DOM rendering adheres to a pull model of grabbing and rendering when the runtime engine is ready rather than allowing user code to push render commands

Virtual DOM allows React to be render engine agnostic (enabling React Native, etc) - allows forming engineering teams around products rather than platforms

Design principles composition scalability common abstraction stability interop schedule management experience tooling

elements are smallest UI unit and are different from components React only updates what is necessary by diffing elements when walking the DOM tree

Components are similar to functions -> props are arbitrary inputs for basic cases, functional (with prop inputs) vs Class components with a render method are equivalent from the runtime's point of view Components are composable and can be composed of user-defined components props are read-only -> like functional pure functions (components must act like pusre functions with respect to props)

- using PropTypes is good because
  - gives a clear declaration of the shape of a component's properties
  - removes need to be defensive when interacting with props (offloading to runtime)

attach state as this.state in constructor (always call super(props)) be sure to free resources (e.g. timers, etc) with the component's lifecycle using componentDidMount and componentWillUnmount (lifecycle hooks)

this.state has special meaning so adding additional fields to class is fine and can be done at will as needed

tick method is built-in and runs every second (could've used this for timer updates)

state

- do not modify state directly except for in ctor (use setState() -> this re-renders a component)
- state updates may be async (props and state updates may be batched by react internally)
  - do not rely on their state to calculate the next state
  - setState has a special form setState((prevState, props) => {}) to handle thi
- state updates are merged with previous state (shallow merging)
- state should remain encapsulated and should be passed down from parent to child via props

events

- similar to standard DOM
- named with camelCase
- pass a function as handler (rather than string)
- cannot return false to prevent default behavior (must call e.preventDefault() explicityl)
- React defines synthetic events according to the W3C spec

- generally do not need to explicitly call addEventListener
- need to bind methods to this in constructor if passing method as event handler
  - JS methods are not bound by default
  - can use experimental property initializer syntax (propName = () => {}) (enabled by default in create-react-app
  - can also use arrow functions in the callback (downside is that it creates a different function each time, which is usually fine but triggers a re-render if passed as a prop to children)

conditional rendering

- use if statements, switch statements, etc in render
- can assign and store elements in variables
- can embed any expressions in JSX with {}
- to prevent component from rendering, return null

lists and keys

- can use map and return elements to render multiple elements
- when creating lists of elements, need to add a key attribute on the element which is a unique string
- keys are used by react to identify modified elements (only need unique IDs in list items amongst siblings)
- use the item index as a last resort
- keys only make sense in the context of the surrounding array

forms

- HTML form elements naturally keep internal state in React
- form submission and navigation can be controlled using controlled components
- forms elements naturally manage and mutate internal state
  - assign state var to value of element and assign change event handler to onChange
  - call setState in change handler, element re-renders on change and thus makes the state var the single source of truth
- textarea uses value attribute in React
- select uses value attribute on outer element in React to set current value
- to handle multiple input elements add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name
- can use uncontrolled components as an alternative

lifting state up

- recommended to lift shared state up to closest common ancestor
- add state and handler to ancestor, handler should refer to e.target.value
- pass state as prop through child chain to target, along with handler

- use one single source of truth, add state to component that needs it first, and when another needs it lift it up to closest common ancestor

composition vs. inheritance

- prefer composition over inheritance
- components like generic boxes do not know children ahead of time, should use props.children as rendered output of the component in that case e.g. {props.children}
- specialization e.g. Widget -> ContainerWidget, is achieved through composition by passing props to the more generic component that configures the component to render as needed
- to reuse functionality across components, extract into shared non-UI class for reuse
- FB has not discovered any inehritance use-cases that cannot be solved more easily than using props and composition

steps for development

1. break UI into a component hierarchy
2. build static version in react
3. identify minimal but complete representation of UI state
4. identify where state should live (e.g. which components or where in the component hierarchy)
5. add inverse data flow, e.g. modify state higher in component hierarchy tree by passing callbacks down the component hierarchy

JSX

- syntactic sugar for React.createElement(component, props, ...children)
- component - a react component which must be in scope
  - scope is either defined by module system or accessible through the global React when loading through the script tag
  - can access scoped components through dot notation (e.g. MyComponents.DatePicker) when a module exports many components
- user defined components must be capitalized
- reading dynamic components into the result of an expression requires the target var to be capitalized
- props can be specified in many ways
  - wrapped in {} in parent
  - if statements and for loops are not expressions in js so cannot be used directly in JSX
  - can pass a string literal as a prop, which is then HTML unescaped
  - props default to true
  - can use spread operator to pass attributes directly
- when passing open and close tags, anything between is accessible as props.children
  - booleans, null and undefined are ignored
  - some falsey values are still rendered

