MOBA2

MOBILE WEB:
COMPONENT DRIVEN UIs

OVERVIEW

- Component Driven UIs
- Web Components
- Other Tools and Libraries
- Introduction to React.js

OVERVIEW

- Component Driven UIs
- Web Components
- Other Tools and Libraries
- Introduction to React.js

MODERN USER INTERFACES

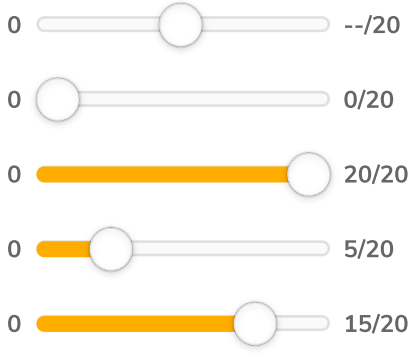
- Modern user interfaces are complicated
- People expect compelling, personalized experiences
- Should work across devices
- More logic embedded into UIs
- Large UIs are brittle, painful to debug

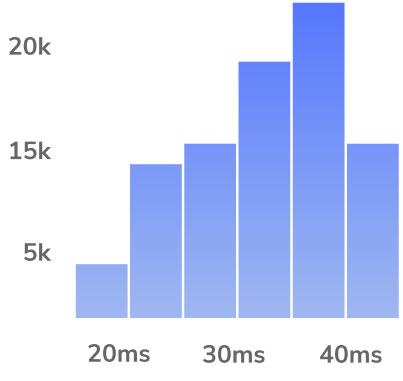
Speaker notes

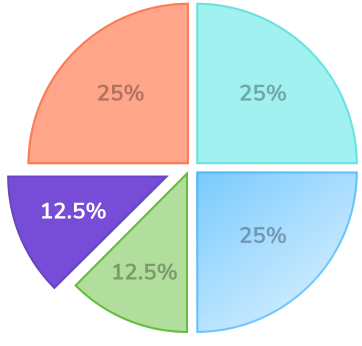
These slides are based on:

- https://www.componentdriven.org
- https://www.chromatic.com/blog/component-driven-development/
- https://www.chromatic.com/blog/ui-component-explorers---your-new-favorite-tool/
- https://storybook.js.org

COMPONENTS


0
--/20
0
0/20
0
20/20
0
5/20
0
15/20


20k
15k
5k
20ms
30ms
40ms


25%
25%
12.5%
25%
12.5%

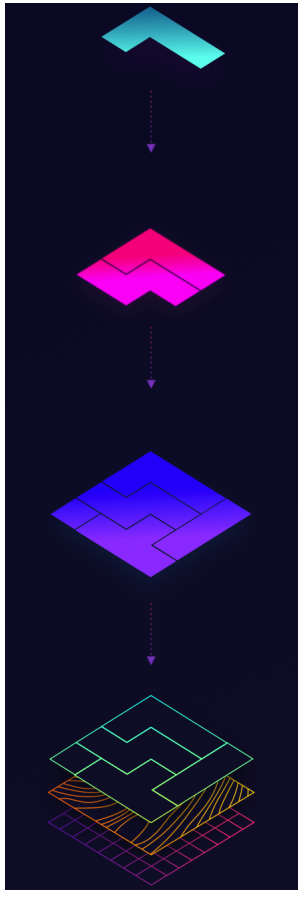
WHY COMPONENTS?

- Necessary to break UIs down in a modular way
- Components enable interchangeability
- Isolate state from application business logic
- Decompose complex screens into simple components
- Each component has a well-defined API and states
- Components can be recomposed to build different UIs

WHAT ARE COMPONENTS?

- Standardized, interchangeable building blocks of UIs
- Encapsulate the appearance and function of UI pieces

COMPONENT DRIVEN DEVELOPMENT



- Build one component at a time

  Avatar, Button, Input, Tooltip

- Combine components

  Form, Header, List, Table

- Assemble pages

  Home page, Settings page, Profile page

- Integrate pages into your project

  Web app, Marketing site, Docs site

BENEFITS

- Focus development
- Increase UI coverage
- Target feedback
- Build a component library
- Parallelize development
- Test visually

Speaker notes

https://www.chromatic.com/blog/component-driven-development/

- Focus development: Working on a single component by manipulating an entire app into a certain state is painful and laborious. Certain states can be difficult or impossible to achieve within the full app context in development (think certain loading or error states).
- Increase UI coverage: Enumerating all relevant states means you can be confident you've not missed anything and the component works in all possible scenarios.
- Target feedback: Looking it up in an explorer is a much easier way for a colleague to review a new or changed component; Focusing on one component at a time allows communication (especially between design and development) to happen with much higher precision.
- Build a component library: Supercharge component reuse within your app and organization.
- Parallelize development: Working one component at a time allows you to share tasks between different team members in a way that is just not possible at the level of "screens".
- Test visually: Component explorers allow for a class of "visual" tests, analogous to traditional automated tests, in an area (UIs) that has often defied automated testing. In particular, they allow a form of "Visual TDD" that has the same benefits as TDD, but in the UI arena.

TOOLS: COMPONENT EXPLORERS

- Showcase the components in various test “states”
- A state is essentially a visual test case
- Test a given component in all important states
- Workflow where you build one component at a time

COMPONENT STORY FORMAT (CSF)

- Open standard for component examples
- Based on JavaScript ES6 modules
- Simple to write component “stories”
- Doesn’t require vendor-specific libraries
- Declarative syntax

https://github.com/ComponentDriven/csf

STORYBOOK

- Frontend for building UI components and pages in isolation
- Suitable for UI development, testing, and documentation
- Mock hard-to-reach edge cases as stories
- Drop the finished UI components into your app
- Open source and free

https://storybook.js.org

Speaker notes

- https://github.com/storybookjs/storybook
- https://storybook.js.org/docs/react/why-storybook

STORYBOOK

Component Driven Development

COMPONENTS AND FRAMEWORKS

- Web Components
  - Stencil, Polymer, ...
- Client side UI logic and components
  - React, Vue, ...
- Presentation layer frameworks
  - Ionic, jQuery Mobile, ...
- Native Components
  - React Native, NativeScript, ...

