

Практическая работа

Курс: Разработка интерфейса на JavaScript

Дисциплина: Основы JavaScript

Тема занятия №32: Асинхронность. Промисы. Async-Await. Fetch

Асинхронность

Проверим, например результат синхронных операций.
Выведем обычный консоль лог. А затем второй.

```
console.log('Start')  
  
console.log('Start 2')
```

Они выполнятся последовательно.

Но мы можем сделать что-то асинхронно. Например, используя браузерную функцию Set Timeout. Она именно браузерная (пришла с браузерным API и доступна у объекта window), не JS.

```
window.setTimeout(function() {  
  console.log('Inside timeout, after 2000 seconds')  
}, 2000)
```

Если мы запишем после этого ещё один консоль лог.

```
console.log('Start')  
  
console.log('Start 2')  
  
window.setTimeout(function() {  
  console.log('Inside timeout, after 2000 seconds')  
}, 2000)  
  
console.log('End')
```

Как это работает?

Видит первую строчку. Закинул её в стек и выполнил. Следующая строчка пустая. Третья строчка снова консоль лог. Закинул в стек и выполнил. Потом

пустая строка. А дальше видит, что там set Timeout он её регистрирует где-то там у себя (в WEB API) и начинает ждать. Дальше программа не тормозит она выполняется дальше. Пустая строка. И потом end закидывает в стек и выполняет.

То есть то, что синхронно выполнится сразу. А асинхронно выполнится позже и выполнения скрипта не блокируется.

Но почему не блокируется? Как работает? Почему в нужный момент времени вызывается?

Здесь кроется важный концепт, который называется Event Loop.

Чтобы его понять есть специальный сервис для наглядности.

<http://latentflip.com/loupe/>

```
console.log('Start')

console.log('Start 2')

setTimeout(function() {
  console.log('Inside settimeot')
}, 5000)

console.log('End')
```

Сначала всё заносится в стек. Как только видит Set timeout то регистрирует её в Web API. Она отработывает определённое время, которые вы дали для Set timeout. Потом заносит в Callback Queue. Где работает цикл (Event Loop), который делает итерацию по этому массиву из callbacks и снова закидывает их обратно в стек.

Теперь попробуем вызвать функцию для обработчика событий.

```
$.on('button', 'click', function() {
  console.log('Button clicked')
})

console.log('Start')

// console.log('Start 2')

// setTimeout(function() {
//   console.log('Inside settimeot')
// }, 10000)

console.log('End')
```

Callback Queue называется также очередью. Например, если мы много раз нажмём на кнопку. Он всё занесёт в этот массив. И ему нужно время чтобы всё выполнить это. (Тут для наглядности замедлили всё).

Есть один концепт, который часто спрашивают на собеседованиях. Это Settimeout 0

```
console.log('Start')

setTimeout(function() {
  console.log('Inside settimeot')
}, 0)

console.log('End')
```

По сути, время не задано. И эта уже синхронная операция. Но что произойдёт?

Он выполнится после всех операций. Так как он заносится в Web API и ждёт там пока закончатся обычные операции. И только после этого выполнит.

Промисы.

Перед тем как начать тему промисы. Давайте смоделируем через асинхронные операции и работу callback функций работу сервера.

```
console.log('Request data...')

setTimeout(() => {
  console.log('Preparing data...')

  const backendData = {
    server: 'aws',
    port: 2000,
    status: 'wokring'
  }

  |
}, 2000)
```

То есть мы здесь за 2 секунды типа обращаемся к серверу. Получаем какие-то данные. Теперь понадобится ещё какое-то время чтобы эти данные вернуть обратно клиенту.

