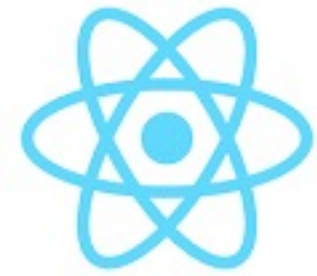


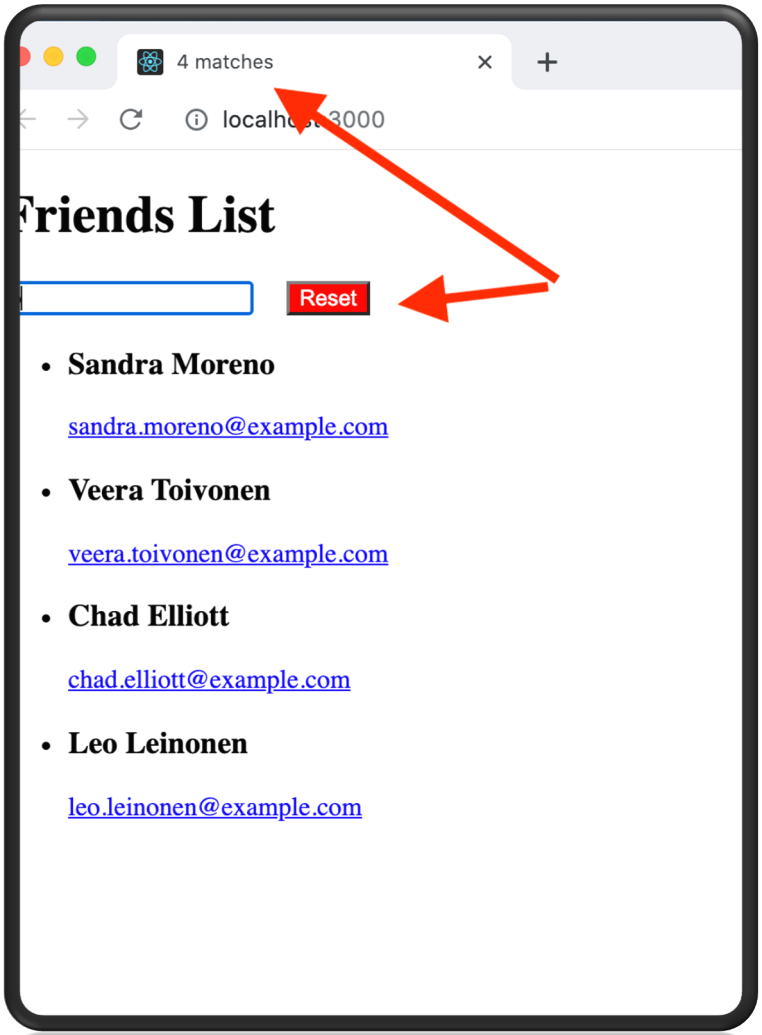
ReactJS.

The Component model (Contd)

Topics

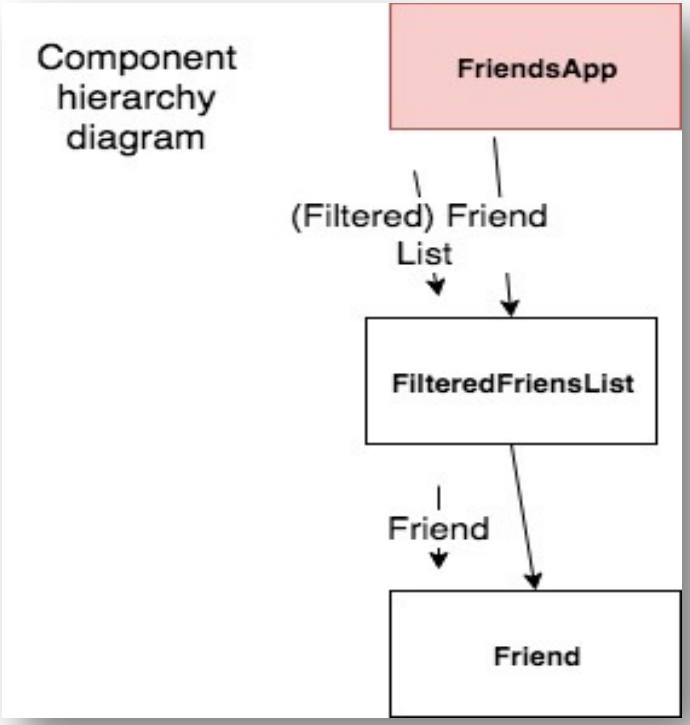
- **Hooks and Component Lifecycle.**
- **Data Flow patterns – Data Down, Action Up pattern.** ✔

