Асинхронность

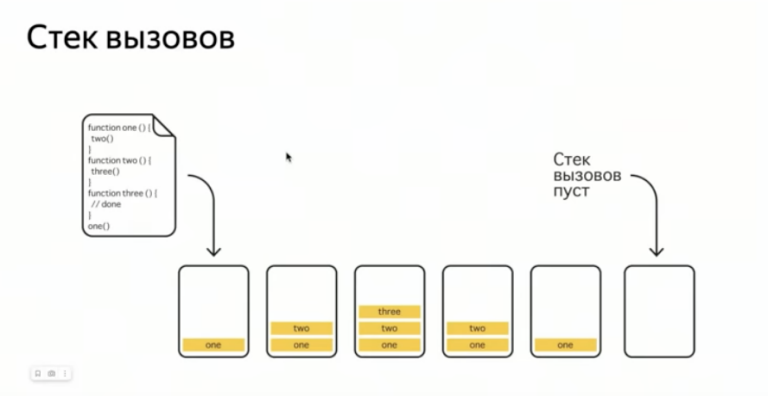
Status	Done
URL	https://www.youtube.com/watch?v=XFn-nC-YZg8&t=113s

Event Loop

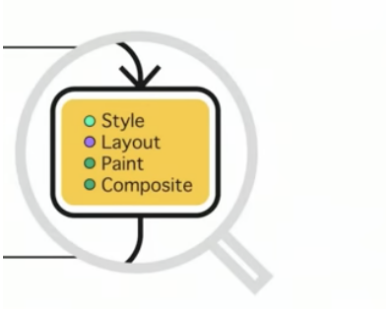
- Две важные задачи:
    - Выполнять код
    - Обновить интерфейс
- Чтобы все казалось, что это выполняется параллельно, приходится переключаться между задачами

Стек вызовов

- Место, куда функция попадает на выполнение и откуда начинает выполняться
- First In Last Out



Обновление интерфейса (этапы)



📕 Чтобы разобраться подробнее, можно почитать статью “Оптимизация производительности фронтенда. Часть 2”

Задачи, тики и Web API

Определения

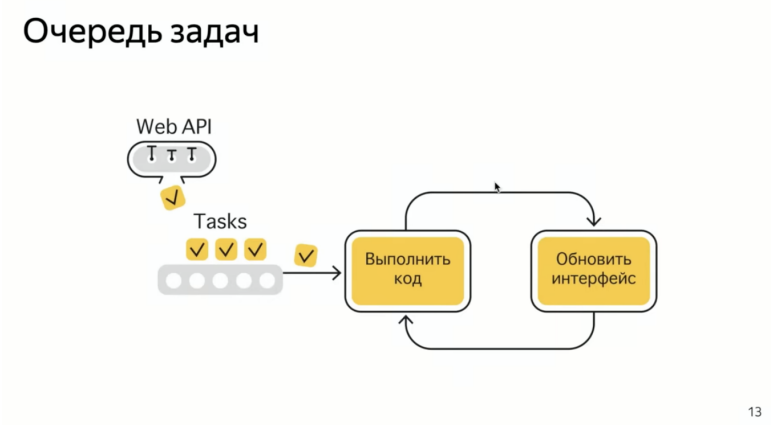
- Задача - это JavaScript-код, который выполняется в стеке вызовов
- Тик - выполнение задачи в стеке вызовов
- Web API - свойства и методы глобального объекта Window
- У Web API есть синхронные и асинхронные методы

Синхронные методы

- Синхронные выполняются “сейчас”, в текущем тике
  - Нет ничего страшного в том, чтобы, например, добавить кнопку и сразу ее перекрасить - все эти операции произойдут за один тик, и для пользователя не будет никакого моргания

Асинхронные методы

- Асинхронные - регистрируем обработчик, который будет выполнен когда-то потом
  - Таких задач может быть много, их надо как-то совмещать
  - Для этого существует очередь задач



- Для взаимодействия с очередью задач есть два способа - setTimeout и setInterval. setTimeout более надежный

```
setTimeout(()=>{
  //Поставить задачу в очередь через 1000мс
}, 1000)
```

- Через 1000мс задача не выполнится, а только поставится в очередь. Аналогично и с event-ами

```
document.body.addEventListener('click', ()=>{
  // Поставить задачу в очередь, когда наступит событие
})
```

Время на выполнение задачи

- Мониторы моргают очень быстро - поэтому задачи должны быть маленькими, чтобы успевать выполняться за время одной перерисовки
- Обработка больших задач: надо их разбивать на маленькие

```
function longTask () {
  toDoSomethingFirst()

  setTimeout(()=>{
    toDoSomethingLater()
  }, 0)
}
```

