Design a Dynamic Frontend with React

React Context API and Redux

A Day in the Life of a MERN Stack Developer

The manager of an online sports news channel reached out, expressing concerns about the inefficient operation of the site during peak hours, particularly in updating match details every half an hour.

To address this issue and ensure customers receive the value they are paying for, you were tasked with overseeing backend programming. This involves managing data fetching activities, providing subscription details, updating data, setting up timers, and ensuring the accurate updating of prices and quantities of the products.

By leveraging the key concepts of React Hooks and understanding their interrelationships, you can complete these tasks and provide an effective solution for the given scenario.

Learning Objectives

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

- Differentiate between React context API and Redux
- 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 development

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

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

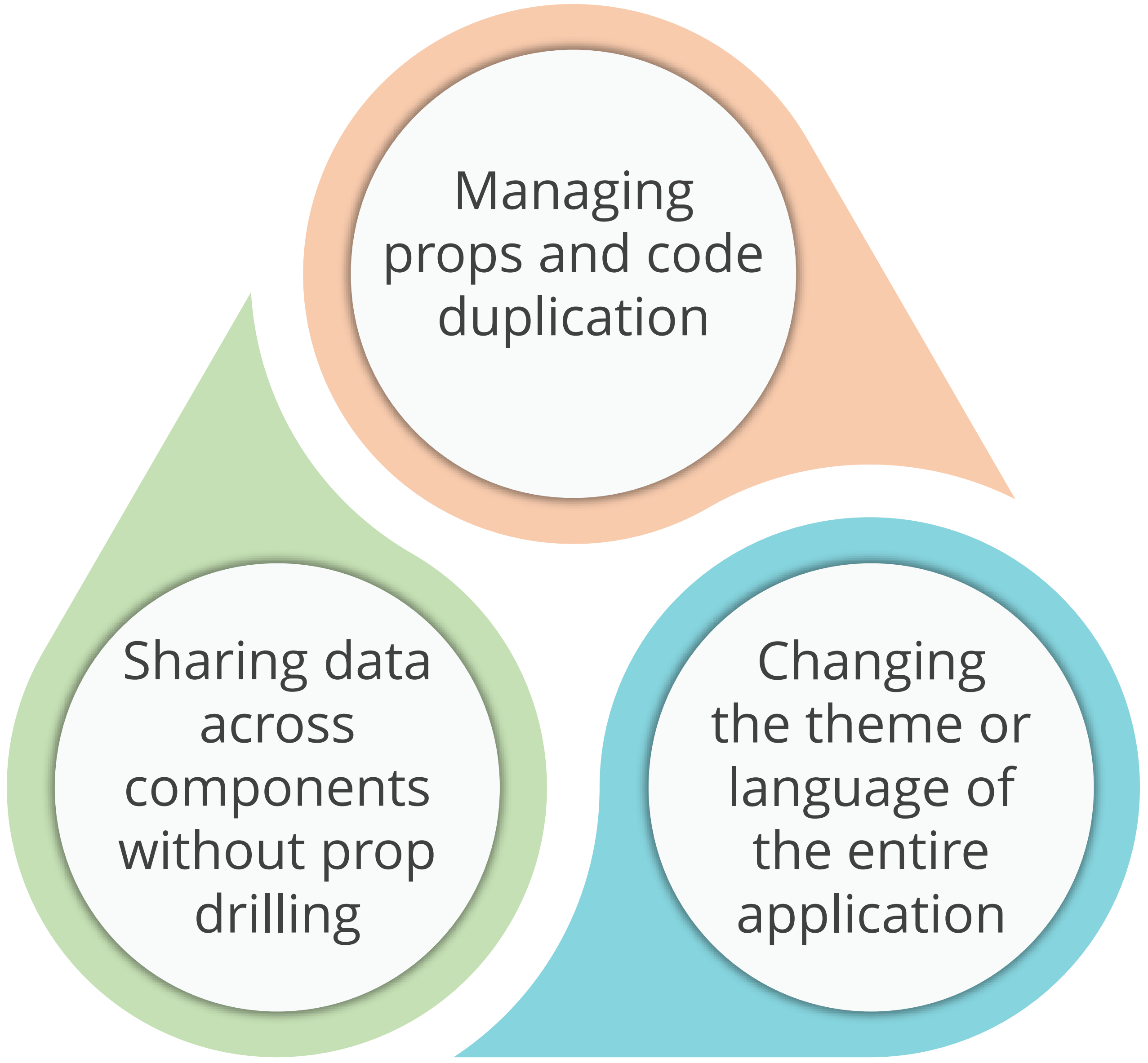
Context API: Applications

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:


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

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

Creating a Context in React

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

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

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.

```
// 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**.

```
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.

```
// 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:

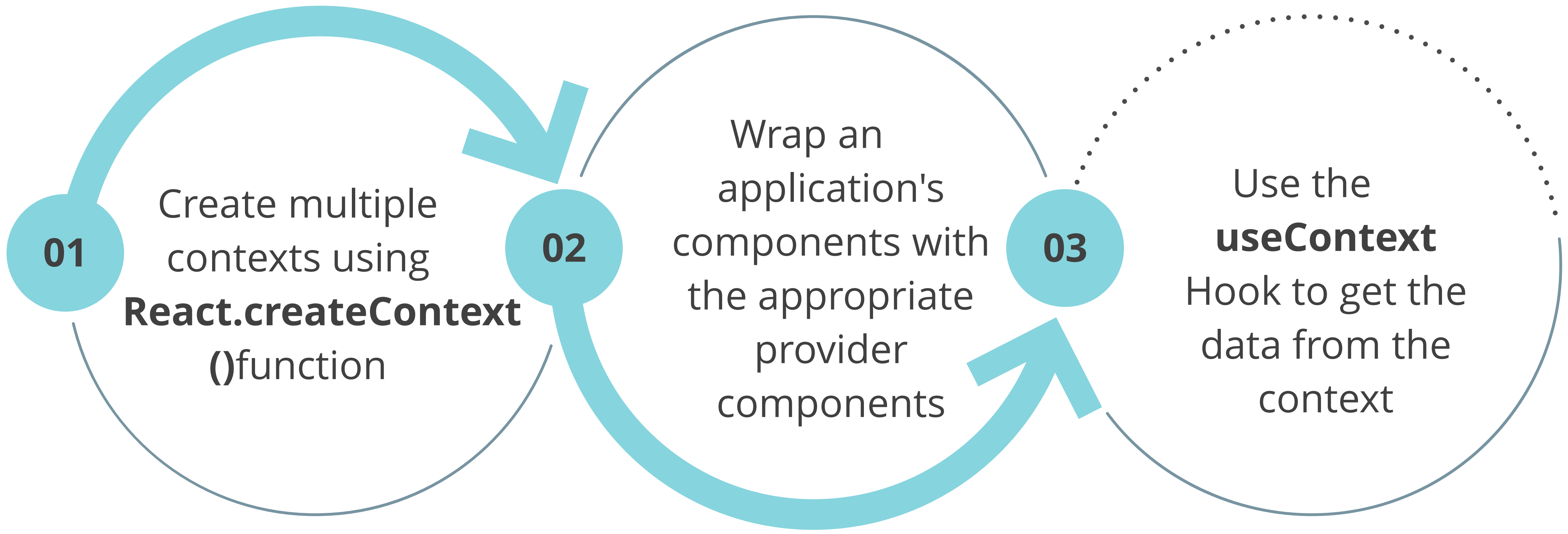
```
import React, { useContext } from 'react';