Sample App 1 - Features



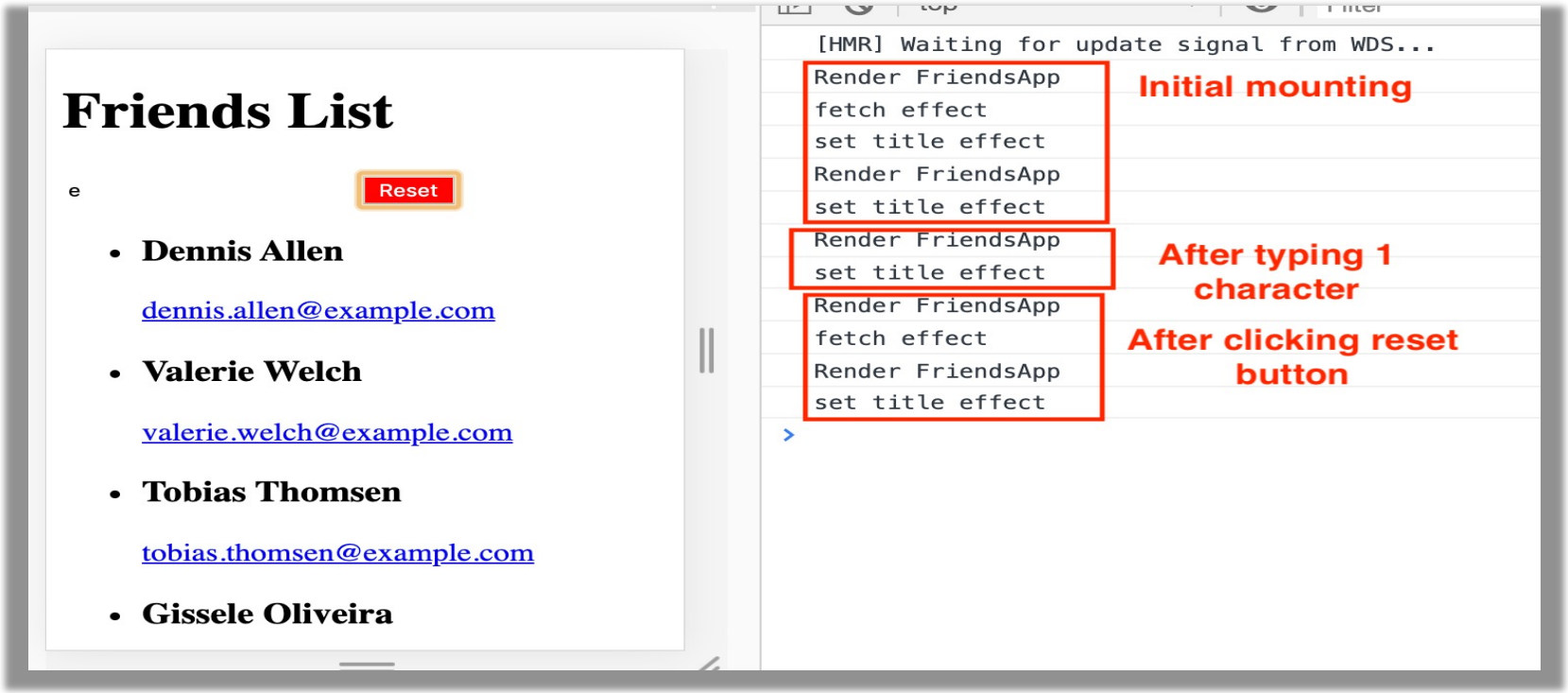
- **App UI changes:**
  1. **A 'Reset' button – load a new list of friends from the API – overwrites current list.**
  2. **Browser tab title - show # of matching friends (side effect).**
- **See lecture archive for source code**

