Asynchronous operations in JavaScript

Alessio Vecchio

Event-based, callbacks

- JavaScript code is **event-based**: code is executed in response to events
- **Events**: the user presses a button, a timer fires, some data is received from the network, etc
- We used **callbacks** to specify what to do in response to events
- **Callback**: a function that is passed as an argument to another function
  - The second function calls back the first one when the event occurs

An example: timer

- Every 1000 ms the timer fires and the callback is executed

```
let counter = 0;

function mycallback() {
  // In this example, the code is simple
  // just printing some message
  console.log("Counter: " + counter++);
  if(counter === 5) {
    clearTimeout(timerhandle);
  }
}

let timerhandle = setInterval(mycallback, 1000);
```

Another example: user's input

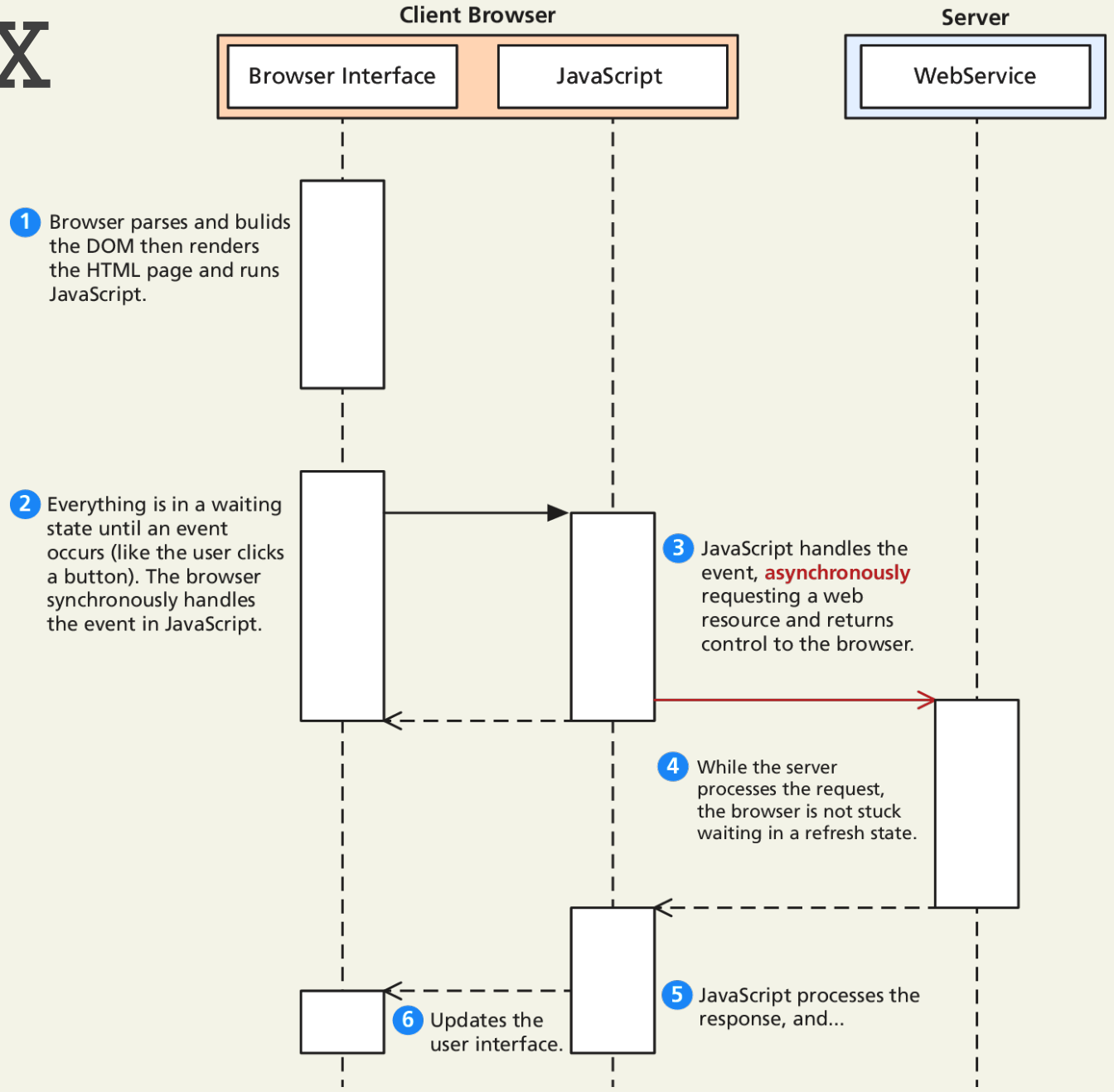
```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Reacting to user's input</title>
<script>
function mycallback() {
  alert("Hi!");
}
function init() {
  const b = document.getElementById("btn");
  b.addEventListener("click", mycallback);
}
window.onload = init;
</script>
</head>
<body>
<button id="btn">Press Me</button>
</body>
```

- There are 2 callbacks

Using the network asynchronously: AJAX

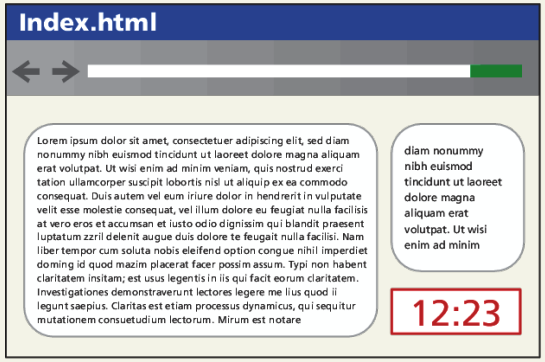
- **Asynchronous JavaScript** and **XML** (AJAX) is a set of techniques used on the client-side to create interactive web applications
- Web applications can retrieve data from the server asynchronously without interfering with the display and behavior of the existing page
- The request is carried out in the background
- Information is retrieved using **XMLHttpRequest**
- The idea is to **not load a new web page** document

