

# UI Hard Parts

---

In UI Hard Parts we develop an under-the-hood understanding of building user interfaces in the web browser.



# Will Sentance

---

CEO & Founder at **Codesmith**

Frontend Masters

## **Academic studies**

Harvard University

Oxford University

# Principles of Engineering



## Analytical problem solving

Can you break down complex challenges and develop a strategy for solving the problem



## Technical communication

Can I implement your approach just from your explanation



## Engineering approach

How you handle 'not knowing', debugging, code structure, patience and reference to documentation



## Non-technical communication

Empathetic and thoughtful communication, supportive approach to the Codesmith community



## JavaScript and programming experience

Concepts like higher order functions, closure, objects and algorithms

# UI the Hard Parts

## **State, View, JavaScript, DOM & Events**

---



Two goals to UI engineering: Display content - state/data - so the user can 'view' it then let them interact with it



The web browser's ad hoc history has led to the features that let us do this being spread across languages, APIs and environments



Requires JavaScript and the DOM to combine with help of Webcore, Web IDL, the DOM API, Event API

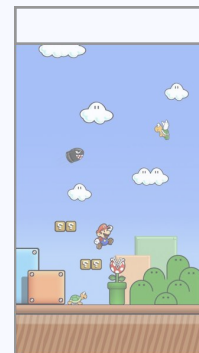
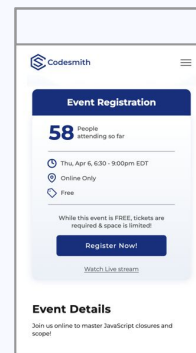
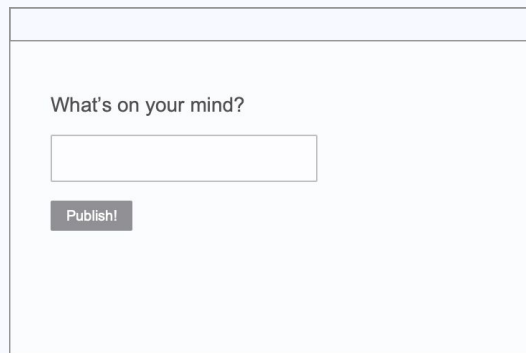


But lets us build dynamic interactive experience at the center of all modern applications

# Almost every piece of technology needs a visual 'user interface' (UI)

Content displayed on a screen (output from the computer) that a user can change by doing something

- Video stream from zoom
- Likes on tweet
- Comments in a chat
- Mario's position on the screen



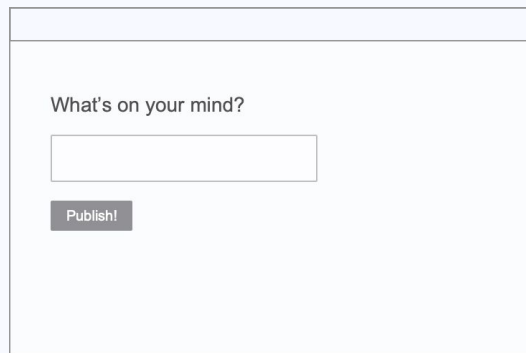
*What are other UIs and what's the interactivity?*

Just two 'simple' goals - (1) display content & (2) let user change it

**Goal 1:** Display content (data inside computer/from internet)

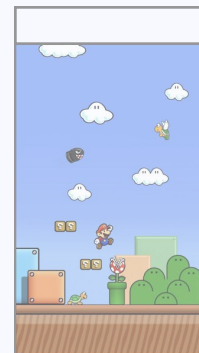
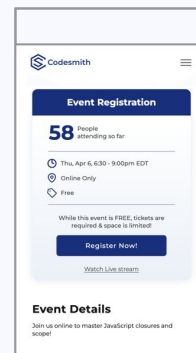
**Goal 2:** Enable the user to interact (tap, click etc) with the content they see and change it (presumably change the underlying data - otherwise it'd be inconsistent)

*Let's start with Goal 1*



What's on your mind?

Publish!



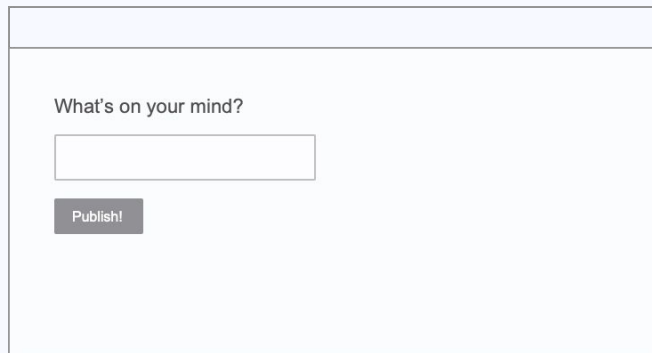
*Code to display content ▼*

1

2

3

## Goal 1 - Display the content



A screenshot of a web form. It has a light gray header bar. Below the header, the text "What's on your mind?" is displayed. Underneath the text is a white rectangular text input field. Below the input field is a dark gray button with the text "Publish!" in white.

*But we still have to think about location  
(coordinates), browser & device variations...*

app.html ▼

What's on your mind?

<input>

<button>Publish!</button>

<div>

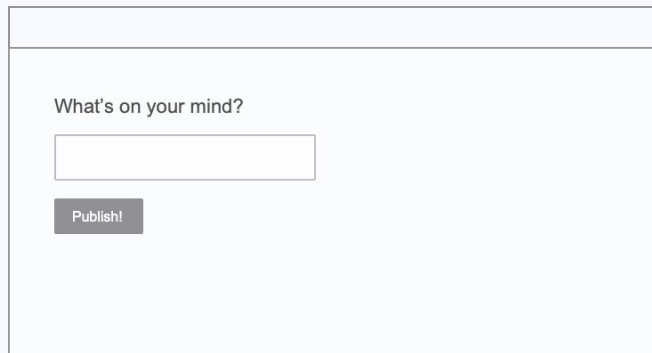
<video src="/carpool.mp4">

<p>Love Les Mis!</p>

<p>♥ 7</p>

</div>

**Solution:** A list of elements in C++ to add to the page - the DOM

A wireframe diagram of a web form. It features a header bar, a main content area with the text "What's on your mind?", a text input field, a "Publish!" button, and a footer area.