Sample App 1 - Design



- **3 state variables:**
  1. List of friends retrieved from API.
  2. Text box content.
  3. **Reset button toggle.**
- **2 side effects:**
  1. **'Fetch API data' - dependent on** reset button **toggle change.**
  2. **'Set browser tab title' - dependent on length of matching list changing.**

Sample App 1 - Events



Sample App 1 - Events.

- **On mounting of** FriensApp **component:**
  Both effects execute (Set browser tab to 0 matches)
  → **'Fetch data' effect changes 'friends list' state**
  → **Component re-renders + 'Set browser tab' effect executes.**
- **On typing a character in the text box:**
  'Text' state change
  → FriendsApp **rerenders + Matching friends list length changes**
  → **'Set browser title' effect executes.**
- **On clicking Reset button:**
  'Reset toggle' state changes
  → FriendsApp **rerenders**
  → **'Fetch data' effect executes**
  → **'Friends list' state changes**
  → **FriendsApp re-renders + Matching list length changes.**
  → **'Set browser title' effect executes**

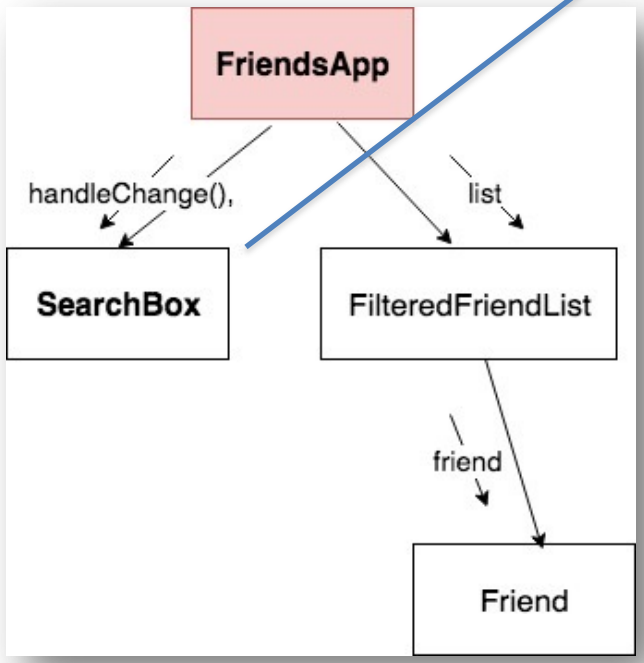
Topics

- **Hooks and Component Lifecycle.** ✔
- **Data Flow patterns – Data Down, Action Up pattern.** ✔

Sample App 2

(Data down, actions up pattern or Inverse data flow pattern)

- **What if a component's** state **is influenced by an** event **in a subordinate component?**
- **Solution: The** data down, action up pattern.



Data down, Action up.

Pattern:

1. Stateful **component** (FriendsApp) **provides a callback to the subordinate** (SearchBox)**.**
2. Subordinate **invokes callback when the event** (onChange) **occurs.**

```
const FriendsApp = () => {
  const [searchText, setSearchText] = useState("");
  const [friends, setFriends] = useState([]);

  useEffect(() => {…
  }, []);

  const filterChange = text =>
    setSearchText(text.toLowerCase());

  const updatedList = friends.filter(friend => {…
  });
  return (
    <>
      <h1>Friends List</h1>
      <SearchBox handleChange={filterChange } />
      <FilteredFriendList list={updatedList} />
    </>
```