PropTypes

- requires prop-types library
- can use Flow or TypeScript instead
- assign special .propTypes property which export validators on the resulting props assigned to a Component
- primitives, node, element, instanceof a class, limited to one of a list of enum, one of many types, array of a type, object with properties of type, an object with a shape
- use isRequired to warn if a prop is not provided
- can specify custom validators (also for arrayOf and objectOf)
- can require that it only has a single child
- can specify default props through .defaultProps

Refs and the DOM

- parents usually just communicate via props, rerender with new props
- React provides escape hatch to handle special cases to interact directly with DOM or elements
    - manage focus, text selection or media playback
    - trigger imperative animations
    - integrate with 3rd party DOM libraries
- avoid using refs
- pass ref callback which receives the DOM element and can then be stored on the Component
- may not use refs on functional components
- can very rarely pass child refs to parent by exposing a special prop in child and passing a callback to the child via the parent through that prop which then assigns ref to parent prop
- if ref callback is inlined then it well get called twice during updates (can avoid by defining callback as a bound function on a class)

uncontrolled components

- use refs instead of callbacks
- use defaultValue on the element

optimizing performance

- use the production build
- profile components using the Chrome performance tab
- React decides if a component should be updated (if state or props has changed)
- can override shouldComponentUpdate(nextProps, nextState) and return false or true for special situations
- avoid mutation
- use immutable data structures

reactive programming

It is convenient to distinguish roughly between three kinds of computer programs. Transformational programs compute results from a given set of inputs; typical examples are compilers or numerical computation programs. Interactive programs interact at their own speed with users or with other programs; from a user point of view, a time-sharing system is interactive. Reactive programs also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not the program itself. Interactive programs work at their own pace and mostly deal with communication, while reactive programs only work in response to external demands and mostly deal with accurate interrupt handling. Real-time programs are usually reactive. However, there are reactive programs that are not usually considered as being real-time, such as protocols, system drivers, or man-machine interface handlers.

https://gist.github.com/staltz/868e7e9bc2a7b8c1f754 http://www.reactivemanifesto.org/ https://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming

Reactive programming is programming with asynchronous data streams. On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams. A stream is a sequence of ongoing events ordered in time. It can emit three different things: a value (of some type), an error, or a "completed" signal. Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.

We capture these emitted events only asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted. Sometimes these last two can be omitted and you can just focus on defining the function for values. The "listening" to the stream is called subscribing. The functions we are defining are observers. The stream is the subject (or "observable") being observed. This is precisely the Observer Design Pattern.

Conal Elliott Functional Reactive Programming (FRP) is a specific programming model with a specific semantics. (Actually, there are two variants, which are sometimes called "classic FRP" and "arrow FRP".) I've given a summary in an answer to "What is (functional) reactive programming?". As I said there, the two key properties for me have always been (a) precise & simple denotation and (b) continuous time. I regret that this model came to be called "functional reactive programming", for a few reasons:

That name omits "time", and time is central for me. The term "functional" has so little specific/clear meaning. I much prefer Peter Landin's suggested replacement "denotative". (See the quotes and reference in this blog comment.) It's easy for people to incorrectly think they know what the term means because they know meanings (more or less) of each of the three words. For descriptiveness & accuracy, I prefer the term "denotative continuous-time programming" (suggested by Jake McArthur in a conversation a while back) over "functional reactive programming".

I wrote a very short piece on the origin of FRP in the blog post Early inspirations and new directions in functional reactive programming.

http://conal.net/blog/posts/early-inspirations-and-new-directions-in-functional-reactive-programming

Discrete[edit] Formulations such as Event-Driven FRP and Elm before version 0.17 require that updates are discrete and event-driven.[3] These formulations have pushed for practical FRP, focusing on semantics that have a simple API that can be implemented efficiently in a setting such as robotics or in a web-browser.[4]

In these formulations, it is common that the ideas of behaviors and events are combined into signals that always have a current value, but change discretely.[5]

Continuous[edit] The earliest formulation of FRP used continuous semantics, aiming to abstract over many operational details that are not important to the meaning of a program.[6] The key properties of this formulation are:

Modeling values that vary over continuous time, called "behaviors" and later "signals". Modeling "events" which have occurrences at discrete points in time. The system can be changed in response to events, generally termed "switching." The separation of evaluation details such as sampling rate from the reactive model. This semantic model of FRP in side-effect free languages is typically in terms of continuous functions, and typically over time.[7]

Redux Motivation

- code must manage more state than ever before and increasing in complexity in UI
- over time, systems become opaque and programmer loses control over understanding of the system
- more demands on web-based UI devs
- requires mixing asynchronicity and complexity which are very error prone

from Flux

flux-concepts

These are the important high-level concepts and principles you should know about when writing applications that use Flux.

Overview

Flux is a pattern for managing data flow in your application. The most important concept is that data flows in one direction. As we go through this guide we'll talk about the different pieces of a Flux application and show how they form unidirectional cycles that data can flow through.

