Using the useReducer Hook

```
import React, { useReducer } from 'react';
const initialState = { count: 0 };
function reducer(state, action) {
            switch (action.type) {
            case 'increment':
            return { count: state.count + 1 };
            case 'decrement':
            return { count: state.count - 1 };
            default:
            return state;
            }
}
function CounterWithReducer() {
const [state, dispatch] = useReducer(reducer, initialState);
return (
<div> <p>Count: {state.count}</p>
<button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
<button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
</div>
);
}
```

- An alternative to **useState** for managing complex state logic.
- Especially useful when state transitions **depend on the previous state.**

Using the useRef Hook

```
import React, { useRef, useEffect } from 'react';

function FocusInput() {
const inputRef = useRef();

useEffect(() => {
inputRef.current.focus();
}, []);

return <input ref={inputRef} />;
}
```

- Creates mutable references to DOM elements or other values that persist across renders.
- Useful when you need to access and interact with DOM elements directly.
- Let's you reference a value that's **not needed for rendering.**

Using the useCallback & useMemo Hook

```
import React, { useState, useCallback, useMemo } from
'react';
function MemoizedComponent({ a, b }) {
const memoizedCallback = useCallback(() => {
// Function logic that depends on 'a' and 'b'
}, [a, b]);

const memoizedValue = useMemo(() => {
// Expensive computation using 'a' and 'b'
return a + b;
}, [a, b]);

return (
<div> <p>Memoized Value: {memoizedValue}</p>
<button onClick={memoizedCallback}>Memoized
Action</button>
</div>
);
}
```

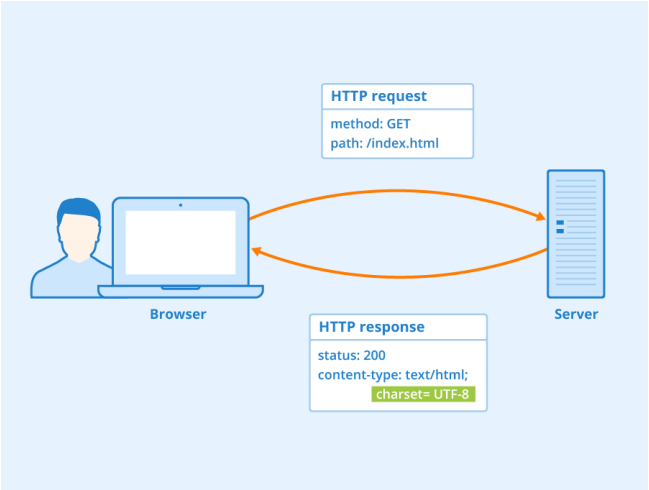
- useCallback memoizes functions to prevent unnecessary re-renders.
- useMemo memoizes the result of an expensive computation to avoid recomputation.

React JS

Lecture 3 (Routing – Movie Project)

Non-react websites

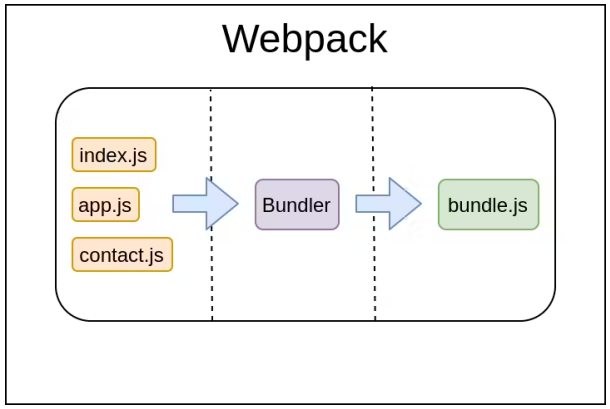
- Browser requests of your URL to a server and sends back a full HTML page
- When any new link is clicked, another request is sent to the server! (Forever requests)



React websites

Single-Page Application (SPA) vs. Multi-Page Application (MPA):

- React Router is primarily used in Single-Page Applications (SPAs). SPAs load a single HTML page and dynamically update content without fully reloading the page. React Router handles the routing within the same HTML document.
- Normal HTML requests for links or pages are typically used in Multi-Page Applications (MPAs). MPAs load entirely new HTML documents when you click on a link or navigate to a new page.



react-router – different routers

- BrowserRouter
  - Based on HTML5 history API
  - Requires server handling dynamic requests (response to any possible URI)
- HashRouter
  - Based on window.location.hash
  - Can be used with static websites (server-less)
- MemoryRouter & StaticRouter – mostly used for testing

react-router – routing

- Route
  - path ⟶ [component[1]|render[1]|children[2]]
    [1] - renders only when path matches [2] - always renders
  - exact
    - no sub-path makes match
    - not by default
  - Enhances inner components‘ props
- Link
  - Renders link to given route
  - Nastily customizable