*DOM* - ordered list (object) of page (document) elements we can add to using HTML

- Order/structure set by where we write each HTML line

*Layout engine* - works out page layout for specific browser/screen

*Render engine* - produces the composite 'image' for graphics card



app.html ▼

What's on your mind?

<input>

<button>Publish!</button>

<div>

<video src="/carpool.mp4">

<p>Love Les Mis!</p>

<p>♥ 7</p>

</div>

Model of Page (DOM - C++)

- Text: What's on your mind?

- Input field

- Button: Publish!

- Division

- Video: /carpool.mp4


- Paragraph: Love Les Mis!

- Paragraph: ♥ 7

We have 'semi-visual code'  
for structure & content

What's on your mind?

Publish!



Love Les Mis!

♥ 7

Create element, Add content, Where to add, Add to  
page

*But I'm displaying visual content - how does it look  
(colors etc)?*

app.html ▼

What's on your mind?

<input>

<button>Publish!</button>

<div>

<video src="/carpool.mp4">

<p>Love Les Mis!</p>

<p>♥ 7</p>

</div>

<link rel="stylesheet" href="app.css">

app.css ▼

button {background: slateblue}

Model of Page (DOM - C++)

- Text: What's on your mind?

- Input field

- Button: Publish!

- Division

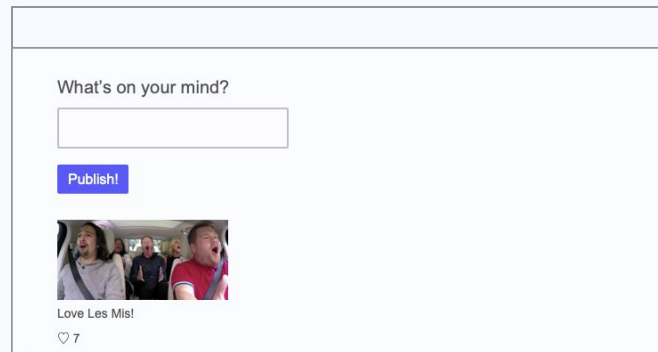
- Video: /carpool.mp4

- Paragraph: Love Les Mis!

- Paragraph: ♥ 7

- Linked file: app.css

## Adding CSSOM for styling and formatting



- CSS object model (CSSOM) mirrors the DOM

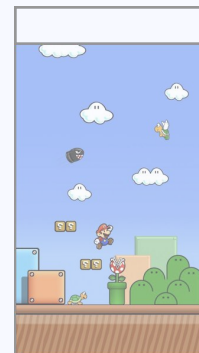
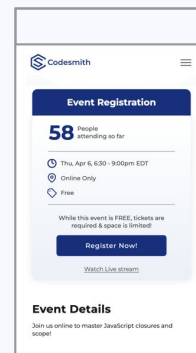
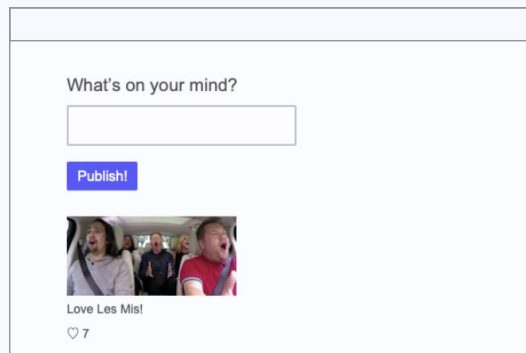
*Goal 1 (display content/data) complete!*

Just two simple goals - (1) display content & (2) let user change it

**Goal 1:** Display the content (data inside computer/from internet) ✓

- HTML + CSS

**Goal 2:** Enable the user to interact (tap, click etc) with the content they see and change it (presumably change the underlying data - otherwise it'd be inconsistent)



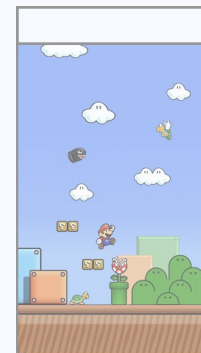
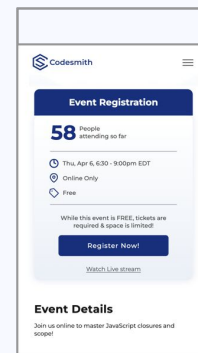
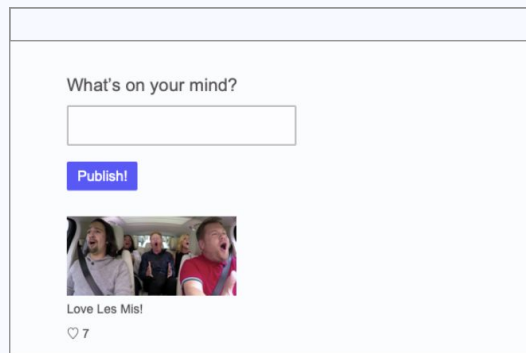
## Goal 2: Enable user to change content they see - but **problem**

Pixels  $\neq$  Data

- We're trained to think so by the real world where metaphysical and epistemic are tied
- 'Data' vs 'View'

But HTML 1x display/paint of content

User can change what they see e.g. input field



app.html ▼

What's on your mind?

<input>

<button>Publish!</button>

<div>

<video src="/carpool.mp4">

<p>Love Les Mis!</p>

<p>♥ 7</p>

</div>

Model of Page (DOM - C++)

- Text: What's on your mind?

- Input field

- Button: Publish!

- Division

- Video: /carpool.mp4


- Paragraph: Love Les Mis!

- Paragraph: ♥ 7

## Typing in the input field changes DOM 'data'

What's on your mind?

Publish!



Love Les Mis!

♥ 7

Does change data from from " " -> "Hi!"