AJAX

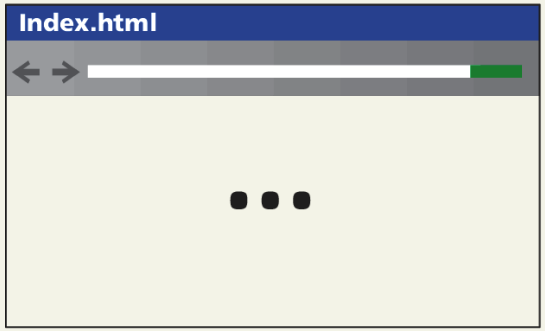

Client Browser
Browser Interface
JavaScript
Server
WebService
1 Browser parses and bulids the DOM then renders the HTML page and runs JavaScript.
2 Everything is in a waiting state until an event occurs (like the user clicks a button). The browser synchronously handles the event in JavaScript.
3 JavaScript handles the event, asynchronously requesting a web resource and returns control to the browser.
4 While the server processes the request, the browser is not stuck waiting in a refresh state.
5 JavaScript processes the response, and...
6 Updates the user interface.

AJAX

Consider a webpage that displays the server's time

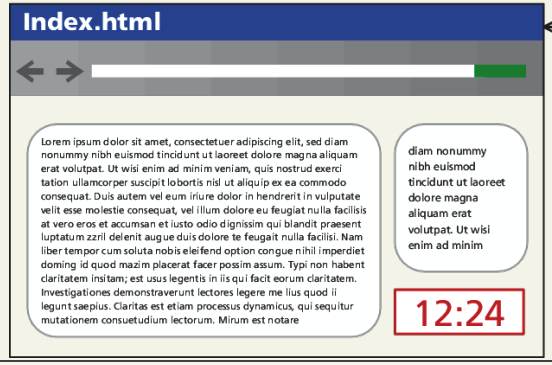


1 The page loads and shows the current server time as a small part of a larger page.



2 A **synchronous** JavaScript call makes an HTTP request for the "freshest" version of the page.

While waiting for the response, the browser goes into its waiting state.

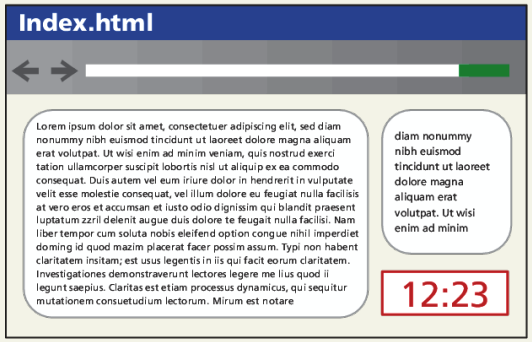


3 The response arrives, so the browser can render the new version of the page, and the functionality in the browser is restored.

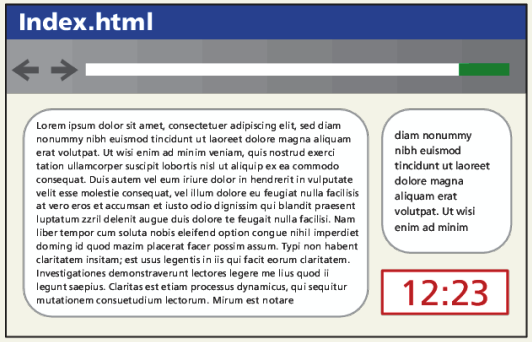
```
<html>
        <head>
        ...
        </head>
        <body>
        ...
        <div id='serverTime'>
                12.24
        </div>
        ...
        </body>
</html>
```

AJAX

Consider a webpage that displays the server's time

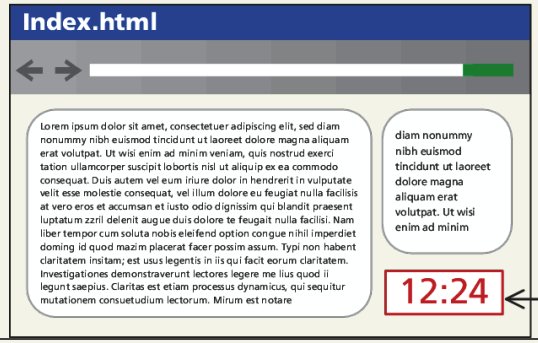


1 The page loads and shows the current server time as a small part of a larger page.



2 An **asynchronous** JavaScript call makes an HTTP request for just the small component of the page that needs updating (the time).

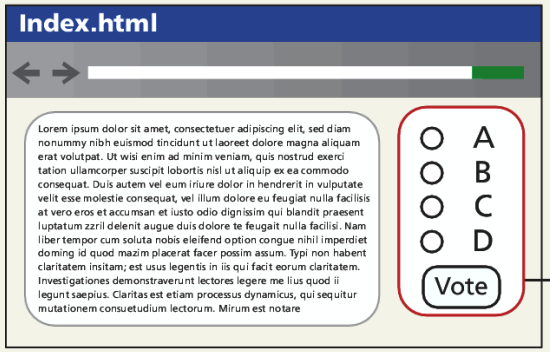
While waiting for the response, the browser still looks the same and is responsive to user interactions.



3 The response arrives, and through JavaScript, the HTML page is updated.

AJAX

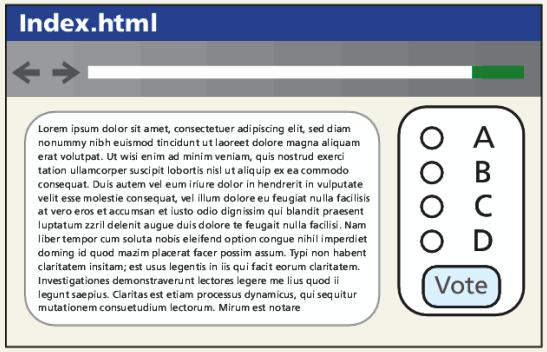
GET Requests

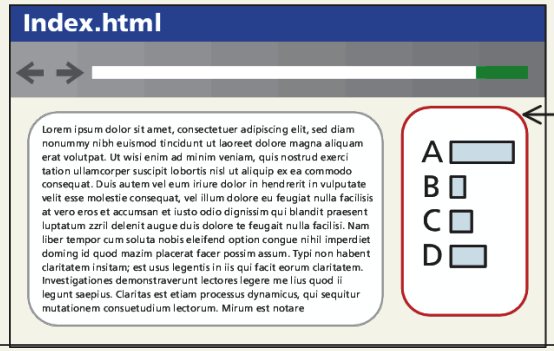


1 The HTML page contains a poll that posts votes asynchronously.

2 An **asynchronous** vote submits the user's choice.

Meanwhile, the browser remains interactive while the request is processed on the server.