- Другой способ: postMessage. Так мы можем без задержки добавлять задачи в очередь

```
window.addEventListener('message', (e) => {
  console.log(e.data)
})

window.postMessage('...')
```

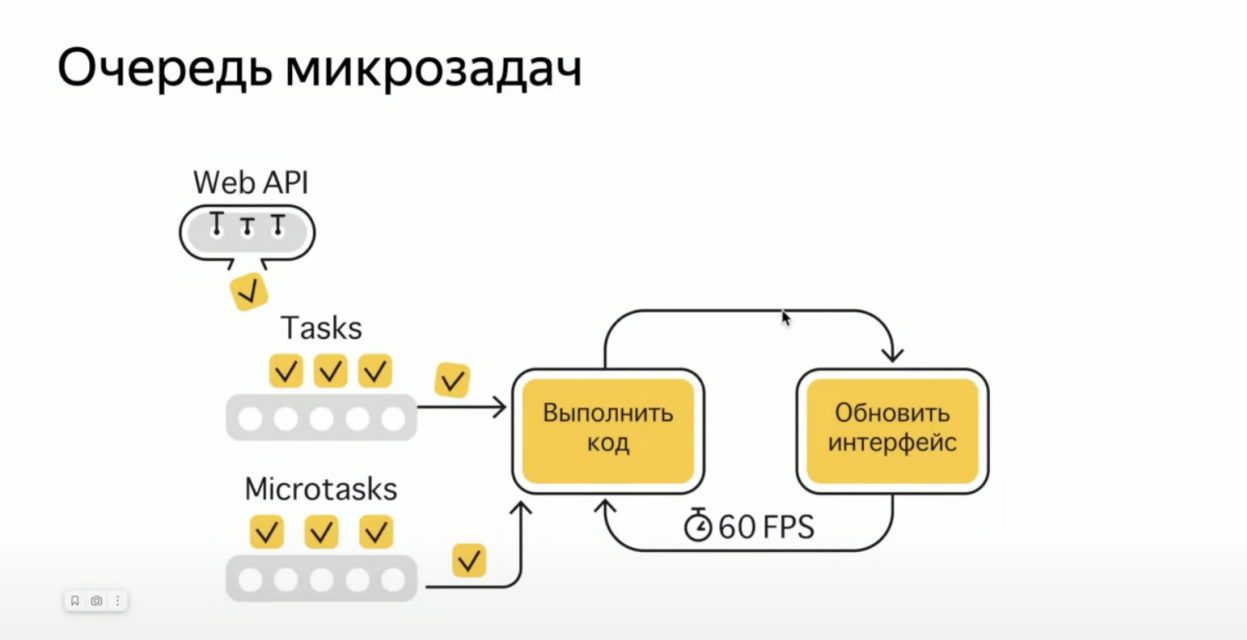
- Тем не менее, event loop - это один поток. Но существуют Web Worker API

```
const worker = new Worker('worker.js')

worker.addEventListener('message', ()=> {
  //получить данные
})

// отправить данные
worker.postMessage('...')
```

- Worker-ы не могут взаимодействовать с DOM-ом, но зато могут разгрузит вычисления

Очередь микрозадач

- Выполняем задачи из этой очереди, пока они там есть.



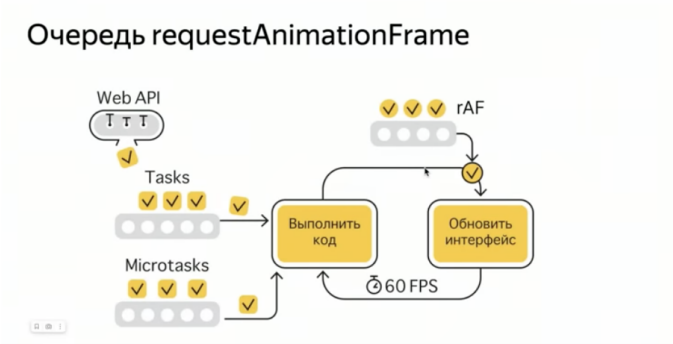
- Взаимодействие с очередью микрозадач

```
Promise.resolve.then(()=> {
  //микрозадача
})

async function () {
  await //микрозадача
}

queueMicrotask(() => {
  //микрозадача
})

new MutationObserver(() => {
  //микрозадача
}).observe(document.body, {childList: true, subtree: true })
```

- Посмотреть, как это работает, можно вот тут: JS Visualizer 9000

requestAnimationFrame

- Дает возможность завязаться на момент перед обновлением интерфейса
- Тут тоже своя очередь



- Выполняются все задачи, кроме тех, что было добавлено в текущем витке (то есть то, чего не было на момент начала)
- Повторяющаяся задача

