ASYNCHRONOUS NETWORKING

Concurrency & JS

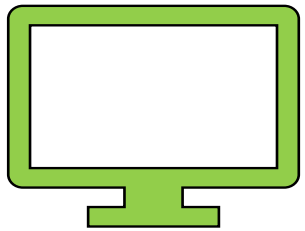
OVERVIEW

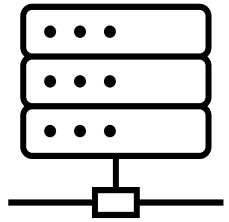
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

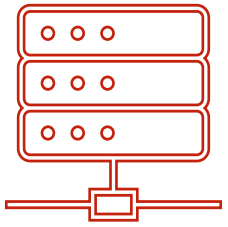
Servers





Latest News API

Super cool cats API



AJAX gives us the framework to design this solution.

But how can this be done in Javascript **specifically?**

Before answering that, need to define what **asynchronous** means

CONCURRENCY 101

Synchronous Programming

- Program executes top-down
- Guaranteed that previous instructions fully complete before executing the next ones
- Very simple to reason about

Asynchronous Programming

- Program has multiple flows of control (called “threads”)
- Threads can interleave or run at the same time
- Much harder to reason about

CONCURRENCY 101 (CONT.)

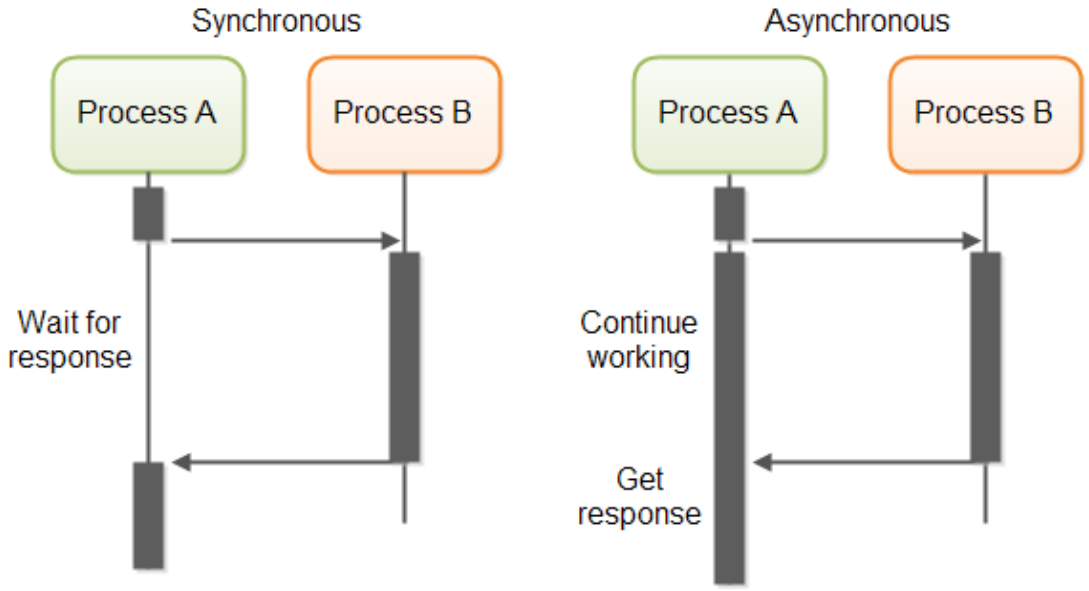


Image credit: Medium

- Synchronous
  - Process A executes first
  - Waits for Process B to finish
  - Then continues
- Asynchronous
  - Process executes first
  - Asks Process B to do its work
  - Whilst Process B is executing, Process A also continues
  - At some point in the future, Process A gets Process B's result

CONCURRENCY MODELS

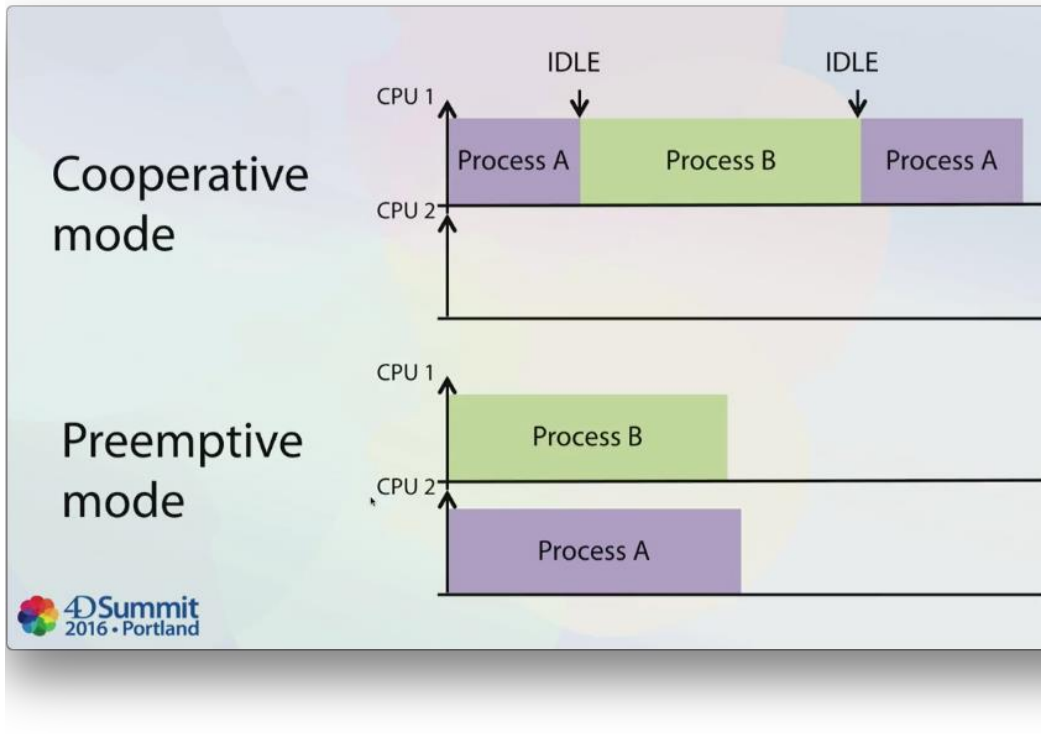
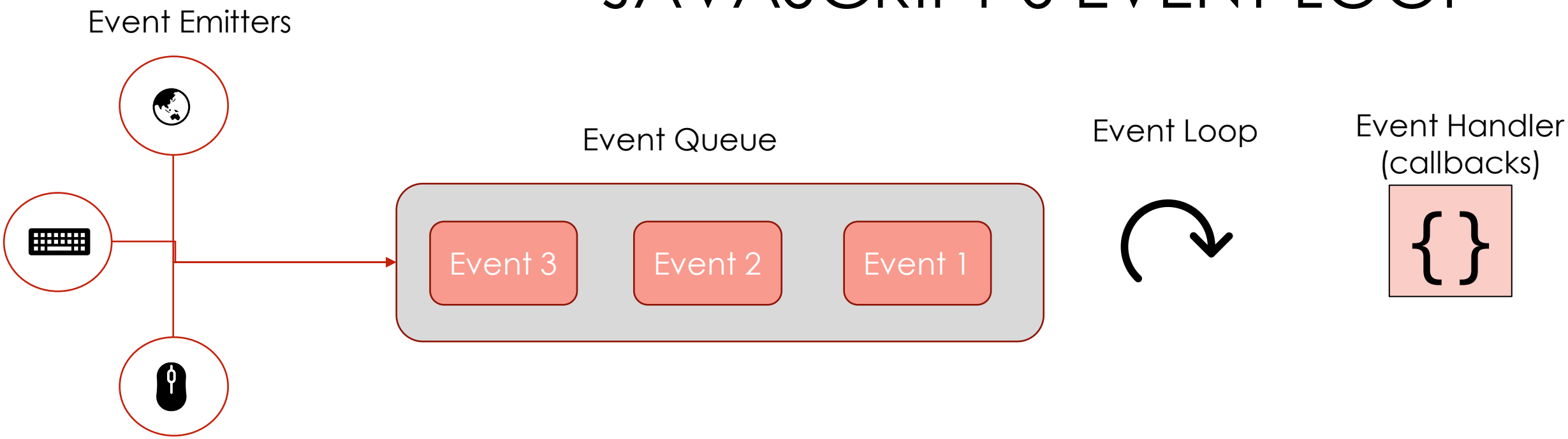