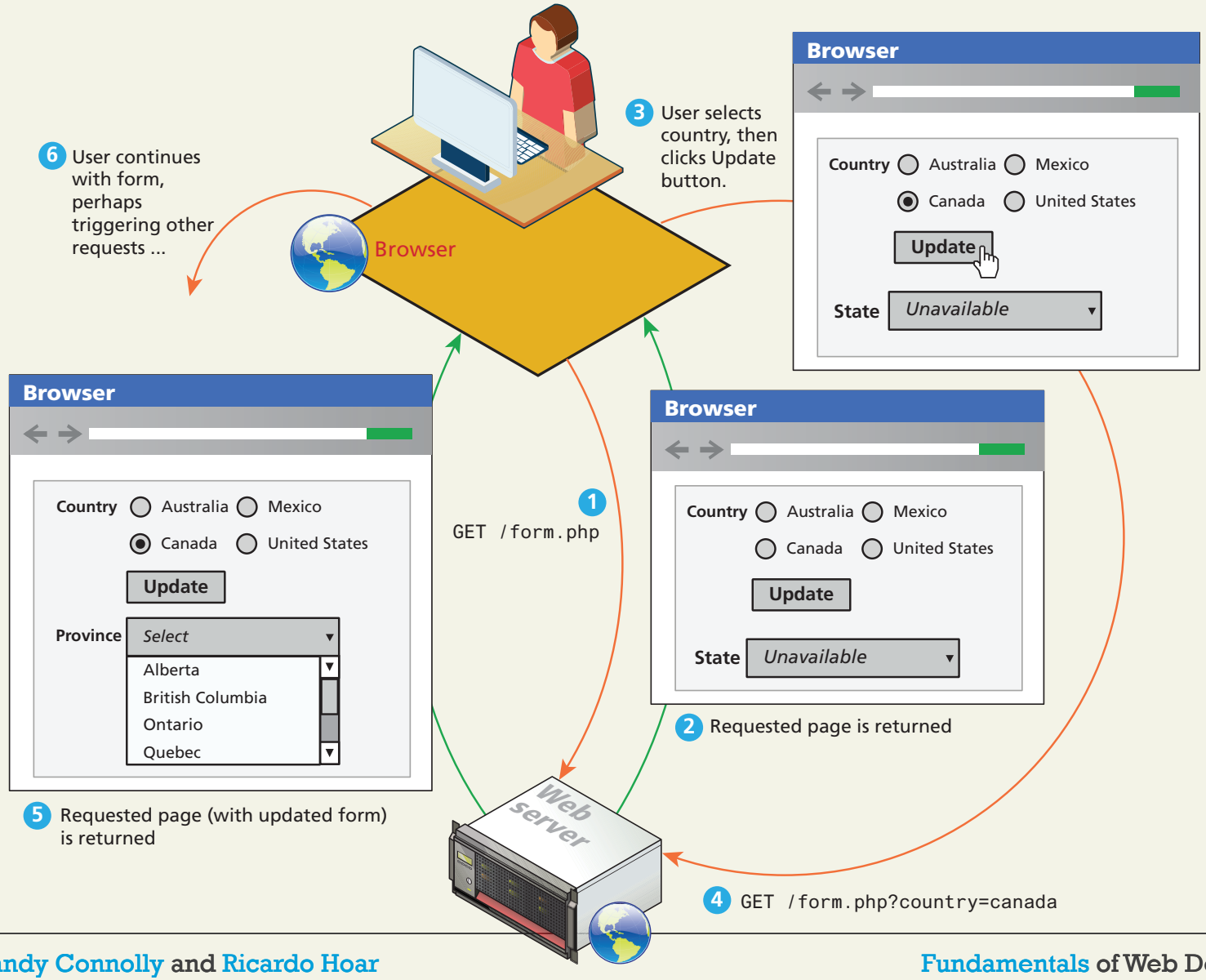




3 The response arrives, and is handled by JavaScript, which uses the response data to update the interface to show poll results.

