LUND UNIVERSITY

React

EDAF90 Web Programming

Per Andersson

html and css

html and css

- great for static information
- separate semantic and layout

but lacks support for

- reuse/templates
- parameters
- iteration
- conditions
- dynamic behaviour
- most you find in any programming language . . .

solution

use JavaScript:

- generate html trees and add them to the DOM
- update the DOM when application state changes
- cumbersome when only using JavaScript
- early approach: libraries, such as jQuery, $, helps

Current approach: react, vue.js, angular

- components:
  - templates: mix of html/JSX and JavaScript
  - component state: JavaScript object
  - business logic: JavaScript functions
- framework synchronise template, application state and DOM

React render functions

```
function HelloWorld() {
  return (<h1>Hello, world!</h1>);
}
```

A render function:

- returns a DOM like tree
- easy to create instances
  - uses JSX to describe the structure
  - babel translates JSX to JavaScript code - builds a tree of react elements
- react injects the tree into the DOM
- re-render when needed

JSX - Build the DOM with expressions

JSX

- looks like html, built from “html tags” and react components
- all tags must be closed, xml syntax works: `<img />`
- must be one root tag
- can use multiple lines
- use () to avoid automatic semicolon insertion!!!
- Babel compiles JSX down to `React.createElement()` calls

```
const element = (
<span>
  <h1>Hello, world!</h1>
  <p>Some more text...</p>
</span>);
```

JSX attributes

- JSX tags can have attributes
- React DOM uses camelCase
- html: **`class`**, JSX: `className`
- html: **`for`**, JSX: `htmlFor`
- mapps to html attributes when injected to the DOM

```
const element1 = <div tabIndex="0"></div>;
const element2 = <img src="picture.jpeg" />;
```

Embedded JavaScript

JSX can contain embedded JavaScript

- syntax: { *JavaScript expression* }
- use in:
  - attribute values
  - tag content
- the embedded JavaScript expression may evaluate to another JSX expression

```
const name = 'Per';
const element1 = <h1>Hello {name}</h1>;

const imgSrc = 'picture.jpeg';
const element2 = <img src={imgSrc} />;

const element3 = (<span>{element1}
  <p>You are transformed to {doSomeMagic(name)}</p></span>);
```

Condition and iteration

Use JavaScript for conditional rendering and iteration

```
function MyWarning(props) {
  return (<h1> {props.message ?
           "Warning: " + props.message : "Well done" }
         </h1>);
}

function TodoList({arrayOfTodos}) {
  return (<ul>
            {arrayOfTodos.map(todo => <li>{todo}</li>)};
          </ul>);
}
```

React Component

Rect Component

react component:

- JavaScript function
  - called when the components need rendering
  - must return a react DOM (a JSX expression)
- instantiate it in JSX using the function name, example: `<HelloWorld />`
- function name must start with capital letter
- in JSX:
  - lowercase tag name - standard html element
  - uppercase tag name - a react component

Component Example

```
function HelloWorld(){
  return (
    <>
      <h1>Hello</h1>
      <p>EDAF90 – web programming</p>
    </>
  );
}
```

Attributes - `props`

a parent can pass data to a child

- parent set the value using attributes in JSX
- the values are passed in a `props` object
- you can use any JavaScript value, including arrays, objects and functions

Prop Example

```
const element = (<Hello name="Per"/>);

function Hello(props) {
  return (
    <span>
      <h1>Hello {props.name}</h1>
    </span>
  );
}
```

Prop children Example

`props` also contain the children of the component, an object or an array.

```
const element =
(<Hello name="Per">
  I am a child <Button />
</Hello>);

function Hello(props) {
  return (
    <span>
      <h1>Hello {props.name}</h1>
      <p>{props.children}</p>
    </span>
  );
}
```

Event Handling

Handling Events

- event names are camelCase in JSX
- must use `preventDefault`, returning **`false`** do nothing
- pass a function: `onClick ={myCallbackFunction}`
- called with one argument, a synthetic event according to the W3C spec

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }
  return (
    <a href="#" onClick={handleClick}>Click me</a>
  );
}
```

State

Component State

State is a components memory:

- preserves data between renderings
- managed by react
- use state hook to get a snapshot of the state
- update using a set function

useState hook

```
function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} times
    </button>
  );
}
```

Render Cycle

1. render - root element and all children
2. commit - update the DOM
3. wait for external event
4. call all event handlers, queues all state updates
5. execute the state updates in the queue
6. the event handler updates the state
7. `if (Object.is(oldState, newState)goto 1 else goto 3`

The render function must be a pure function of state and `props`.

State over time

use state at render time:

```
const [cnt, setCnt] = useHook(0);
handleClick(){
  setCnt(cnt+1);
  setCnt(cnt+1);
}
```

use the latest state:

```
const [cnt, setCnt] = useHook(0);
handleClick(){
  setCnt(newCnt => newCnt+1);
  setCnt(newCnt => newCnt+1);
}
```

Object and Array in State

- state must be immutable
- a new value triggers re-render
- compares using `Object.is`, must be a new object or array to trigger re-render
- copy object: `newState = {...oldState}`
- copy array:
  - `newState = [...oldState]`
  - `newState = oldState.map(`
    `(e, index)=> index===modifyIndex ? newElement : e)`
- safe array functions:
  - `concat, filter, slice, map`
  - or copy first `[...arr]`
- deep structures, copy all modified objects:
  `{outer..., inner: { outer.inner, v: newValue }}`

Rules for State

- in render functions - treat state as read only
- only update state in event handlers
- event handlers are local functions in the render function
  - `[state, setState] = useHook()` is in the closure
  - `setState` do not change the local snapshot
  - re-render ⟶ new closure and event handler functions
  - handlers always have the state viewed by the user
- the state must be an immutable data structure, copy and modify
- setting the state to a new value triggers a re-rendering of the react tree, the root component and its children
- react updates the state after all event handlers finish

Lifting State Up

Child → Parent communication

- prop pass data down in the tree
- data can be any JavaScript value, including functions
- pass the `setState` function:
  - children can update the parents state
  - only use in handlers, never during rendering

Sibling ↔ Sibling communication

- lift upp state - store the state in their nearest common ancestor
- drill down props - pass the state trough all intermediate components

Lifting State Up

```
function GreatestCommonAncestor() {
  const [cnt, setCnt] = useState(0);
  function handleClick() { setCnt(cnt + 1); }
  return (<><MyView cnt={cnt} /><MyButton cnt={cnt} onClick={
     handleClick} /></>)
}

function MyView({ cnt }) {
  return (<p>counter is {cnt}</p>);
}
function MyButton({ cnt, onClick: handleClick }) {
  return (<button onClick={handleClick}>you have clicked me {
     cnt} times</button>);
}
```

Lists and key

Lists and key

- JavaScript embedded in JSX may return a collection of react elements
- updating the DOM is expensive
- react only update parts of the DOM that have been changed
- you must help react how an array changes:
  - each element must have a `key` property
  - unique among siblings
  - the value must be preserved over time
  - avoid array index as key (changes when elements are deleted)
- using the key react can detect
  - elements changed value
  - elements have been added to the list
  - elements have been deleted from the list

`key` example

```
function MyList(props) { return (
  <ul>
    {props.list.map(row => (
      <li key={row.id}>
        {row.text}
      </li>
    ))}
  </ul>);
}
```

Hooks —
Connecting to react internals

React Basics

Render code

- computes the react DOM tree from `props` and state
- must be pure
- no control of when or how often it is called (console.log will be messy)

Event handlers

- call is triggered by external events like user actions or network trafic
- changes steate

Hooks

- react basics do not cover all needs
- hooks into the internals of react
- covers more use cases than react basics
- named `useSomething()`
- you have seen `useState()` which provides memory to the componets

Rules of hooks:

- called from render code directly, or other hooks
- must be called in the same order every re-render
- only call from top level (not insida conditions or loops)

Effects

“side effects that are caused by rendering itself, rather than by a particular event”

- example: fetch data from a server when the component is viewed

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Code here will run after *every* render
  });
  return <div />;
}
```

Effects

```
useEffect(() => {
  // initialisation_code
  return
    () => {/* clean_up_code */}
  }, [list of dependencies]);
```

- `initialisation_code` is always followed by `clean_up_code`
- `initialisation_code` is run after component is mounted in the DOM, and when a variable in the dependency list is updated
- `clean_up_code` is run when the component is removed from the DOM and before `initialisation_code` is run due to a value change

Effects in Ticker example

```
import { useEffect, useState } from 'react';

function Ticker({delay}) {
  const [cnt, setCnt] = useState(0);

  useEffect(() => {
    const id = setInterval(_=> setCnt(cnt => cnt+1), delay);
    return () => {clearInterval(id)}
  }, [delay]);

  return (<p> ticks: {cnt} </p>);
}
```

Effects - Running Ticker

component mounts:

- run `initialisation_code` (closure 1)

`delay` is changed:

- run `clean_up_code` (closure 1)
- run `initialisation_code` (closure 2)

`delay` is changed:

- run `clean_up_code` (closure 2)
- run `initialisation_code` (closure 3)

. . . component is deleted:

- run `clean_up_code` (closure 3)

Effects - Development mode

In development mode:

- components are mounted twice (and deleted once)
- stress test the clean up code in effects
- will mess upp any `console.log`

Context — broadcasting in a tree

Context, broadcasting in a tree

- `props` are convenient when communication components are few and close
- do not scale
- context can broadcast a value to all components in a tree
- three steps:
  - create the context
  - provide the value for a subtree
  - us the value in a component

Context, create

Create the context in a separate file: MyContext.js

```
import { createContext } from "react";

export const MyContext = createContext('default value');
```

Context, provide a value

Provide a value in a JSX expression

```
import { MyContext } from 'MyContext';

function App() {
  return (
    <MyContext.Provider value={'top level'}>
      <ViewMyContext />
      <MyContext.Provider value={'subtree'}>
        <ViewMyContext />
      </MyContext.Provider>
    </MyContext.Provider>
  );
}
```

Context, use

Use the context in a component

```
import { useContext } from 'react';
import { MyContext } from './MyContext.js';

export function ViewMyContext() {
  const myValue = useContext(MyContext);
  return <p>context is: {myValue}</p>
}
```

Reducers —
Separating state logic

Reducers

Using handelrs:

- `nextState = handler(event, currentState)`
- mixing UI code with state logic
- hard to reuse state logic in different UI components

Reducers

- UI code emits actions
- `nextState = reducer(action, currentState)`
- easy to reuse state logic
- clearer which operations are allowed on the state
- easy to test state logic

Action — a plain JavaScript object

- `type` property, which action to execute on the state
- contains all information needed to perform the action

```
{
  type: "Added",
  task: { id, text, moreData}
}
```

Reducer — a plain JavaScript function

```
function tasksReducer(tasks, action) {
  switch (action.type) {
    case 'deleted': {
      return tasks.filter((t) => t.id !== action.id);
    }
    // more actions
    default: {
      throw Error('Unknown action: ' + action.type);
    }
  }
}
```

useReducer

```
function MyComponent() {
  const [tasks, dispatch] = useReducer(tasksReducer,
     initialTasks);

  function handleChangeTask(task) {
    dispatch({
      type: 'changed',
      task: task,
    });
  }

  return <button onChange={handleChangeTask}>update</button>}
}
```

Redux —
combining context and reducers

Redux

- first introduced by facbook
- soved the never ending message count bug
- simplifies component dependencies
- component $\longrightarrow$ state $\longrightarrow$ component

Redux — TasksContext.js

```
const TasksContext = createContext(null);
const TasksDispatchContext = createContext(null);

export function useTasks() {
  return useContext(TasksContext);
}

export function useTasksDispatch() {
  return useContext(TasksDispatchContext);
}

function tasksReducer(tasks, action) {
  // your code
}
```

Redux — TasksContext.js

```
export function TasksProvider({ children }) {
  const [tasks, dispatch] = useReducer(
    tasksReducer,
    initialTasks
  );

  return (
    <TasksContext.Provider value={tasks}>
      <TasksDispatchContext.Provider value={dispatch}>
        {children}
      </TasksDispatchContext.Provider>
    </TasksContext.Provider>
  );
}
```

Redux — App.js

```
function App() {
  return (
    <TasksProvider>
      <MyAppComponents />
    </TasksProvider>
  );
}
```

Redux — MyComponent.js

```
function MyComponent() {
  const tasks = useTasks();
  const dispatch = useTasksDispatch();

  function handleAdd(text){
       dispatch({
          type: 'added',
          text: text,
        });
       }
  }
  return (
    <MyUI onAdd={handleAdd} data={tasks} />
  );
}
```

Summary — Rules of React

Rules of React

render code

- called by react, when needed
- must be pure function of `props` and state (`useState`, `useReducer`)
- no side effects, do not update state
- may use hooks

Rules of React

event handlers and callback passed to `useEffect`

- called by the browser
- update state to trigger re-render
- declare inside component function
  - operates on data viewed by the user
  - `setState` and `dispatch` in the closure
  - latest state: `setState(currentState => computeNewState)`

Rules of React

state

- owned by react
- read by render code, get a snapshot
  - `useState`
  - `useReducer`
- update
  - `setState` in event handlers and effect callbacks
  - reducers computes the next state
- immutable data structure

Rules of React

hooks

- called by your render code and custom hooks
- must be called in the same sequence every re-rener
- call from top level, not in conditions or loops
- may call other hooks
- pure functions

Rules of React

reducer

- called by react, once for each a `dispath()`
- pure function
- remember, state is immutable

Bootstrapping React

`ReactDOM.render()`

- updates the DOM
- takes two parameters:
  - a react element (JSX expression)
  - a DOM element
- updates the DOM if the element already was part of the DOM
- optimised, only updated the delta

bootstrap the app

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