Image Credit: 4D

- Two primary "models" of concurrency:
  - **Pre-emptive multitasking**
  - **Co-operative multitasking**
- Pre-emptive Multitasking:
  - Code runs in parallel on different CPU cores
  - Really good for CPU-intensive workloads
  - Not important to us here
- Co-operative Multitasking:
  - Tasks that need to be done are queued up and executed in a loop
  - Really good for IO-intensive workloads
  - Get the benefits of asynchronous programming with the reasonability of synchronous programming
- Javascript's model is *Co-operative Multitasking*:
  - Tasks that need to be done are "events"
  - Event-handlers are scheduled to execute by the event loop
  - Each event handler runs **synchronously**

JAVASCRIPT'S EVENT LOOP



Each JS Engine generally follows this procedure:

1. Execute your JS script top-down, registering any handlers
2. Wait for events in a loop
3. If an event comes and there is a handler for it, execute the handler ***to completion***
4. Repeat (2) ad infinitum

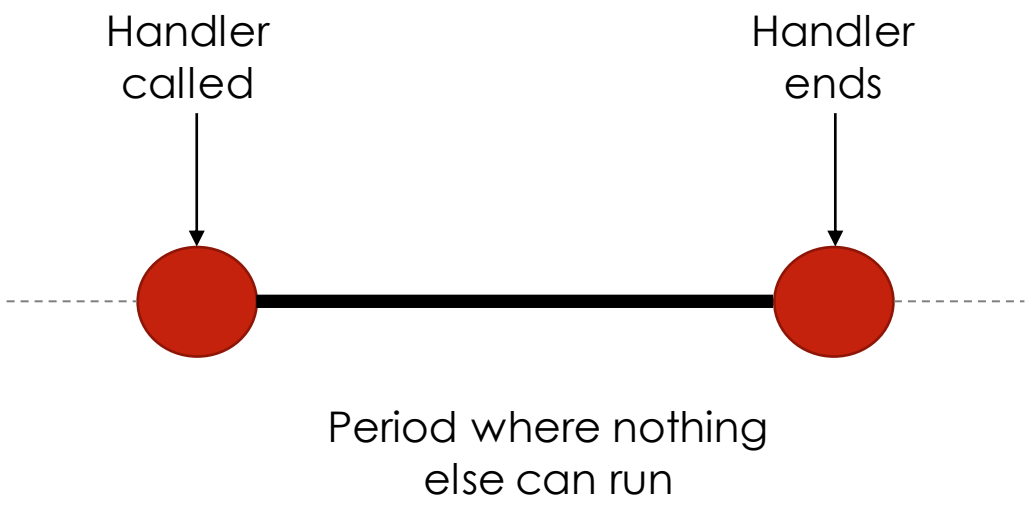
Important:
Browsers handle:

- Counting down timers
- Networking
- Adding to the event queue

EVENT LOOP DEMO

See examples/event-loop

EVENT LOOP CONSIDERATIONS



- The event loop runs on a **single** thread
  - Long-running event handlers block everything else
  - Makes the page unresponsive
- Tips on avoiding blocking:
  - Make sure all networking is done asynchronously
  - CPU-intensive calculations should be pushed to a backend server to process

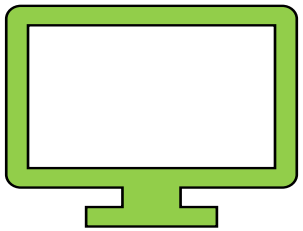
BLOCKING THE LOOP DEMO

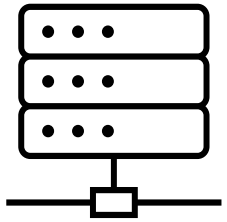
See examples/blocking-the-loop

PROGRESSING

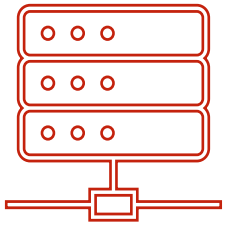
Client

Servers





Latest News API



Super cool cats API

We know now that JS has an event loop we can use to asynchronously execute tasks/events

But we still don't know what event to use to do networking!

The first of 3 approaches is the focus of next lecture

SUMMARY

- Today:
  - A brief introduction to concurrency
  - How concurrency is done in Javascript

- Coming Up Next:
  - Networking with `XMLHttpRequest()`

ASYNCHRONOUS NETWORKING

Networking with Promises & `fetch()`

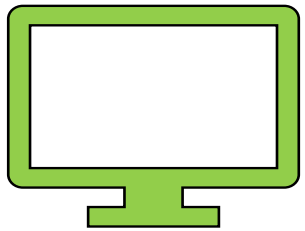
OVERVIEW

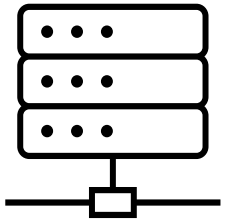
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

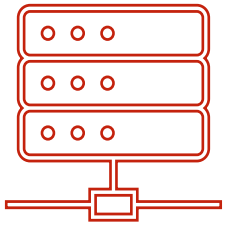
Servers





Latest News API

Super cool cats API



`XMLHttpRequest()` provides one way to do asynchronous fetching.

ES2015 introduced a new way via `fetch()` and **Promises**

Before talking about `fetch()`, what is a Promise?

PROMISE

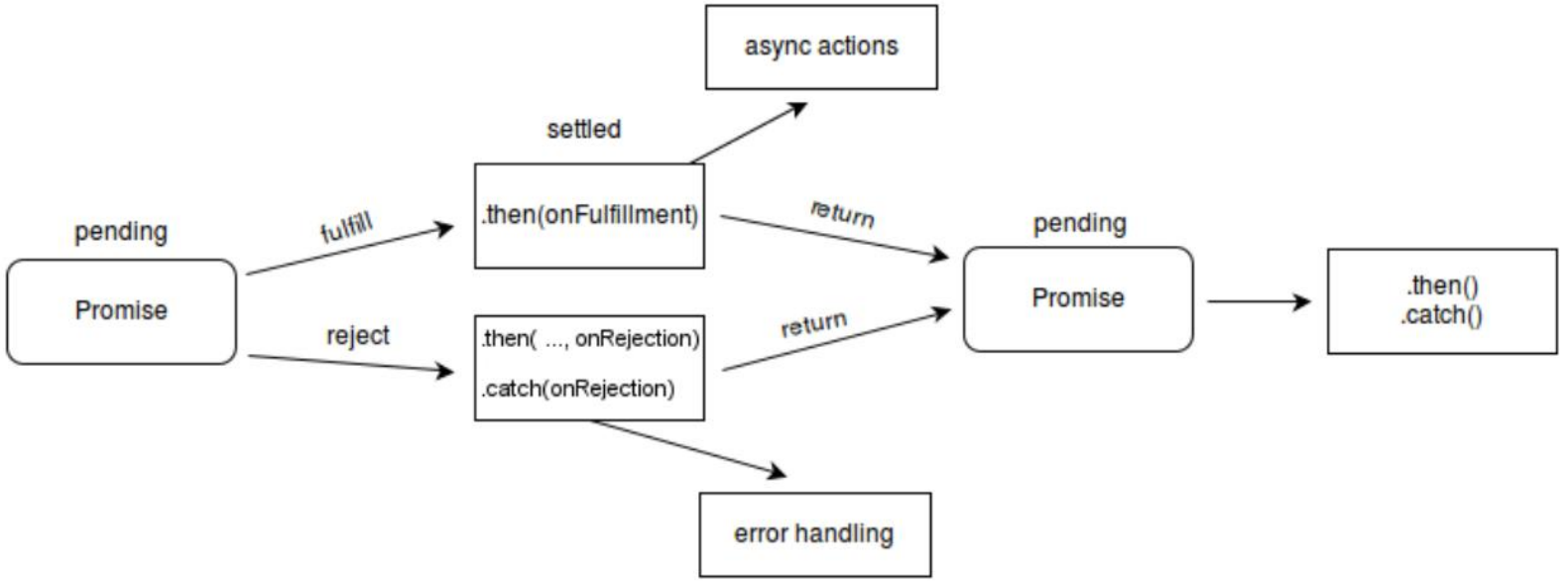


Image Credit: MDN

ES2015 Promises

- Proxy for a future value
- Evaluated asynchronously
- Support chaining, branching, error handling