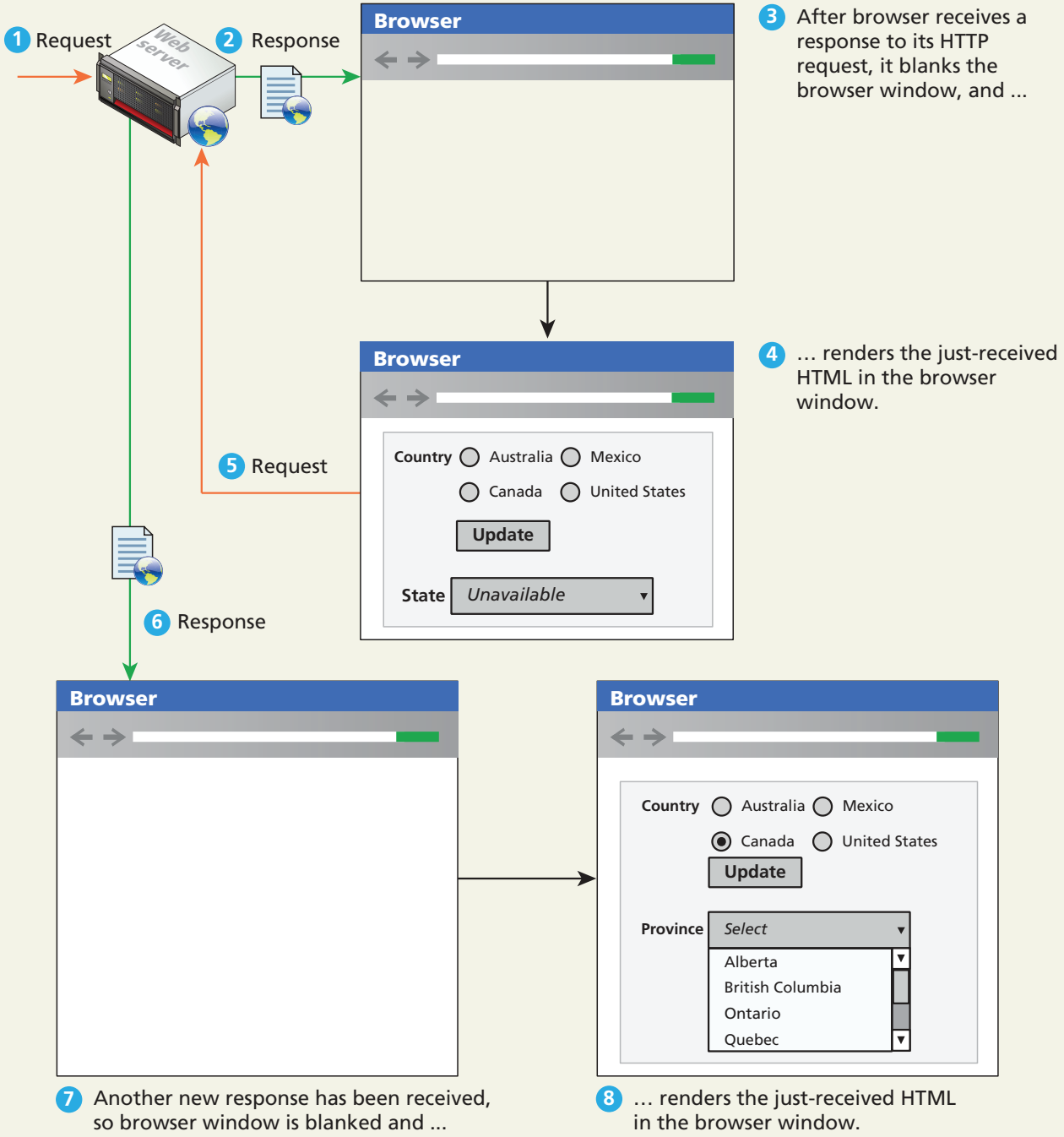
AJAX

First the Normal Request Response Loop



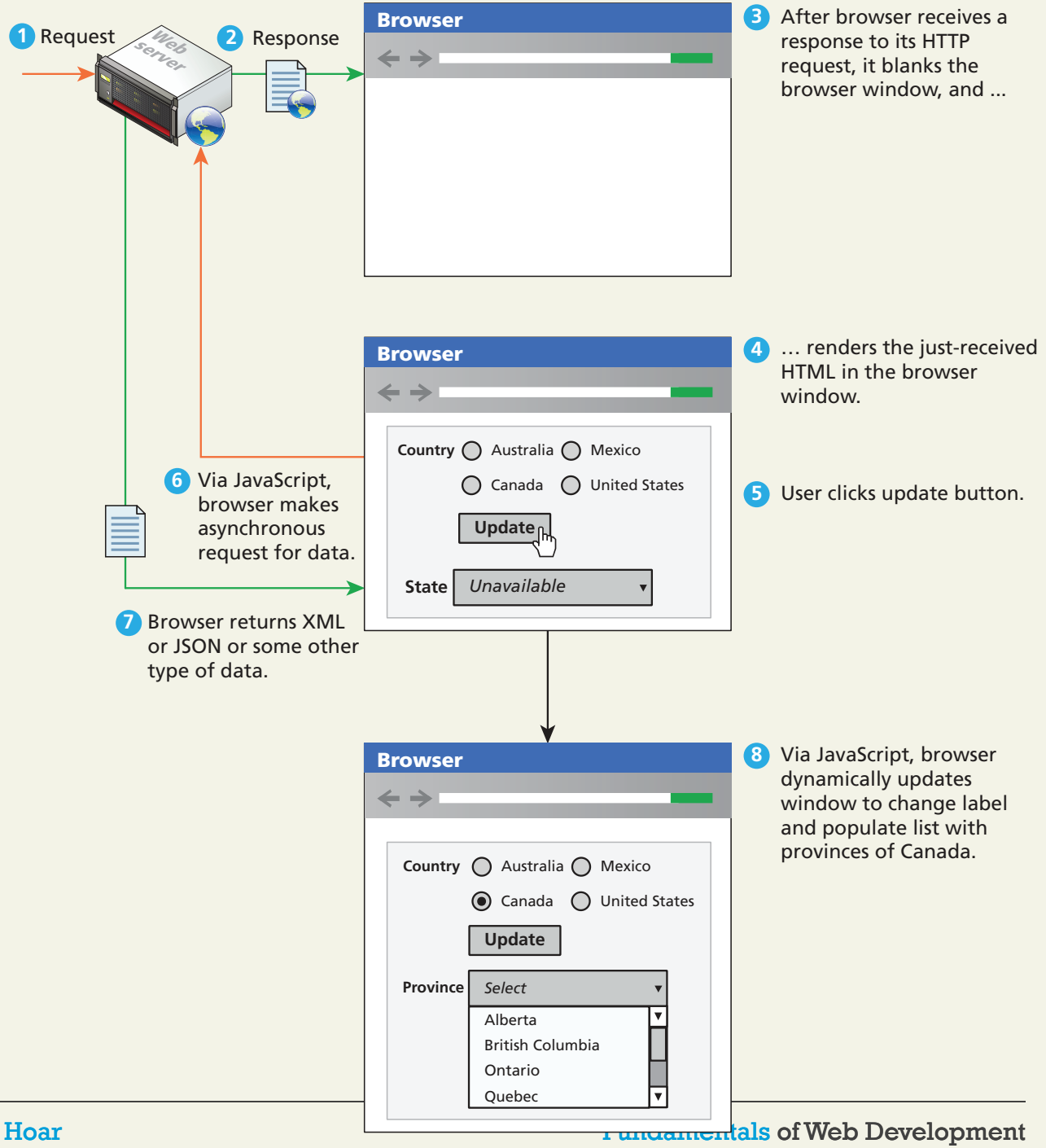
AJAX

The problem



AJAX

The solution



Using the network asynchronously: AJAX

- **XMLHttpRequest**
  - used to send HTTP or HTTPS requests to a web server
  - The data received from the server can be an XML document, JSON, or plain text
  - Data from the response can be used directly to modify the DOM of the currently active document in the browser window