- Read on:
  1. https://github.com/remix-run/react-router/tree/dev/examples
  2. https://reactrouter.com/en/main/route/route
  3. https://reactrouter.com/en/main/components/link

Using the useHistory & useLocation & useParams Hook

```
import { BrowserRouter, Route, useHistory, useLocation,
useParams } from 'react-router-dom';

function MyComponent() {
const history = useHistory();
const location = useLocation();
const params = useParams();

return (
<div>
<p>Current URL: {location.pathname}</p>
<button onClick={() => history.push('/new-
route')}>Navigate</button>
<p>Route Parameter: {params.id}</p>
</div>
);
}
```

- Hooks from the react-router library for client-side routing.
- useHistory: Provides access to the navigation history.
- useLocation: Gives information about the current URL.
- useParams: Extracts parameters from the URL.

example:

Download the package

$npm install react-router-dom

In index.js

import {BrowserRouter } from ’react-router-dom’;

In App.js

import { Routes, Route, Link} from 'react-router-dom'

index.js / or main.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter } from 'react-
router-dom';
import App from './App';

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

App.js

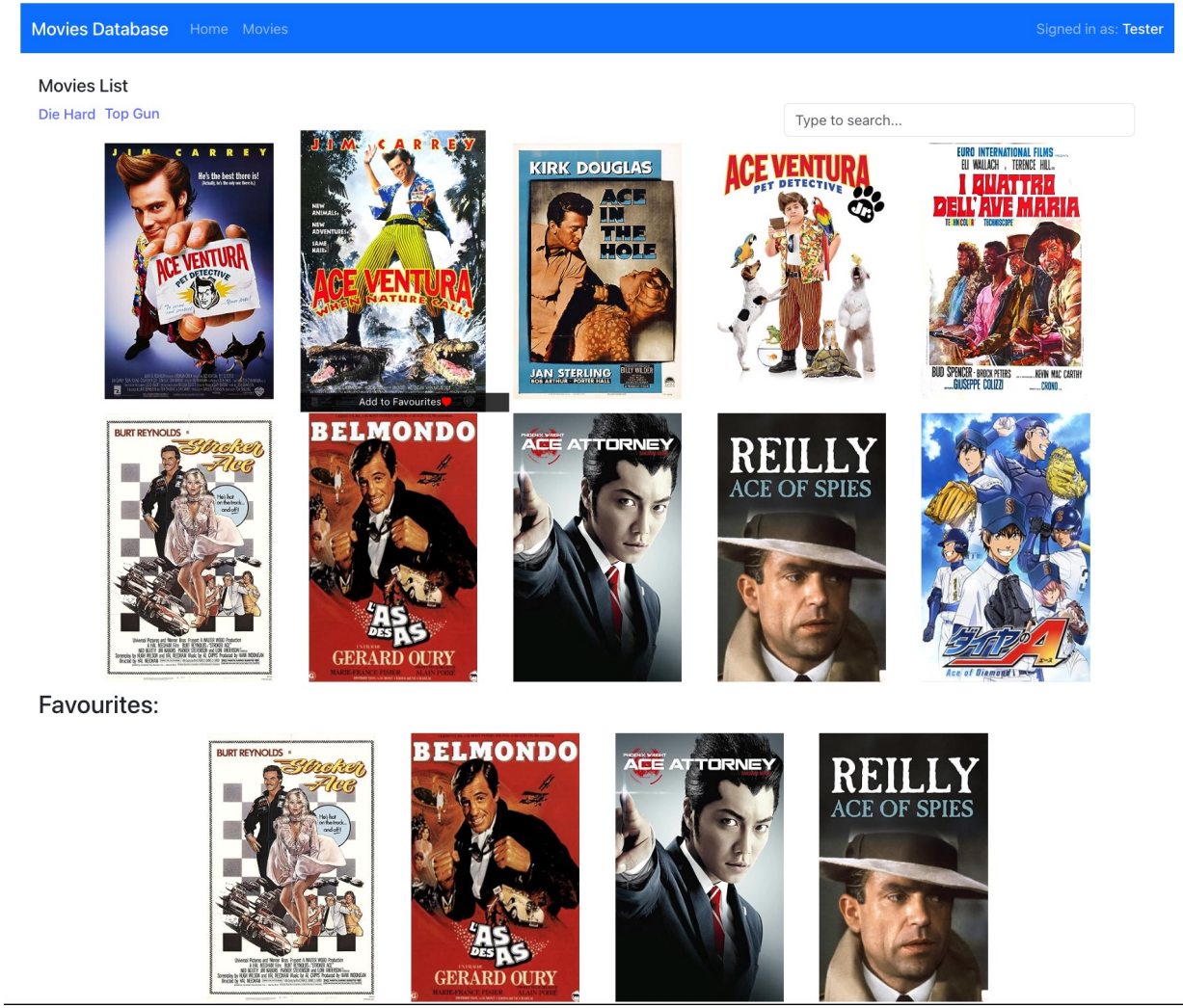
```
import React from 'react';
import { Route, Routes } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <div>
      <Routes>
        <Route path="/about" element={<About/>} />
        <Route path="/" element={<Home/>} />
      </Routes>
    </div>
  );
}

export default App;
```

