Building Ajax Applications

In which we dip into the curious world of Old-School Ajax Web Development with stateful global references and asynchronous interactions and updates.

Before we begin, where do I put my code?

```html
<!DOCTYPE html>
<html>
  <head>
    <link href="/path.to/my/css/stylesheets.css" rel="stylesheet">
  </head>
  <body>
    <p>Some content here.</p>
    <script type="text/javascript" src="http://path.to/shared/library.js"></script>
    <script type="text/javascript" src="/path.to/my/awesome/app.js"></script>
    <script>
      document.addEventListener("DOMContentLoaded", function () {
        console.log('Hello, browser!')
      })
    </script>
  </body>
</html>
```

Before we begin, where do I put my code?

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <link href="/path.to/my/css/stylesheets.css" rel="stylesheet">
5      <script type="text/javascript" src="http://path.to/shared/library.js" defer></script>
6      <script type="text/javascript" src="/path.to/my/awesome/app.js" defer></script>
7      <script defer>
8        document.addEventListener("DOMContentLoaded", function () {
9          console.log('Hello, browser!')
10       })
11     </script>
12   </head>
13   <body>
14     <p>Some content here.</p>
       <button id="my-button">CLICK ME!</button>
15   </body>
16 </html>
```

Asynchronous programming

same order that requests were made. Responses are not
required to come back in the

- The ECMAScript runtime is **single-threaded**.
- If a piece of code is using the thread, nothing else can execute.
- If an event occurs during that time, its callback is queued for execution.
- When the thread is freed, a queued callback is chosen for execution.
- Long running executions are killed by the browser.

- Events can be generated by the browser, by the user, by timers and by the network.
- Zero or more callbacks can be associated to a single event.

Handling user events

```
1 document.getElementById('my-button').addEventListener('click', function (evt) {
2   this //=> points to the clicked element
3   evt.stopPropagation() //=> stops the event from bubbling
4   evt.preventDefault() //=> prevents the default behaviour
5 })
```

Handling user events with jQuery

```
1 $('#my-button').on('click', function (evt) {
2   this //=> points to the clicked element
3   return false //=> stops the event from bubbling
4                //=> and prevents the default behaviour
5 })
```

Handling browser events

```
1 document.addEventListener("DOMContentLoaded", function (evt) {
2   // Executes when the page is fully parsed.
3   this //=> points to the document object
4 })
```

Handling browser events with jQuery

```
1 $(function (evt) {
2   // Executes when the page is fully parsed.
3   this //=> points to the document object
4 })
```

Handling timer events

```
1 var timeoutId = window.setTimeout(function (a, b) {
2   // Will get executed in about 500 ms.
3   this //=> points to `window`
4   a //=> 'foo'
5   b //=> 'bar'
6 }, 500, "foo", "bar")
7
8 window.clearTimeout(timeoutId) //=> remove callback from queue.
```

Why isn't this working?

```
1  function delay (arr) {
2    for (var i = 0; i < arr.length; i++) {
3      setTimeout(function () {
4        console.log(arr[i])
5      }, 0)
6    }
7  }
8
9  delay([ 1, 2, 3 ])
10 // prints:
11 //   undefined
12 //   undefined
13 //   undefined
```

Would this work?

```
1 function delay (arr) {
2   for (var i = 0; i < arr.length; i++) {
3     setTimeout(function (i) {
4       console.log(arr[i])
5     }, 0, i)
6   }
7 }
```

How about this?

```
1 function delay (arr) {
2   for (var i = 0; i < arr.length; i++) {
3     (function (i) {
4       setTimeout(function () {
5         console.log(arr[i])
6       }, 0)
7     }(i))
8   }
9 }
```

Handling XMLHttpRequest events

```
1  var xhr = new XMLHttpRequest()
2  xhr.open('GET', '/path.to/backend/application')
3  xhr.onreadystatechange = (function() {
4    // Called when the state of the request changes.
5    // 0 => UNSENT, 1 => OPENED, 2 => HEADERS_RECEIVED, 3 => LOADING, 4 => DONE
6    if (xhr.readyState === 4 && xhr.status === 200) {
7      xhr.responseText //=> contains the payload as a string
8      xhr.responseXML //=> contains the payload parsed as XML
9      try {
10       var data = JSON.parse(xhr.responseText)
11     } catch (e) {
12       // Can't parse this.
13     }
14   }
15 })
16 xhr.send()
17 xhr.abort() //=> cancels the running request
```

Handling XMLHttpRequest events with jQuery's Deferred and Promise classes