However:

1. Doesn't display anywhere else
2. We can't run any code in DOM

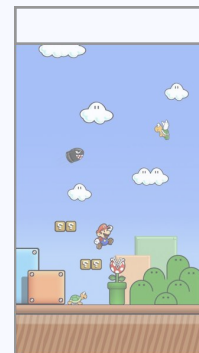
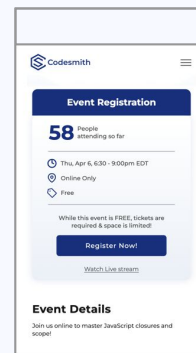
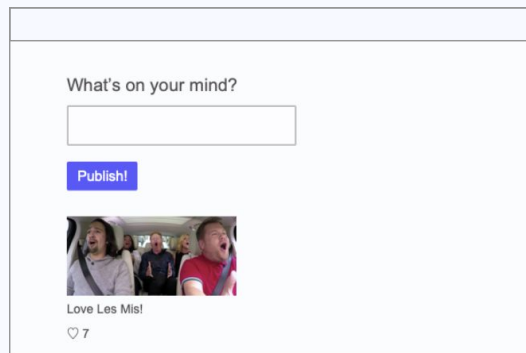
## Goal 2: Enable user to change content they see - but **problem**

Step 2 impossible w/ our approach to Step 1

- HTML 1x display/paint of content
- DOM - Display 'data' but can't run code

We have to use JavaScript to:

- Create & save content/data
- Run code to change it



*app.html* ▼

```
1 <script src="app.js"></script>
```

*app.js*

```
1 let post = "Hey 🙌"  
2 post = "Hi!"  
3 console.log(post)
```

Data (and code to change it) is only available in JavaScript

JS file of code added using html code

JavaScript is an entire runtime

- Thread of execution to run code
- Memory for data
- Call stack & event loop to allow code (functions) to run asynchronously

We have data now - but how do we **display** it?  
With WebIDL & WebCore

app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
// document - object of methods for DOM access
1 let post = "Hi!"
2 const jsDiv = document.querySelector('div')
3 jsDiv.textContent = post
4 console.log(jsDiv) // extremely misrepresentative!
```

## Webcore gives JavaScript access to the DOM & pixels

Goal 1: *Display* content/data to the user

1. Data created & stored in JS: `post = "Hi!"`
2. Get DOM element (a link in JS): `document.querySelector('div')`
3. Store link to element, in JS: `const jsDiv`
4. Use built-in 'property-methods' to add content/data from JS to DOM element: `jsDiv.textContent = post`
5. DOM automatically displays content on page (the 'view')

**Success! But SO much harder than `<div>7</div>`**

- How to technically communicate lines 2 and 3?
-



app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = ""
2
3 const jsInput = document.querySelector('input')
4 const jsDiv = document.querySelector('div')
5
6 function handleInput(){
7   post = jsInput.value
8   jsDiv.textContent = post
9 }
10
11 jsInput.oninput = handleInput
```

## Solving for Goal 2: Users can change what they see (& underlying data) by interacting

User interaction (typing, clicking etc) is 'registered' by DOM elements (they know when they're tapped)

But how can this cause a change in data which is elsewhere (in Javascript)?

**DOM events** and **handler** functions - lets user action/input ('events') change data in JavaScript

```
jsInput.oninput = handleInput
```

By saving a function definition to the DOM element - when the user action ('event') happens, the function will be 'called back' into JS and executed automatically

**So what?**

app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = ""
2
3 const jsInput = document.querySelector('input')
4 const jsDiv = document.querySelector('div')
5
6 function handleInput(){
7   post = jsInput.value
8   jsDiv.textContent = post
9 }
10
11 jsInput.oninput = handleInput
```

## We've solved for UI goals 1 & 2!

**User actions can trigger running of JS code (!) to:**

- (1) Save/change underlying data (in JS)
- (2) Change what data is displayed (in the DOM)
  - Note that we have to manually re-update the DOM with our data - there's no 'propagation'

We have a full User interface (UI)

**Goal 1:** Display content (data inside computer/from internet) as the 'view' for users to see ✓

**Goal 2:** Enable the user to interact (tap, click etc) with the 'view' they see and change it (by changing the underlying data & updating the view) ✓

# Now we can display content and let our users change it

JavaScript, the DOM, Events & Handlers give us our interactive applications

## **HTML + the DOM work together for one-time display**

Our remarkably intuitive HTML adds elements to the DOM exactly where we write them in code then the layout and render engines display the content

## **Apps let users interact (change what they see)**

But this requires changing underlying data - which is only available in JavaScript - we get to add that data to the DOM (and do many other things) with WebCore & WebIDL

## **Events and handlers let users run JavaScript**

Which can then get info on the user's action and (1) make change to underlying data then (2) redisplay the new data - so the user sees the consequence

## **From HTML to JavaScript - descriptive to imperative**

With HTML we could state (describe) our displayed content in intuitive ordered code but without data it was not interactive. Adding data added interactivity but also significant complexity. The rest of UIHP (and much of UI engineering) aims to reduce that complexity

## UI the Hard Parts

# One-way Data Binding

---



Popular paradigm for tackling the essential UI engineering challenge - data/view consistency



Invaluable at scale (state/view with 1000s of points of interaction) & implemented in React, Angular & Vue



Enables us to build scalable apps with or without frameworks, debug complex UIs & answer the toughest interview Qs

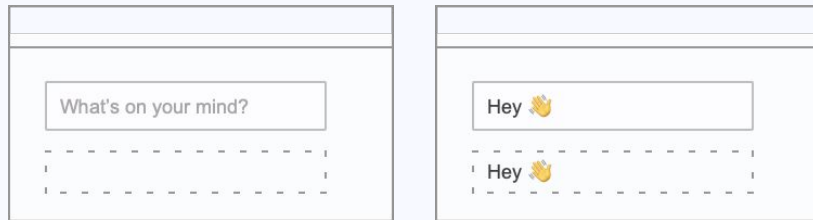
app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = ""
2 const jsInput = document.querySelector('input')
3 const jsDiv = document.querySelector('div')
4 jsInput.value = "What's on your mind?" // affect view!
5
6 function handleInput(){
7   post = jsInput.value
8   jsDiv.textContent = post // affect view!
9 }
10 function handleClick(){
11   jsInput.value = "" // affect view!
12 }
13 jsInput.oninput = handleInput
14 jsInput.onclick = handleClick
```