// 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 is 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:

```
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

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

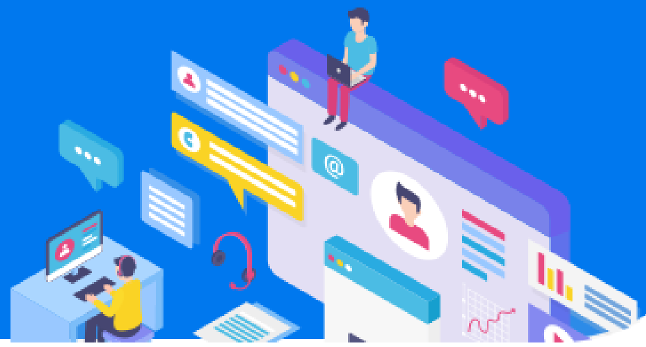
Using Multiple Contexts: Example

```
// 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

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

Assisted Practice

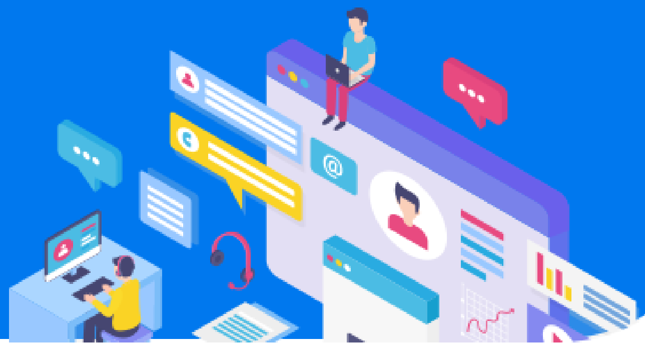


Creating a Theme Button Using ContextAPI **Duration: 10 Min.**

Problem Statement:

You are given a project to develop a React App which uses the concept of 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

Redux

Redux: Introduction

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

Centralized state	Predictable sate changes	Immutable state	Middleware support
Supports a single state tree	Provides unidirectional data flow	Prevents direct changes	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