OVERVIEW

- Component Driven UIs
- Web Components
- Other Tools and Libraries
- Introduction to React.js

WEB DEVELOPMENT

In many instances you're either copying huge chunks of HTML out of some doc and then pasting that into your app ...

A Guide to Web Components

HTML should be ...

- ... expressive enough to create complex UI widgets
- ... extensible to fill in any gaps with our own tags

This is eventually possible with Web Components

Speaker notes

In many instances you're either copying huge chunks of HTML out of some doc and then pasting that into your app (Bootstrap, Foundation, etc.), or you're sprinkling the page with jQuery plugins that have to be configured using JavaScript. It puts us in the rather unfortunate position of having to choose between bloated HTML or mysterious HTML, and often we choose __both__

A Guide to Web Components

WEB COMPONENTS

- Bundle markup and styles into custom HTML elements
- Fully encapsulate all of their HTML and CSS
- Introduced by Alex Russell at Fronteers Conference 2011

EXAMPLE: IMAGE SLIDER

```
<div id="slider">
  <input type="radio" name="slider" id="slide1" selected="false" checked>
  <input type="radio" name="slider" id="slide2" selected="false"> ...
  <div id="slides">
    <div id="overflow">
      <div class="inner">
        <img src="images//rock.jpg">
        <img src="images/grooves.jpg">...
      </div>
    </div>
  </div>
  <label for="slide1"></label>
  <label for="slide2"></label>...
</div>
```

codepen.io/robdodson/pen/rCGvJ

EXAMPLE: BETTER IMAGE SLIDER

```
<img-slider>
  <img src="images/sunset.jpg" alt="a dramatic sunset">
  <img src="images/arch.jpg" alt="a rock arch">
  <img src="images/grooves.jpg" alt="some neat grooves">
  <img src="images/rock.jpg" alt="an interesting rock">
</img-slider>
```

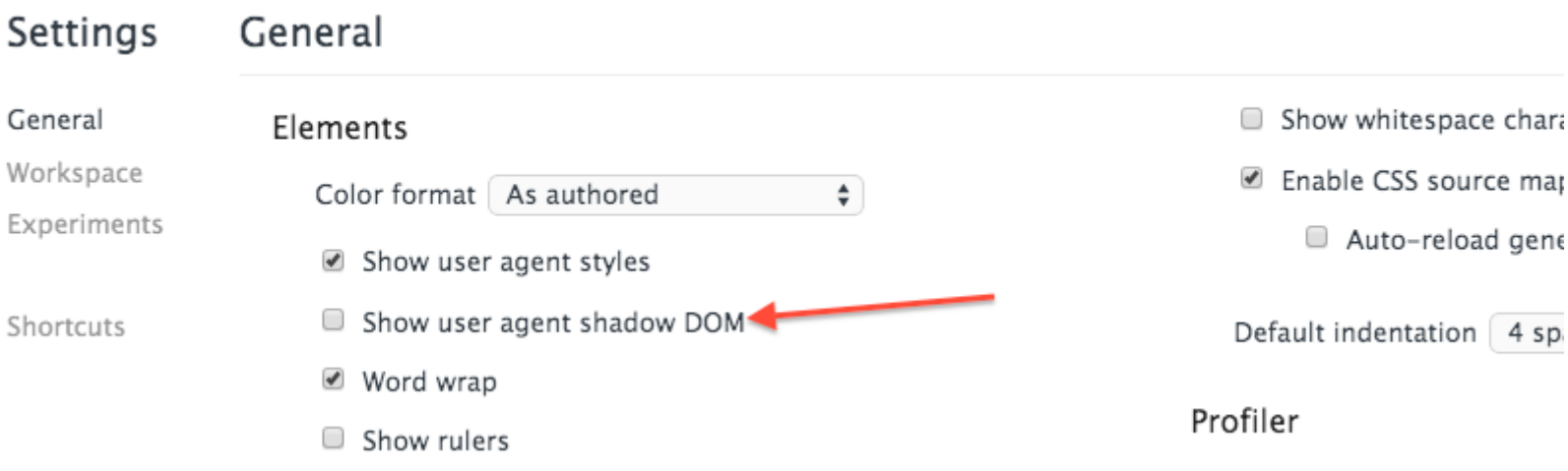
THE VIDEO ELEMENT

```
<video src="./foo.webm" controls></video>
```

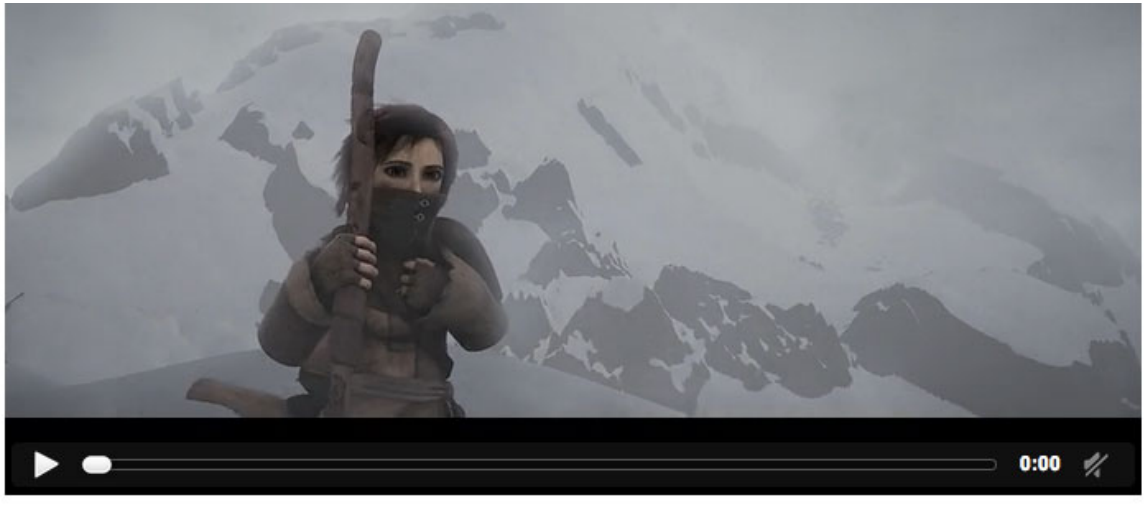
- There's a play button, a scrubber, timecodes, a volume slider
- A way to build the *video* element from these parts was needed
- Browser makers created a secret place: the *Shadow DOM*