Let's expand our UI to be more sophisticated



Adding another handler - this one for clicks

We're changing our 'view' based on multiple possible user interactions. In practice there will be 1000s that can all affect data/view

**We need a way to make these changes as predictable as we possibly can - how?**

app.html ▼

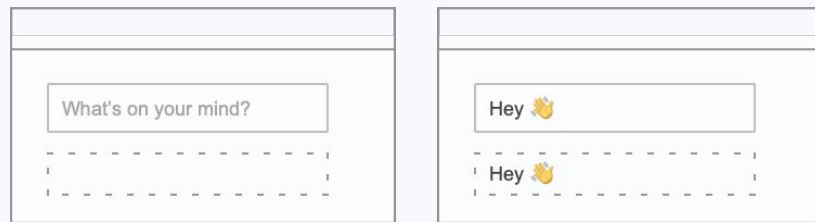
```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = undefined // Data
2 const jsInput = document.querySelector('input')
3 const jsDiv = document.querySelector('div')
4
5 function dataToView(){ // affect view!
6   jsInput.value =
7     post == undefined ? "What's on your mind?" : post
8   jsDiv.textContent = post
9 }
10
11 function handleClick(){
12   post = "" // Update data
13   dataToView() // Convert data to view
14 }
15
16 function handleInput(){
17   post = jsInput.value // Update data
18   dataToView() // Convert data to view
19 }
20
21 jsInput.onclick = handleClick
22 jsInput.oninput = handleInput
23
24 dataToView()
```

Restrict every change to view to be via

- (1) An update of 'data' and
- (2) A run of a single *dataToView* convertor function



- This is just one approach - but an immensely popular one (React, Next, Vue, Angular, Svelte)
- Tees us up for semi-visual coding with a virtual DOM

app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = undefined // Data
2 const jsInput = document.querySelector('input')
3 const jsDiv = document.querySelector('div')
4
5 function dataToView(){ // affect view!
6   jsInput.value =
7     post == undefined ? "What's on your mind?" : post
8   jsDiv.textContent = post
9 }
10 function handleInput(){
11   post = jsInput.value
12 }
13 function handleClick(){
14   post = ""
15 }
16
17 jsInput.oninput = handleInput
18 jsInput.onclick = handleClick
19
20 setInterval(dataToView, 15)
```

Added benefit that we can declare our data/view relationship once and when the data updates the view will update!



Remember our goals of UI: Display content (underlying data) as 'view' and enable the user to change it (both the underlying data and corresponding view of it)

- We need to run the dataToView convertor on repeat but we're able to automate the progress

# We've added predictability to our Data and View flow

UI application complexity grows exponentially with # of elements - one-way data binding can ensure predictability and ease of reasoning

## **1000s of ways for users to interact in a modern app**

Each piece of 'view' (element on page) can be clicked/typed/moused over - all of which need to change data and view

## **We need a predictable structure**

Better to be restricted but predictable than overly flexible and it be impossible to identify which line of code caused our view/data to change

## **Every interaction must change JS data and then update all dependent views**

Changes to views via (1) An update of 'data' and (2) A run of a single dataToView convertor

## **One-way data binding**

Restricting all our user's changes to flow to associated underlying data and all changes of view to flow through a single dataToView convertor is a restrictive but powerful step to simplify our reasoning about UI by order of magnitude



# What's tomorrow?



## **More flexible Virtual DOM & UI Composition**

JavaScript is not at all visual or declarative - we'll make it more so



## **Functional components & properties**

We'll get visual/declarative development but with added flexibility & reusability



## **State hooks, Diffing & DOM reconciliation**

We need to solve the performance crisis we create by solving the first 3 hard parts

## UI the Hard Parts

# Virtual DOM

---



Most misunderstood concept in UI development



Enables us to have a more visual (or declarative) experience of coding UIs with JavaScript



Requires significant optimizations (diffing, reconciliation) to be performative

app.html ▼

```
1 <input/>
2 <div></div>
3 <script src="app.js"></script>
```

app.js

```
1 let post = "" // Data
2 const jsInput = document.querySelector('input')
3 const jsDiv = document.querySelector('div')
4
5 function dataToView(){ // affect view!
6   jsInput.value = post
7   jsDiv.textContent = post
8 }
9
10 function handleInput(){
11   post = jsInput.value
12   if (post == "Will"){ jsDiv.remove() } // affect view!
13 }
14
15 jsInput.oninput = handleInput
16
17 setInterval(dataToView, 15)
```

Returning to our full UI with auto updating views (from data)



- We're describing a key part of the UI in *dataToView*:
  - The contents (data) and how to display it
- But even our 'containers' are 'data' of sorts
- What if we also described the element as well then our *dataToView* convertor is a complete description of the data + view?

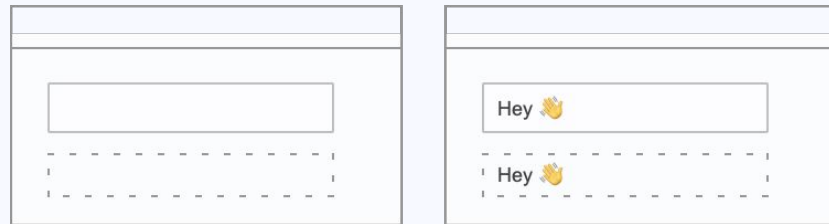
app.html ▼