```
1 var promise = $.getJSON('/path.to/backend/application').then(
2   function (data, textStatus, xhr) {
3     // Things worked!
4   },
5   function (xhr, textStatus, errorThrown) {
6     // Something failed!
7   }
8 )
```

- Always use callbacks (or promises, or deferreds) for long running tasks.
- Consider using IIFEs or the "bind" function when dealing with asynchronous tasks.
- Use specialized librairies to orchestrate asynchronous tasks (e.g. async.js, Bluebird, ES2015 Promises)

Ajax programming

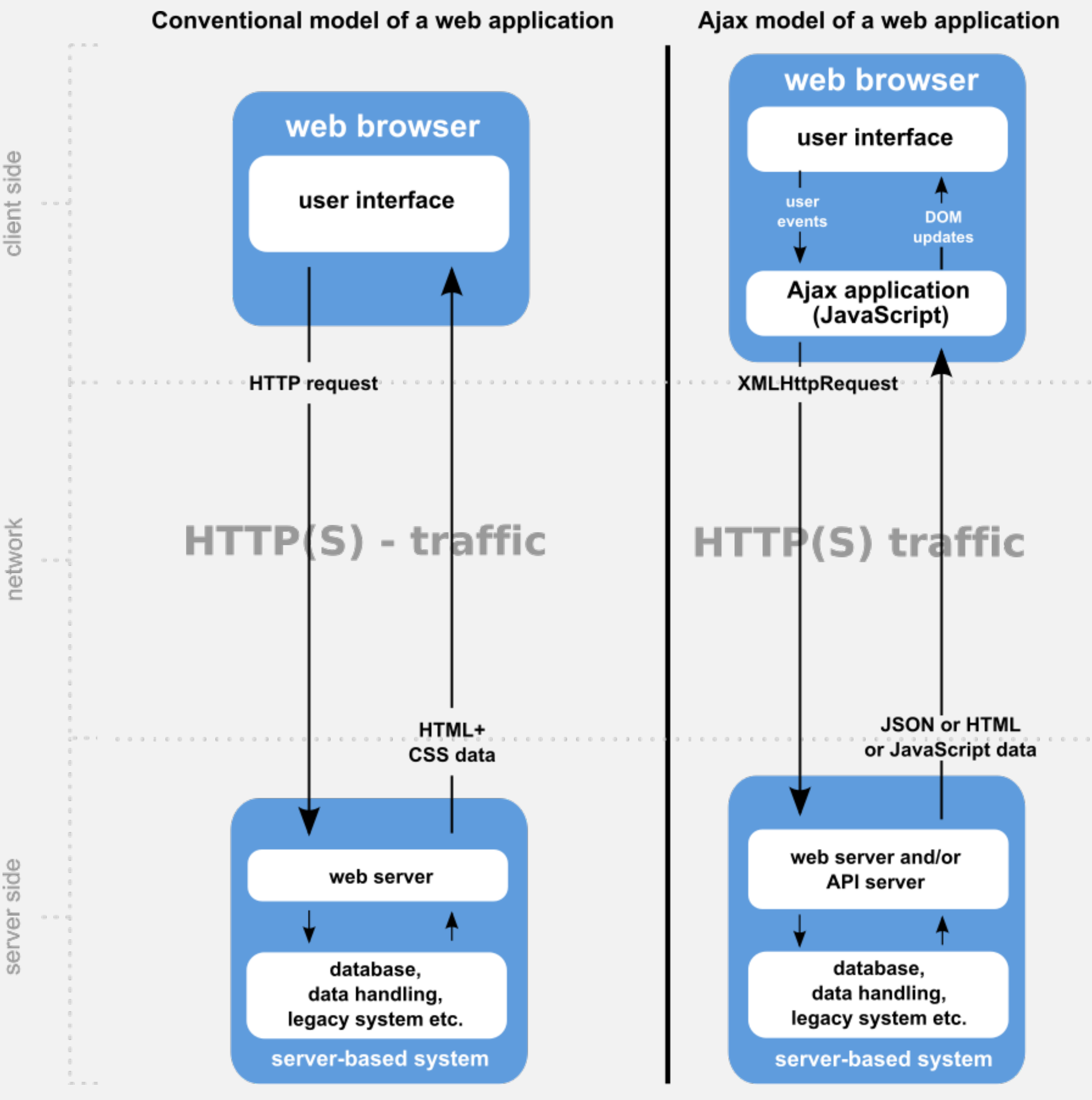
