ASYNCHRONOUS NETWORKING

Networking with Promises & `fetch()`

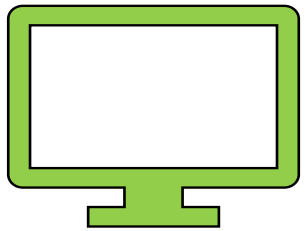
OVERVIEW

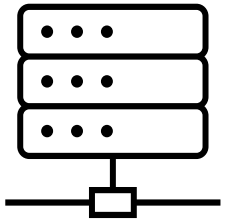
- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

Client

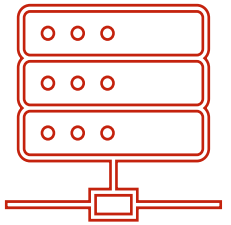
Servers





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

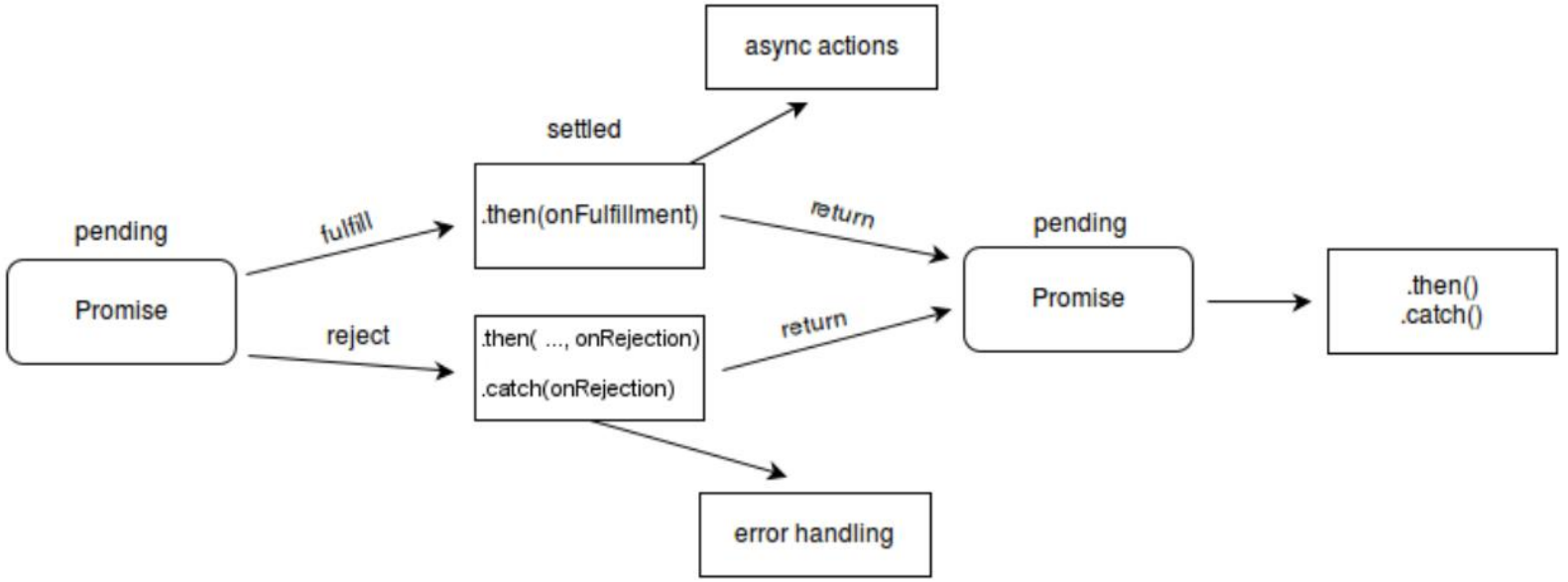


Image Credit: MDN

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

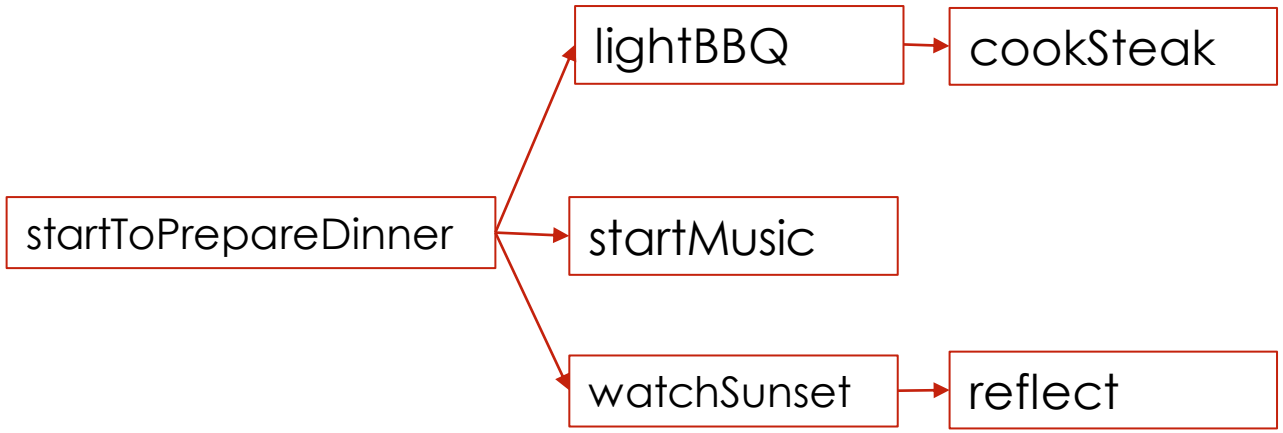
CHAINING

```
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

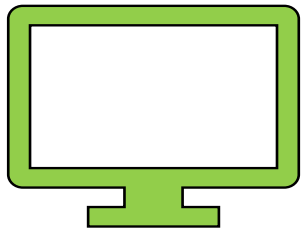
Fetch

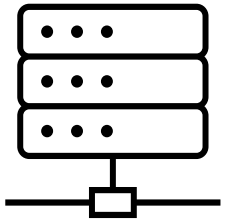
- Only works on browsers with Promise support (less of a problem nowadays)
- Promises not easily cancellable
- More complex functionality implemented via the Streams API which has a non-trivial learning curve

MOVING FORWARD

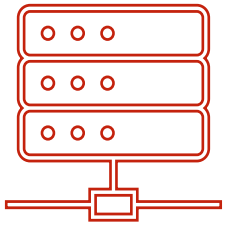
Client

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()`