You can activate *Show user agent shadow DOM* in the browser's DevTools

Speaker notes



THE VIDEO ELEMENT



nts | Network Sources Timeline Profiles Resources Audits Console

```
▼<video id="video" controls preload="none" poster="http://media.w3.org/2010/05/sintel/
poster.png">
  ▼#shadow-root (user-agent)
    ▼<div>
      ▼<div>
        ▼<div>
          ▶<input type="button">
          ▶<input type="range" step="any" max="0">
           <div style="display: none;">0:00</div>
           <div>0:00</div>
          ▶<input type="button">
          ▶<input type="range" step="any" max="1" style="display: none;">
          ▶<input type="button" style="display: none;">
          ▶<input type="button" style="display: none;">
```

TEMPLATES

- The `template` element
- Not rendered on the page until it is activated using JavaScript

```
<template>
  <h1>Hello there!</h1>
  <p>This content is top secret :)</p>
</template>
```

Speaker notes

Example: Image Slider

```
<template>
  <style></style>
  <div id="slider">
    <input type="radio" name="slider" id="slide1" selected="false" checked>
    <input type="radio" name="slider" id="slide2" selected="false">
    ...
  </div>
</template>
```

Put all of its HTML and CSS into a `template`

SHADOW DOM

Select an element and call its *attachShadow* method

```
<!-- HTML -->
<div class="container"></div>


// JavaScript
var host = document.querySelector('.container')
var root = host.attachShadow({mode: 'open'})
root.innerHTML = '<p>How <em>you</em> doin?</p>'
```

SHADOW HOST AND SHADOW ROOT

- Shadow Host
  - Element that *attachShadow* is called on
  - The only piece visible in the element hierarchy
  - The place where the element is supplied with content
  - Example: the *video* element is the shadow host
- Shadow Root
  - Document fragment returned by *attachShadow*
  - It and its descendants are hidden
  - But they're what the browser will actually render

Speaker notes

SHADOW BOUNDARY

- Separates DOM in the parent document from the shadow DOM
- Separates CSS in the parent document from the shadow DOM

EXAMPLE: IMAGE SLIDER

```
<template>
  <!-- Full of slider awesomeness -->
</template>

<div class="img-slider"></div>

// Add the template to the Shadow DOM
var tmpl = document.querySelector('template')
var host = document.querySelector('.img-slider')
var root = host.attachShadow({mode: 'open'})
  .appendChild(tmpl.cloneNode(true))
```

USING SLOTS

- Open problem: Image paths are hard coded in the template
- To pull items into the shadow DOM use the *slot* tag
- Projects elements from the shadow host into the shadow DOM

```
<template>
  ...
  <div class="inner">
    <slot name="imgs"></slot>
  </div>
</template>

<div class="img-slider">
  <img src="images/rock.jpg" alt="an interesting rock">
  <img src="images/grooves.jpg" alt="some neat grooves">
  <img src="images/arch.jpg" alt="a rock arch">
  <img src="images/sunset.jpg" alt="a dramatic sunset">
</div>
```

SHADOW DOM CSS

- Pseudo classes and elements for the Shadow DOM
- `:host`
  Selects the shadow host element
- `:host-context(<selector>)`
  Shadow host based on a matching parent element
- `::slotted(<compound-selector>)`
  Matches nodes that are distributed into a slot

http://robdodson.me/shadow-dom-css-cheat-sheet/

CUSTOM ELEMENT

```
class ImageSlider extends HTMLElement {
  constructor() {
    super()
    const shadowRoot = this.attachShadow({mode: 'closed'})
    shadowRoot.innerHTML = `
      <style></style>
      <div class="slider">
        ...
      </div>
    `
  }
}

customElements.define('image-slider', ImageSlider)
```

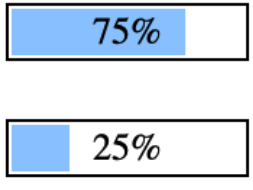
Speaker notes

Custom Element

- Name must contain a hyphen
- Prototype must extend *HTMLElement*
- New element is registered with *customElements.define*

ANOTHER EXAMPLE – DEMO

```
<custom-progress-bar class="size">
<custom-progress-bar value="25">
<script>
  document.querySelector('.size').progress = 75;
</script>
```



WEB COMPONENTS SUMMARY

Based on these pieces:

- Shadow DOM
- Custom Elements
- HTML Templates
- CSS additions

https://github.com/WICG/webcomponents
https://developer.mozilla.org/en-US/docs/Web/Web_Components

BROWSER SUPPORT

- Web Comonents were introduced in 2011
- By now, Web Components should be everywhere
- Browser support: good
  caniuse.com/#search=Web%20Components
- Reason for slow progress: vendors couldn’t agree
- Web Components were a Google effort

WEB COMPONENT LIBRARIES

- Stencil: Web Component compiler
  https://stenciljs.com
- Lit (Successor of Polymer)
  https://lit.dev
- X-Tag: Mozilla's alternative
  www.x-tags.org

OVERVIEW

- Component Driven UIs
- Web Components
- Other Tools and Libraries
- Introduction to React.js

EXAMPLE (WBE RECAP)