What do cleaning products have to do with Web development?

- **Asynchronous.** The application leverages JavaScript's single-threaded and eventful nature for server communication (XHR) and HTML manipulation.
- **JavaScript.** The application is completely written in ECMAScript.
- **And.** Used to connect words of the same part of speech, clauses, or sentences that are to be taken jointly.
- **XML.** Back in 2005, XML was cutting edge stuff. APIs can consume and produce anything though. JSON is the current favorite among developers.

Ajax applications run in a **single HTML page**. They respond to user events and then may send data to the backend.

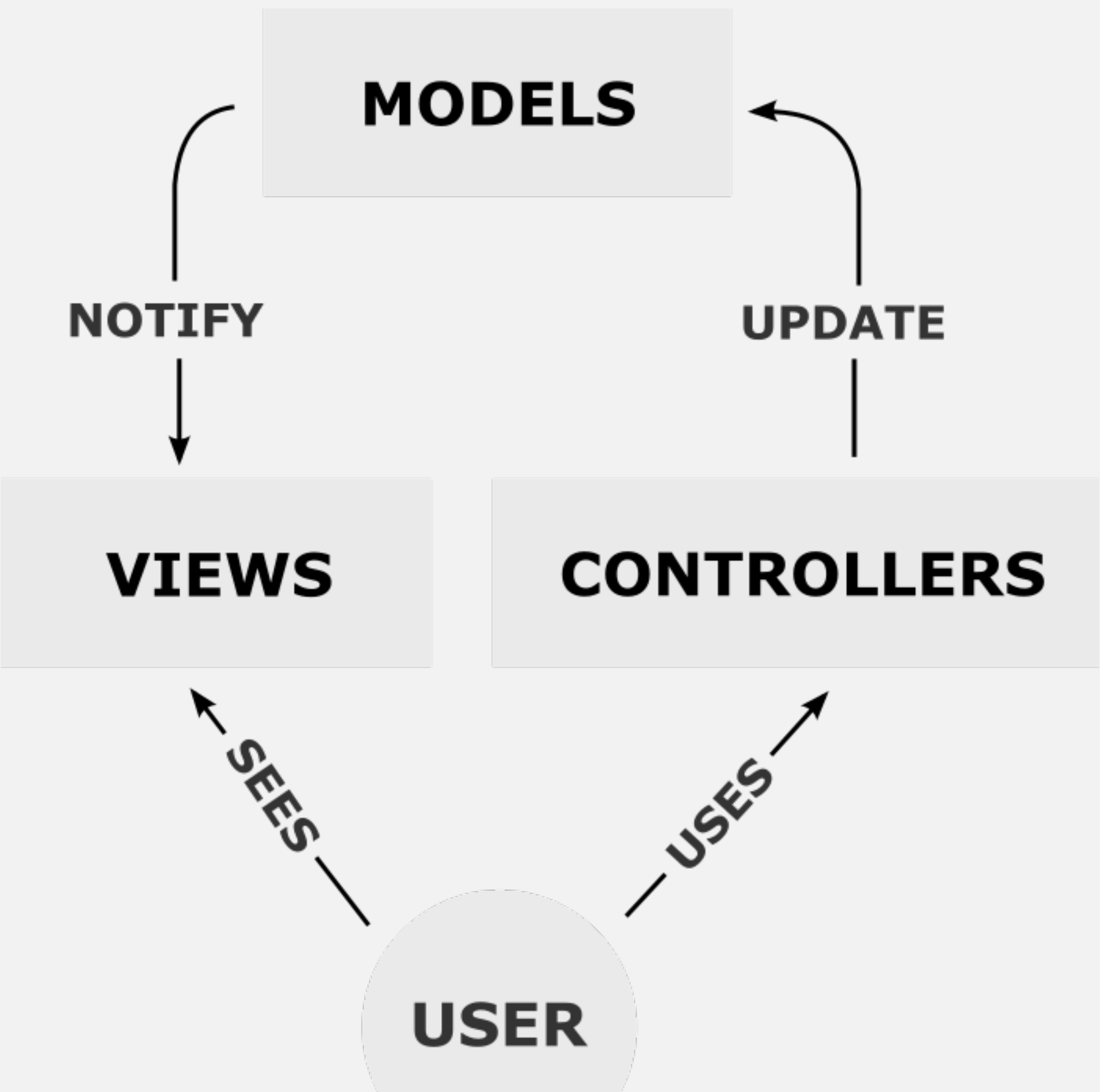
When data is sent back, the application may **modify the DOM** (Document Object Model, the rendered HTML page) accordingly.

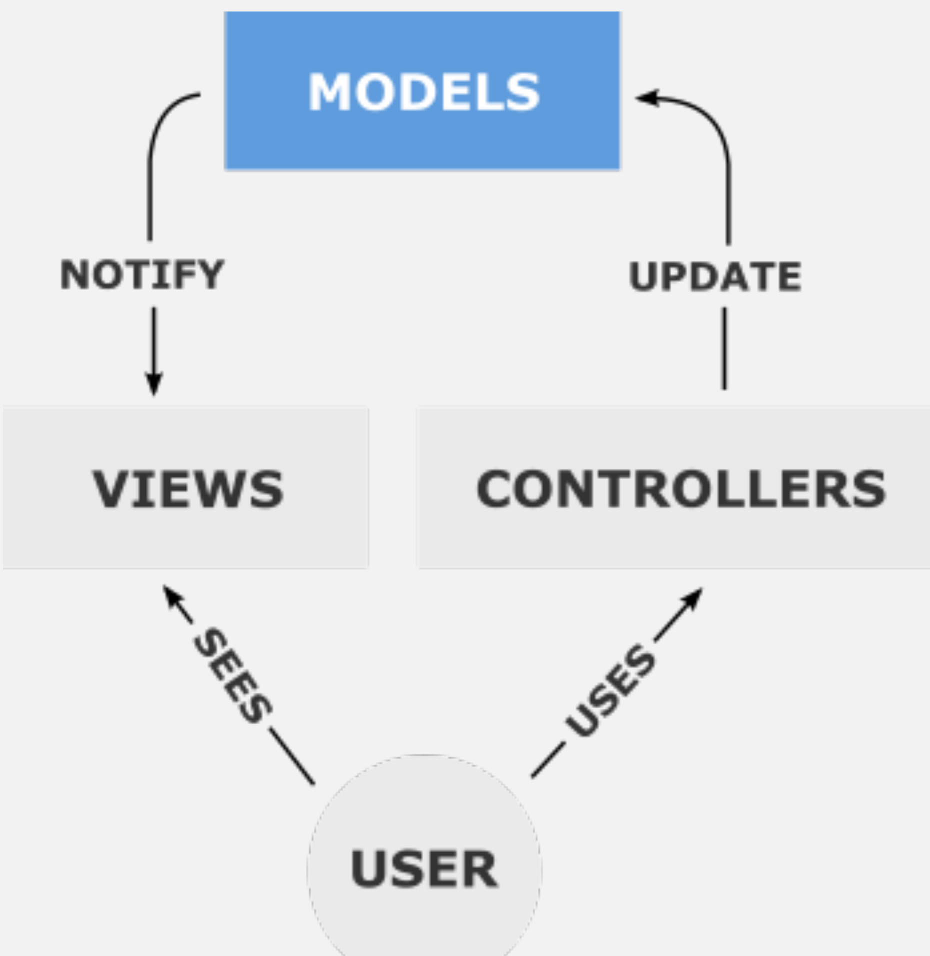
The backend may respond with raw data (data-centric APIs), with HTML fragments (content-centric APIs) or with ECMAScript snippets (code-centric APIs). **Data-centric** APIs, formatted in JSON are usually favoured.



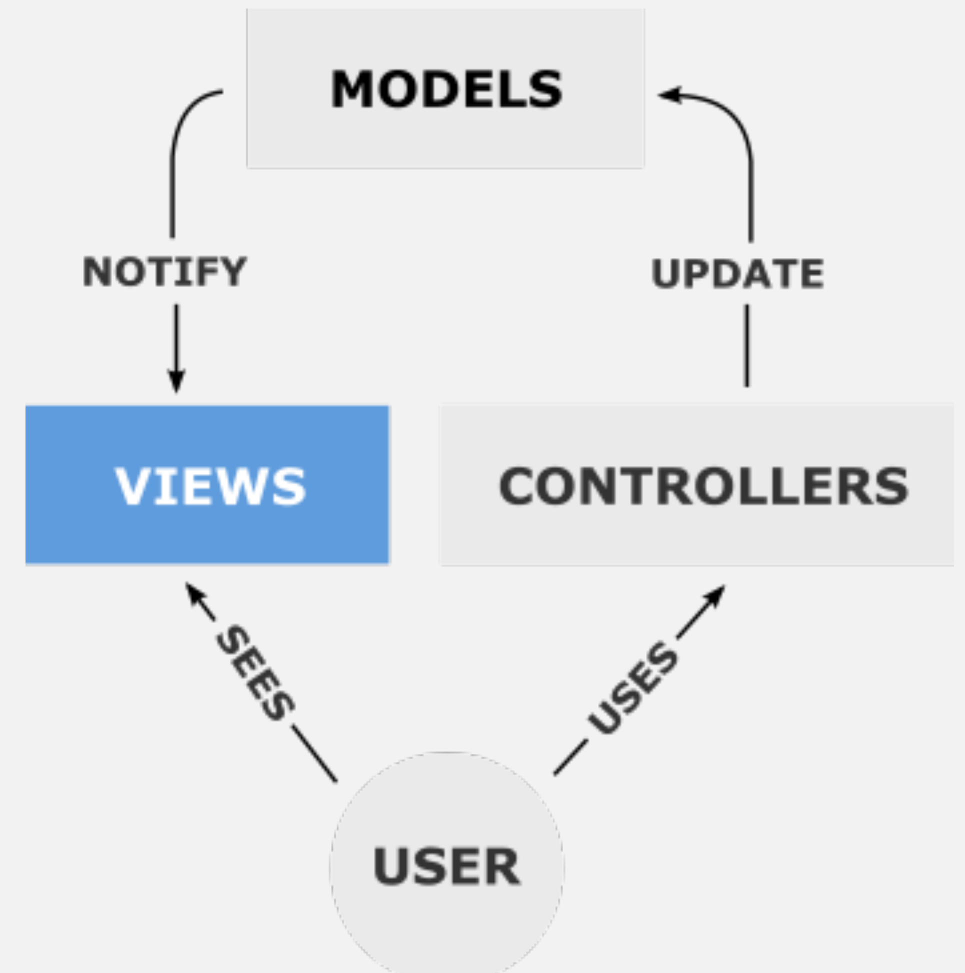