```
1 <script src="app.js"></script>
```

app.js

```
1 let post = ""; let jsInput; let jsDiv // Data
2
3 function dataToView(){ // affect view!
4   jsInput = document.createElement('input')
5   jsDiv = post == "Will" ? "" : document.createElement('div')
6
7   jsInput.value = post
8   jsDiv.textContent = post
9   jsInput.oninput = handleInput
10
11   document.body.replaceChildren(jsInput, jsDiv)
12 }
13
14 function handleInput(){
15   post = jsInput.value
16 }
17 setInterval(dataToView, 15)
```

Function that fully creates element & relates data/view is known as a **UI component**



- Combines everything into 1 function - Creates elements, sets their data/contents, attaches handlers & displays via DOM
- Ties data to view - handles how user can change data (here, via input) then re-runs to update view with new data
- React, Angular, Vue, Svelte all have these

## String interpolation gives us 'visual' code

- Known as Template literals
- The closer our code can be to 'visual' (mirroring what its visual/graphic output will look) the easier for us as developers
- Could we do the something similar with our main code creating visual elements?

*mypage.html* ▶

*mypage.js* ▼

```
1 let name = "Jo"
2
3 let textToDisplay = "Hello, "
4 textToDisplay = textToDisplay.concat(name)
5 textToDisplay = textToDisplay.concat("!") // "Hello, Jo!"
6
7 // alternative
8
9 textToDisplay = `Hello, ${name}!` // "Hello, Jo!"
```

## Could we emulate HTML with semi-visual coding?

Starting with a 'unit of code' representing each piece of 'view'

- JavaScript array with all the details
- Element 0 the type, element 1 the contents

And a function that takes in that list and produces a DOM element as output

- A 'mirror' in our code the actual DOM element

*Puts all our element details in an array and our 'create element, add contents' instructions into a convert function*

*mypage.html ▶*

*mypage.js ▼*

```

1  let name = "Jo"
2
3  const divInfo = ['div', `Hi, ${name}!`]
4
5  function convert(node){
6      const elem = document.createElement(node[0])
7      elem.textContent = node[1]
8      return elem
9  }
10
11  const jsDiv = convert(divInfo)
12
13
```

# Creating a JavaScript ('virtual') DOM

1. Blocks of code representing each piece of 'view'

- JavaScript array with all the details
- Function that takes in that array and produces a DOM element as output
- `convert`

2. Then *where* we place them in the code mirrors the order they show up on the page

`mypage.html` ▶

`mypage.js` ▼

```

1  let name = ""; let jsInput; let jsDiv; let vDOM
2
3  function createVDOM (){
4      return [['input', name, function handle (){name = jsInput.value}],
5              ['div', `Hello, ${name}!`]]
6  }
7  function updateDOM() {
8      vDOM = createVDOM()
9      jsInput = convert(vDOM[0]) // jsInput
10     jsDiv = convert(vDOM[1]) // outputDiv
11     document.body.replaceChildren(jsInput, jsDiv) // DOM
12 }
13 function convert(node){
14     const element = document.createElement(node[0])
15     element.textContent = node[1]
16     element.value = node[1]
17     element.oninput = node[2]
18     return element
19 }
20 setInterval(updateDOM, 15)

```

# Declarative UI as a programming paradigm

Goal: See a JS nested list representation<sup>1</sup> of your actual DOM elements (and contents) so you can get semi-visual coding 😊

- Via nested objects or arrays of info (as we default to in UIHP) - or functional components in React

For it to be guaranteed accurate (& not out of sync) - you need your data in JS to be the only source of truth

- You can only do a JS 'DOM' if you've set on 'one-way data binding'<sup>2</sup> - otherwise it's not guaranteed to be reflective of the actual DOM (which has been altered by the user!)

So we literally take any change to the DOM (e.g. user action like typing letters in an input field) and immediately replace it by

- (i) sending the info to 'data' store in JS
- (ii) have that update contents of the JS DOM<sup>3</sup>
- (iii) convert that to the DOM (view)

We have semi-visual coding - our JS DOM contents and positioning on the JS page is reflected in our actual view

- We 'declare' (a) what page structure we want and (b) how we want its contents to depend on our data<sup>4</sup>
- And then it appears on the page (even an empty form field is the propagation of data - our 'empty string')

1. This is the 'virtual DOM'. You don't have one by default - only objects representing access to the respective DOM elements

2. Every bit of view at all times depends on underlying JS state - not whatever a user has started typing in

3. [I need to update the schema to include this 'JS DOM' step]

4. Note there's two parts to the 'declaration': structure/positioning & contents



## UI the Hard Parts

# **Composition & Functional components**

---



Our JavaScript code mirrors our output with our JS DOM - we can make it flexible by mapping over it



We can use reusable functions to create JS (V) DOM elements each with specific data



We get UI composition with units of UI (data, view and handlers) that we can reuse and combine flexibly to build out our applications

# Lists are central to UI development

We have a list of 2 arrays with their element details - but in reality it will likely be many (49 video elements in a zoom meeting, 100 tweets in a timeline)

We need tools for dealing with lists

- To apply code (functionality) to each element - here to take the info and connect to a DOM element `vDOM.map()`

To convert an array elements to individual inputs (arguments) to a function (for functions that don't take in arrays) spread syntax `...`

*mypage.html* ▶

*mypage.js* ▼

```
1 const vDOM = [  
2   ['input', name, function handle () {name = jsInput.value}],  
3   ['div', `Hello, ${name}!`]   
4 ]
```

## 'Mapping' over our vDOM

- Now adding new 'elements' doesn't require any new code
- The map call will handle as many elements in the vDOM as there are

*mypage.html* ▶

*mypage.js* ▼

```
1 let name = ''
2 const vDOM = [
3   ['input', name, function handle () {name = jsInput.value}],
4   ['div', `Hello, ${name}!`]
5 ]
6
7 function convert(node){
8   const element = document.createElement(node[0])
9   element.textContent = node[1]
10  element.value = node[1]
11  element.oninput = node[2]
12  return element
13 }
14
15 const elems = vDOM.map(convert)
```

## Spread syntax for arrays & objects

To add our array of DOM elements we need to convert them to individual arguments

- Spread syntax allows an iterable such as an array expression or string to be expanded
- To work with the `append` method (which can't handle arrays)

*mypage.html* ▶

*mypage.js* ▼