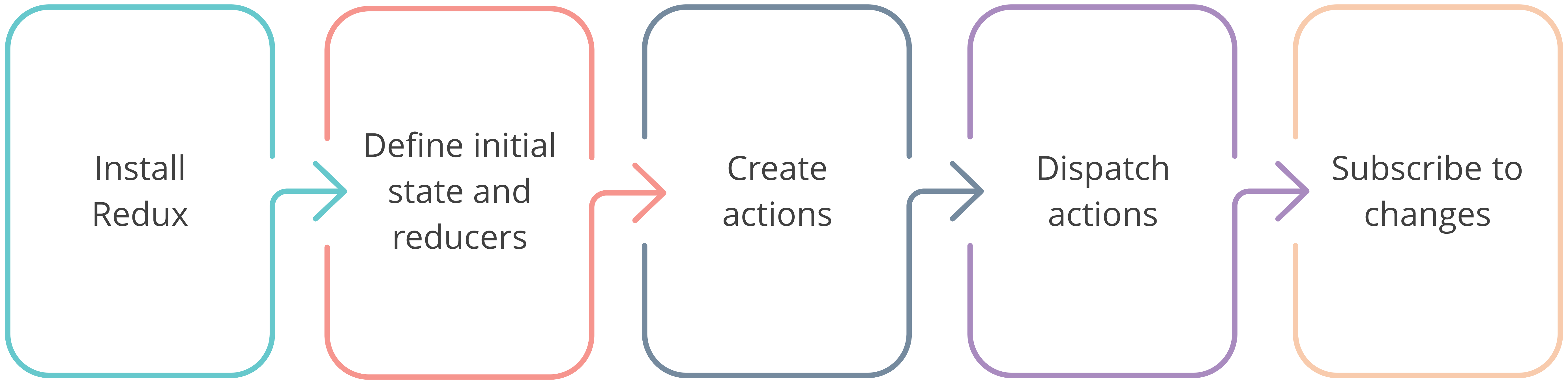
Maintains a single source of truth

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

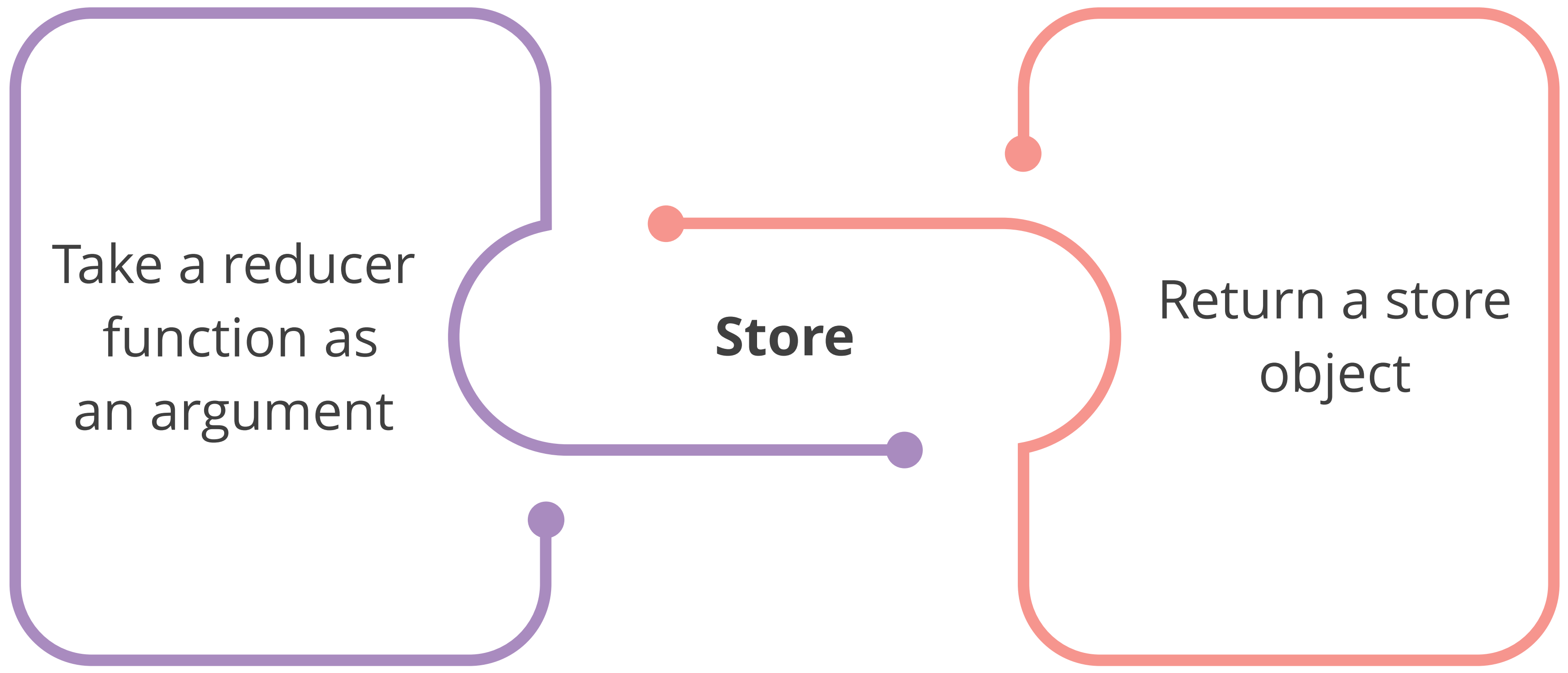
03 Combine multiple reducers into a single root reducer

04 Ensure immutability

About Store

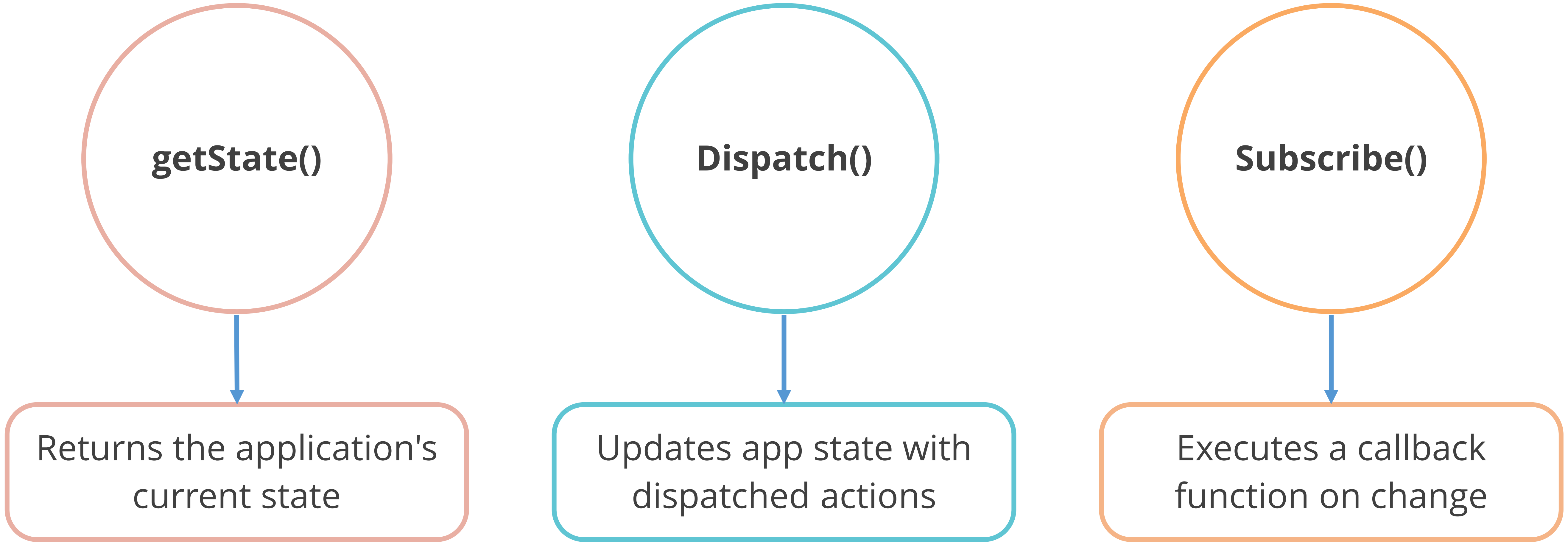
Stores are objects in Redux that hold the application's state.

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

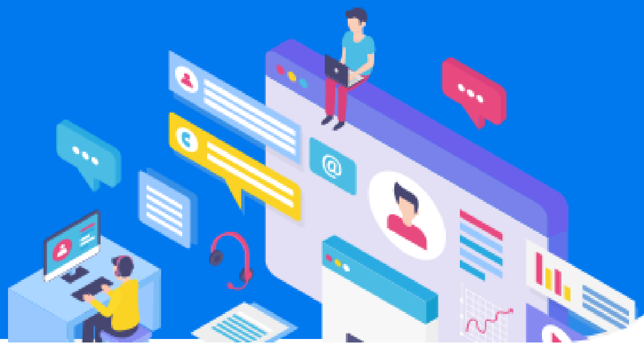


About Store

The store has three primary responsibilities:


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

Assisted Practice

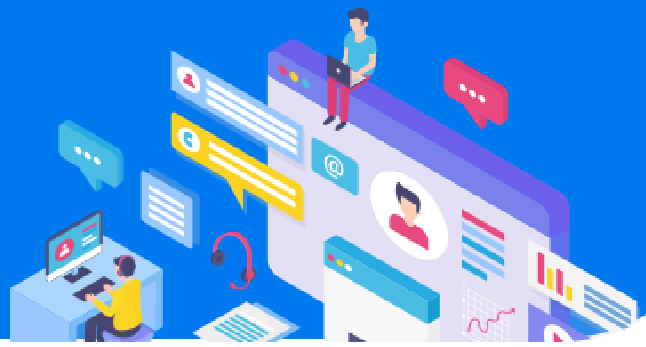


Creating a Application Using the Principles of Redux **Duration: 10 Min.**

Problem Statement:

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

Assisted Practice: Guidelines



Steps to be followed:

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

Multiple Reducers

They handle different parts of the state which form 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:

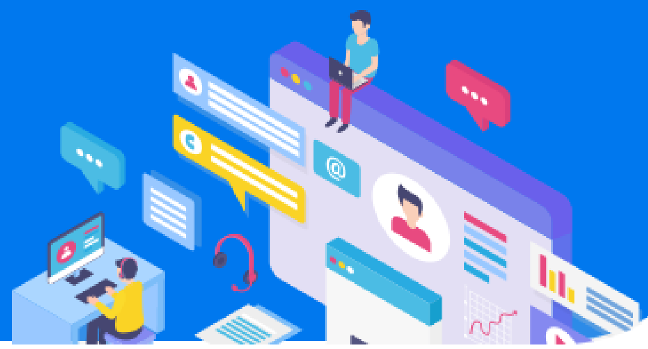
1. Takes an object with reducers as its argument
2. Returns a single reducer

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

Assisted Practice

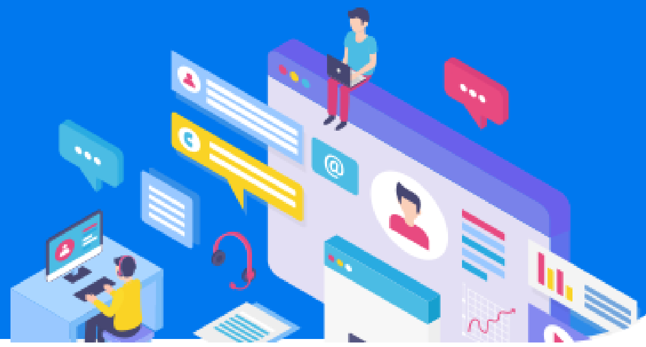


Demo with Combine Reducers **Duration: 10 Min.**

Problem Statement:

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

Assisted Practice: Guidelines



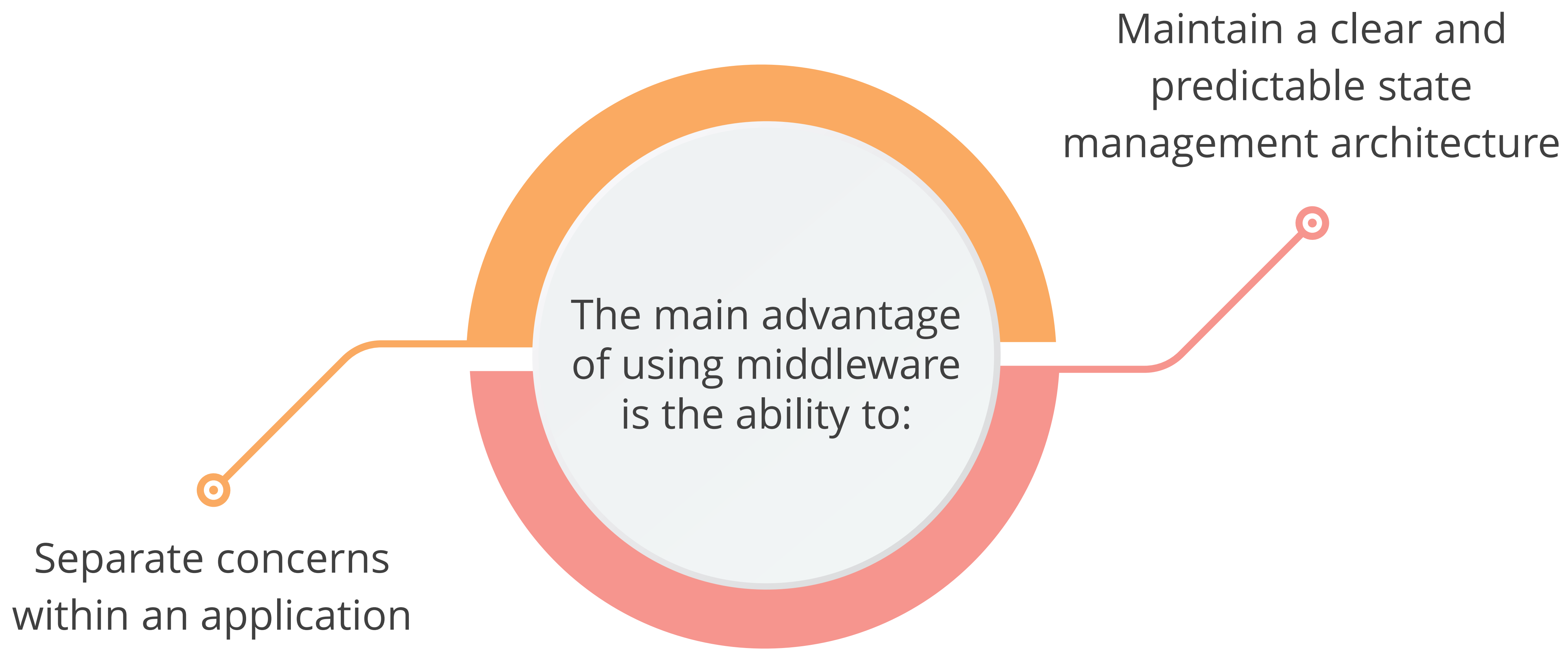
Steps to be followed:

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

Data Retrieval Using Redux

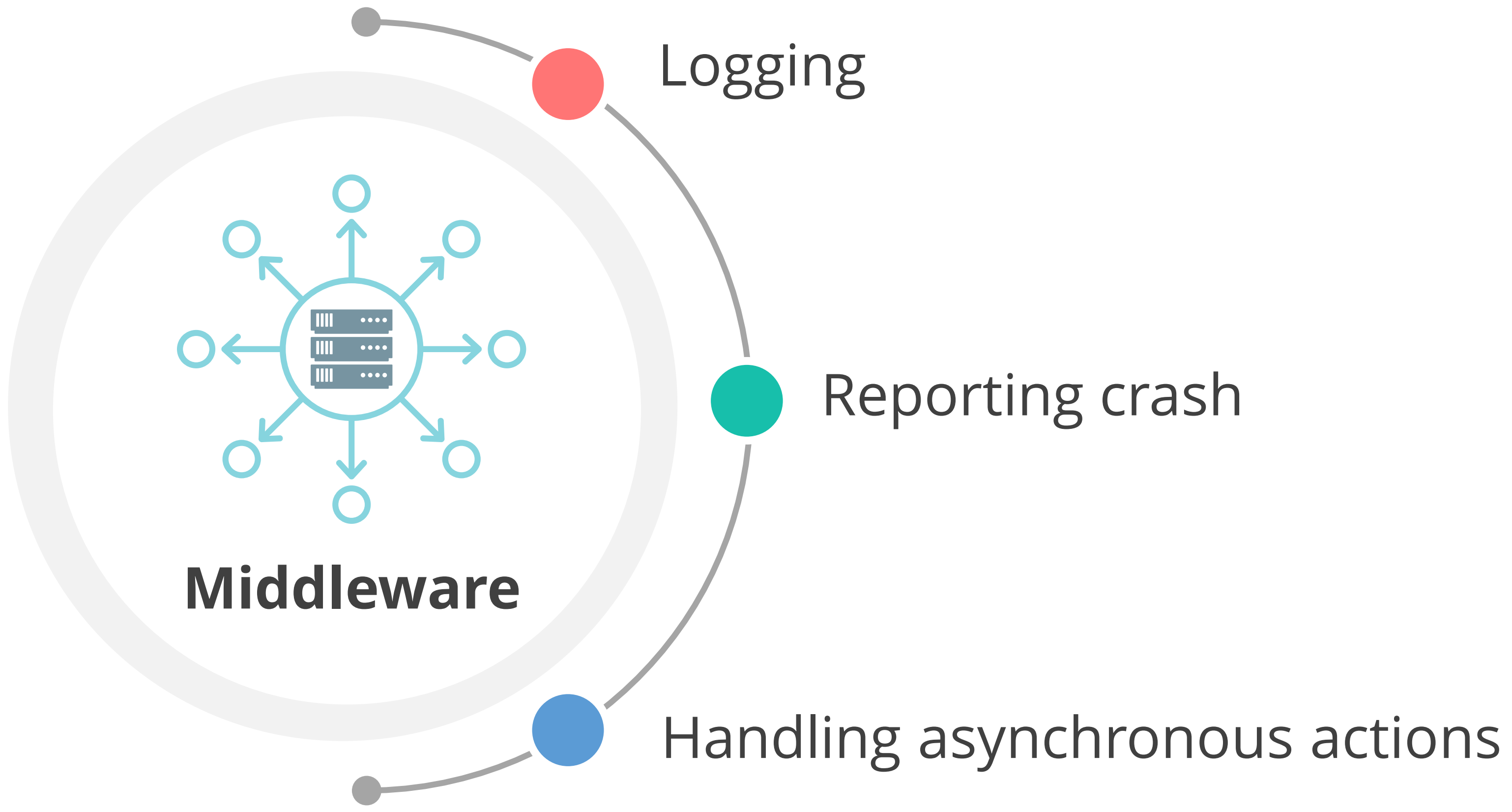
About Middleware

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


Maintain a clear and predictable state management architecture
The main advantage of using middleware is the ability to:
Separate concerns within an application