Can be in one of 4 states:

- "Pending" – not evaluated yet
- "Fulfilled" – Successfully evaluated
- "Rejected" – Failed to evaluate
- "Settled" – Either rejected or fulfilled

Other useful features:

- Can be orchestrated (`Promise.all`, `Promise.race`, `etc.`)
- "Promise-like" objects can be used with Promises

BASIC API USAGE

```
1  // Creates a brand new Promise
2  const myPromise = new Promise( executor: (resolve, reject) => {
3      // if the action succeeds, call resolve() with the result
4      // or, if the action failed, call reject() with the reason
5  });
6
7  myPromise.then(
8      () => {
9          // this callback will be called if myPromise is fulfilled
10     },
11     () => {
12         // this specific callback will be called if myPromise is rejected
13     }
14 );
15
16 // In addition to giving a callback for errors in .then(), you can give a
17 // catch-all error handler as .catch()
18 myPromise.catch(
19     () => {
20         // handle the problem here
21     }
22 );
23
```

Constructor:

- Accepts a callback that takes `resolve()` and `reject()` functions
- Fulfillment = calling `resolve()`
- Rejection = calling `reject()`

.then:

- Most common way to chain promises.
- Executes the next action if the previous one fulfilled

.catch:

- Catch-all error handler for the chain above

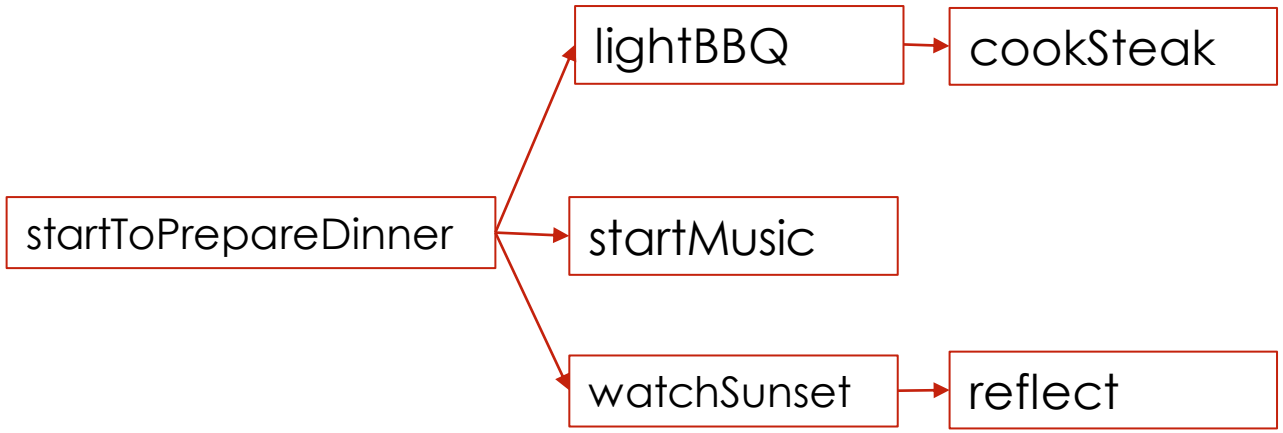
CHAINING

```
1  const dinnerPlans = startToPrepareDinner()
2      .then(lightBBQ)
3      .then(stokeFire)
4      .then(grillSteak)
5      .then(EAT)
6      .catch(eatNothing) // in case we burn ourselves
7
8  const restPlans = dinnerPlans
9      .then(watchYoutube)
10     .then(eveningStroll)
11     .catch(goToBed) // maybe the internet was out
```

- Nested execution of dependent operations
- Each `.then()` runs iff the previous promise fulfilled
- Any error/rejection that happens passed to the next-nearest `.catch()`
- After a `.catch()`, more operations can occur
- Every `.then()` returns a *new* promise which wraps the previous one.
  - Even if the given callback doesn't return a promise.
  - Stored as Russian dolls
  - But executed as a stack i.e. most-nested happens first

BRANCHING

```
1   const dinnerPlans = startToPrepareDinner();
2
3   dinnerPlans
4       .then(lightBBQ).then(cookSteak);
5
6   dinnerPlans
7       .then(startMusic);
8
9   dinnerPlans
10      .then(watchSunset).then(reflect);
```



- Multiple .then()'s on the same promise = branching
- When the parent promise is resolved, all .then()'s invoked in order
- Allows for complex control-flow on fulfillment

ERROR-HANDLING

```
const rejectedPromise = new Promise( executor: (resolve, reject) => reject( reason: "oops"))
    .then(() => "Never reached") Promise<string>
    .catch((error) => {
        // the rejection means we'll end up in here
        console.log(error);
    });

const exceptionPromise = new Promise( executor: (resolve, reject) => throw new Error("oops"))
    .then(() => "Never reached") Promise<string>
    .catch((err) => {
        // even though there wasn't an explicit rejection,
        // any exceptions cause an implicit rejection

        // rethrowing...
        throw err;
    }) Promise<string>
    .catch((err) => {
        // exceptions can also be rethrown and recaught in further down .catch() clauses
        // quite useful
        console.log(err);
    })
```

- Errors/Exceptions always cause rejections
- Explicit rejections done via calling `reject()`
- Any exceptions cause an implicit rejection
- `.catch()` clauses can handle errors or pass them to the next `.catch()` by rethrowing
- `.finally()` is also available that will run regardless of if an error occurred or not

ERROR-HANDLING GOTCHAS

```
1  function foo() {
2      try {
3          // explicit rejection
4          const baz = new Promise( executor: (res, reject) => {
5              reject( reason: "oops");
6          });
7      } catch (e) {
8          // do we enter here?
9          console.log(e);
10     }
11 }
```

- Promises always asynchronous
- Current function context *always* completes before a Promise is settled
- This means Promises don't work with `try/catch` like on the left!
- Good idea to add an event listener to the window to handle the `unhandledrejection` event

PROMISE ORCHESTRATION

```
1   Promise.all([...])
2
3   Promise.allSettled([...])
4
5   Promise.any([...])
6
7   Promise.race([...])
8
9   Promise.reject(val)
10
11  Promise.resolve(val)
12
```

- The `Promise` class has some utilities for easy orchestration
- `Promise.all()`: returns a promise that resolves iff all of the promises passed to it resolve
- `Promise.allSettled()`: returns a promise that resolves once all of the promises passed to it are resolved
- `Promise.any()`: returns a promise that resolves if at least one of the promises passed to it resolves
- `Promise.race()`: returns a promise which resolves as soon as one of the promises passed to it resolves
- `Promise.reject()`: immediately return a rejected promise with a value
- `Promise.resolve()`: immediately return a resolved promise with a value

PROMISE-LIKE OBJECTS (THENABLES)

```
1   class customThenable {
2       then(onFulfill, onReject) {
3           console.log("inside a thenable!");
4           onFulfill();
5       }
6   }
7
8   Promise.resolve(new customThenable()).then(
9       () => console.log("used a custom thenable")
10  );
11
12  // output:
13  // inside a custom thenable!
14  // used a custom thenable
```