"Ajax-vergleich-en.svg" by DanielSHaischt available under the Creative Commons Attribution-Share Alike 3.0 Unported license. Copyright @ 2013, Daniel S. Haischt.
Modified by François-Xavier Guillemette, redistributed under the same license.

The application code can quickly turn to spaghetti. The classic **Model-View-Controller** design pattern is used to separate concerns and ease maintenance.

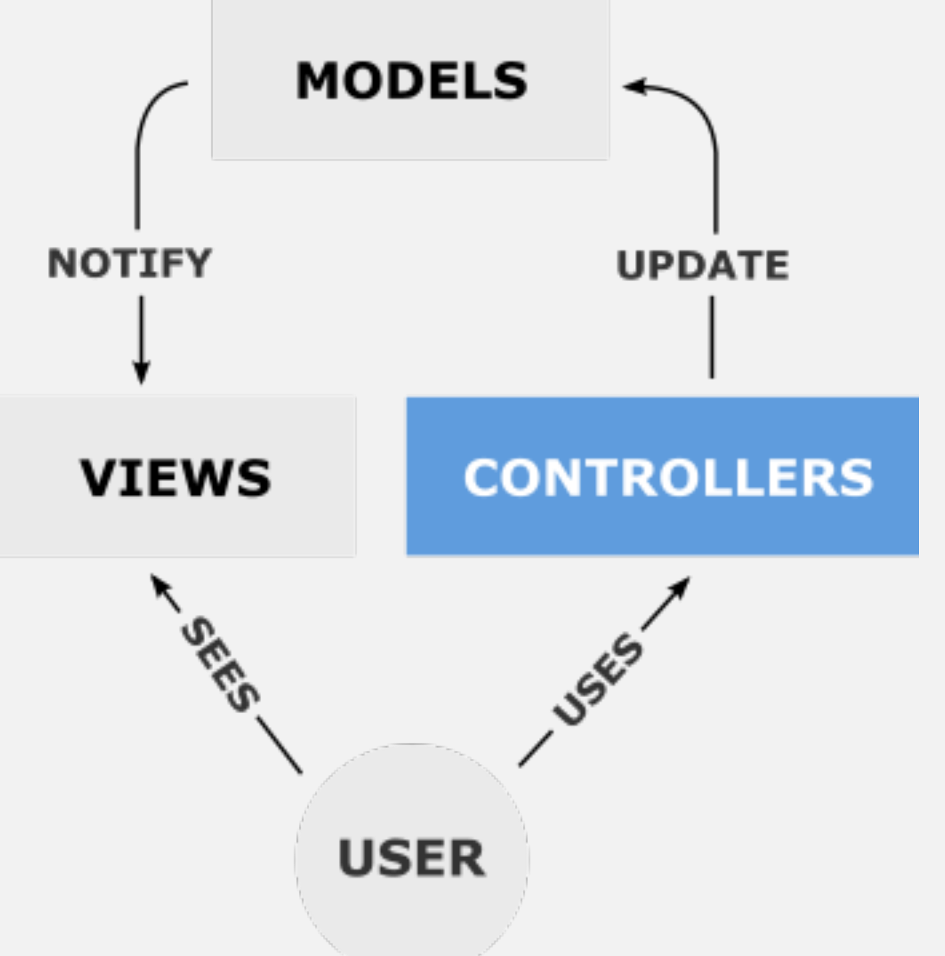

MODELS
NOTIFY
UPDATE
VIEWS
CONTROLLERS
SEES
USES
USER



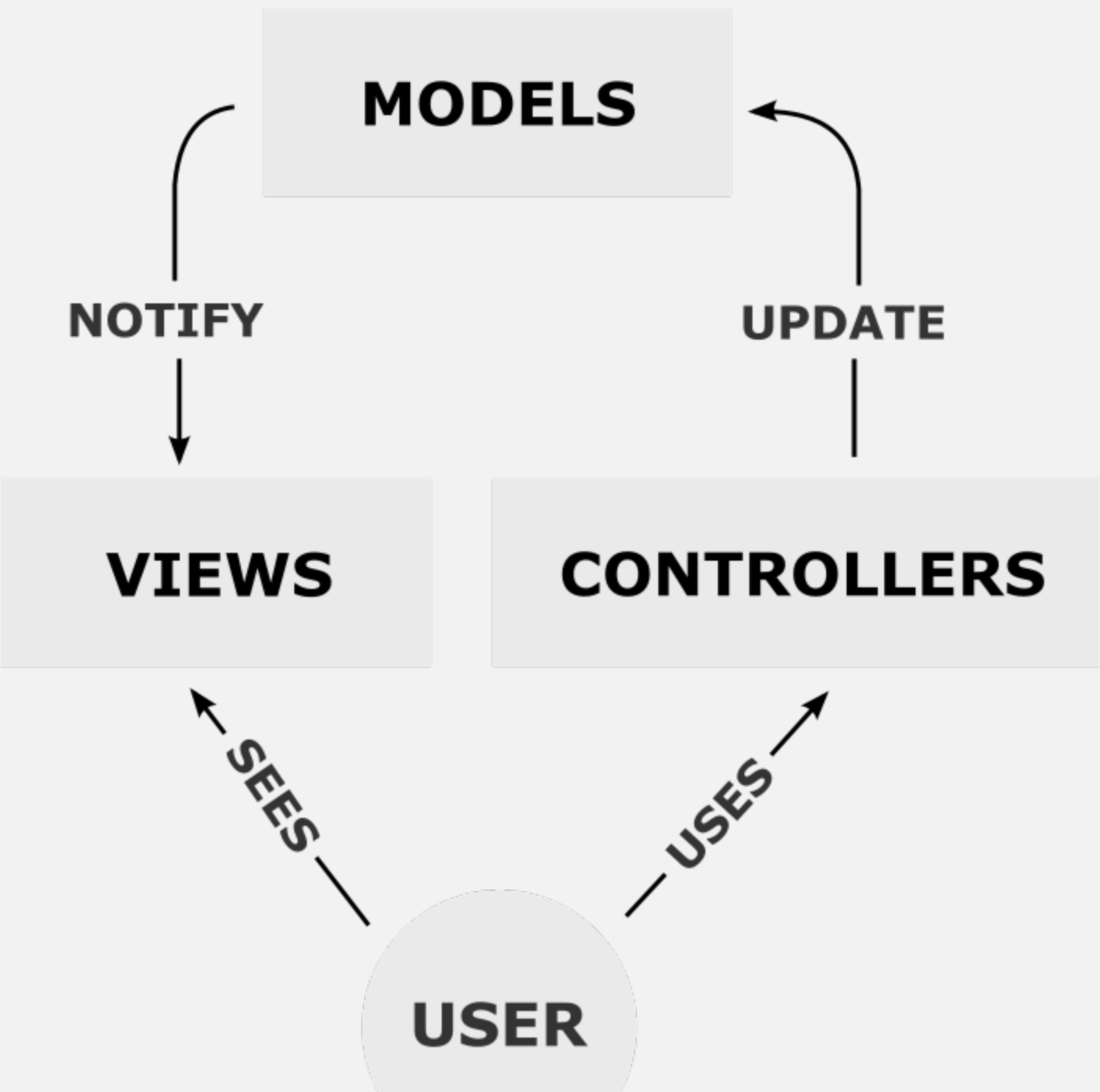
- **Models** hold the application state, including the data that is rendered in the DOM.
- They maintain a list of subscribers which they notify when the state changes.
- Models get updated by Controllers and other Models.



- **Views** render the models' data by modifying the DOM.
- They subscribe to events emitted by the Models.
- Views emit events generated by the user.



- **Controllers** subscribe to events emitted by the Views.
- They react to the events by possibly calling upon services to synchronize data with the server or updating Models.


MODELS
NOTIFY
UPDATE
VIEWS
CONTROLLERS
SEES
USES
USER

Querying the DOM