XMLHttpRequest

- To send an HTTP request you have to create an **XMLHttpRequest** object

```
let x = new XMLHttpRequest();
```

- Then you have to specify the URL, and send the request

```
x.open("GET", <URL of the data>);
x.send();
```

- When transaction completes, the object will contain the response body received from the server and the HTTP status

XMLHttpRequest

- Callback functions are executed when events occur
- Both GET and POST methods can be used
- It is possible to receive notifications about the progress of the operation, if it takes a lot of time
- The request can also be synchronous

An example

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<title>Reacting to user's input</title>
<script src="xmlreqexample.js"></script>
</head>
<body>
<button id="btn">
Press Me
</button>
</body>
```

An example

```
function success(text){
  const p = document.createElement("p");
  document.body.appendChild(p);
  p.innerText = "The reply is: " + text;
}
function fail(text) {
  const p = document.createElement("p");
  document.body.appendChild(p);
  p.innerText = "Problem: " + text;
}
```

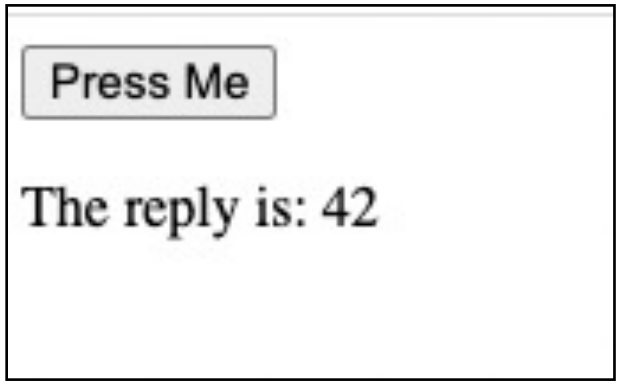
An example

```
function startrequest(){
  const x = new XMLHttpRequest();
  x.open("GET", "http://localhost/xmlrequest/long.php");
  x.onload = function() {
      console.log("Reply received");
      if(x.status == 200) success(x.response);
      else fail(x.statusText);
    };
  x.onerror = function() {
      // executed if request cannot be sent
      fail("Network Error");
    };
  x.send(); // this method returns immediately
  console.log("Request sent");
}
```

An example

```
function init(){
  const b = document.getElementById("btn");
  b.addEventListener("click", startrequest);
}

window.onload = init;
```



An example

```php
<?php
// Sleep for 3 seconds
sleep(3);
echo "42";
?>
```

Other methods of XMLHttpRequest

- send(string)

  Sends the request to the server.

  string: used for POST requests

- setRequestHeader(header, value)

  Adds HTTP headers to the request.

  header: provides the header name

  value: provides the header value

Example:

```
xmlhttp.open("POST","myscript.php");
xmlhttp.setRequestHeader("Content-type",
                         "application/x-www-form-urlencoded");
xmlhttp.send("name=Mario&age=33");
```

XMLHttpRequest progress