- Any object or class with a `then()` considered "promise-like" or a "**thenable**".
- Can be used with Promise chaining
- Useful for fine-grained control over how chaining works for custom types

THE FETCH API

```
1  // only the URL is required
2  fetch("http://example.com/movies.json", {
3      method: "POST", // this object is optional
4  })
5      // return the body as JSON
6      .then(res => res.json())
7
8      // finally access the JSON
9      .then(js => console.log(js));
10
```

- Promise-based native JS API to download remote resources
- Resolves if a Response is received, even if the HTTP status code is not 200
- Rejects if there is any network error
- Access the result of the request via chaining `then()`s
- Optional 2nd argument to `fetch()` can control the Request options e.g.
  - Authentication
  - CORS
  - HTTP method

PROMISE + FETCH DEMO

See examples/promise-fetch

FETCH LIMITATIONS

XMLHttpRequest

- Works even on very old browsers
- Gives large download progress
- Easily cancelled

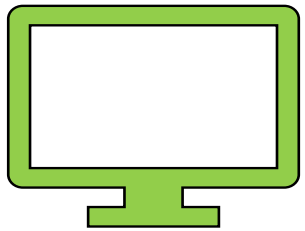
Fetch

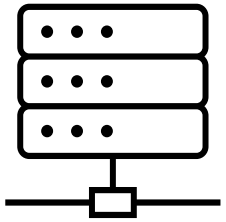
- Only works on browsers with Promise support (less of a problem nowadays)
- Promises not easily cancellable
- More complex functionality implemented via the Streams API which has a non-trivial learning curve

MOVING FORWARD

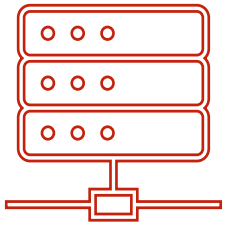
Client

Servers





Latest News API



Super cool cats API

`fetch()` is a very flexible and nice API to work with.

However, chaining and callbacks still removes us from writing code like we are used to.

Can we do even better?

SUMMARY

- Today:
  - Promises
  - Using Promises with `fetch()`

- Coming Up Next:
  - Networking with `async/await & fetch()`

ASYNCHRONOUS NETWORKING

Networking with `async/await & fetch()`

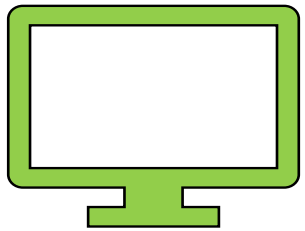
OVERVIEW

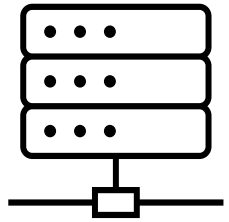
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

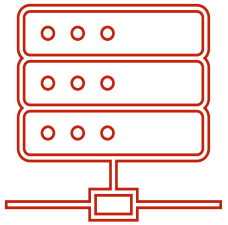
Servers





Latest News API

Super cool cats API



We've built up a nice asynchronous framework with callbacks, then Promises.

Both have a similar problem: code written does not look like the functions we are used to.

That's where `async/await` come in

ASYNC/AWAIT

```
1   // Promise version
2   fetch( input: "http://example.com/movies.json",  init: {
3       method: "POST", // this object is optional
4   }) Promise<Response>
5       .then(res => res.json()) Promise<any>
6       .then(js => console.log(js));
7
8
9   // async/await version
10  async function getMovies() {
11      const url = "http://example.com/movies.json";
12
13      const res = await fetch(url);
14      const js = await res.json();
15
16      console.log(js);
17  }
18
```

- Keywords introduced in ES2017
  - Supported by virtually all browsers
  - So-called "coroutines" i.e. resumable functions
- Built on top of Promises
- `async` functions always return a Promise
- `await` expressions always unwrap a resolved Promise
- Rejections are thrown as exceptions
- Program asynchronous code in a familiar and intuitive way again

UNDER THE HOOD
AN INTUITIVE UNDERSTANDING

```
async function foo(v) {
    const w = await v;
    return w;
}
```

```
resumable function foo(v) {
  implicit_promise = createPromise();
  // 1. Wrap v in a promise.
  promise = Promise.resolve(v)
  // 2. Attach handlers for resuming
  // foo.
  promise.then(() => {
    resume(foo);
  }, (err) => {
    throw(err);
  });
  // 3. Suspend foo and return
  // implicit_promise
  w = suspend(foo, implicit_promise);
  implicit_promise.resolve(w);
}
```

Image Credit: Zain Afzal

N.B. It is *as-if* this is what happens. This is not real code.

- Non-promise values/thenables are awaitable like promises
- `v` always wrapped into a Promise
- This Promise then chained with injected logic to resume the function at the point where `await` was used
- JS runtime suspends execution of the function whilst returning a Promise to the caller that represents the suspended function's eventual result
- The returned Promise resolves once the suspended function is resumed and reaches the end of its body

ERROR-HANDLING REVISITED

```
1   // Promise version
2   function foo() {
3       // should reject with an unreachable domain error
4       const baz = fetch( input: "http://oops");
5
6       baz.catch(err => console.log(err));
7   }
8
9   // async/await version
10  async function Foo() {
11      // should reject with an unreachable domain error
12      const baz = fetch( input: "http://oops");
13
14      try {
15          await baz;
16      } catch (err) {
17          console.log(err);
18      }
19  }
20
```

- `async/await` functions always maintain their function scope
- Errors and rejections handled by `try/catch` again
- Careful:
  - `await`'ing a Promise will cause an exception
  - `async` functions called synchronously will still return a pending Promise

ASYNC/AWAIT FETCH DEMO

See examples/asyncawait-fetch

PROMISES OR ASYNC/AWAIT?

Promise

- Can pass around and attach different handlers
- Maintains a stack trace so takes more memory
- Supported more on older browsers

`async/await`

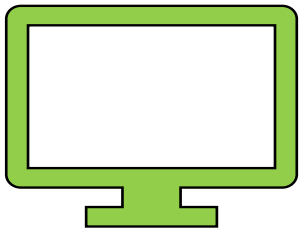
- Easier to read
- Potentially harder to debug
  - Debuggers will jump around source code
- More performant in some cases
- More familiar

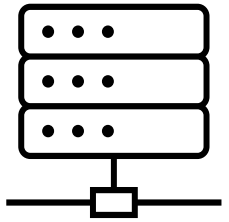
Ultimately, which to use comes down to your specific needs / environment / targets, etc.

FINALLY!

Client

Servers

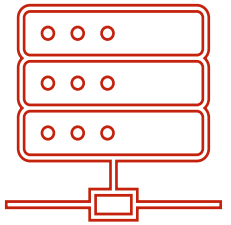




Latest News API

We've finally built enough understanding to make an educated decision about how to do asynchronous networking in JS!