```

1  const vDOM = [
2    ['input', name, handle],
3    ['div', `Hello, ${name}!`]
4  ]
5
6  function handle (){name = jsInput.value}
7
8  function convert(node){
9    const element = document.createElement(node[0])
10   element.textContent = node[1]
11   element.value = node[1]
12   element.oninput = node[2]
13   return element
14 }
15
16 const elems = vDOM.map(convert)
17 // convert(vDOM[0]) - push to new elems array
18 // convert(vDOM[1]) - push to new elems array
19 document.body.replaceChildren(...elems);
20 // spread out elems array into individual arguments

```

With our new  
element-flexible code we  
can 'compose' our code

- Adding to our list of arrays each new element we want to see in order on our page
- They'll show up on the page exactly as they're placed in code

We're getting semi-visual coding! 🙌

mypage.html ▶

JS + Webcore

mypage.js ▼

```
1 let name = ''; let vDOM; let elems
2
3 function createVDOM () {
4   return ["input", name, function handle (e) { name = e.target.value; },
5         ["div", `Hello, ${name}!`,
6         ["div", "Great job"]]]
7 }
8 function updateDOM() {
9   vDOM = createVDOM()
10  elems = vDOM.map(convert)
11  document.body.replaceChildren(...elems)
12 }
13 function convert(node) {
14   const element = document.createElement(node[0])
15   element.textContent = node[1]
16   element.value = node[1]
17   element.oninput = node[2]
18   return element
19 }
20 setInterval(updateDOM, 15)
```



Even more composition -  
creating multiple (similar)  
VDOM elements each with  
different data

- We can produce our VDOM elements with a function so that we can easily add new elements
- Post defines data -> view relationship where the data (each element in the array posts) changes for each of the posts
- Can also add more logic to the Post/Input functions to make them more powerful

mypage.html ▶

JS + Webcore

mypage.js ▼

```
1 let posts = ["Gez", "Gerry", "Fen", "Ursy"]; let vDOM; let elems;
2
3 function Post(msg){ return ["div", msg] }
4
5 function createVDOM (){
6   return ["input", name, handle,
7     ...posts.map(Post)]
8 }
9 function handle (e){posts.push(e.target.value)}
10
11 function updateDOM() {
12   vDOM = createVDOM()
13   elems = vDOM.map(convert);
14   document.body.replaceChildren(...elems)
15 }
16 function convert(node){
17   const element = document.createElement(node[0])
18   element.textContent = node[1]; element.value = node[1]
19   element.oninput = node[2]
20   return element
21 }
22
23 setInterval(updateDOM, 15)
```

# All great and awesome but we need efficiency now

We love the vDOM for semi-visual coding - however performance is a nightmare - we need some improvements

## Hooks

We're running this updateDOM function every 15 ms - CSS animations and smooth scrolling are at risk (and user action handlers get blocked even!).

We could only run on data change - but now our VDOM is not 'real' - it's only rendered 'as is' if we've remembered to run the updateDOM function on every data change (including from servers etc) - which is unlikely - so introduce a 'state hook'

## Diffing

Only some of our DOM elements need recreating from scratch - we can compare archived VDOMs and workout what changes to actually make using an algorithm

## Automatic update on data change without looping

- Instead of directly updating name, we run a function 'updateName' to do so
- And of course it's low key running updateDOM for us
- As long as we restrict our team from ever changing data directly and only let them do so using the update function then we're fine
- We'd likely lock down name so that it can't be accessed directly

mypage.html ▶

JS + Webcore

mypage.js ▼

```
1 let name = ""; let vDOM; let elems
2
3 function updateName(value){
4   name = value
5   updateDOM()
6 }
7 function createVDOM (){
8   return ["input", name, function handle (e){updateName(e.target.value)}],
9         ["div", `Hello, ${name}!`],
10        ["div", "Great job!"]]
11 }
12 function updateDOM() {
13   vDOM = createVDOM()
14   elems = vDOM.map(convert);
15   document.body.replaceChildren(...elems);
16 }
17 function convert(node){
18   const element = document.createElement(node[0])
19   element.textContent = node[1]; element.value = node[1]
20   element.oninput = node[2]
21   return element
22 }
23 updateDOM()
```



Turning it into something we can use with any data

- We can make our **updatename** function an `updateData` function and have it work for any piece of data in our app - we just hook into it
- So they call it a hook...
- We could switch to running `requestAnimationFrame` rather than `updateDOM` directly on data change - so it never prioritizes over animations (CSS etc) - truly optimized wrt other priorities now

`mypage.html` ▶

JS + Webcore

`mypage.js` ▼

```
1  const data = {name: ''}; let vDOM; let elems
2
3  function updateData(label, value){
4    data[label] = value
5    updateDOM()
6  }
7
8  function createVDOM (){
9    return ["input", data.name, handle],
10           ["div", `Hello, ${data.name}!`,
11            ["div", "Great job!"]]
12 }
13
14 function handle (e){updateData('name', e.target.value)}
15
16 function updateDOM() {
17   vDOM = createVDOM()
18   elems = vDOM.map(convert);
19   document.body.replaceChildren(...elems);
20 }
21
22 function convert(node){
23   const element = document.createElement(node[0])
24   element.textContent = node[1]; element.value = node[1]
25   element.oninput = node[2]
26   return element
27 }
28
29 updateDOM()
```

All the benefit of composition - but it's still dangerously inefficient

- Only some of our elements need recreating from scratch
- Couldn't we recreate our vDOM from scratch to give us 'composition' **but** then:
- Write an 'algorithm' (smart instructions) to check what elements actually **differ** - and only change the DOM elements that need updating

mypage.html ▶

JS + Webcore

mypage.js ▼

