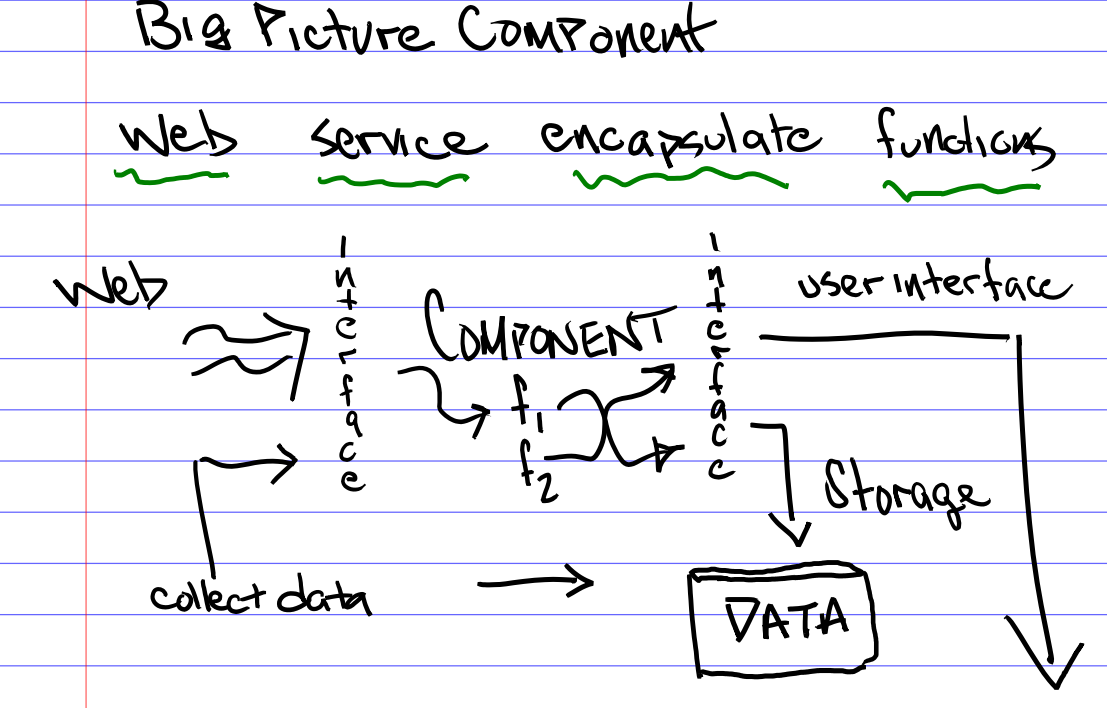
7/7/20
edit 11/9/20

Components

Big Picture Component

Web service encapsulate functions



A component is defined by the system that uses it.

Two way effect

A system is a group of interacting entities that form a unified whole.

A system is described by its spatial and temporal boundaries, surrounded and influenced by its environment, described by its structure and purpose and expressed by its functioning.

History of the word System

1824 Sadi Carnot
System ≐ Working Substance
e.g. water vapor in steam engine

1850 Rudolf Clausius
Included the surroundings

system = Working Substance + surroundings

1945 Ludwig von Bertalanff

General Systems theory

models, principles and laws that apply to generalized systems.

System theory views the world as a complex system of interconnected parts.

A system is scoped by defining its boundary

Model System: ① structure & ② behaviour — system | environment (all that is not in the system)

Open Systems

- free exchange of matter with environment (Gaia/Earth)
- e.g. ① Gaia/Earth
  ② data networks

Closed System

- exchange of energy
- e.g. computer, Biosphere 2

System Model

- multiple views
  concept, analysis, design
  implementation, deployment
  structure, behavior, input data
  output data views.

In Information & Computer Science

system is a { Hardware System + Software System }

which has

① components as its structure

② Inter-process communication as its behavior

Components in React

Before React

Class Component Pseudo-code.

```
Form extends ViewObject
  render
    this.button = new Button
    this.el.append(button)

  later...                     form is handling DOM
    this.el.replace(new Message)
```

React Elements

An element is a plain object describing a component or a DOM node.

An element is a way to tell React what you want to see

An element has no methods.

An element is immutable w/ two fields

type: String | ReactClass   ("or")

props: Object

When type == String

React Elements are plain objects.

```
{ type: 'button',          <- lower case string
  props: {
    className: 'button'
    children: {
      type: 'b'
      props: {
        children: 'ok!'     <- string
      }
    }
  }
}
```

Creates

```
<button class='button'>
  <b>
    ok!
  </b>
</button>
```

Elements are created from components.

Ex 1)

```
class Welcome extends React.Component
  render() {
    return <h1> Hello {this.props.name}
           </h1>
  }
```

2)

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

1) class based Component

2) function based Component

Both return a React.Element or other Components, arrays, strings numbers.

Component names are capitalized.

Virtual DOM ~ element of nested elements

"Fibers" internal objects holding additional information about the component tree.

Model — Controller — View

Transform React UI is a projection of data.
Abstraction Reusable pieces
Composition Build new abstractions
State Some projections have specific state
Memoization Cache for UI functions
Lists Holds item states, maps over
Continuations Handle list-of-lists, bind this
State Map Build state passing to children
Memoization Map Map Memo. is hard, low payoff
Algebraic Effects Shortcut between abstractions. A React "context"

Fundamental Concepts

link! UI Are not Pure functions of the Model

Transformation

```
NameBox(name)          <- in
  returns
    { weight: 'bold',
      label: name       } out
    }
```

Abstraction

Reusable pieces that don't leak their implementation

Composition

1)

```
FancyBox(children)
  return { borderstyle: 'style',
           children: children
         }
```

2)

```
UserBox(user)
  return
    FancyBox([
      'Name: ',
      NameBox(user.first +
              user.last)
    ])
```

State

- Immutable data model
- "thread functions through that can update state."