Super cool cats API



SUMMARY

- Today:
  - `async/await`
  - Using `async/await` with `fetch()`
- Asynchronous networking with Javascript accomplished by:
  - `XMLHttpRequest` (legacy)
  - `fetch()` and Promises
  - `fetch()` and `async/await`
- AJAX technologies continue to evolve
- But the fundamentals rarely change

ASYNCHRONOUS NETWORKING

Client-Server Model + AJAX

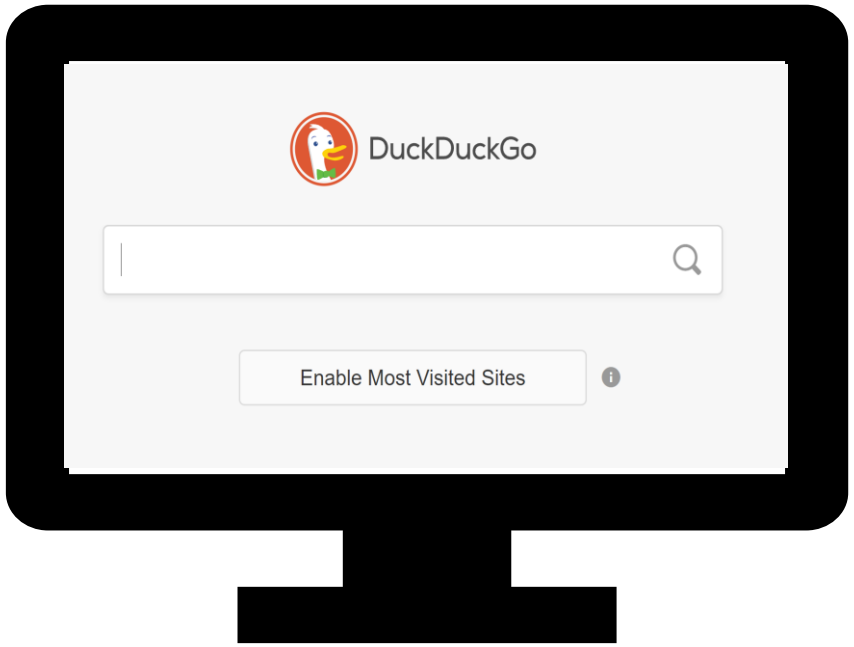
OVERVIEW

- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

CLIENT-SERVER INTERACTION

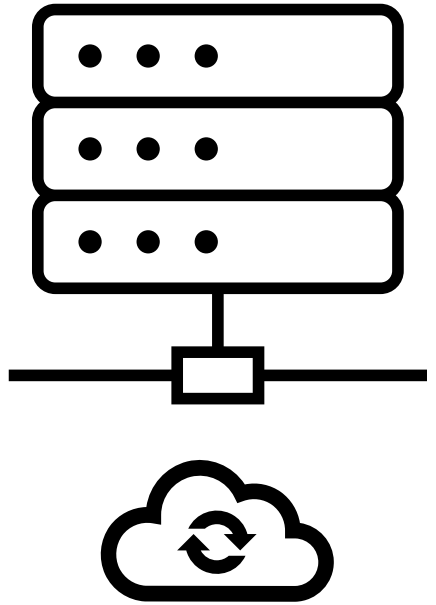
Client

Smart phones, laptops, etc.



Server

Racks of machines in datacentres



CLIENT-SERVER INTERACTION

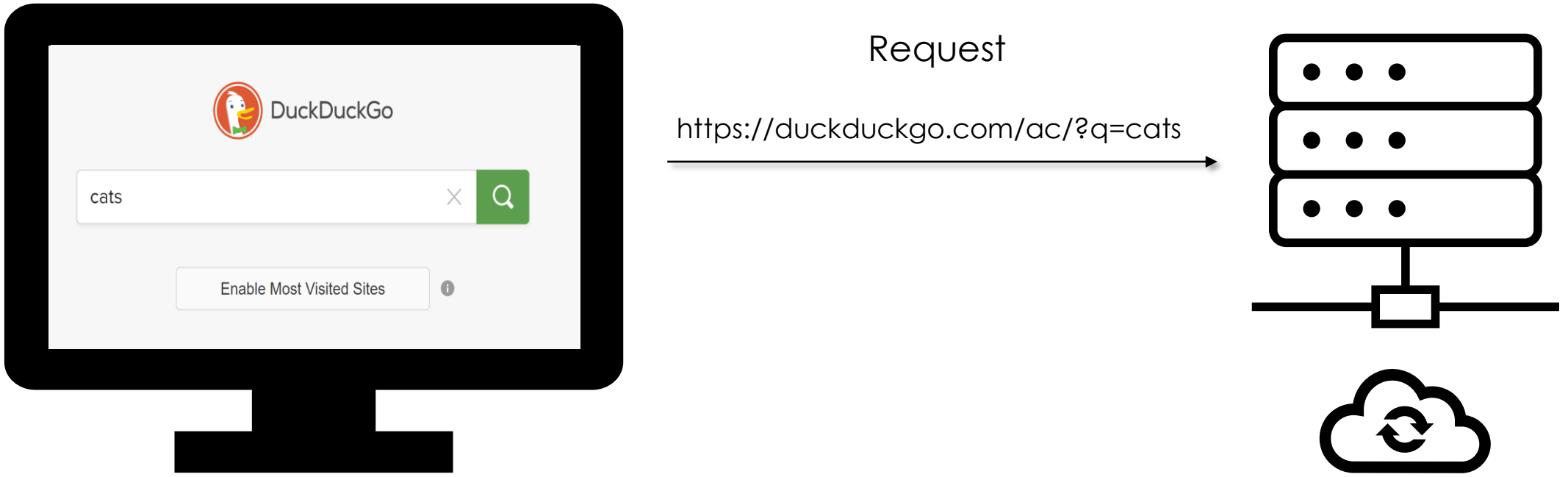
Client

Makes a request to a server

e.g. Search for websites that contain ‘cats’

Server

Waiting for requests



CLIENT-SERVER INTERACTION

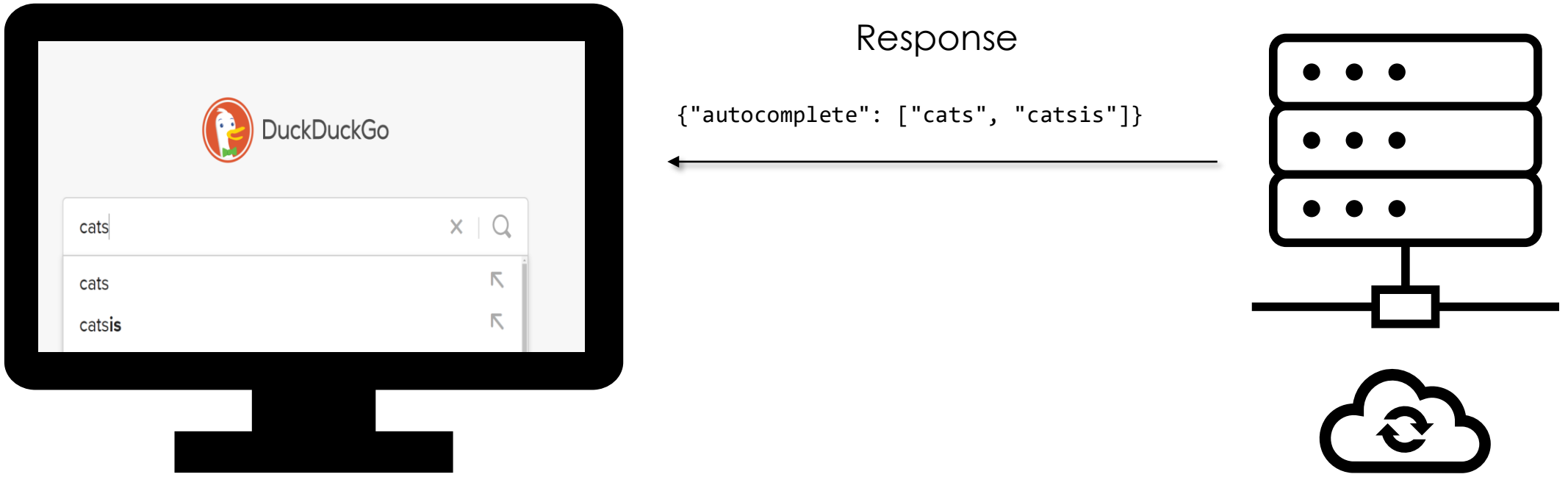
Client

Receives payload as a server response

Renders the HTML & CSS, executes the JS

Server

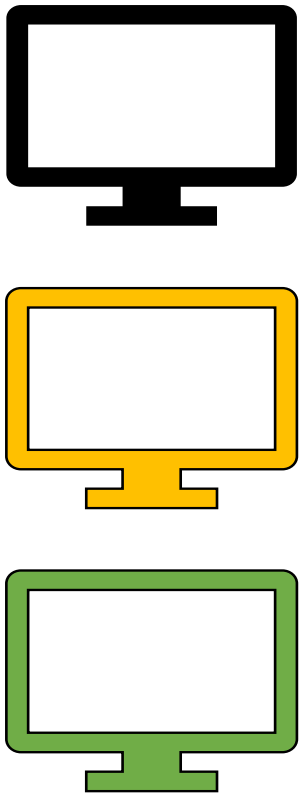
Validates arguments, executes API, returns data