```
1 elements = document.getElementsByClassName('some-class')
2 // matches all `<div class="some-class"></div>` elements.
3
4 element = document.getElementById('some-id')
5 // matches the first `<div id="some-id"></div>` element.
```

Querying the DOM

```
1 var allForms = document.getElementsByTagName('form')
2 var myForms = Array.prototype.filter.call(allForms, function (formEl) {
3   var matches = false
4   if (formEl.id === 'my-form') {
5     Array.prototype.forEach.call(formEl.children, function (child) {
6       if (child.tagName === 'INPUT'
7             && child.getAttribute('type') === 'submit'
8               && child.classList.contains('btn-large')) {
9         matches = true
10      }
11    })
12  }
13  return matches
14 })
15 // matches all elements like:
16 //
17 //     <form id="my-form">
18 //       <input type="submit" class="btn-large some-other-class">
19 //     </form>
```

Querying the DOM

```
elements = document.querySelectorAll('.some-class')
// matches `<div class="some-class"></div>` elements.

element = document.querySelector('#some-id')
// matches the first `<div id="some-id"></div>` element.

elements = document.querySelectorAll(
    'form#my-form > input.btn-large[type="submit"]')
// matches all:
//
//     <form id="my-form">
//       <input type="submit" class="btn-large some-other-class">
//     </form>
```

Querying the DOM with jQuery

```
1  elements = $('.some-class')
2  // matches `<div class="some-class"></div>` elements.
3
4  element = $('#some-id')
5  // matches the first `<div id="some-id"></div>` element.
6
7  elements = $('form#my-form > input.btn-large[type="submit"]')
8  // matches all:
9  //
10 //     <form id="my-form">
11 //       <input type="submit" class="btn-large some-other-class">
12 //     </form>
```

Modifying the DOM

```
1  var form = document.getElementById('my-form')
2
3  var input = document.createElement('input')
4  input.setAttribute('type', 'text')
5  input.setAttribute('name', 'username')
6  form.appendChild(input)
7
8  var submit = document.querySelector('input[type="submit"]')
9  form.removeChild(submit)
10
11 var title = document.createTextNode('A Formidable Form')
12 var h1 = document.createElement('h1')
13 h1.appendChild(title)
14 document.body.insertBefore(h1, form)
15
16 // <body>
17 //   <h1>A Formidable Form</h1>
18 //   <form id="my-form">
19 //     <input type="text" name="username">
20 //   </form>
21 // </body>
```

Modifying the DOM with jQuery

