Agenda

React

React & State

React & Redux

Reading Redux state

Triggering Actions

Reducing into state

Redux Middleware

Functional programming concepts

Hooks

Component Lifecycle

- Classes vs Hooks

Context

Composition

Building tools & webpack

Context & Composition

Composition

Higher Order Components =

Function that takes a component as arguments and renders/adapts them

typically named withXYZ()

Render props

Props that contain a function that returns a component instance when called.

Can be any prop that is called in components render (including props.children).

Context

Allows sharing data across a subtree of components

Context

Code for composition & context examples...

Functional Programming

JS/TS

TypeScript is a strict syntactical superset of **JavaScript**, with **optional static typing**. Existing JavaScript programs are also valid TypeScript programs.

JavaScript is a high-level, **multi-paradigm**, interpreted language using dynamic typing, prototype-based object orientation and **first class functions**. JavaScript supports event driven, **functional** and imperative (including object-oriented and prototype-based) **programming** styles.

JS/TS

Multi-paradigm means you are responsible for how your program is designed.

Very easy to depart from functional structures.

Principles of Functional Programming

Functions as first class constructs

Immutability

Purity

Recursion instead of looping

Principles of Functional Programming

Functions as first class constructs

Functions can be passed as parameters and stored in variables

Functions always return a value

JS

Functions are first class constructs

Functions return undefined if no explicit return is used (functional contract upheld)

Principles of Functional Programming

Higher Order Functions

= Functions that take functions as parameters and/or return functions

Used to compose the behavior of a program using functions.

JS/TS

Higher Order Functions = Functions that take functions as parameters and/or return functions

Principles of Functional Programming

Immutability

Values (state) of an object cannot be changed

Helps reduce complexity

Can be used to increase execution efficiency

JS/TS

Immutability - upheld for String, Number, undefined, null

Use Object.assign, Spread Operator, JSON.stringify / parse to uphold immutable structures.

Deep cloning in code is relatively performant

Readonly, Object.freeze and libraries like Immutable.js can help

Principles of Functional Programming

Purity

Pure Functions always return the same value for a specific set of inputs.

Principles of Functional Programming

Non-pure = Side Effects

Change outside a function when a function is called.

Non-pure functions have side effects

Principles of Functional Programming

Special case with pure functions and arguments with side effects

Use of closures can result in non-pure functions

Ways to make functions pure again

- Dependency injection
- Effect functors (higher order functions)

JS/TS

Pure functions - just make sure you have no side effects.

Very easy to have side effects and mutate data

- all real world application functionality depends on it

Principles of Functional Programming

No looping instead recursion with a function

Functional array manipulation as built in functions on arrays.

.map

.reduce

.slice

Higher Order Functions, Currying, Partial Application & Bind

Higher Order Functions, Currying, Partial Application

Currying - splitting functions with multiple parameters into multiple functions with a single parameter. Supports higher order function constructs. Utilizes closures.

Partial application - Preparing a function hierarchy with some values to make different higher order functions from a single base possible.

.bind as JS escape hatch

Functional programing in React

JavaScript only has functions, no classes...

React class component behavior can be constructed by using regular JS functions and prototypal inheritance.

Is a a pain in the a$$ and excludes novice JavaScript devs.

Class component syntax & TS just shorthand for the required complex functional structures.

JSX/TSX

JSX/TSX is actually just shorthand for loads of function calls

Each element is compiled into React.createElement(...) calls and pushed into the virtual DOM

Components are not actually rendered at all.

Functional components

Components take props and transform them into UI using createElement()

```
function MyComponent(props) {
 return <div>props.title</div>;
}
```

Functional TS components

Typescript React.FunctionComponent

```
const MyComponent: React.FC<Props> = (props) => {
 return (<div>props.title</div>)
}
```

Purity

React Components = Functions

Pure Functions = Pure Components

Just like Pure functions Components which are only a result of their inputs are easier to understand, debug and reason about!

Pure Components always result in the same output for a set of props values.

Data Flow

Unidirectional flow

- reducing component dependencies and imperative coding complexities

The only communication between component instances is through props.

Props

Props fully define a pure components behavior

= a component doesn't own it's props

= props must be treated as immutable

Passing functions

Component instances cannot be connected to each other in a reasonable way. A component cannot have direct side effects on other components. Dependency injection to allow controlled side effects.

= pass a function that triggers the externa effect

Still a pure component since props allow function values

State

Component state breaks purity - makes your application harder to reason about and more likely to have bugs.

State is a required attribute of any non-trivial application!