- In order to compare various approaches
- Create a list from an array

```
/* input: */
let data = ["Maria", "Hans", "Eva", "Peter"]

<!-- DOM structure to be created: -->
<ul>
  <li>Maria</li>
  <li>Hans</li>
  <li>Eva</li>
  <li>Peter</li>
</ul>
```

DOM SCRIPTING

```
function List (data) {
  let node = document.createElement("ul")
  for (item of data) {
    let elem = document.createElement("li")
    let elemText = document.createTextNode(item)
    elem.appendChild(elemText)
    node.appendChild(elem)
  }
  return node
}
```

- Simple abstraction: a `List` component
- Based on DOM functions

DOM SCRIPTING

```
function init () {
  let app = document.querySelector(".app")
  let data = ["Maria", "Hans", "Eva", "Peter"]
  render(List(data), app)
}

function render (tree, elem) {
  while (elem.firstChild) { elem.removeChild(elem.firstChild) }
  elem.appendChild(tree)
}
```

DOM SCRIPTING ENHANCED

```
function domElt (type, attrs, ...children) {
  let node = document.createElement(type)
  if (attrs) Object.keys(attrs).forEach(key => {
    node.setAttribute(key, attrs[key])
  })
  for (let child of children) {
    if (typeof(child) instanceof HTMLElement) node.appendChild(child)
    else node.appendChild(document.createTextNode(child))
  }
  return node
}
```

DOM SCRIPTING ENHANCED

- Abstraction enables a simpler `List` component
- DOM functions hidden in function `domElt`

```
function List (data) {
  return domElt("ul", {}, ...data.map(item => domElt("li", {}, item)))
}
```

JQUERY

```
function List (data) {
  return $("<ul>").append(...data.map(item => $("<li>").text(item)))
}

function render (tree, elem) {
  while (elem.firstChild) { elem.removeChild(elem.firstChild) }
  $(elem).append(tree)
}
```

- `List` returns a jQuery object
- Minor modification to the `render` function needed

REACT.JS

```
const List = ({data}) => (
  <ul>
    { data.map(item => (<li key={item}>{item}</li>)) }
  </ul>
)
ReactDOM.render(
  ( <List data={["Maria", "Hans", "Eva", "Peter"]} /> ),
  document.getElementById('app')
)
```

- XML syntax in JavaScript: JSX
- Needs to be translated to JavaScript
- More in a moment...

VUE.JS

```
<div id="app">
  <ol>
    <li v-for="item in items">
      {{ item.text }}
    </li>
  </ol>
</div>
```

```
var app4 = new Vue({
  el: '#app',
  data: {
    items: [
      { text: 'Learn JavaScript' },
      { text: 'Learn Vue' },
      { text: 'Build something awesome' }
    ]
  }
})
```

https://vuejs.org

Speaker notes

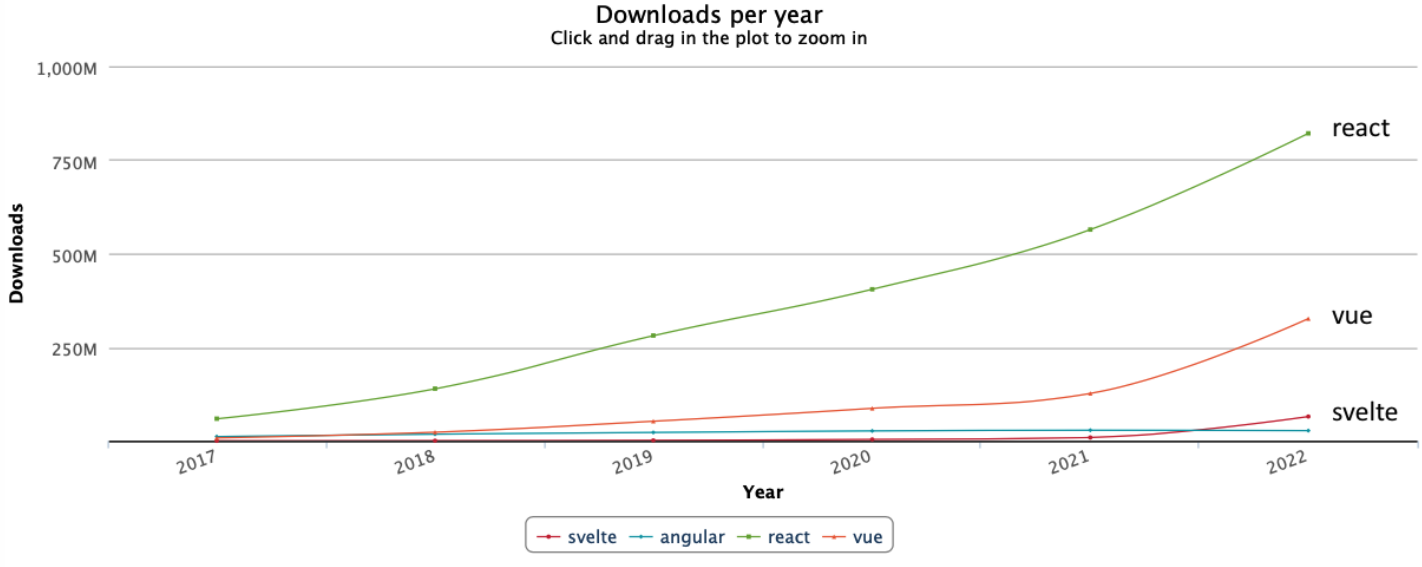
- Progressive framework for building user interfaces
- Core library is focused on the view layer only
- Capable of powering sophisticated Single-Page Applications