CLIENT-SERVER REALITY

Webapp Client**s**

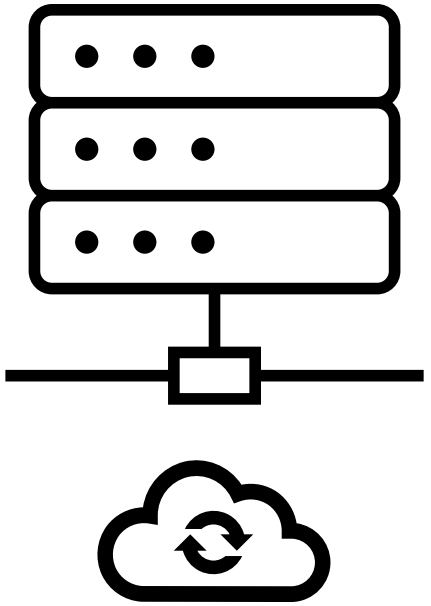
Server



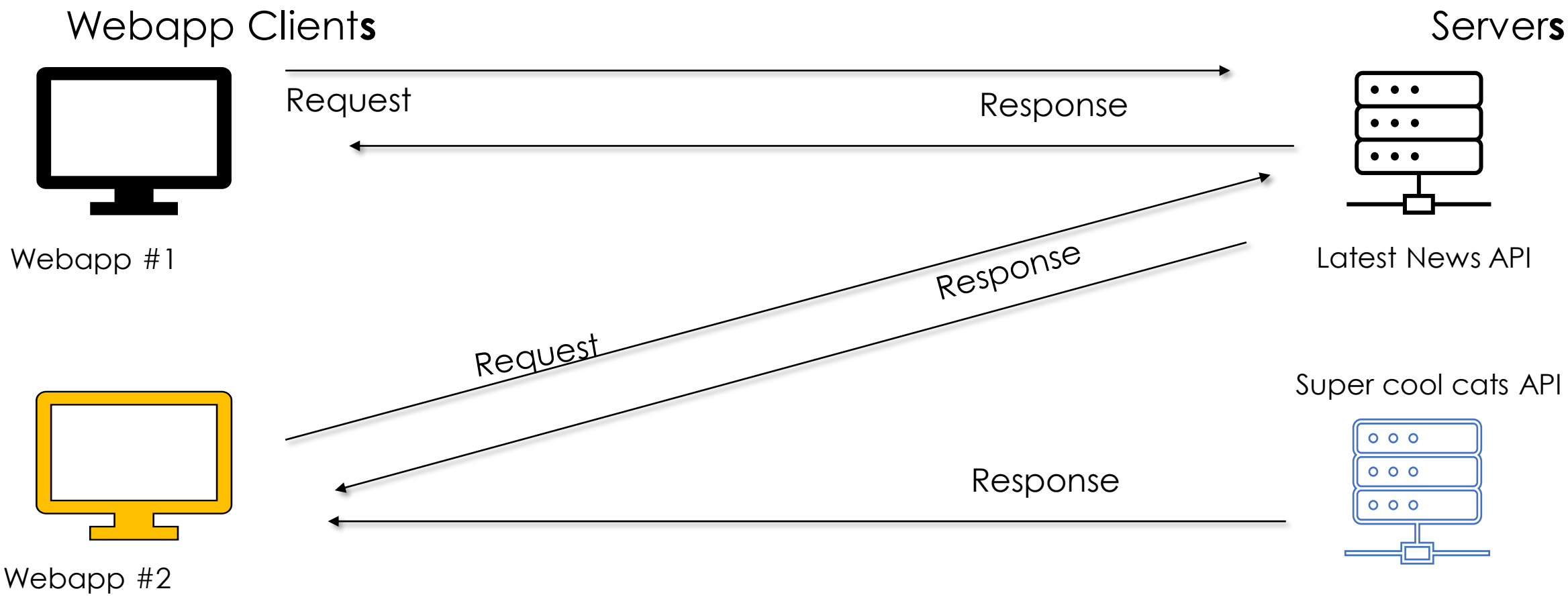
Response

Response

Response



CLIENT-SERVER REALITY²



LATENCY & THROUGHPUT: THE PROBLEM

Client-side

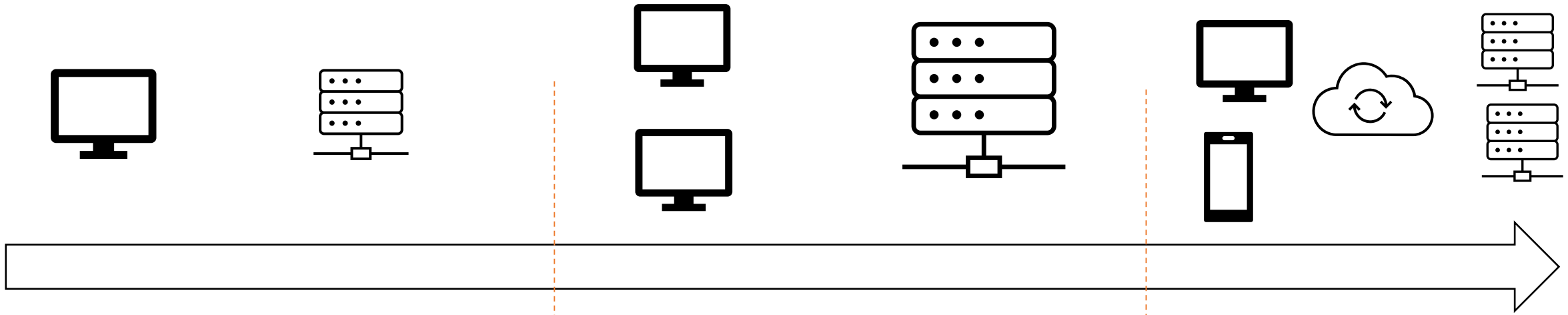
- Interactivity + responsiveness = low **latency** is most important
- Page loads ideally fast + once
- Losing your place in a page due to reload is bad UX

Server-side

- Need to handle at least thousands of requests
- Resources (time, power, # of servers) are expensive
- Maximising **throughput** is most important

EVOLUTION OF THE CLIENT-SERVER

A BRIEF HISTORY



1990s & early 2000s

- Clients = majority desktop web browsers
- On-site servers
- Era of "static webpages" i.e. **server-side rendering (SSR)**
- Scale by upgrading server

Mid-2000s – Early 2010s

- Majority of clients still desktop browsers
- JS adding much more interactivity client-side
- Still scale server vertically

~2010s – Present

- iPhone ushers in the smart phone era
- Clients now IoT and mobile
- Almost all rendering is client-side
- Cloud server-hosting is dominant
- Scale servers horizontally

LATENCY & THROUGHPUT: SOLUTIONS?

Client-side

- Leave presentation strictly to the frontend
- Initial page load contains minimal necessary data
- Fetch data **asynchronously** as needed ("lazy-loading")

Server-side

- Scale to handling millions of requests per second by **asynchronous** event processing
- Whilst waiting for I/O, serve another request => never idle
- Increase load-balancing by adding more servers

DEFINING AJAX

Asynchronous

Javascript

And

XML*

- AJAX = set of techniques to create asynchronous webapps.
- Not any one particular technology
- Allows offline apps, smooth UI/UX, modular frontend & backend

*JSON is used more nowadays rather than XML. So, Ajaj?

Doesn't quite roll off the tongue as well.