```
1 let name = ''
2
3 function createVDOM () {
4   return [
5     ["input", name, function handle (e) { name = e.target.value; }],
6     ["div", `Hello, ${name}!`],
7     ["div", "Great job!"]
8   ]
9 }
10
11 function findDiff(prevVDOM, currentVDOM) {
12   for (let i = 0; i < currentVDOM.length; i++) {
13     if(JSON.stringify(prevVDOM[i]) !== JSON.stringify(currentVDOM[i])) {
14       // change the actual DOM element related to that vDOM element!
15     }
16   }
17 }
18
19 const vDOM1 = createVDOM()
20 name = 'Will'
21 const vDOM2 = createVDOM()
22
23 findDiff(vDOM1, vDOM2)
```



## Finally - composable UI that's not a disaster!

- Integrating this 'diffing algorithm' makes our code for building user interfaces 'semi-visual' (enabling composition)
- And yet not untenably inefficient

mypage.html ▶

mypage.js ▼

```
1  let name = ''; let vDOM = createVDOM(); let prevVDOM; let elems
2  function createVDOM () { return [
3      ["input", name, handle],
4      ["div", `Hello, ${name}!`],
5      ["div", "Great job!"]
6  ]
7  function handle (e) { name = e.target.value
8
9  function updateVDOM () {
10     if (elems == undefined) {
11         elems = vDOM.map(convert)
12         document.body.append(...elems)
13     } else {
14         prevVDOM = [...vDOM]
15         vDOM = createVDOM()
16         findDiff(prevVDOM, vDOM)
17     }
18 }
19 function convert(node) {
20     const element = document.createElement(node[0]);
21     element.textContent = node[1]
22     element.value = node[1]
23     element.oninput = node[2]
24     return element
25 }
26 function findDiff(prevVDOM, currentVDOM) {
27     for (let i = 0; i < currentVDOM.length; i++) {
28         if (JSON.stringify(prevVDOM[i]) !== JSON.stringify(currentVDOM[i])) {
29             elems[i].textContent = currentVDOM[i][1]
30             elems[i].value = currentVDOM[i][1]
31             elems[i].oninput = currentVDOM[i][2]
```

# We have a groundbreaking approach to UI

## **Single source of truth for our data**

Everything the user sees stems from the data. It's restrictive but desperately predictable - and enables data propagation to the view

## **Enables UI composition with JavaScript**

A JavaScript DOM that enables a semi visual form of coding our UIs - where units of code that describe units of view can be positioned in the code to indicate their order on the webpage

## **Makes reasoning about state/view much easier**

The downside of semi-visual code is it takes the data and info on the view at this moment and creates it in full (that's why it's so easy to reason about - we don't have to figure out the change from the previous data/view - just current data generating current view).

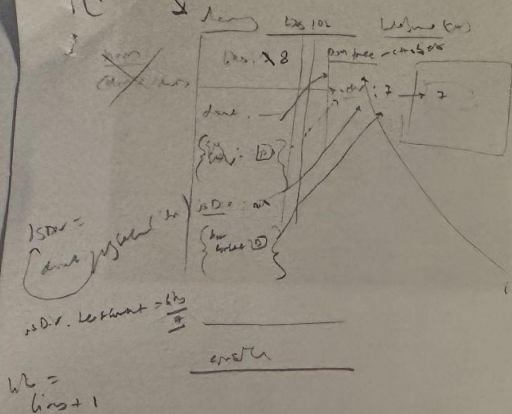
## **But requires techniques like hooks & VDOM diffing for efficiency**

If we rebuilt the DOM itself every time any data changed that would be entirely unnecessary. Instead we can spot the actual differences and only change those (DOM reconciliation)

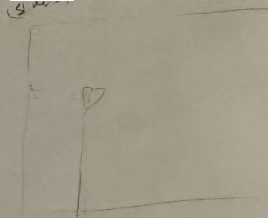
# UIHP ARCHIVE

---

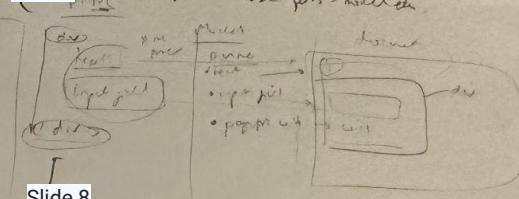
Slide 12/13



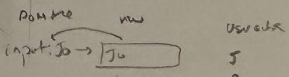
Slide 5



Slide 7

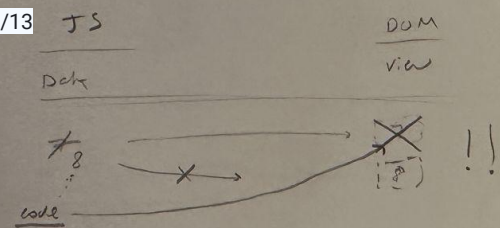


Slide 8

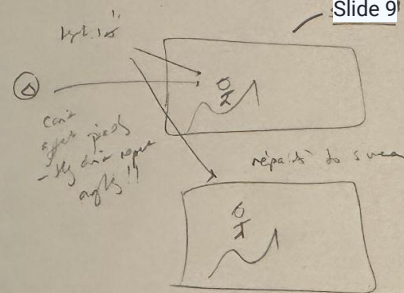


Slide 13

Slide 12/13



Slide 9



March 31 2023

Slide 14

istoput = donnerb. ges. Salzw. (impur)

isopropyl. gungster (200)

13. isomorph. type =  $\sqrt{2}$

Cell 5 W

graphical

update D.h. ( )

$n_{\text{rel}} = \frac{\text{jump value}}{\text{...}}$

$$\text{isopleth. } \frac{dC}{dx} = \frac{m}{\sigma_0}$$

Menu


dearest

{ ganz seltsam. 10 }

name: "Id"

sample : { sample : 10 }  
val : 10

Age: { culture: 14

pdale Solo 

 Slide 14

Slide 15 -> 16

deho esche

(c)

(slide 15)

Let's think about  
this really  
(visual)  
So only for a  
non-visual sound

(b)

" " - cole

150

To

Mar 31 2023

Slide 15

(75)

Not we

✓

(over)

معاذ

(D) Reparat

Here the previous view is one of two (I)ndependent sources of truth (ie we have to reason about this part of the view and how to change it if necessary)

Delta

we

✓ c

code

1

Great Hall

Here the previous view (user typing 'Jo') updates our data but then our data regenerates our entire view - our prev is of no relevance



# Slide 26

DOM  
set.html (updateDOM, 15)

JSms updateDOM()