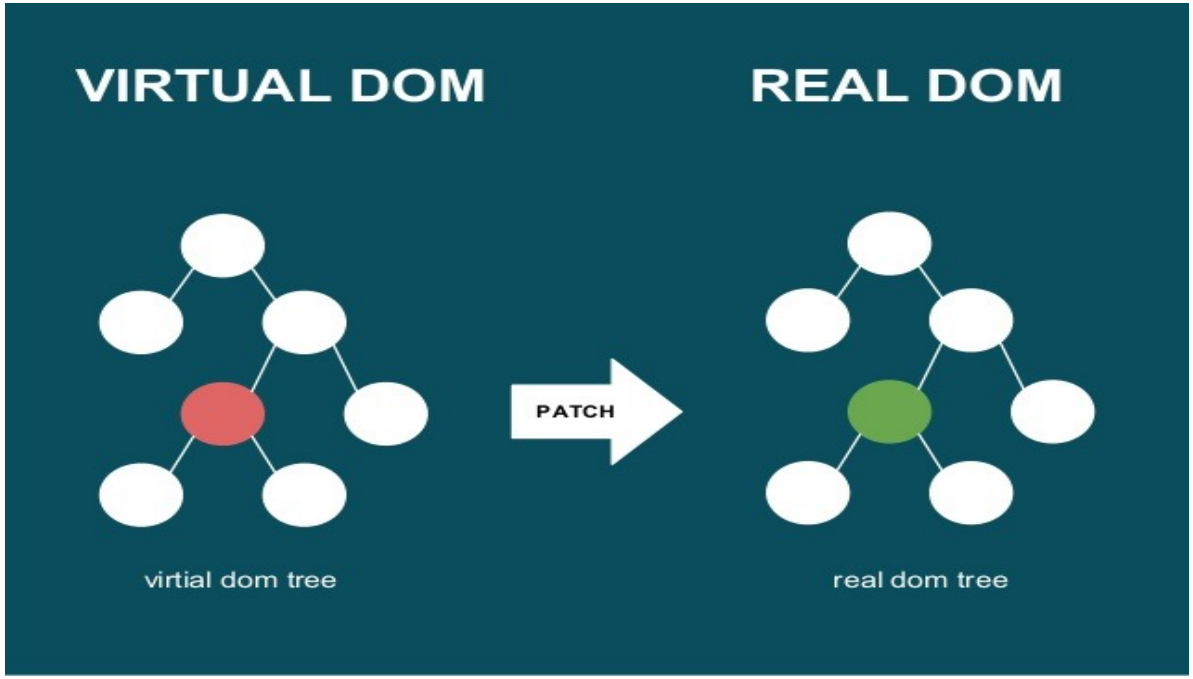
```
const SearchBox = props => {
  const onChange = event => {
    event.preventDefault();
    const newText = event.target.value.toLowerCase();
    props.handleChange(newText);
  };

  return <input type="text" placeholder="Search"
           onChange={onChange} />;
};
```

Summary

- **A state variable change always causes a component to re-render.**
  - **State change logic is usually part of an event handler function.**
  - **Event hadler may be in a subordinate component.**

- **Side effects:**
  - **Always execute at mount time.**
  - **The dependency array will either reference a state variable, a value computed from a state variable, or a prop.**
    - **Can be multiple entries**
  - **Callback performs the side-effect, and may also cause a state change.**

- **Data flows downward, actions flow upward.**



React internals.

Modifying the DOM

- **DOM – an internal data structure representing the browser's current 'display area' ; DOM always** in sync **with the display.**

- **Traditional performance best practice:**
  1. **Minimize access to the DOM.**
  2. **Avoid expensive DOM operations.**
  3. **Update elements offline, then reinsert into the DOM.**
  4. **Avoid changing layouts in Javascript.**
  5. **. . . etc.**

- **Should the developer be responsible for low-level DOM optimization? Probably not.**
  - **React provides a** *Virtual DOM* **to shield developer from these concerns.**

The Virtual DOM

- **How React works:**
  1. **It create a lightweight, efficient form of the DOM – the Virtual DOM.**
  2. **Your app changes the V. DOM via components' JSX.**
  3. **React engine:**
     1. **Perform *diff* operation between current and previous V. DOM state.**
     2. **Compute the set of changes to apply to real DOM.**
     3. **Batch update the real DOM.**