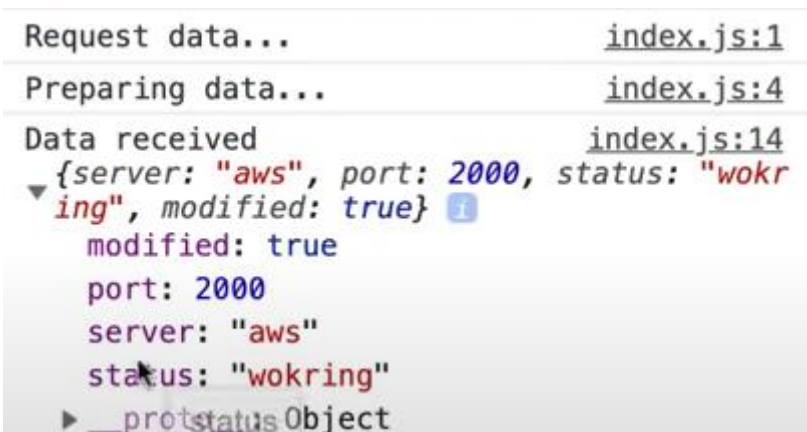
```
console.log('Request data...')

setTimeout(() => {
  console.log('Preparing data...')

  const backendData = {
    server: 'aws',
    port: 2000,
    status: 'wokring'
  }

  setTimeout(() => {
    backendData.modified = true
    console.log('Data received', backendData)
  }, 2000)
}, 2000)
```

И получаем примерно такой результат



```
Request data... index.js:1
Preparing data... index.js:4
Data received index.js:14
{server: "aws", port: 2000, status: "wokring", modified: true}
  modified: true
  port: 2000
  server: "aws"
  status: "wokring"
  __proto__: Object
```

То есть мы реализовали последовательную асинхронность через callback функции.

Чем плох данный подход? Тем что мы получаем большую вложенность. И если там будет сразу несколько запросов обращения на сервер, код будет намного больше в наших функциях, то такой код поддерживать достаточно сложно. Для упрощения этой задачи на помощь пришли промисы.

Создаём переменную через глобальный объект Promise. Мы должны туда передать функцию callback, которая принимает два параметра. Resolve и Reject. Эти два параметра тоже функции.

И далее в функции нам нужно написать какой-то асинхронный код. Можно и синхронный, но нет никакого смысла в этом.

Давайте запишем через промисы первую асинхронную функцию, которая была в set timeout.

```
const p = new Promise(function(resolve, reject) {
  setTimeout(() => {
    console.log('Preparing data...')
    const backendData = {
      server: 'aws',
      port: 2000,
      status: 'wokring'
    }
  }, 2000)
})
```

Возникает вопрос как работать этими данными и получить доступ до внутренних данных.

Для этого существуют две функции resolve(успешно) и reject (не успешно) когда закончен промис. Например, наш промис успешно завершил работу.

```
const p = new Promise(function(resolve, reject) {
  setTimeout(() => {
    console.log('Preparing data...')
    const backendData = {
      server: 'aws',
      port: 2000,
      status: 'wokring'
    }
    resolve()
  }, 2000)
})
```

И вызовем метод then. Чтобы проверить что туда пришло.

```
p.then(() => {
  console.log('Promise resolved')
})
```

А теперь давайте дальше допишем функционал чтобы работал также как и на примере callback функции.

```
const p = new Promise(function(resolve, reject) {
  setTimeout(() => {
    console.log('Preparing data...')
    const backendData = {
      server: 'aws',
      port: 2000,
      status: 'wokring'
    }
    resolve(backendData)
  }, 2000)
})

p.then(data => {
  console.log('Promise resolved', data)
})
```

```
p.then(data => {
  const p2 = new Promise((resolve, reject) => {
    setTimeout(() => {
      data.modified = true
      resolve(data)
    }, 2000)
  })

  p2.then(clientData => {
    console.log('Data received', clientData)
  })
})
```

Всё работает также как и раньше, но через промисы. У нас такая же вложенность. Тогда какой смысл в них?

Поэтому изменим наш промис другим образом тоже через then.


```

p.then(data => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      data.modified = true
      resolve(data)
    }, 2000)
  })
}).then(clientData => {
  console.log('Data received', clientData)
})

```

Вот в чём прелесть промисов. У нас получилась один уровень вложенности. И более наглядно видно. Один промис выполнен, за ним следующий и так далее.

Чем хороши промисы? Если мы работаем с какими-то асинхронными операциями. У нас могут возникать ошибки. И с помощью промисов их удобно отлавливать.