```
<!DOCTYPE html>
<title>Waiting for Magical Unicorns</title>
<progress id=p></progress>
<script>
  var progressBar = document.getElementById("p"),
                    client = new XMLHttpRequest();
  client.open("GET", "magical-unicorns")
  client.onprogress = function(pe) {
      if(pe.lengthComputable) {
        progressBar.max = pe.total;
        progressBar.value = pe.loaded;
      }
    }
  client.onloadend = function(pe) {
      progressBar.value = pe.loaded;
    }
client.send();
</script>
```

from **WHATWG**

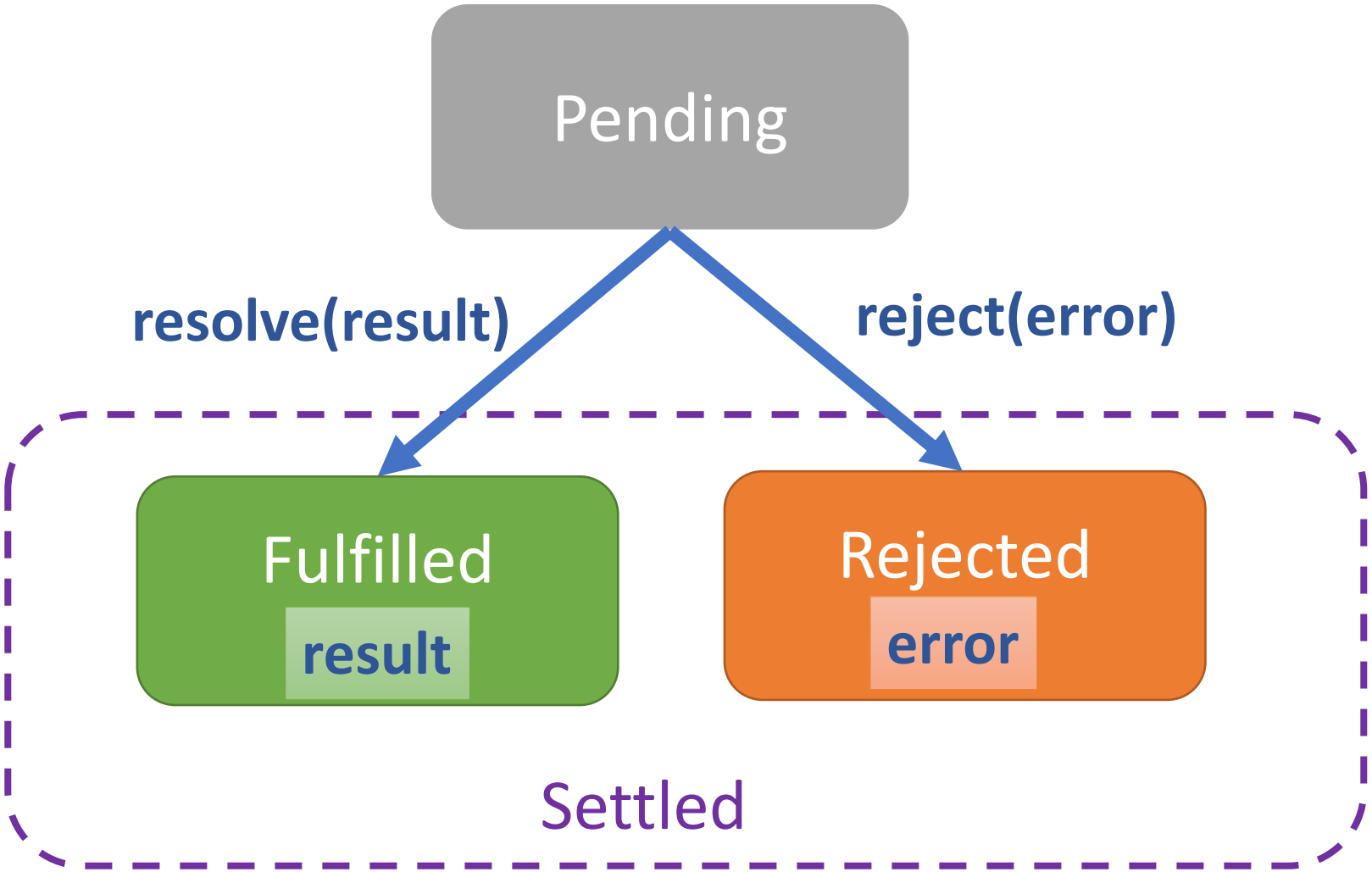
Fetch

- **fetch** is a modern version of XMLHttpRequest
- Fetch relies on the concept of **Promises** to cope with asynchronicity
  - More linear code compared to multiple levels of callbacks
  - Easier management of errors

Promise

- **Promise**: an object that represent the future result of an asynchronous operation
- A Promise can be:
  - *pending*: this is the initial state, the Promise is neither fulfilled nor rejected.
  - *fulfilled*: the operation was completed successfully
  - *rejected*: the operation failed.
- Handlers can be associated with the fulfilled and rejected states
- When the Promise reaches the fulfilled or rejected state the corresponding handler is executed

Promise


Pending
resolve(result)
reject(error)
Fulfilled
result
Rejected
error
Settled

Promise

We can use the following methods to associate actions with a Promise:

- **then():** sets the handler that is triggered in case of fulfillment. Can be used also to provide the handler in case of rejection.
- **catch():** sets the handler that is triggered in case of rejection.
- **finally():** sets the handler that is triggered when the Promise is settled (fulfill or reject).

A simple example: Promise 1

```
let prom1 = new Promise(function(res, rej) {
  // This code is executed when the Promise is created
  // This code generally takes some time to complete
  console.log("1");
  // Simulate computation with a useless cycle
  for(let i=0; i<100000000; i++);
  // When the computation is over:
  // - if succesfull, return a result using the
  // first argument function. It is called res
  // in this example, but it is often named resolve
  // - if unsuccessful, return a reason explaining the
  // problem using the second argument function (rej/rejected)
  if(Math.random() < 0.5)
    res("value 1");
  else
    rej("reason 1");
  }
);
```

A simple example: Promise 1

```
prom1.then(
/* first argument is the function
   to be executed when the Promise
   completes succesfully */
function(risultato) {console.log("Risultato: " + risultato)},
/* second argument is the function
   to be executed when the Promise
   completes with an error (it is rejected) */
function (errore) {console.log("Errore: " + errore)}
);

console.log("2");
```

```
➜  promise node promise1.js
1
2
Errore: reason 1
➜  promise node promise1.js
1
2
Risultato: value 1
➜  promise
```

Using catch for errors: Promise 2

- Let's write it differently using **catch** (same behavior)

```
let prom1 = new Promise(function(res, rej) {
  // This code is executed as soon as the Promise is created
  // This code generally takes some time to complete
  console.log("1");
  // Simulate computation with a useless cycle
  for(let i=0; i<100000000; i++);
  // When the computation is over:
  // - if succesfull, return a result using the
  // first argument function. It is called res
  // in this example, but it is often named resolve
  // - if unsuccessful, return a reason explaining the
  // problem using the second argument function
  if(Math.random() < 0.5)
    res("value 1");
  else
    rej("reason 1");
  }
);
```

Using catch for errors: Promise 2

- Let's write it differently using **catch** (same behavior)

```
prom1.then(
  /* Handlers for the two cases can be better
  separated using then for the resolve and catch for reject */
  function(risultato) {console.log("Risultato: " + risultato)}
);


 prom1.catch(
   function(errore) {console.log("Errore: " + errore)}
 );

 console.log("2");
```

```
➜  promise node promise2.js
1
2
Risultato: value 1
```

Using catch for errors: Promise 2

Problem: if the Promise is rejected before the handler for the rejection is set the following error can be raised:

```
1
2
Errore: reason 1
node:internal/process/promises:246
triggerUncaughtException(err, true fromPromise );
^

[UnhandledPromiseRejection: This error originated
either by throwing inside of an async function without
a catch block, or by rejecting a promise which was not
handled with .catch(). The promise rejected with the
reason "reason 1".] {
code: 'ERR_UNHANDLED_REJECTION'
}
```

The handler is executed but an error is shown

Make it better: Promise 3