Movie project

http://www.omdbapi.com



Movie project

Important component

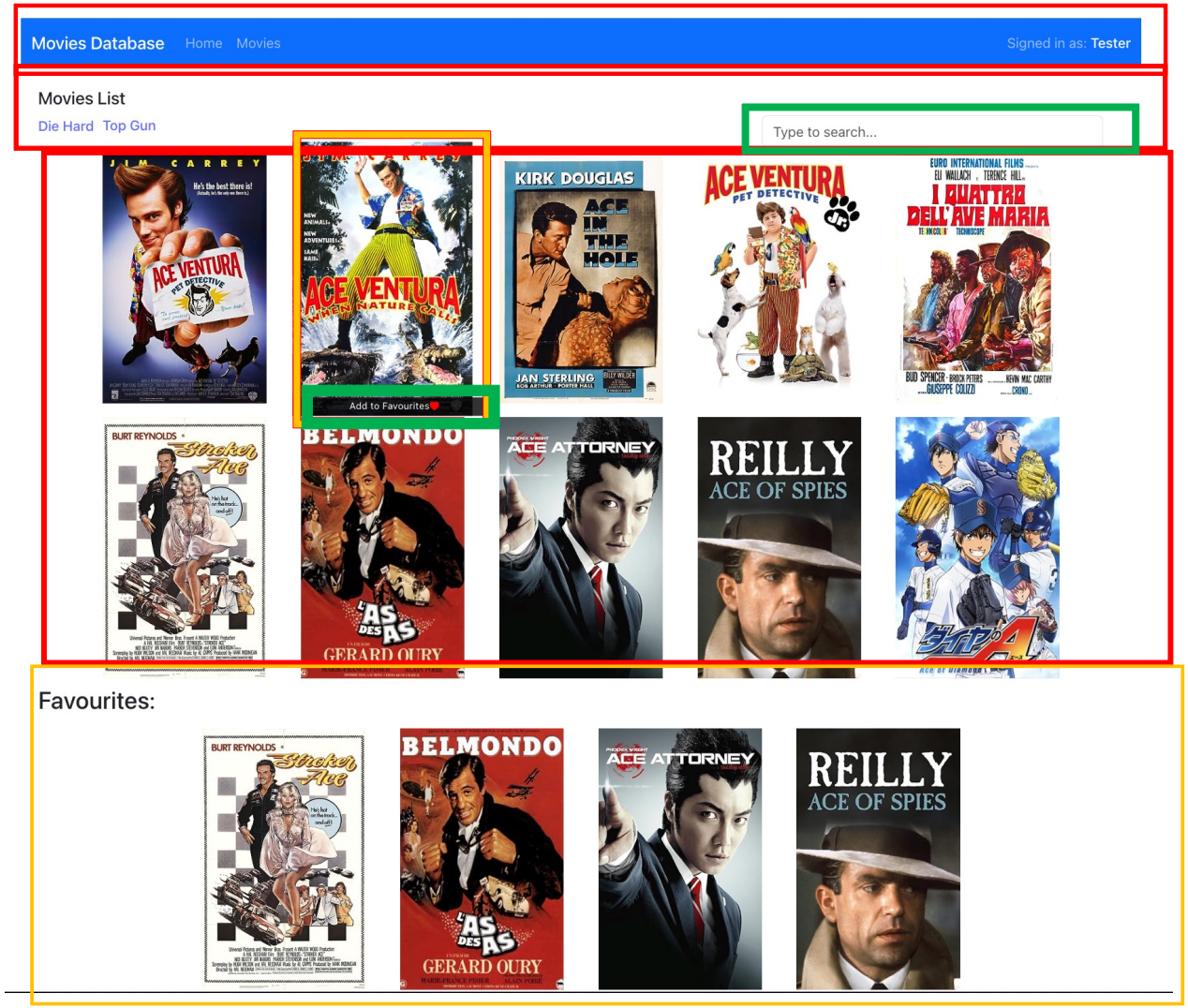
App.js
Navbar.js
Movielist.js
Movie.js
Searchbar.js
Favorites.js

Important values!