- **Benefits:**
  a) **Cleaner, more descriptive programming model.**
  b) **Optimized DOM updates and reflows.**

Unidirectional data flow & **Re-rendering**

(revised from previous lecture)

- **What happens when the user types in the text box?**

  User types a character in text box

  → *onChange* ***event handler executes***

  → ***Handler changes a state variable***

  → ***React re-renders*** *FriendsApp component*

  → ***React re-renders children (****FilteredFriendList****) with new prop values.***

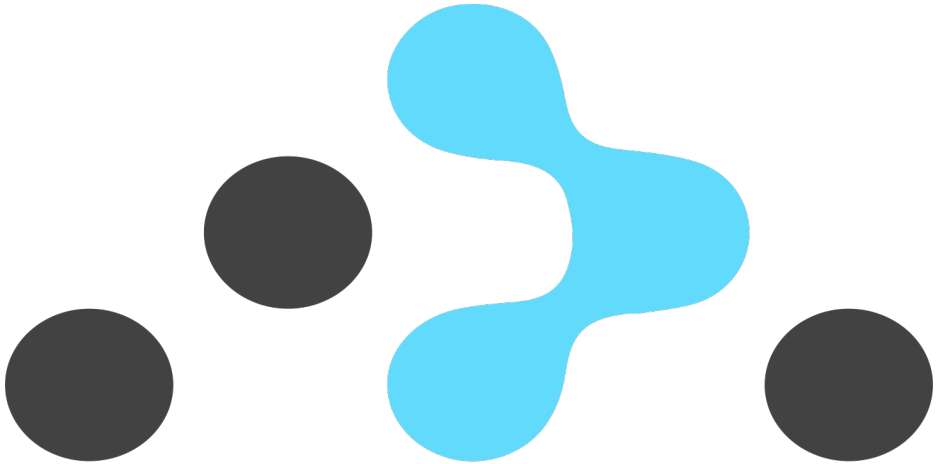
  → ***React re-renders children of*** *FilteredFriendList****.***

  → ***Some components may have unmounted, new ones (re)mounted.***

  → ***(Pre-commit) React diffs the changes between the current and previous <u>Virtual DOM</u>***

  → ***(Commit) React batch updates the <u>real DOM</u>.***

  → ***Browser repaints screen***



Navigation

The React Router

Routing - Introduction

- **Allows multiple views / pages in an app.**

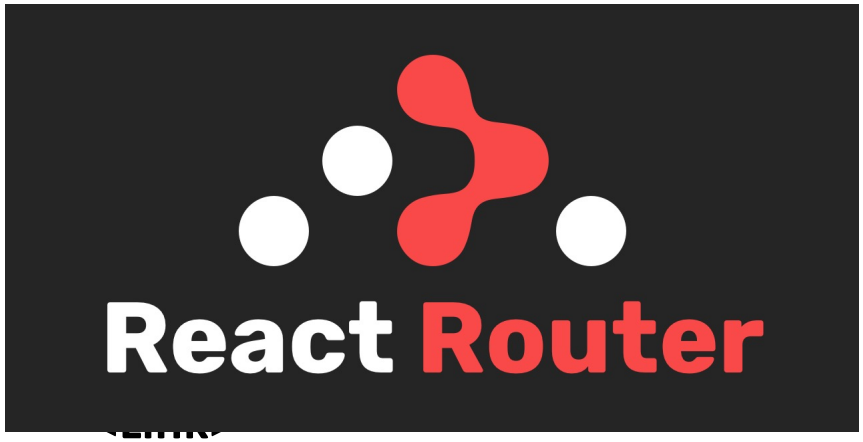
- **Keeps the URL in sync with the UI.**