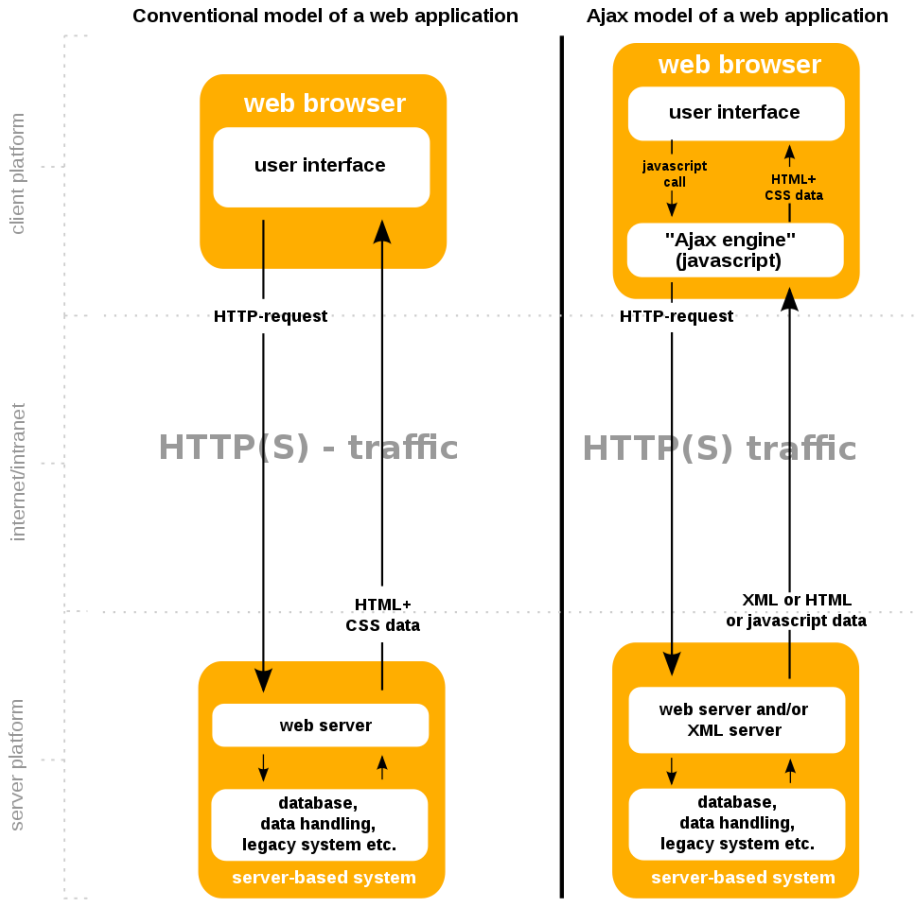


Image Credit: [Wikipedia]

SUMMARY

- Today:
  - Client-Server Model
  - AJAX

- Coming Up Next:
  - Concurrency & JS

ASYNCHRONOUS NETWORKING

Networking with Promises & `fetch()`

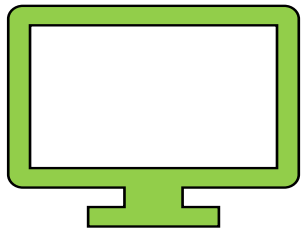
OVERVIEW

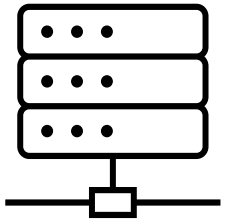
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

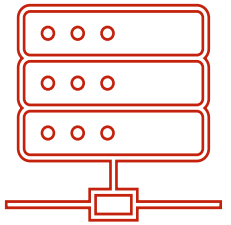
Servers





Latest News API

Super cool cats API



`XMLHttpRequest()` provides one way to do asynchronous fetching.

ES2015 introduced a new way via `fetch()` and **Promises**

Before talking about `fetch()`, what is a Promise?

PROMISE

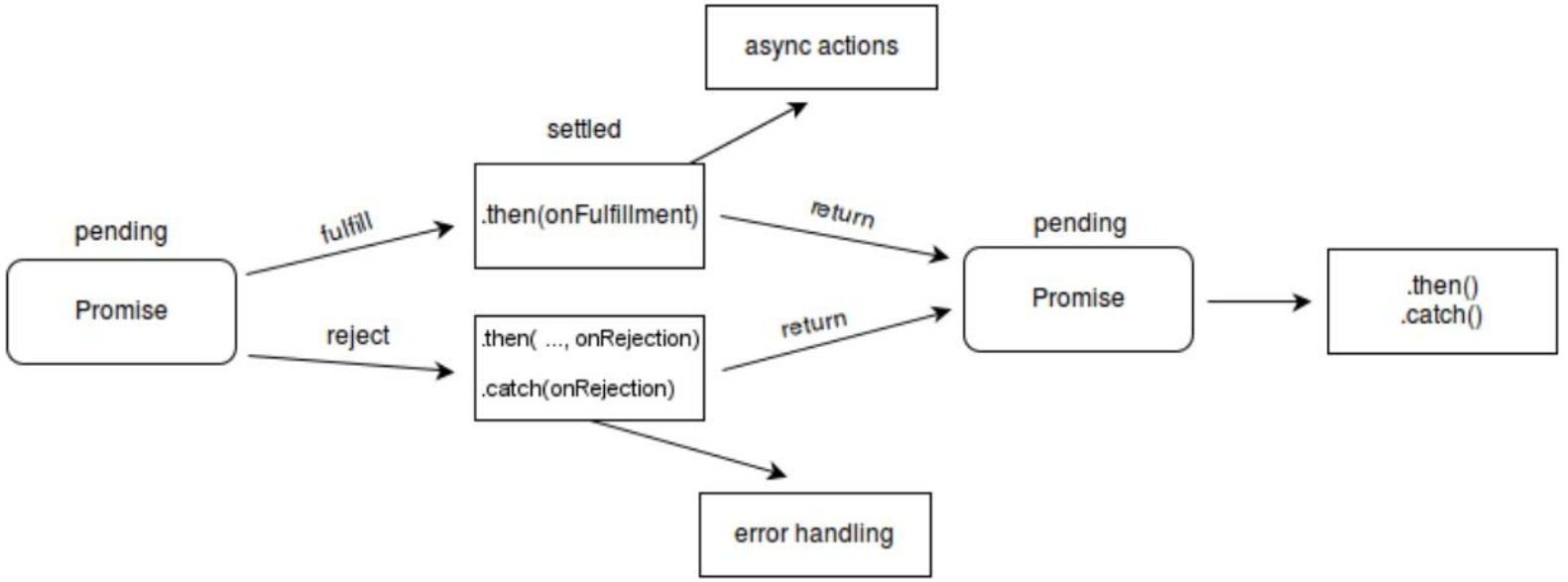


Image Credit: MDN

ES2015 Promises

- Proxy for a future value
- Evaluated asynchronously
- Support chaining, branching, error handling

Can be in one of 4 states:

- "Pending" – not evaluated yet
- "Fulfilled" – Successfully evaluated
- "Rejected" – Failed to evaluate
- "Settled" – Either rejected or fulfilled

Other useful features:

- Can be orchestrated (`Promise.all`, `Promise.race`, `etc.`)
- "Promise-like" objects can be used with Promises

BASIC API USAGE

```
1  // Creates a brand new Promise
2  const myPromise = new Promise( executor: (resolve, reject) => {
3      // if the action succeeds, call resolve() with the result
4      // or, if the action failed, call reject() with the reason
5  });
6
7  myPromise.then(
8      () => {
9          // this callback will be called if myPromise is fulfilled
10     },
11     () => {
12         // this specific callback will be called if myPromise is rejected
13     }
14 );
15
16 // In addition to giving a callback for errors in .then(), you can give a
17 // catch-all error handler as .catch()
18 myPromise.catch(
19     () => {
20         // handle the problem here
21     }
22 );
23
```

Constructor:

- Accepts a callback that takes `resolve()` and `reject()` functions
- Fulfillment = calling `resolve()`
- Rejection = calling `reject()`

.then:

- Most common way to chain promises.
- Executes the next action if the previous one fulfilled