```
1  var form = $('#my-form')
2
3  var input = document.createElement('input')
4  input.setAttribute('type', 'text')
5  input.setAttribute('name', 'username')
6  form.append(input)
7
8  form.find('input[type="submit"]').remove()
9
10
11 var title = document.createTextNode('A Formidable Form')
12 var h1 = document.createElement('h1')
13 h1.appendChild(title)
14 $(form).before(h1)
15
16 // <body>
17 //   <h1>A Formidable Form</h1>
18 //   <form id="my-form">
19 //     <input type="text" name="username">
20 //   </form>
21 // </body>
```

Modifying the DOM

```
1 var form = document.getElementById('my-form')
2
3 form.innerHTML = "<input type='text' name='username'>"
```

Modifying the DOM with jQuery

```
1 var form = $('#my-form')
2
3 form.html("<input type='text' name='username'>")
```

For moderately-sized applications, no one does it by hand anymore. Currently favoured frameworks and libraries include React (with Flux or Redux), Angular 2 and Ember. The examples at todomvc.com are a good reference.

Some exercices

- Orchestrate 2 parallel XMLHttpRequest calls so that a function is called only when both Ajax calls have successfully completed. The function must have access to the two fetched JSON objects.
- Use "https://api.github.com/users/fxg42" as the first URL and "https://api.github.com/users/fxg42/repos" as the second.

Exercices (cont.)

- Do the same exercice but use jQuery's "$.getJSON(…).then(…)" function to make the Ajax calls.
- Do the same exercice but use jQuery's "$.getJSON(…)" function and the "success" callback argument to make the Ajax calls.

Exercices (cont.)

- Write an Ajax application that fetches a GitHub user's info (the username must taken from an HTML form) and displays it on the page.

Orchestrating by hand (1 of 4)

```
1 var user, repos
2 var id = "fxg42"
3 fetchUser(id)
4 fetchUserRepos(id)
5
```

Orchestrating by hand (2 of 4)

```
6 function fetchUser (id) {
7   var xhr = new XMLHttpRequest()
8   xhr.open('get', "https://api.github.com/users/"+id)
9   xhr.onreadystatechange = (function() {
10    if (xhr.readyState === 4 && xhr.status === 200) {
11      user = JSON.parse(xhr.responseText)
12      join()
13    }
14  })
15  xhr.send()
16 }
17
```

Orchestrating by hand (3 of 4)

```
18 function fetchUserRepos (id) {
19   var xhr = new XMLHttpRequest()
20   xhr.open('get', "https://api.github.com/users/"+id+"/repos")
21   xhr.onreadystatechange = (function() {
22     if (xhr.readyState === 4 && xhr.status === 200) {
23       repos = JSON.parse(xhr.responseText)
24       join()
25     }
26   })
27   xhr.send()
28 }
29
```

Orchestrating by hand (4 of 4)

```
30 function join () {
31   if (user !== undefined && repos !== undefined) {
32     displayUserInfo(user, repos)
33   }
34 }
35
36 function displayUserInfo (user, repos) {
37   console.log(user, repos)
38 }
```

Orchestrating with jQuery Promises

```
1   var id = "fxg42"
2   $.when(fetchUser(id), fetchUserRepos(id)).then(displayUserInfo)
3
4   function justData (data, status, promise) { return data }
5
6   function fetchUser (id) {
7     return $.getJSON("https://api.github.com/users/"+id).then(justData)
8   }
9
10  function fetchUserRepos (id) {
11    return $.getJSON("https://api.github.com/users/"+id+"/repos").then(justData)
12  }
13
14  function displayUserInfo (user, repos) {
15    console.log(user, repos)
16  }
```

Orchestrating with the (future) Fetch API

```
1  var id = 'fxg42'
2
3  Promise.all([
4    fetch("https://api.github.com/users/"+id).then(function (resp) {
5      return resp.json()
6    }),
7    fetch("https://api.github.com/users/"+id+"/repos").then(function (resp) {
8      return resp.json()
9    })
10 ]).then(function (payloads) {
11   displayUserInfo(payloads[0], payloads[1])
12 })
13
14 function displayUserInfo (user, repos) {
15   console.log(user, repos)
16 }
```

Orchestrating with the Fetch API and ES2015

```
1 const id = 'fxg42'
2
3 Promise.all([
4   fetch(`https://api.github.com/users/${id}`).then(resp => resp.json()),
5   fetch(`https://api.github.com/users/${id}/repos`).then(resp => resp.json())
6 ]).then(displayUserInfo)
7
8 const displayUserInfo = ([user, repos]) =>
9   console.log(user, repos)
```