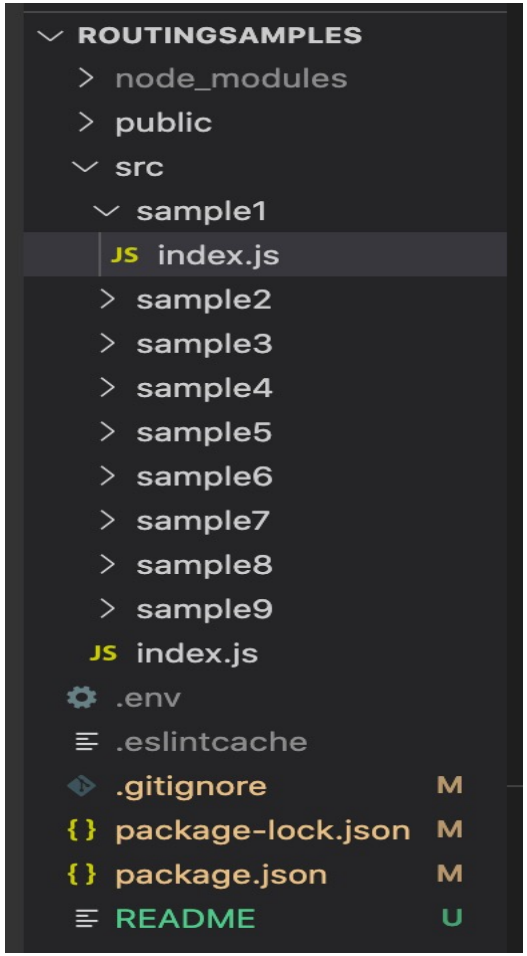
- **Adheres to traditional web principles:**
  1. **Addressability.**
  2. **Information sharing.**
  3. **Deep linking.**
  - **1st generation AJAX apps violated these principles.**

- **Not supported by the React framework.**
  - **A separate library is required: React Router.**

Demos

- **See the archive.**
- **Each sample demos a routing feature**





Basic routing configuration

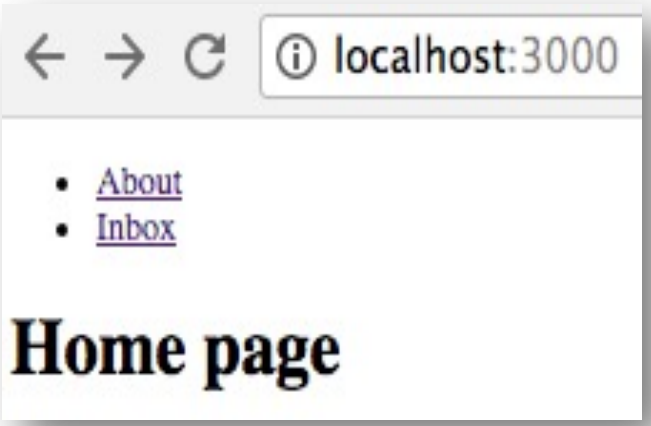
	URL	Components
1	/	Home
2	/about	About
3	/inbox	Inbox

```
17  const App = () => {
18    return (
19      <BrowserRouter>
20        <Switch>
21          <Route path="/about" component={About} />
22          <Route path="/inbox" component={Inbox} />
23          <Route exact path="/" component={Home} />
24          <Redirect from="*" to="/" />
25        </Switch>
26      </BrowserRouter>
27    );
28  };
29
30  ReactDOM.render(<App />, document.getElementById("root"));
```

- **Declarative routing.**
- <BrowserRouter> **- matchs browser's URL with a <Route> path.**
- **Matched** <Route**> declares component to be mounted.**
- <Route> **path supports regular expression pattern matching.**
  - **Use** exact **prop for precision.**
- Use <Redirect> **to avoid HTTP 404 error.**
- <Switch> **- <u>only</u> <u>one</u> of the nested Routs can be active.**
- ReactDOM.render**() passed an app's Router component.**
- **Ref. src/sample1**

Hyperlinks

- **Use the <Link> component for internal links.**
  - **Use anchor tag for external links - <a href . . . . . >**
- **Ref. src/sample2/**




```
6   const Home = () => {
7     return (
8       <>
9         <ul>
10          <li>
11            <Link to="/about">About</Link>
12          </li>
13          <li>
14            <Link to="/inbox">Inbox</Link>
15          </li>
16        </ul>
17        <h1>Home page</h1>
18      </>
19    );
20  };
```

Absolute URL

- **<Link> changes browser's URL address (event)**

  → React Router handles event by consulting its routing configuration

  → Component unmounting/mounting occurs → Browser updates screen

Dynamic segments.