Higher order components

Higher order components

Higher Order Components =

Function that takes a component as parameter and returns a component

Remember components are functions…

= functional programming

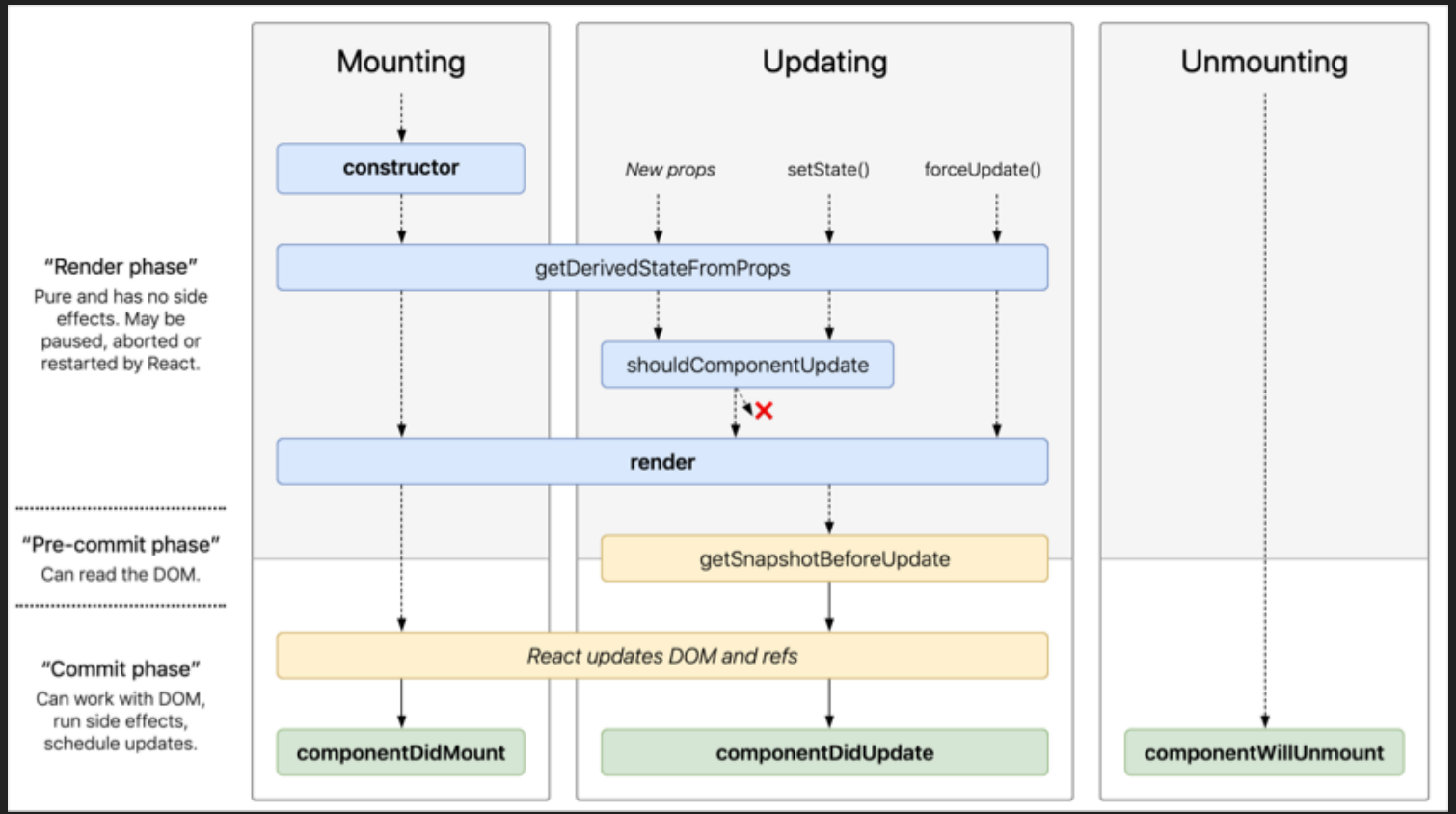
Component lifecycle classes vs functions

Component Lifecycle - only in Class components

Allows for controlling side effects such fetching data and browser interactions

Follows a strict order - functions automatically called by React

Component Lifecycle - only in Class components



Hooks

Difference between hooks and class state/lifecycle

Lifecycle supports predetermined events and are called by React at specific points in time on class based components

Hooks offer a more flexible model for customizing behavior using a number of built in hooks, composable into more complex behavior.

Hooks

Hooks are another step towards more functional constructs in React.

Allows function components to use state and interact with the component lifecycle

Syntax uses destructuring

Hooks

useState(initialValue)

useEffect(didUpdate, valueConditionsArray)

useContext(context)

Hooks

State hooks - state properties are unnamed so order the order of useState calls determines which values are tracked into the destructuring.

```
const [state, setState] = useState(initialState);
```

calling setState does not merge properties - state is fully replaced, updates can use current state.

```
setState(newState);
```

Hooks

Effect hooks (escape hatch) runs on every render when sen

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

Hooks

Context hook consume context

```
const value = useContext(MyContext);
```

Hooks

Custom hooks - combine other hooks in a function named useXYZ

Building react app

Task runner = react-scripts

Transpiling TS, TSX, JSX

Pack for web (Webpack…)

Webpack & React - CRA

Full transpiling and dependency management

Scripts and stylesheets get minified and bundled together to avoid extra network requests.

Missing files cause compilation errors instead of 404 errors for your users.

Result filenames include content hashes so you don’t need to worry about browsers caching their old versions.

Webpack

Making it work for the web

Import other types (png, pdf …)

Automatically deploy and parse web files such as: css, css modules, sass

Browserlist support to setup css prefixes and workarounds

Public folder for non module based files

Babel & transpiling

Default setup that cannot be changed without customizing the entire build process

Eject allows full customization

Adapting in place without ejecting

npm packages that rewire webpack etc

Adapting react-scripts - fork react-scripts and deploy custom npm package

Agenda

React

React & State

React & Redux

Reading Redux state

Triggering Actions

Reducing into state

Redux Middleware

Functional programming concepts

Hooks

Component Lifecycle

- Classes vs Hooks

Context

Composition

Building tools & webpack

Tack

& Övriga frågor