Memoization Not Impt!

- cache result of pure function

Lists

map over data (i.e. state)
render per item's state

Continuations ★

- difficult to boilerplate list UI
- move boilerplate out of business logic by defering execution

By currying (using bind), we pass state through from outside our core functions (free from boilerplate)

```
FancyUserList(users)
  return FancyBox
    UserList.bind(null,
                  users)

box = FancyUserList(data.users)

resolveChildren = box.children(
                    likesPerUser,
                    updateUserLikes)
resolveBox = {
  ...box,
  children: resolvedChildren
}
```

State Map

We can move the logic of extracting and passing state to a low-level function we use alot.

```
FancyBoxWithState(
1)  children
2)  stateMap
3)  updateState ) {
      return
        FancyBox(
          children.map(child =>
            child.continuation(
              stateMap.get(child.key),
              updateState )
          )
}}
```

FancyBoxWithState

```
UserList(users) return
  users.map(user => {
    continuation:
      FancyNameBox.bind(null, user),
    key: user.id
```

Note: a second mechanism for state is not necessary – UI can be considered pure function from Model → View and without need of state.

Algebraic Effects

- gives rise to a uniform representation of all computational effects.
- based off of algebraic effect handlers
- provides handlers for exceptions plus all other computational effects
- can redirect output
- wrap state modifications in transactions.
- schedule asynchronous threads.

Example: avoid intermediate ceremony imposed by monads

```
ThemeBorderColorRequest() {}

FancyBox(children)
  color = raise new ThemeBorderColorReq.
  return {
    borderWidth: '1px"
    borderColor: color
    children: children
  }

BlueTheme(children) {
  return try {
    children()
  }
  catch effect ThemeBorderColorRequest
    -> [, continuation] {
  continuation("blue")

App(data)
  return
    BlueTheme(
      FancyUserList.bind
        (null, data.users))
```

Simple Example

card component

defined by small, med, large

Render

```
<style>
  small {
    container-small
    card-small
  }

  medium {
    container-med
    card-med
  }

  large {
    container-large
    card-large
  }
</style>
```

Composable CSS Components

idea: collect component css to create single CSS with

1) media-query break points
2) color pallet classes eg. blue-400
3) Post CSS javascript for
   a) browser specific positioning
   b) "lensing" data down

Render CSS

Necessary Component Convention

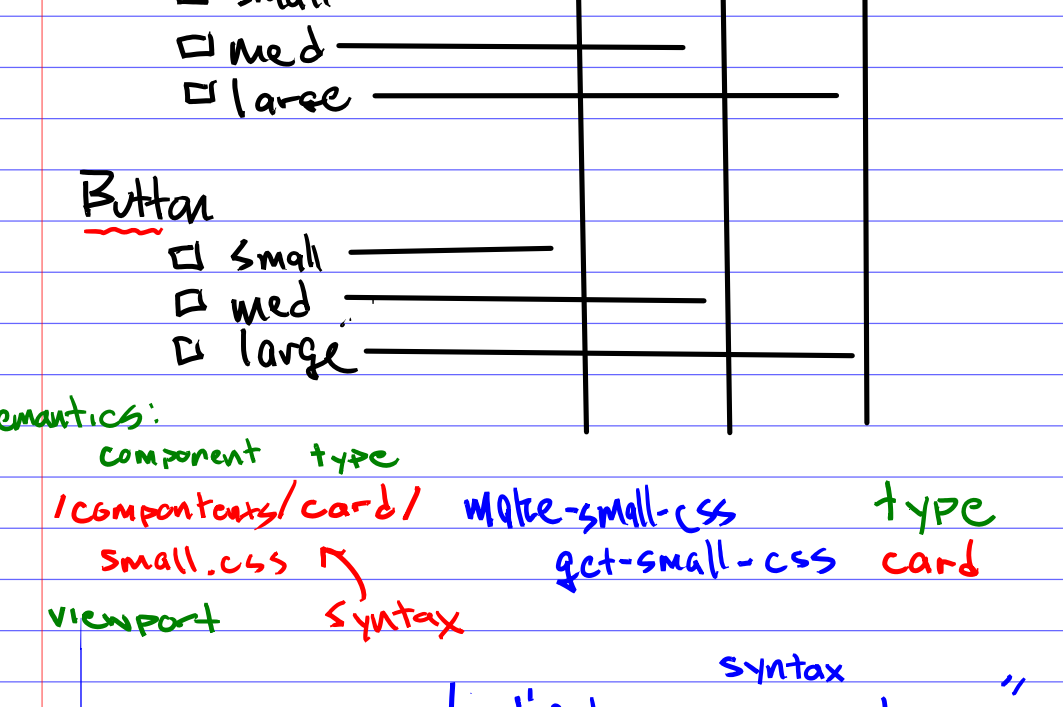
make-small-css
./components/*/make-small-css

make-medium-css
e.g. card/make-medium.css

make-large-css
e.g.
container/make-large-css

Define in many (small, med, large)

Use in many media-query-{small, med, large}



Semantics:
component type
/componteuts/card/small.css
viewport syntax

make-small-css type
get-small-css card

syntax
"action-size-type"
symantics
make-small-card
framework syntax
get-small-css card