using Vite that integrates Redux Toolkit and Context API to demonstrate scalable state management and real-time data handling. The application will use Redux slices to manage match data and user state, createAsyncThunk for asynchronous API calls, and Context API to control theme toggling across components. This project showcases modular architecture, async data flows, and clean separation of UI and business logic in a dynamic sports dashboard.

Creating a React Application with Redux Store

Duration: 25 Min.

Perform the following:

1. Set up a new React project using Vite
2. Set up Redux store with slices and async thunks
3. Implement Context API for theme toggling
4. Build App component with Redux and Context integration
5. Update main.jsx
6. Run and verify the application

Expected deliverables: A fully functional React application built with Vite, demonstrating the use of Redux Toolkit for state management, Context API for theme control, and asynchronous data fetching with createAsyncThunk. The application will include modular Redux slices, live match data rendering, user session handling, and global theme toggling. It will showcase clean project structure, maintainable code, and an interactive UI, serving as a practical reference for combining Redux and Context in modern React applications.

Key Takeaways

- React Context API is best suited for managing simple or medium-complexity state across components without the need for additional libraries.
- Redux is more powerful and scalable for managing complex state in large applications, especially when multiple components require access to shared data.
- Context API provides a way to avoid prop drilling by allowing global data to be accessed anywhere in the component tree.
- Redux uses actions, reducers, and a centralized store to provide a predictable and structured state management pattern.
- Redux Toolkit simplifies the Redux setup process and encourages writing clean and maintainable code.
- Effective state management enhances performance, reduces bugs, and improves the maintainability of React applications.

Thank You