SVELTEJS

- Framework for building UIs, like Vue or React
- Svelte is a compiler, unlike React or Vue
- No virtual DOM, code compiled to vanilla JS
- Truly reactive framework, no complex state management libraries

https://svelte.dev

NPM STATS



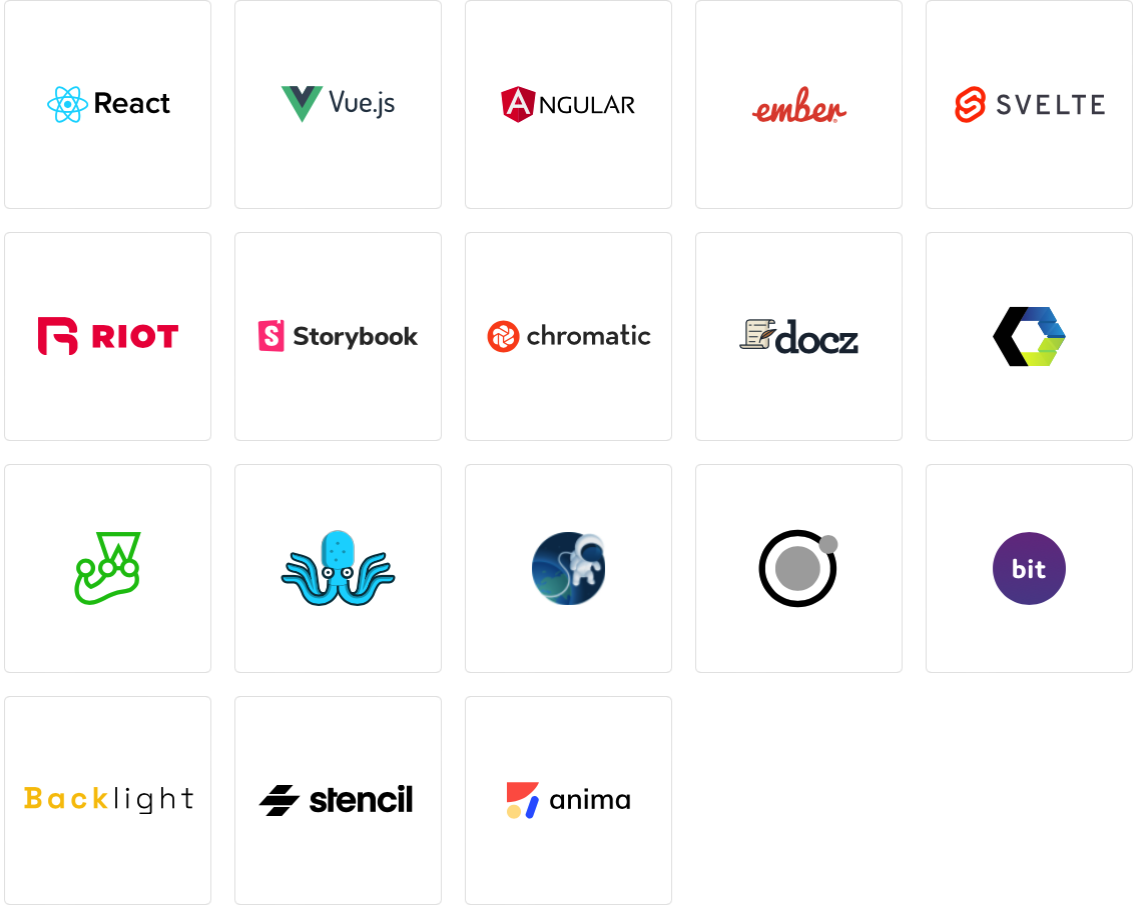
Total number of downloads between *2017-01-01* and *2022-12-31*:

package	downloads
react	2,269,072,195
vue	623,085,259
angular	133,375,554
svelte	79,180,699

Speaker notes

https://npm-stat.com/charts.html?package=react&package=vue&package=svelte&package=angular&from=2017-01-01&to=2022-12-31

COMPONENT DRIVEN DEVELOPMENT



DESIGN AND PROTOTYPING

Sketch
Figma
Xd
invision
Framer
UXPin
ZEPLIN
Abstract
zeroheight
draftbit
Interplay.
relate
penpot

OVERVIEW

- Component Driven UIs
- Web Components
- Other Tools and Libraries
- Introduction to React.js

WHAT IS REACT?

A JavaScript library for building user interfaces

- It’s not a mega framework
- It’s not a full-stack solution

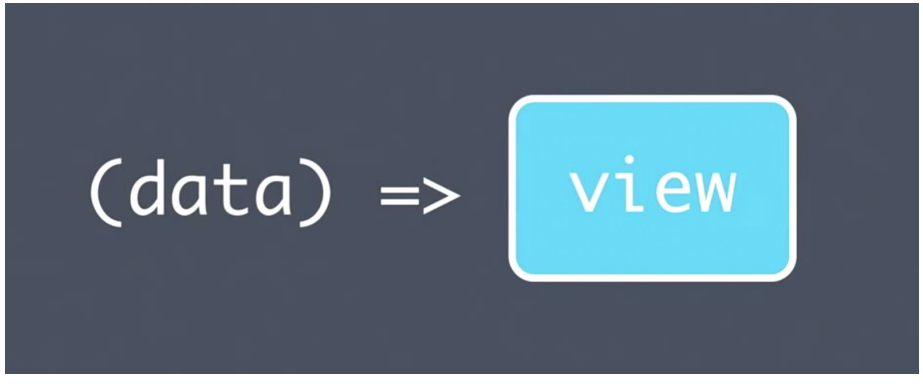
WHAT IS REACT?