Flux Parts

- Dispatcher
- Store
- Action
- View

Dispatcher

The dispatcher receives actions and dispatches them to stores that have registered with the dispatcher. **Every store will receive every action.** There should be only one singleton dispatcher in each application.

Example:

1. User types in title for a todo and hits enter.
2. The view captures this event and **dispatches** an "add-todo" action containing the title of the todo.
3. **Every store** will then receive this action.

Store

A store is what holds the data of an application. Stores will register with the application's dispatcher so that they can receive actions. **The data in a store must only be mutated by responding to an action.** There should not be any public setters on a store, only getters. Stores decide what actions they want to respond to. **Every time a store's data changes it must emit a "change" event.** There should be many stores in each application.

Examples:

1. Store receives an "add-todo" action.
2. It decides it is relevant and adds the todo to the list of things that need to be done today.
3. The store updates its data and then emits a "change" event.

Actions

Actions define the internal API of your application. They capture the ways in which anything might interact with your application. They are simple objects that have a "type" field and some data.

Actions should be semantic and descriptive of the action taking place. They should not describe implementation details of that action. Use "delete-user" rather than breaking it up into "delete-user-id", "clear-user-data", "refresh-credentials" (or however the process works). Remember that all stores will receive the action and can know they need to clear the data or refresh credentials by handling the same "delete-user" action.

Examples:

1. When a user clicks "delete" on a completed todo a single "delete-todo" action is dispatched:

```
{
  type: 'delete-todo',
  todoID: '1234',
}
```

Views

Data from stores is displayed in views. Views can use whatever framework you want (In most examples here we will use React). **When a view uses data from a store it must also subscribe to change events from that store.** Then when the store emits a change the view can get the new data and re-render. If a component ever uses a store and does not subscribe to it then there is likely a subtle bug waiting to be found. Actions are typically dispatched from views as the user interacts with parts of the application's interface.

Example:

1. The main view subscribes to the TodoStore.
2. It accesses a list of the Todos and renders them in a readable format for the user to interact with.
3. When a user types in the title of a new Todo and hits enter the view tells the Dispatcher to dispatch an action.
4. All stores receive the dispatched action.
5. The TodoStore handles the action and adds another Todo to its internal data structure, then emits a "change" event.
6. The main view is listening for the "change" event. It gets the event, gets new data from the TodoStore, and then re-renders the list of Todos in the user interface.

Flow of data

We can piece the parts of Flux above into a diagram describing how data flows through the system.

1. Views send actions to the dispatcher.
2. The dispatcher sends actions to every store.
3. Stores send data to the views.

- *(Different phrasing: Views get data from the stores.)*

Data flow within Flux application

(There is also another node in the diagram accounting for actions that do not originate from views, which is common)

CQRS (https://martinfowler.com/bliki/CQRS.html)

- started with CRUD CQRS naturally fits with some other architectural patterns.

As we move away from a single representation that we interact with via CRUD, we can easily move to a task-based UI. CQRS fits well with event-based programming models. It's common to see CQRS system split into separate services communicating with Event Collaboration. This allows these services to easily take advantage of Event Sourcing. Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using eventual consistency. For many domains, much of the logic is needed when you're updating, so it may make sense to use EagerReadDerivation to simplify your query-side models. If the write model generates events for all updates, you can structure read models as EventPosters, allowing them to be MemoryImages and thus avoiding a lot of database interactions. CQRS is suited to complex domains, the kind that also benefit from Domain-Driven Design.

In particular CQRS should only be used on specific portions of a system (a BoundedContext in DDD lingo) and not the system as a whole. In this way of thinking, each Bounded Context needs its own decisions on how it should be modeled.

beware that for most systems CQRS adds risky complexity.

CQS - It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, Asking a question should not change the answer.[1] More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.

Command query responsibility segregation (CQRS) applies the CQS principle by using separate Query and Command objects to retrieve and modify data, respectively

Event Sourcing https://martinfowler.com/eaaDev/EventSourcing.html We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.

Core Concepts

- to change state dispatch an action
    - actions have clear types which facilitates an understanding of everything that is going on in an application
- can describe an application’s state as a plain object
- reducers tie actions to state as functions which handle actions (events) and return a modified state
- best to split these up to manage specific portions of overall state, thus scoping reducers
- main idea is that describe how application state is updated over time in response to actions

Three Principles

- single source of truth

    - the state of your whole application is stored in an object tree within a single store

- state is read-only
    - the only way to change the state is to emit an action, an object describing what happened
- changes are made with pure functions
    - to specify how the state tree is transformed by actions, you write pure reducers
- see Flux, Elm, RxJS and Baobab for inspiration

Actions

- payloads of data sent from application to store
- only source of information for store
- must return an object with a type property which specifies the action type