- **Parameterized URLs, e.g. /users/*22*, /users/*12*/purchases**
  - **How do we declare a parameterized path in the routing configuration?**
  - **How does a component access the parameter value?**
- **Ex: Suppose the Inbox component shows messages for a specific user, where the user's id is part of the browser URL**

  e.g /inbox/123 where 123 is the user's id.

- **Solution:** <Route path='/inbox/:userId' component={ Inbox } />
  - **The colon (:) prefixes a parameter in the path; Parameter name is arbitrary.**
  - **Ref src/sample3**

Dynamic segments.

```
3
4   const BaseInbox = props => {
5     return (
6       <>
7         <h2>Inbox page</h2>
8         <h3>Messages for user: {props.match.params.userId} </h3>
9       </>
10    );
11  };
12
13  export default withRouter(BaseInbox);
14
```

- withRouter() **function: Returns a new, enriched component.**
  - **Injects routing props into a component:**
    - **props.match.params.(parameter-name)**
    - **props.history**
- **More than one parameter allowed.**

  e.g. /users/:userId/categories/:categoryName

Nested Routes

- Objective: A component's child is dynamically determined from the browser's URL (Addressability).
- EX.: (See src/sample4) Given the route:

  <Route path='/inbox/:userId' component={ Inbox } />,

  when the browser URL is:

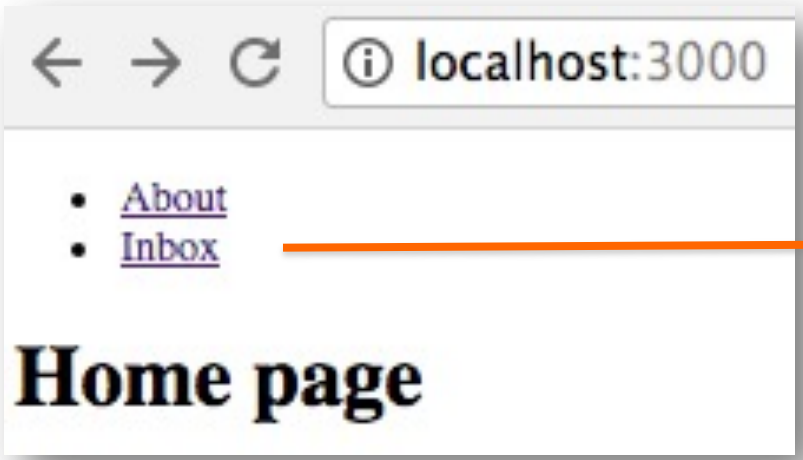
  1. /inbox/XXX/statistics render Inbox + Stats components.
  2. /inbox/XXX/draft then render Inbox + Drafts components.

```
const BaseInbox = (props) => {
  return (
    <>
      <h1>Inbox page</h1>
      <Messages id={props.match.params.userId} />
      <Route path={`/inbox/:userId/statistics`} component={Stats} />
      <Route path={`/inbox/:userId/drafts`} component={Draft} />
    </>
  );
};
```

Nested routes

Extended <Link>

- Objective: Pass additional props via a <Link>.
- EX.: See /src/sample5/.




```
const userId = 'id1234'
const beta = 'something else'

<Link
    to={{
    pathname: "/inbox",
    state: {
        user: userId,
        beta: beta
    }
    }}
>Inbox </Link>
```

```
const Inbox = props => {
  const { user, beta } = props.location.state;
  return (
    <>
      <h2>Inbox page</h2>
      <p>{`Link Props: ${user}, ${beta}`}</p>
    </>
  );
};
```

```
<Route path="/inbox" component={Inbox} />
```

Routing

- **More later**