A JavaScript library for building user interfaces

- Facebook, Instagram
- First introduced in 2013

https://reactjs.org

React wraps an imperative API with a declarative one



Speaker notes

IMPERATIVE STYLE

```
// Imperative Programming
var base = 20;
var solution = (base+1)*2;
dealWithUltimate(solution);

// Imperative GUI Programming
$('.config').prop('checked', false);
$('.namein').attr('placeholder','Your name');
```

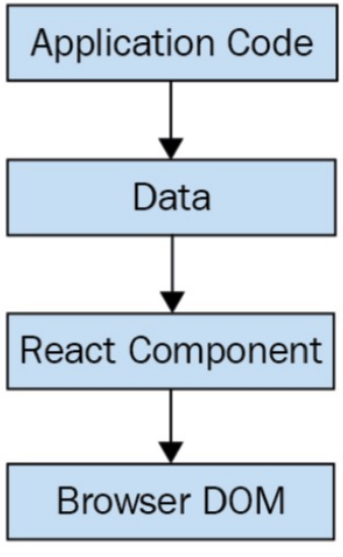
DECLARATIVE UI STRUCTURE

Example:
You want to add a class to a paragraph when a button is clicked

- Imperative Programming (jQuery, ...)
    - Perform steps
    - select paragraph
    - add class to it
- Declarative Programming (React, ...)
    - Describe what the UI should look like
    - Render page

Declarative programming is very well suited for UI development

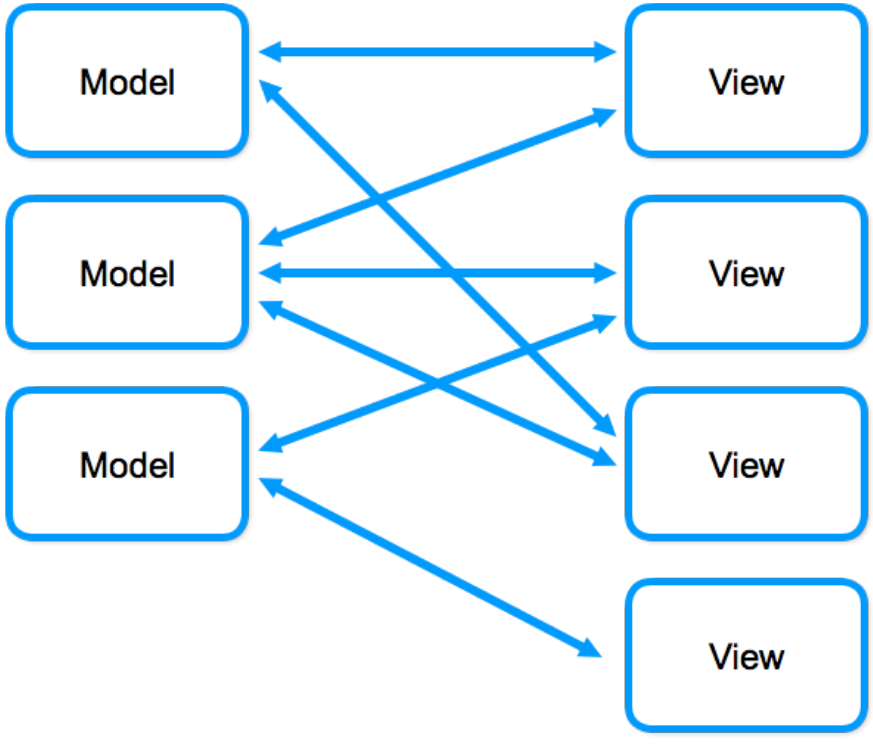
REACT IS JUST THE VIEW



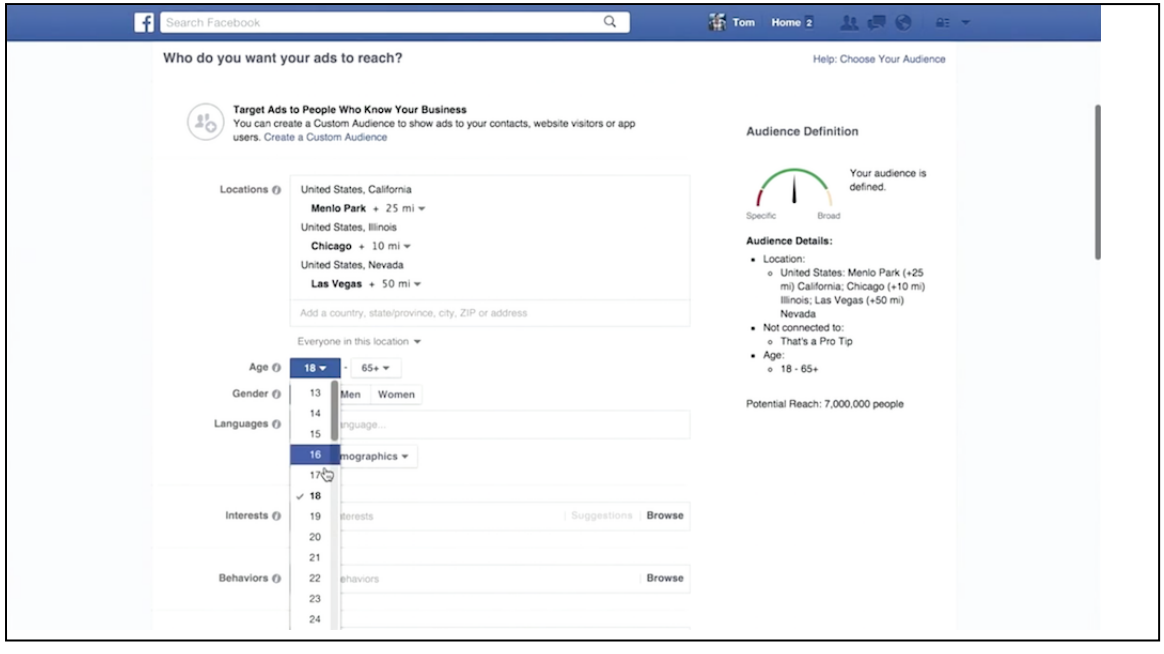
- Application logic generates some data
- React component uses the data to generate the HTML and CSS code
- Avoids two way data binding

Speaker notes

Two-way data binding is a great idea, but often it produces more pain than benefits