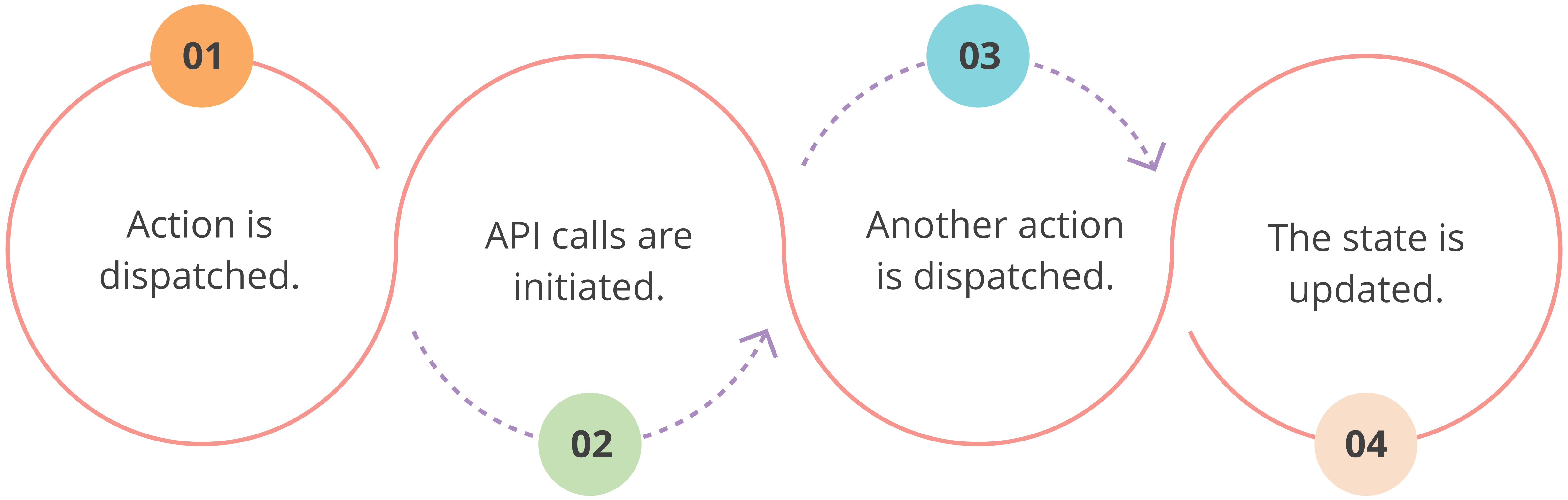
About Middleware

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

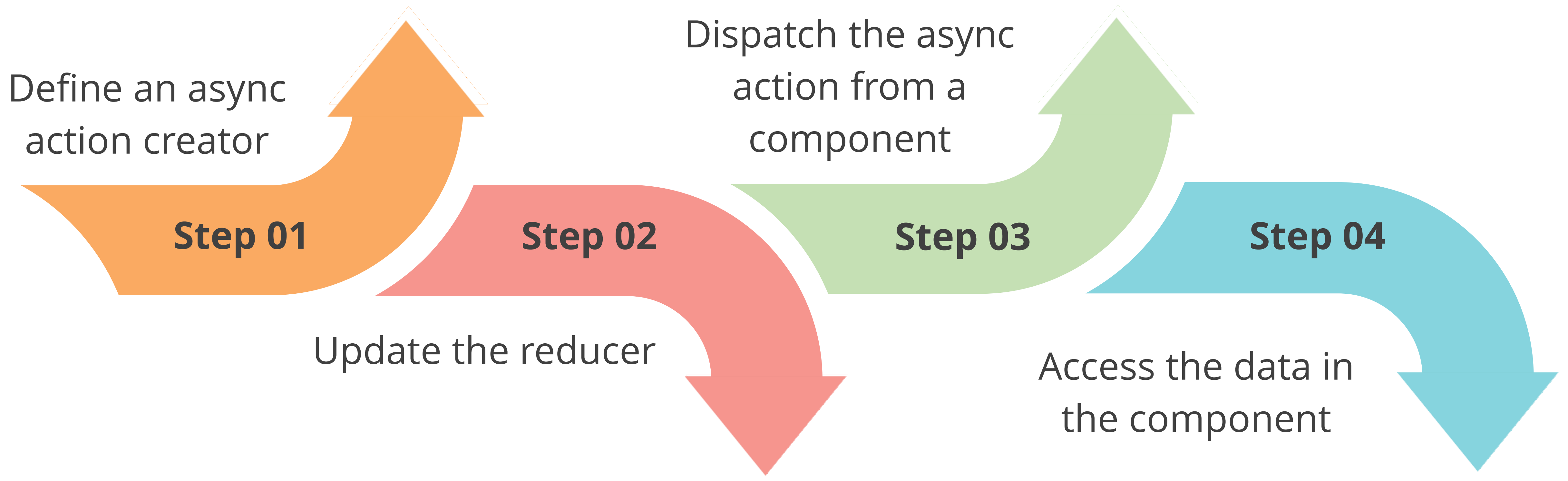
To handle async actions in Redux, use the following middleware:

1 **Redux-thunk**
Allows developers to write action creators

2 **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:

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:

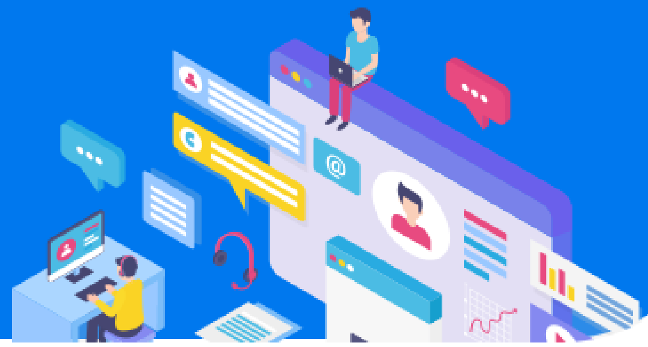
```
.then(response => response.json())
      .then(data => {
        dispatch(fetchDataSuccess(data));
      })
      .catch(error => {
        dispatch(fetchDataFailure(error.message));
      });
  }, []);
  return (
    <>
      {loading ? <p>Loading...</p> : null}
      {error ? <p>{error}</p> : null}
      {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

Assisted Practice

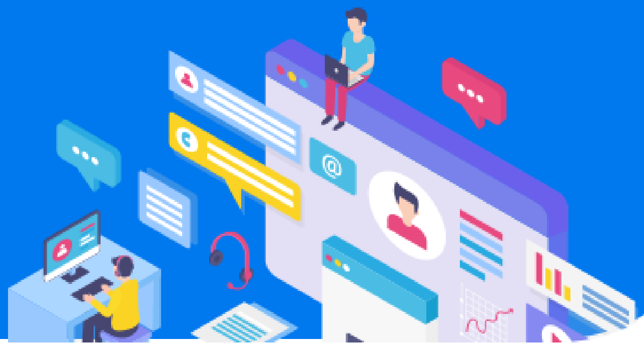


Demo with Asynchronous Data Retrieval Using Redux **Duration: 10 Min.**

Problem Statement:

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

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. Create a new file called Weather.js
6. Create a new file called WeatherForm.js
7. Open the existing file App.js in the src folder
8. Create a new file called index.js
9. Create a new file called .env
10. Update the axios.get URL in actions.js
11. Run the app and view it in the browser

Introduction to Redux

Redux Toolkit

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

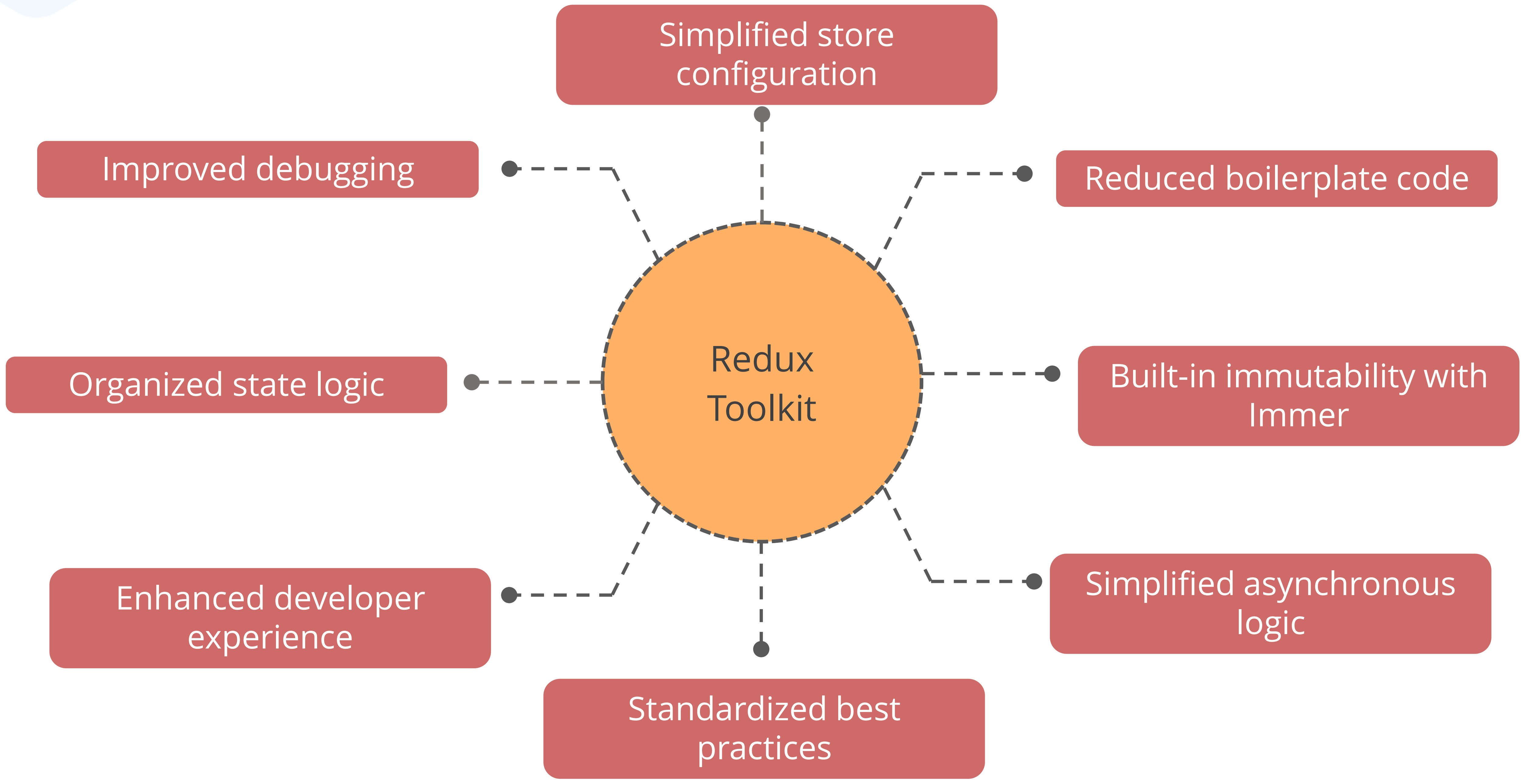
It aims to address the common complaints about Redux being too verbose and complicated by providing a set of tools that streamline Redux development.

Redux Toolkit

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


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

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

Understanding the 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

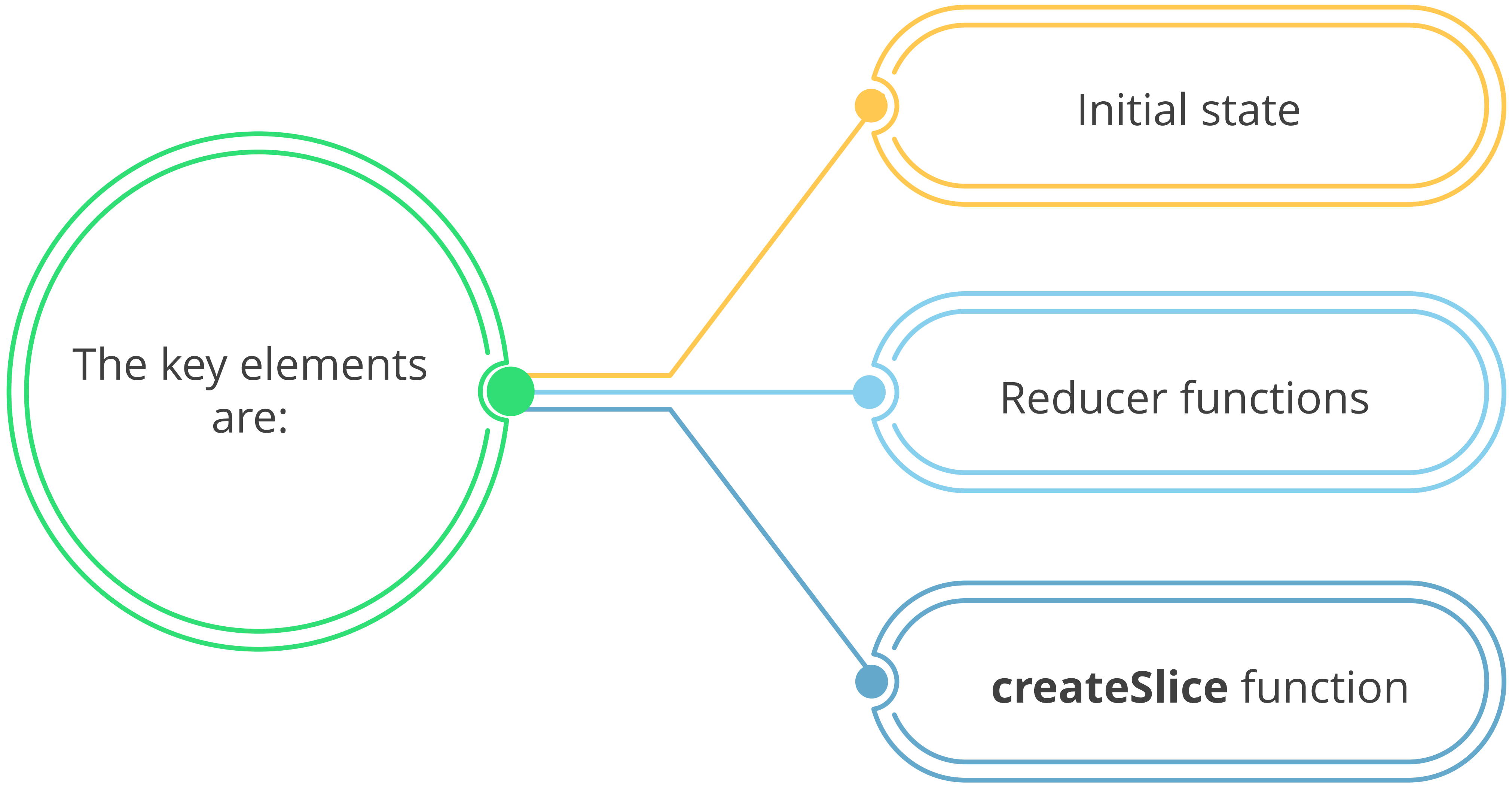
The example showcases a comprehensive store setup with multiple slices, asynchronous logic handling, and custom middleware.