```
new Promise(function(res, rej) {
    console.log("1");
    // Simulate computation with a useless cycle
    for(let i=0; i<100000000; i++);
    if(Math.random() < 0.5) res("value 1");
    else rej("reason 1");
  })
.then(
  /* Then called on the Promise object */
  function(risultato) {console.log("Risultato: " + risultato)}
)
.catch(
  /* catch called on the Promise object */
  function(errore) {console.log("Errore: " + errore)}
);

console.log("2");
```

```
➜  promise node promise3.js
1
2
Errore: reason 1
```

async and arrow: Promise 4

- Let's now introduce some operations that take some time to complete using *setTimer()*
- Let's use arrow functions
- Let's also use *finally*

async and arrow: Promise 4

```
console.log("Creating the Promise and specifying the handlers");
new Promise(function(resolve, reject) {
 console.log("Starting a timer");
 setTimeout(
   ()=>{Math.random()<0.5 ?
        resolve("value 1"):reject("reason 1")}
   , 2000);
})
.then(risultato => {console.log("Risultato: " + risultato)})
.catch(errore => {console.log("Errore: " + errore)})
.finally(() => console.log("Finally"));
console.log("I'm here");
```

```
➜  promise node promise4.js
Creating the Promise and
specifying the handlers
Starting a timer
I'm here
... 2 seconds ...
Risultato: value 1
Finally
```

waiting and waiting: Promise 5

```
console.log("Creating the Promise and specifying the handlers");
new Promise(
  function(resolve, reject) {
    console.log("Starting a timer");
    setTimeout(()=>resolve("value 1"), 2000);
  }
)
.then(
  risultato => {
    console.log("In first then");
    new Promise(
      function (res, rej){
        setTimeout(()=>res(risultato + ", value 2"), 3000);
      }
    ).then(
      risultato2 => {console.log("All results: " + risultato2)})
  }
);
console.log("I'm here");
```

waiting and waiting: Promise 5

- In this example we didn't consider the case of errors to make it simpler

```
➜  promise node promise5.js
Creating the Promise and
specifying the handlers
Starting a timer
I'm here
... 2 seconds ...
In first then
... 3 seconds ...
All results: value 1, value 2
```

Same but different: Promise 6

```
console.log("Creating the Promise and specifying the handlers");
new Promise(
  function(resolve, reject) {
    console.log("Starting a timer");
    setTimeout(()=>resolve("value 1"), 2000);
  }
)
.then(
  risultato => {
    console.log("First promise resolved");
    return new Promise(
      function (res, rej){
        setTimeout(() => res(risultato + ", value 2"), 3000);
      }
    )
  }
)
.then(
  risultato2 => {
    console.log("Second promise resolved: " + risultato2)});
console.log("I'm here");
```

Same but different: Promise 6

- Second then is now moved out, as the first then returns a second Promise
- Less nested, same behavior

```
→  promise node promise6.js
Creating the Promise and specifying the
handlers
Starting a timer
I'm here
... 2 seconds ...
First promise resolved
... 3 seconds ...
Second promise resolved: value 1, value 2
```

Fetch: an example

```
<script>
function addrow(n, u, v, t) {
  const t1 = document.getElementById("t1");
  const li = document.createElement("li");
  const text = document.createTextNode(n + ", " + u + ", "
                                        + v + ", " + t);
  li.appendChild(text);
  t1.appendChild(li);
}
function loadfromservice(){
  fetch("https://api.coingecko.com/api/v3/exchange_rates")
  .then(res => res.json())
  .then(data => {
          const rat = data.rates;
          for (const key in rat) {
            addrow(rat[key].name, rat[key].unit,
                   rat[key].value, rat[key].type);
          }
        }
        );
}
</script>
```

Fetch: an example

```
<body>
<p>Rates downloaded from
https://api.coingecko.com/api/v3/exchange_rates using a
fetch</p>
<button onclick="loadfromservice()">Download</button>
<ul id="t1">
</ul>
</body>
```

fetch

```
fetch("https://api....")
  .then(res => res.json())
  .then(data => {/* actions ... */});
```

- fetch ruturns a **first** Promise object
- the HTTP request to the specified URL is sent
- handlers are registered

fetch

```
fetch("https://api....")
  .then(res => res.json())
  .then(data => {/* actions ... */});
```

- when the HTTP status and headers are received, the **first** promise is resolved, and the handler specified in the **first then** is executed
- the **res.json()** method returns a **second Promise**

fetch

```
fetch("https://api....")
  .then(res => res.json())
  .then(data => {/* actions ... */});
```

- when the body of the HTTP reply is received and converted to a JS object, the **second promise** is resolved and the handler specified in the **second then** is executed
- *data* is the JS object resulting from parsing the HTTP response body

Another example with fetch

```
function loadList(){
  fetch('list.txt')
    .then(response => {
      if (!response.ok) {
      // If an exception is thrown, the Promise returned by
      // this then is automatically rejected
        throw new Error("HTTP error! status: "
                        + response.status);
      } else {
        // The body of the response as text
        return response.text();
      }
    })
    .then(text => {
      let p = document.createElement('pre');
      p.innerText = text;
      document.body.appendChild(p);
    })
    .catch(e => {
       let pe = document.createElement('pre');
       pe.innerText = e.message;
       document.body.appendChild(pe);
    });
}
```

Another example with fetch

```
<button onclick="loadList()">
Load
</button>
```

async/await (ES 2017)

- **async** ad **await** have been introduced to make Promise-based code easier to write
  - syntax is different but concepts are still the same
- A function can be declared async:

  ```
  async function myfunction(){ ...}
  ```

- An async function always returns a Promise
- async functions (in general Promises) can be called using the await keyword:
  - to wait for the result when the promise is fulfilled
  - generating an exception in case of rejection

async 1

- An async function always returns a Promise
- If the returned value is not a Promise, a fulfilled Promise is returned

```
async function myfunction1() {
  return 42;
}

async function myfunction2(){
  return Promise.resolve(42);
}

myfunction1()
.then(r => {console.log(`The answer is ${r}`)});

myfunction2()
.then(r => {console.log(`The answer is ${r}`)})
```

async 2

- If the Promise is rejected, throw an exception