API fetch!
moviesList
searchValue
favoriteList

Important events!

1. Fetch on load (useEffect)
2. Reload on change (useState)
3. OnClick (add to fav)
4. onSubmit search (new fetch)



Movie project

We already have

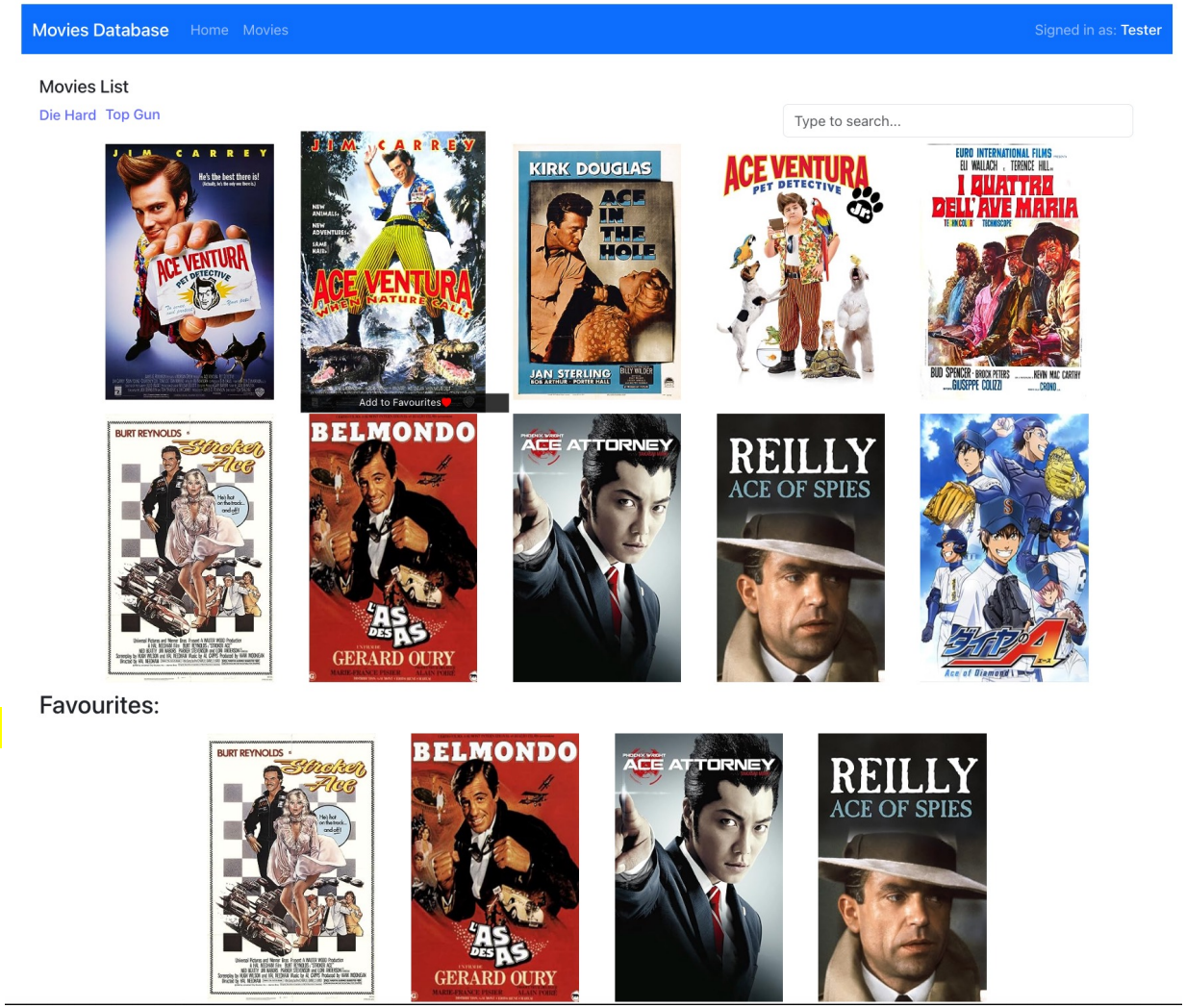
- navbar, home, moviesList (components)

Steps

1. useState → for movies list
2. useEffect → for Fetch API
3. Set the values of movies list
4. In moviesList.js (map all values from movies list to the return)
5. Test it!

Next steps

1. Add search bar component
2. Add search value (useState) must be in the main parent component!
3. Pass the search value function to the search bar component
4. Update (useEffect to handle search value changes
5. Test it!



Movie project

Nest steps..

1. Add to favorites section
2. Favorite list (useState)
3. Add to favorites EVENT! OnClick
4. Favorite list view under the movies list