```
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;
```

Reducing Boilerplate with createSlice

Fundamentals of createSlice

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

Fundamentals of createSlice

Reducer functions:

Each key-value pair corresponds to a specific action type and the logic to update the state in response to this action.

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

Fundamentals of createSlice

createSlice function:

counterSlice is an object that includes automatically generated actions and a reducer based on the provided initial state and reducer functions.

```
// 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: 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: 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

Performed to signal the start of the process

Success

Dispatched with the payload

Failure

Dispatched with an error message

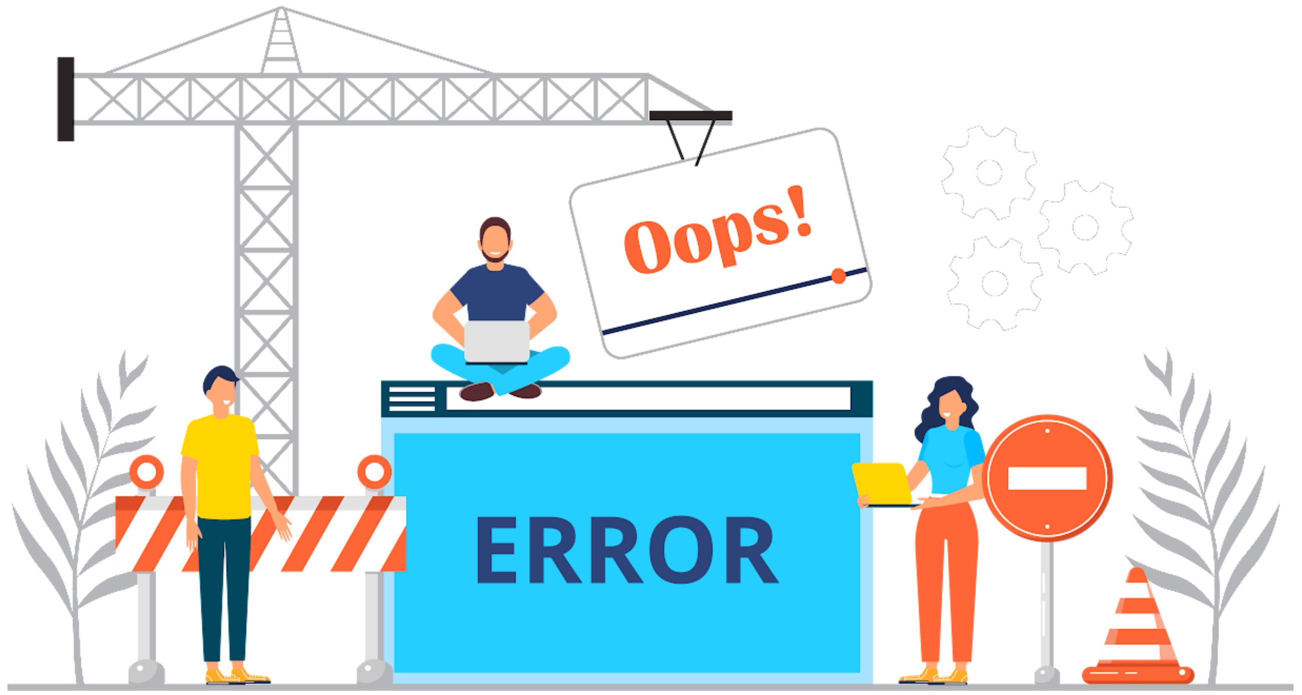
Handling Loading States

Asynchronous operations often involve loading states to inform 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.

Handling Side Effects with createAsyncThunk in Redux Toolkit

Understanding createAsyncThunk

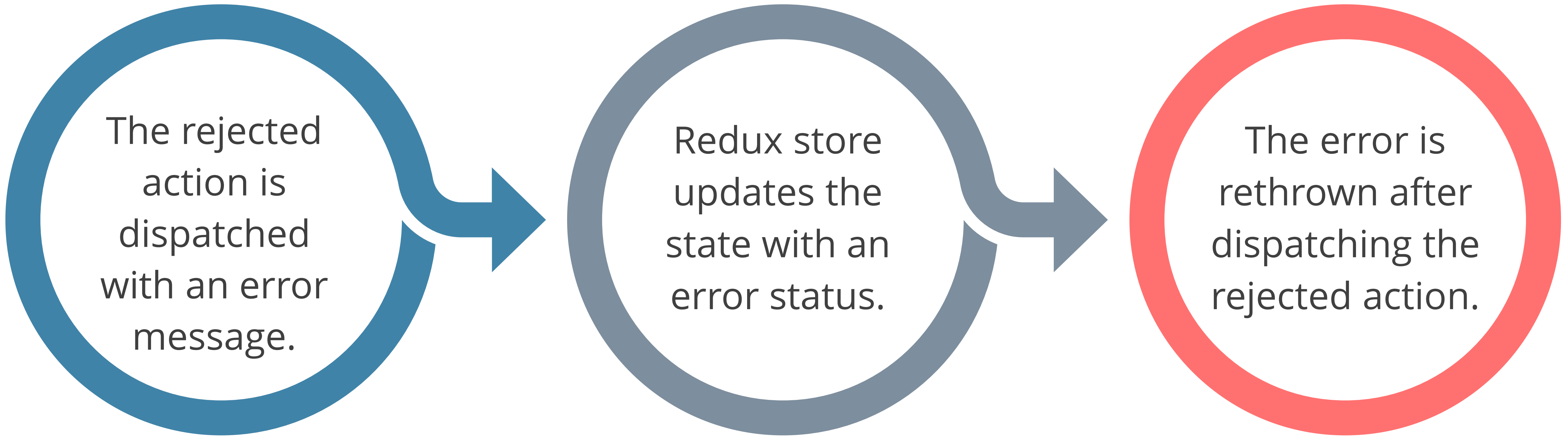
createAsyncThunk is a function that streamlines the process of managing asynchronous logic in Redux applications.

A **thunk** function performs asynchronous operations and dispatch 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.

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.

Thank You