```

root = orderDOM()
data = vDOM.map(orders)

const ([x, [p, 'will', [id]], [p, 'will', [id]], [p, 'will', [id]]])

document.createElement('div')
{
  id: [id]
}

if (true) {
  [p, 'will', [id]] * 3
  .map(orders)
  const ([p, 'will', [id]])
  {
    id: [id]
  }
}
x 3
  
```

could  
define be  
holder

const . append(... childNode)

holder()

name = 'ging'

documents;

```

{
  body: [id]
  footer: [id]
}
  
```

name: "ging"

vDOM

```

[[id], [
  [p, 'will', [id]],
  [p, 'will', [id]],
  [p, 'will', [id]]
]]
  
```

data:

```

x1 {
}
  
```

createVDOM: [id]

updateDOM: [id]

current: [id]

WebSite

WebCore

Var Archis

DOM tree

body

div

p

p

p

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

will

JS

Console

Network

Timer

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML

DOM

HTML



## Part 1 - Event delegation

Now what if we wanted more control over placement - nested elements

- We can use 'sub-arrays' to visually show on page of code that we want these 'p' elements to be the content of the 'div' on our webpage itself
- Our code layout determines our actual layout (semi-visual coding!)
- We have to expand our 'createElement' function to handle a array of elements as the 'content' of another element

mypage.html ▶

JS + Webcore]

mypage.js ▼

```
1 let name = ''; let vDOM; let elems
2
3 function createVDOM(){
4   return [['div', [
5     ['p', "Gez", function handle1 (e){name = e.target.textContent}],
6     ['p', "Ginger", function handle2 (e){name = e.target.textContent}],
7     ["p", `${name}!`, function handle3 (e){name = e.target.textContent}]]]]
8
9 function updateDOM() {
10   vDOM = createVDOM()
11   elems = vDOM.map(convertNested);
12   document.body.replaceChildren(...elems);
13 }
14
15 function convertNested(node){
16   const element = document.createElement(node[0])
17   if (node[1] instanceof Array){
18     const childElems = node[1].map(convertNested);
19     element.append(...childElems)
20   }
21   else { element.textContent = node[1] }
22   element.onclick = node[2]
23   return element
24 }
25
26 setInterval(updateDOM, 15)
```

## Introducing event delegation for performance/efficiency

- In practice we'll have 20, 50, 100+ elements each needing their own event handler function - totally inefficient
- Fort... feature also includes 'event bubbling' - the event floats 'up' and each parent on the DOM tree gets hit with the event - and if it has a handler function that function will run in JavaScript

CUT

mypage.html ▶

mypage.js ▼

```

1 let name = ''; let vDOM; let elems
2
3 function createVDOM(){
4   return [['div', [
5     ['p', "Gez"],
6     ['p', "Ginger"],
7     ['p', `${name}!`]],
8     function handle (e){name = e.target.textContent}]]
9 }
10 function updateDOM() {
11   vDOM = createVDOM()
12   elems = vDOM.map(convertNested);
13   document.body.replaceChildren(...elems);
14 }
15 function convertNested(node){
16   const element = document.createElement(node[0])
17   if (node[1] instanceof Array){
18     const childElems = node[1].map(convertNested);
19     element.append(...childElems)
20   } else { element.textContent = node[1] }
21   element.onclick = node[2]
22   return element
23 }
24 setInterval(updateDOM, 15)

```

## Let's try the decoration approach - \*directives\*

- Each element on the page has a chance to 'do' something in the user's eyes
- In reality that 'doing' is happening in JavaScript (e.g. checking a conditional, a loop etc) and then some updating of the view (DOM)
- We can make our vDOM elements store the functionality

mypage.html ►

JS + Webcore

mypage.js ▼

```
1 let name = ''; let vDOM; let elems;
2
3 function createVDOM () {
4   return [["input", name, function handle (e){name = e.target.value}],
5           ["div", `Hello, ${name}!`, , setColor]]
6 }
7 function setColor(el, data) { if (data == "Will") {el.style.color = "lightgreen" } }
8
9 function updateDOM() {
10   vDOM = createVDOM()
11   elems = vDOM.map(convert)
12   document.body.replaceChildren(...elems)
13 }
14 function convert([type, contents, handler, fn]){
15   const element = document.createElement(type)
16   element.textContent = contents
17   element.value = contents
18   element.oninput = handler
19   if (fn) { fn(element, name) }
20   return element
21 }
22 setInterval(updateDOM,15)
```

# Could we augment our VDOM elements to include additional functionality?

Couple of interesting ways we could try:

1. Write statements that evaluate directly in the array
2. Create functions that take in our element and add functionality to it ('decorate' it) then return it - Directives
3. Create our elements with a function that returns them out but before it does, it can run code to determine exactly what it returns out - functional components

## Alternatively produce our elements with functions

- Let's us add logic/code/ functionality before we return out the VDOM element
- We still get to have declarative UI (see line 11) - but now with some logic added

*mypage.html* ▶

*mypage.js* ▼

```
1 let name = ''; let vDOM; let elems;
2
3 function Input(){
4   return ['input', name, function handle (e){name = e.target.value}]
5 }
6 function Notifcation(){
7   let className = 'darkred'
8   if (name == "Will"){className = 'lightgreen'}
9   return ["div", `Correct ${name}!`, , className]
10 }
11 function updateDOM() {
12   vDOM = [Input(), Notifcation()]
13   elems = vDOM.map(convert);
14   document.body.replaceChildren(...elems)
15 }
16 function convert(node){
17   const element = document.createElement(node[0])
18   element.textContent = node[1]
19   element.value = node[1]
20   element.oninput = node[2]
21   element.classList.add(node[3])
22   return element
23 }
24 setInterval(updateDOM, 15)
```