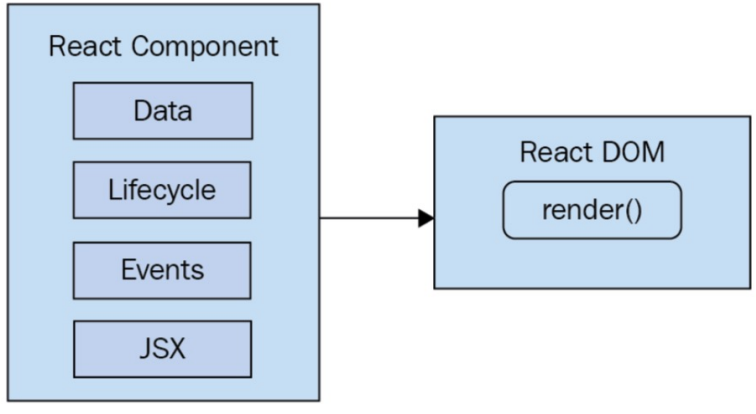


Example: Facebook Ads configuration, a complex user interface



TWO PARTS

- React DOM
  - Performs the actual rendering on a web page
- React Component API
  - Data to be rendered
  - Lifecycle support
  - Events: respond to user interactions
  - JSX: syntax used to describe UI structures



Speaker notes

Simplicity is good

- Having a small API as an advantage
- Internally, there’s a lot going on

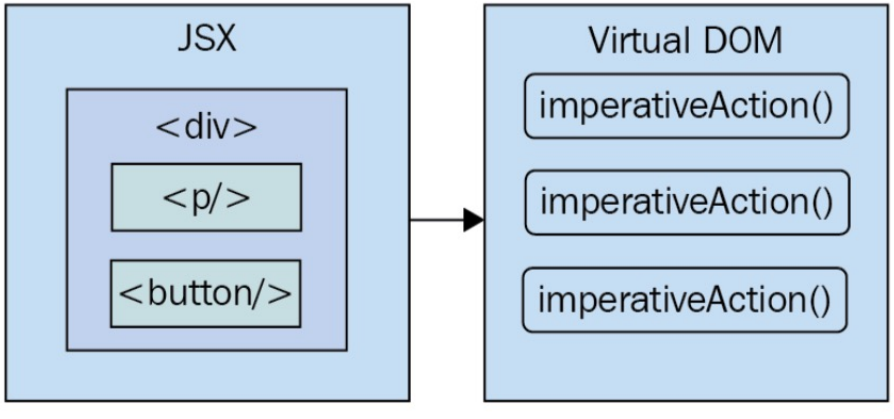
Time and data

- React components rely on data being passed into them
- Data represents the dynamic aspects of the UI
- Result: ordered collection of rendered UI components

React is simple, because it doesn’t have a lot of moving parts

PERFORMANCE MATTERS

- Challenge of the declarative approach: performance
- React uses a Virtual DOM: representation of the real DOM elements in memory
- Calculates differences on rendering and executes only the necessary DOM operations

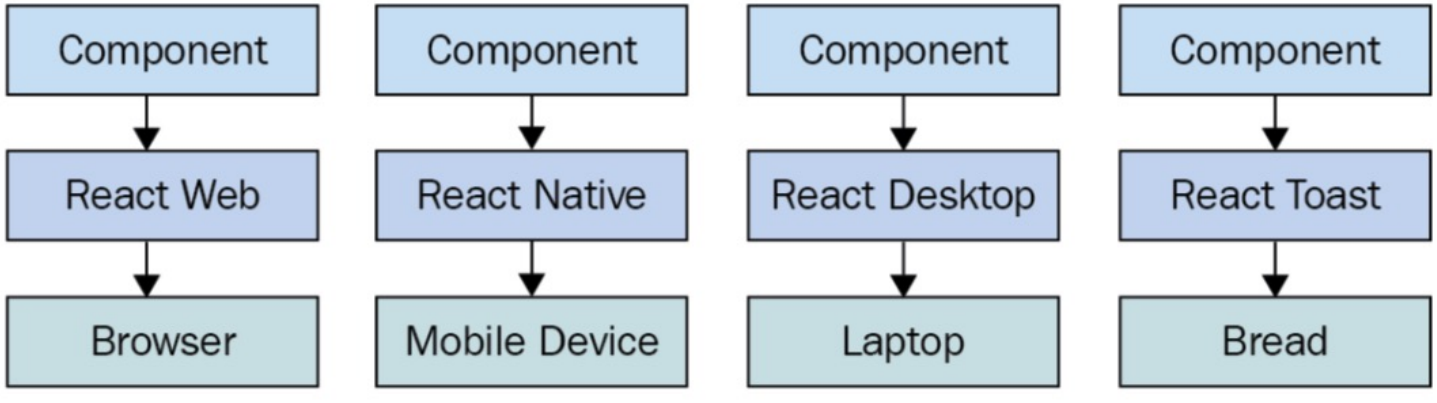


Speaker notes

- Traditional approach: declarative templates (Handlebars?) and imperative code to handle the dynamic aspects of the UI
- Declarative approach: Complete re-rendering into the DOM is bad for performance
- Virtual DOM: only the necessary DOM operations are executed
- JSX syntax translates to low-level operations that we have no interest in maintaining

THE RIGHT LEVEL OF ABSTRACTION

- We don’t necessarily care what the render target is
- React has the potential to be used for any UI



JSX

```
const Hello = () => (
  <p>Hello World</p>
)
```

- Syntax used by React components
- Component renders content by returning some JSX
- HTML markup, mixed with custom tags
- JSX = JavaScript XML (or: JavaScript Syntax Extension?)

https://facebook.github.io/jsx/

HELLO JSX

```
// The "render()" function will render JSX markup and
// place the resulting content into a DOM node. The "React"
// object isn't explicitly used here, but it's used
// by the transpiled JSX source.
import React from 'react'
import { render } from 'react-dom'

// Renders the JSX markup. Notice the XML syntax
// mixed with JavaScript? This is replaced by the
// transpiler before it reaches the browser.
render(
     (<p>Hello, <strong>JSX</strong></p>),
     document.getElementById('app')
)
```

HELLO JSX

- JSX is transpiled into JavaScript statements
- Browsers have no idea what JSX is

BABEL

```
render(

    (<p>Hello, <strong>JSX</strong></p>),
    document.getElementById('app')

);
```

```
'use strict';

render(React.createElement(
    'p',
    null,
    'Hello, ',
    React.createElement(
        'strong',
        null,
        'JSX'
    )
), document.getElementById('app'));
```

https://babeljs.io/repl/

Speaker notes

The parentheses surrounding the JSX markup aren’t strictly necessary. However, this is a React convention, and it helps us to avoid confusing markup constructs with JavaScript constructs.