```
async function myfunction1(z) {
  if(z>0.5) return z;
  else throw "Wrong z value";
}

async function myfunction2(z) {
  if(z>0.5) return Promise.resolve(z)
  else return Promise.reject("Wrong z value");
}

const z = Math.random();

myfunction1(z)
  .then(r => {console.log(`The answer is ${r}`)})
  .catch(e => {console.log(`There was a problem: ${e}`)});

myfunction2(z)
  .then(r => {console.log(`The answer is ${r}`)})
  .catch(e => {console.log(`There was a problem: ${e}`)});
```

```
→  async-await node async2.js
The answer is 0.7867358165194982
The answer is 0.7867358165194982
→  async-await node async2.js
There was a problem: Wrong z value
There was a problem: Wrong z value
```

await 1

- **await** is used to call an async function and wait for the associated promise to settle
  - If fulfilled, the return value
  - If rejected, an exception
- await can only be used in async functions

```
async function myfunction() {
  console.log("In myfunction");
  const z = Math.random();
  if(z>0.5) return z;
  else throw "Wrong z value";
}

async function otherfunction(){
  try {
    console.log("Before calling myfunction");
    const r = await myfunction();
    console.log(`The answer is ${r}`);
  } catch (e) {
    console.log(`There was a problem: ${e}`);
  }
}

otherfunction()
.then(() => {console.log("Execution completed")})
```

```
→  async-await node async3.js
Before calling myfunction
In myfunction
There was a problem: Wrong z value
Exceution completed
→  async-await node async3.js
Before calling myfunction
In myfunction
The answer is 0.9848621870426133
Execution completed
```

await 2

- If the awaited promise is fulfilled: the value is returned
- if the awaited promise is rejected: an exception is thrown

```
→  async-await node async4.js
The result is 42
There was a problem: Not fulfilled
Execution completed
```

```
async function f1(){
  try {
    const r1 = await Promise.resolve(42);
    console.log(`The result is ${r1}`);
    const r2 = await Promise.reject("Not fulfilled");
    console.log("I'm here");
  } catch (e) {
    console.log(`There was a problem: ${e}`);
  }
}

f1()
.then(() => {console.log("Execution completed")});
```

Same examples with async/await

```
<script>
function addrow(n, u, v, t) {
  // same as before
  ...
}
async function loadfromservice(){
  console.log("loadfromservice called");
  const res = await fetch("https://api.coingecko.com/api/v3/exchange_rates")
  const data = await res.json();
  const rat = data.rates;
  return rat;
}
function myf() {
  loadfromservice()
  .then((rat) => {
    for (const key in rat) {
      addrow(rat[key].name, rat[key].unit, rat[key].value, rat[key].type);
    }})
  .catch((e) => alert("Problem: " + e.message));
  }
</script>
</head>
<body>
<p>Rates downloaded from https://api.coingecko.com/api/v3/exchange_rates
using a fetch</p>
<button onclick="myf()">Download</button>
<ul id="t1"></ul>
</body>
```

Same examples with async/await

```
async function loadList(){
  try {
    let response = await fetch('list.txt');
    if (!response.ok) {
      throw new Error("HTTP error! status: " + response.status);
    }
    let text = await response.text();
    let p = document.createElement('pre');
    p.innerText = text;
    document.body.appendChild(p);
  } catch(e) {
    let pe = document.createElement('pre');
    pe.innerText = e.message;
    document.body.appendChild(pe);
  }
}
```

The JS execution model

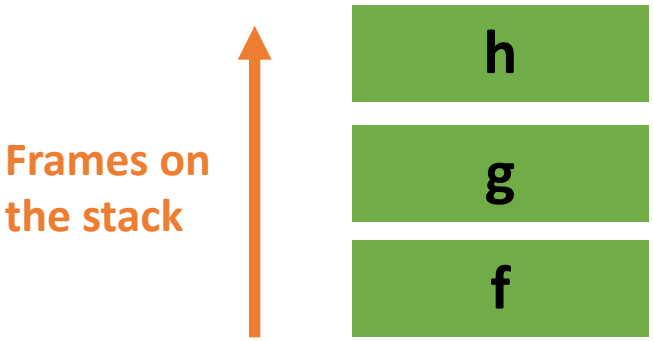
- The event loop

```
while (true) {
        queue.waitForATask();
        queue.removeAndProcess();
}
```

Tasks are inserted in the queue when the user clicks a button, a timer fires, etc

- A synchronous function and all the called functions are in a single task

```
function h(z) {
  return z*2;
}
function g(a) {
  a++;
  const b = h(a);
  return b;
}
function f() {
  let x = 10;
  x = g(x);
  console.log(x);
}
```



f() when button pressed

Same task. When all frames removed, a new task.

tasks

- If code (a function) takes too much to complete, other tasks cannot be processed.
  - The browser gui becomes unresponsive.

```
const timestamp1 = new Date().getTime(); // time value in ms

setTimeout(()=>{
  const timestamp2 = new Date().getTime();
  console.log(timestamp2-timestamp1); // ~1000 is the expected value
  }, 1000); // fires in 1 sec

for(;;){
  const timestamp3 = new Date().getTime();
  if(timestamp3-timestamp1 > 5000) break;
}
```

```
→  async-await node block.js
5001
```

The timer callback is inserted in the queue after 1 sec, but cannot run until current task is complete (5 secs)

tasks

- Always completed, also in case of 0 delay

```
console.log('Current task');
setTimeout(() => {
    console.log('The callback is started');
  }, 0);
console.log('Still current task');
```

```
➜  async-await node block2.js
Current task
Still current task
The callback is started
```

tasks

```
async function myfunction() {
  console.log('Body of myfunction');
  return "ABC";
}

console.log('Before calling myfunction');
myfunction()
  .then((h) => {
    console.log(`In then() handler... ${h}`);
  });
console.log('After calling myfunction');
```

- myfunction is started syncronously, in the current task
- result is delivered later, in other task

```
➜  async-await node
finaltasks.js
Before calling myfunction
Body of myfunction
After calling myfunction
In then() handler... ABC
```