.catch:

- Catch-all error handler for the chain above

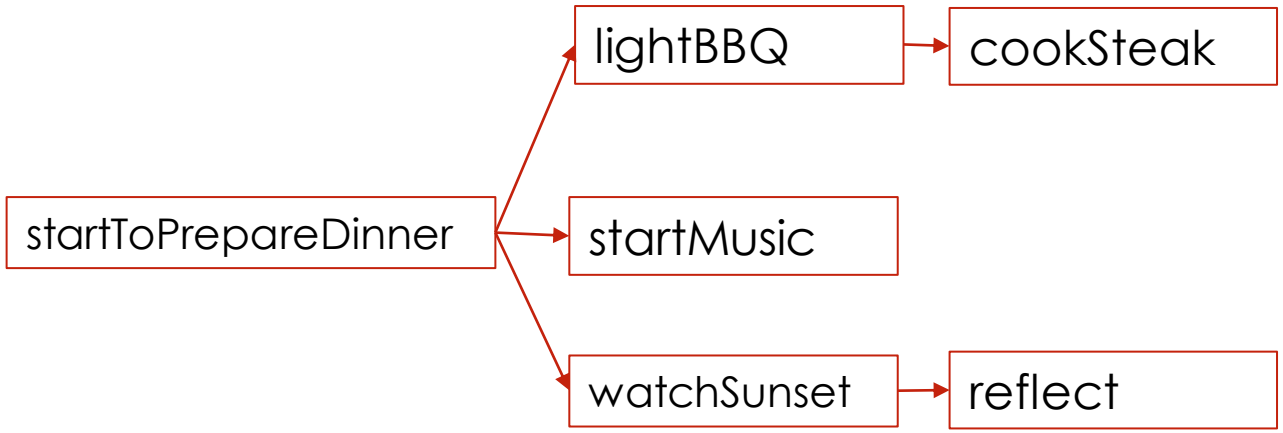
CHAINING

```
1  const dinnerPlans = startToPrepareDinner()
2      .then(lightBBQ)
3      .then(stokeFire)
4      .then(grillSteak)
5      .then(EAT)
6      .catch(eatNothing) // in case we burn ourselves
7
8  const restPlans = dinnerPlans
9      .then(watchYoutube)
10     .then(eveningStroll)
11     .catch(goToBed) // maybe the internet was out
```

- Nested execution of dependent operations
- Each `.then()` runs iff the previous promise fulfilled
- Any error/rejection that happens passed to the next-nearest `.catch()`
- After a `.catch()`, more operations can occur
- Every `.then()` returns a *new* promise which wraps the previous one.
  - Even if the given callback doesn't return a promise.
  - Stored as Russian dolls
  - But executed as a stack i.e. most-nested happens first

BRANCHING

```
1   const dinnerPlans = startToPrepareDinner();
2
3   dinnerPlans
4       .then(lightBBQ).then(cookSteak);
5
6   dinnerPlans
7       .then(startMusic);
8
9   dinnerPlans
10      .then(watchSunset).then(reflect);
```



- Multiple .then()'s on the same promise = branching
- When the parent promise is resolved, all .then()'s invoked in order
- Allows for complex control-flow on fulfillment

ERROR-HANDLING

```
const rejectedPromise = new Promise( executor: (resolve, reject) => reject( reason: "oops"))
    .then(() => "Never reached") Promise<string>
    .catch((error) => {
        // the rejection means we'll end up in here
        console.log(error);
    });

const exceptionPromise = new Promise( executor: (resolve, reject) => throw new Error("oops"))
    .then(() => "Never reached") Promise<string>
    .catch((err) => {
        // even though there wasn't an explicit rejection,
        // any exceptions cause an implicit rejection

        // rethrowing...
        throw err;
    }) Promise<string>
    .catch((err) => {
        // exceptions can also be rethrown and recaught in further down .catch() clauses
        // quite useful
        console.log(err);
    })
```

- Errors/Exceptions always cause rejections
- Explicit rejections done via calling `reject()`
- Any exceptions cause an implicit rejection
- `.catch()` clauses can handle errors or pass them to the next `.catch()` by rethrowing
- `.finally()` is also available that will run regardless of if an error occurred or not

ERROR-HANDLING GOTCHAS

```
1  function foo() {
2      try {
3          // explicit rejection
4          const baz = new Promise( executor: (res, reject) => {
5              reject( reason: "oops");
6          });
7      } catch (e) {
8          // do we enter here?
9          console.log(e);
10     }
11 }
```

- Promises always asynchronous
- Current function context *always* completes before a Promise is settled
- This means Promises don't work with `try/catch` like on the left!
- Good idea to add an event listener to the window to handle the `unhandledrejection` event

PROMISE ORCHESTRATION

```
1   Promise.all([...])
2
3   Promise.allSettled([...])
4
5   Promise.any([...])
6
7   Promise.race([...])
8
9   Promise.reject(val)
10
11  Promise.resolve(val)
12
```

- The `Promise` class has some utilities for easy orchestration
- `Promise.all()`: returns a promise that resolves iff all of the promises passed to it resolve
- `Promise.allSettled()`: returns a promise that resolves once all of the promises passed to it are resolved
- `Promise.any()`: returns a promise that resolves if at least one of the promises passed to it resolves
- `Promise.race()`: returns a promise which resolves as soon as one of the promises passed to it resolves
- `Promise.reject()`: immediately return a rejected promise with a value
- `Promise.resolve()`: immediately return a resolved promise with a value

PROMISE-LIKE OBJECTS (THENABLES)

```
1   class customThenable {
2       then(onFulfill, onReject) {
3           console.log("inside a thenable!");
4           onFulfill();
5       }
6   }
7
8   Promise.resolve(new customThenable()).then(
9       () => console.log("used a custom thenable")
10  );
11
12  // output:
13  // inside a custom thenable!
14  // used a custom thenable
```

- Any object or class with a `then()` considered "promise-like" or a "**thenable**".
- Can be used with Promise chaining
- Useful for fine-grained control over how chaining works for custom types

THE FETCH API

```
1  // only the URL is required
2  fetch("http://example.com/movies.json", {
3      method: "POST", // this object is optional
4  })
5      // return the body as JSON
6      .then(res => res.json())
7
8      // finally access the JSON
9      .then(js => console.log(js));
10
```

- Promise-based native JS API to download remote resources
- Resolves if a Response is received, even if the HTTP status code is not 200
- Rejects if there is any network error
- Access the result of the request via chaining `then()`s
- Optional 2nd argument to `fetch()` can control the Request options e.g.
  - Authentication
  - CORS
  - HTTP method

PROMISE + FETCH DEMO

See examples/promise-fetch

FETCH LIMITATIONS

XMLHttpRequest

- Works even on very old browsers
- Gives large download progress
- Easily cancelled

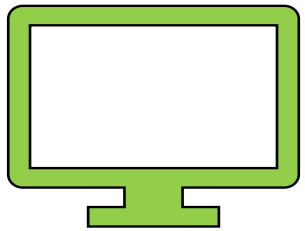
Fetch

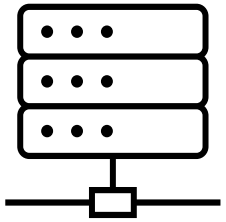
- Only works on browsers with Promise support (less of a problem nowadays)
- Promises not easily cancellable
- More complex functionality implemented via the Streams API which has a non-trivial learning curve

MOVING FORWARD

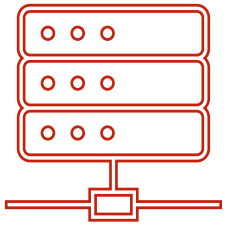
Client

Servers





Latest News API



Super cool cats API

`fetch()` is a very flexible and nice API to work with.

However, chaining and callbacks still removes us from writing code like we are used to.

Can we do even better?

SUMMARY

- Today:
  - Promises
  - Using Promises with `fetch()`

- Coming Up Next:
  - Networking with `async/await & fetch()`