BUILT-IN HTML TAGS

- React comes with HTML components
- So we can render arbitrary HTML tags

```
import React from 'react'
import { render } from 'react-dom'

render((
  <section>
    <header>
      <h1>A Header</h1>
    </header>
  </section>
  ),
  document.getElementById('app')
)
```

Speaker notes

HTML TAG CONVENTIONS

- Use lowercase for the tag name
- Tag names are case sensitive
- Non-HTML elements are capitalized
- We can pass HTML elements any of their standard properties
- Some exceptions, e.g., HTML `class` attribute is called `className` in JSX

DESCRIBING UI STRUCTURES

```
import React from 'react'
import { render } from 'react-dom'

render((
  <section>
    <header>
      <h1>A Header</h1>
    </header>
    <nav>
      <a href="item">Nav Item</a>
    </nav>
    <main>
      <p>The main content...</p>
    </main>
    <footer>
      <small>© 2016</small>
    </footer>
  </section>
  ),
  document.getElementById('app')
)
```

COMPONENTS

```
// Function components return some JSX markup. In this case,
// "MyComponent" encapsulates an HTML structure.

const MyComponent = () => (
  <section>
    <h1>My Component</h1>
    <p>Content in my component...</p>
  </section>
)

render(
  <MyComponent />,
  document.getElementById('app')
)
```

CLASS COMPONENTS

```
class MyComponent extends Component {
  render() {
    // class components have a "render()" method
    return (
      <section>
        <h1>My Component</h1>
        <p>Content in my component...</p>
      </section>
    )
  }
}
render(
  <MyComponent />,
  document.getElementById('app')
)
```

Speaker notes

- Class components *extend* `Component`
- They have a `render()` method
- This method returns the HTML that the component encapsulates
- The new component can then be rendered
- Usually, components are imported and added to the appropriate scope

NESTED ELEMENTS (1)

```
import React from 'react'
import { render } from 'react-dom'

// Imports our two components that render children...
import MySection from './MySection'
import MyButton from './MyButton'

// Renders the "MySection" element, which has a child
// component of "MyButton", which in turn has child text.
render((
  <MySection>
    <MyButton>My Button Text</MyButton>
  </MySection>
  ),
  document.getElementById('app')
)
```

NESTED ELEMENTS (2)

```
// MySection.js

// Renders a "<section>" element. The section has
// a heading element and this is followed by
// "props.children".

export default const MySection = (props) => (
  <section>
    <h2>My Section</h2>
    {props.children}
  </section>
)
```

Speaker notes

Same as a class component:

```
// MySection.js
import React, { Component } from 'react'

// Renders a "<section>" element. The section has
// a heading element and this is followed by
// "this.props.children".

export default class MySection extends Component {
  render() {
    return (
      <section>
        <h2>My Section</h2>
        <em>{this.props.children}</em>
      </section>
    )
  }
}
```

NESTED ELEMENTS (3)

```
// MyButton.js

// Renders a "<button>" element, using
// "props.children" as the text.

export default const MyButton = (props) => (
  <button>{props.children}</button>
)
```

Speaker notes

Same as a class component:

```
// MyButton.js
import React, { Component } from 'react'

// Renders a "&lt;button>" element, using
// "this.props.children" as the text.

export default class MyButton extends Component {
  render() {
    return (
      &lt;button>{this.props.children}&lt;/button>
    )
  }
}
```

NESTED ELEMENTS

- Use `{props.children}` to access nested elements or text
- In class components: `{this.props.children}`
- Braces are used for JavaScript expressions in JSX
- In the example, the button text is passed through *MySection*
- React handles the messy details

DYNAMIC PROPERTY VALUES

```
const enabled = false
const text = 'A Button'
const placeholder = 'input value...'
const size = 50

render((
  <section>
    <button disabled={!enabled}>{text}</button>
    <input placeholder={placeholder} size={size} />
  </section>
  ),
  document.getElementById('app')
)
```

Speaker notes

- JSX has special syntax to embed JS expressions: `{expression}`
- Any valid JavaScript expression can go in between the braces
- Every time JSX is rendered, expressions in the markup are evaluated
- This is the dynamic aspect of JSX content

MAPPING COLLECTIONS

```
const array = [ 'First', 'Second', 'Third' ]

render((
  <section>
    <h1>Array</h1>

    <ul>
      { array.map(i => (
        <li key={i}>{i}</li>
      )) }
    </ul>
  </section>
  ),
  document.getElementById('app')
)
```

No imperative logic needed 😄

Speaker notes

- Call to `array.map()` returns a new array
- The mapping function is returning a JSX element (`<li>`)
- The result of evaluating this expression is an array
- JSX knows how to render arrays of elements
- Object: can map over array `Object.keys()`
- Note the "key" property on `<li>`
    - This is necessary for performance reasons,
    - React will warn us if it’s missing.

OUTLOOK

- Properties and State
- React Hooks
- Developer Tools
- Event Handling
- Reusable Components

READING MATERIAL, SOURCES

DOCS AND TUTORIALS

- React: Quick Start and Docs
  https://reactjs.org/docs/hello-world.html
- Tutorial: Intro To React
  https://reactjs.org/tutorial/tutorial.html
- Babel – a JavaScript compiler
  http://babeljs.io

SOURCES

- React – A JavaScript library for building user interfaces
  https://reactjs.org
- Adam Boduch: React and React Native
  Second Edition, Packt Publishing, 2018
  Packt Online Shop