В колбек функция не очень удобно. Нужно каждую тип ошибки проверять. А в промисах есть для этого команда `catch`. Его обычно пишут в конце.

```

p.then(data => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      data.modified = true
      reject(data)
    }, 2000)
  })
})
  .then(clientData => {
    clientData.fromPromise = true
    return clientData
  })
  .then(data => {
    console.log('Modified', data)
  })
  .catch(err => console.error('Error: ', err))

```

Есть также метод `finally`. Он отработает всегда независимо от правильности или ошибки. Он выполнится в конце.

```

p.then(data => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      data.modified = true
      reject(data)
    }, 2000)
  })
})
.then(clientData => {
  clientData.fromPromise = true
  return clientData
})
.then(data => {
  console.log('Modified', data)
})
.catch(err => console.error('Error: ', err))
.finally(() => console.log('Finally'))

```

Полезные фишки для промисов.
 Напишем функцию sleep

```

const sleep = ms => {
  return new Promise(resolve => {
    setTimeout(() => resolve(), ms)
  })
}

sleep(2000).then(() => console.log('After 2 sec'))
sleep(3000).then(() => console.log('After 3 sec'))

```

Это упрощенная функция для задержки на определённое время.

Также у глобального Promise есть два метода.

Первый это all. Он работает следующим образом он выполнится тогда, когда завершатся все промисы, которые мы ему передаём в параметре(массив)

```

Promise.all([sleep(2000), sleep(5000)]).then(() => {
  console.log('All promises')
})

```

Это нужно, например, когда есть несколько запросов к базе данных разные и вам надо подождать пока они завершатся и только потом выполнять свой.

Второй метод это Race. Он работает, наоборот. В него также передаётся массив промиссов. И наш промис выполнится тогда, когда первый из массива какой-то выполнится.

```
Promise.race([sleep(2000), sleep(5000)]).then(() => {  
  console.log('Race promises')  
})
```

Задача. На использование промисов и обращение к серверу. Сервер у нас будет фейковый. Называется Json placeholder

Мы будем использовать функцию fetch. Это аналог AJAX, но только для промисов.

fetch отправляет запрос по адресу, а затем выполняет функцию, переданную в методе then

Если промис возвращает промис, то можно использовать цепочку вызовов. Сначала напишем функцию delay(аналог sleep) На сайте возьмём ссылку.

```
const delay = ms => {  
  return new Promise(r => setTimeout(() => r(), ms))  
}  
  
const url = 'https://jsonplaceholder.typicode.com/todos'  
  
function fetchTodos() {  
  console.log('Fetch todo started...')  
  return delay(2000)  
    .then(() => {  
      return fetch(url)  
    })  
    .then(response => response.json())  
}  
  
fetchTodos()  
  .then(data => {  
    console.log('Data:', data)  
  })  
  .catch(e => console.error(e))
```

Выглядит всё неплохо. У нас нет вложенностей. Используем промисы. Кстати, попробуйте функцию ещё сократить.

```
function fetchTodos() {
  console.log('Fetch todo started...')
  return delay(2000)
    .then(() => fetch(url))
    .then(response => response.json())
}

fetchTodos()
  .then(data => {
    console.log('Data:', data)
  })
  .catch(e => console.error(e))
```

Async Await

Но благодаря новым подходам в JS async И await мы можем сделать данный подход ещё проще.

Перепишем старый код через новые ключевые слова. Если мы используем await. То результат нужно заносить в переменную, так как оператор Await автоматически обрабатывает промис (then) и поэтому заносим в переменную.

```
async function fetchAsyncTodos() {
  console.log('Fetch todo started...')
  await delay(2000)
  const response = await fetch(url)
  const data = await response.json()
  console.log('Data:', data)
}

fetchAsyncTodos()
```

Но что на счёт метода catch вдруг будут ошибки.

В этом нам поможет try catch

```
async function fetchAsyncTodos() {  
  console.log('Fetch todo started...')  
  try {  
    await delay(2000)  
    const response = await fetch(url)  
    const data = await response.json()  
    console.log('Data:', data)  
  } catch (e) {}  
  console.error(e)  
}  
  
fetchAsyncTodos()
```

Вот и удобство `async await`. Такая функция всегда возвращает промис. Они для удобства ввели его с работой асинхронных функций.