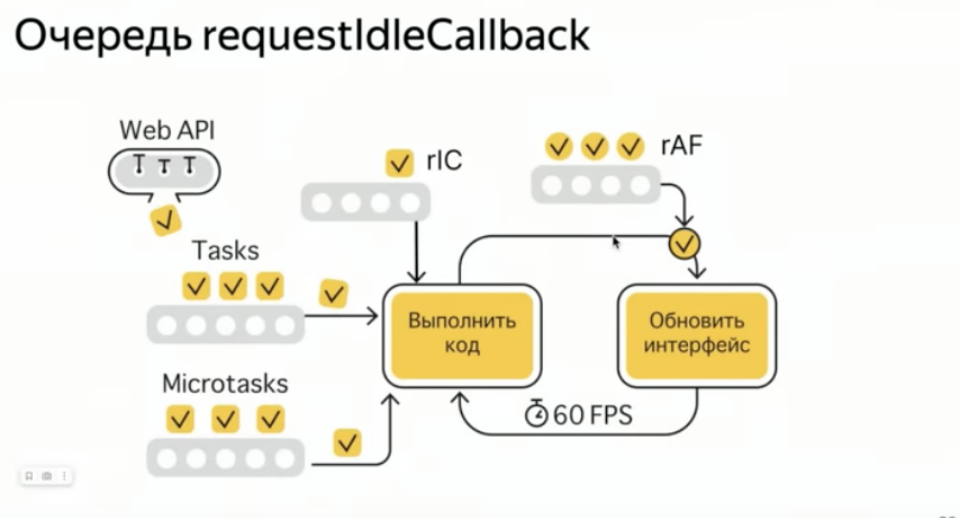
```
let requestId

function animate () {
  requestId = requestAnimationFrame(animate)
}

requestId = requestAnimationFrame(animate)

setTimeOut(() => {
  //отменить анимации через какое-то время
  cancelAnimationFrame(requestId)
}, 3000)
```

requestIdleCallback

- Экспериментальная штука
- Планирует низкоприоритетные задачи (сделает, когда нечем будет заняться)

```
function sendAnalytics () {
    //отправить данные для аналитики
}
```

```
requestIdleCallback(sendAnalytics, {timeout: 2000})
```



Сравнение очередей

- Микрозадачи - самая приоритетная очередь. Выполняем, пока есть задачи
- Задачи - одно или несколько заданий между обновлениями интерфейса (классическая очередь)
- rAF - задачи, выполняемые непосредственно перед обновлением интерфейса. Все задачи из очереди выполняются, но новые - уже в следующем тике
- rIC - в период простоя, когда нет более приоритетных задач

Взаимодействие с очередями

- Задачи - таймеры, события (включая обработку postMessage)
- Микрозадачи - промисы, async, Observer API, queueMicrotask
- rAF/rIC - соответствующие вызовы API

Цикл событий в Node.js

- Работает схожим образом, но набор очередей отличается

📕 Difference between process.nextTick and queueMicrotask (stackOverflow)

📕 NodeJS Event Loop by Deepal Jayasekara

Callback

- Функции обратного вызова

- Проблема монстра Залго: не понятно, синхронно или асинхронно вызывается коллбек (зависит от того, что у нее внутри)
- Жесткая сцепленность:

```
step1((error, data) => {
  if (error){
    //отменить этап 1
  }
  step2((error, data) => {
    if (error){
      //отменить этап 2, потом 1
    }
  })
})
```

- Инверсия управления

```
import { thirdPartyCode } from 'third-party-package'

thirdPartyCode(() => {
  //инверсия управления - не знаем, где и как будет вызываться наша функция
})
```

Promises

- У промиса есть два состояния - выполнение и отказ

```
//Выставить промис на выполнение
const resolvedPromise = new Promise((resolve, reject) => {
  setTimeout(() => {resolve('ok') }, 1000)
})

//Выставить промис на отказ
const resolvedPromise = new Promise((resolve, reject) => {
  setTimeout(() => {reject('not ok') }, 1000)
})

//обработка выполнения
resolvedPromise.then((value) => {
  console.log(value)
})

//обработка отказа
rejectedPromise.then(
  (value) => {
    //
  },
```

```
  (error) => {
    console.log(error)
  }
)

rejectedPromise.catch((error) => {
  console.log(error)
})
```

- Значение выставляется раз и навсегда

```
const promise = new Promise((resolve, reject) => {
  resolve('ok')
  reject('not ok') //уже никак не повлияет на состояние Promise
})
```

- Статические методы resolve и reject - возвращают установленный промис

```
Promise.resolve('ok').then((value) => {
  console.log(value)
})
```

- Метод finally
  - Выполняет какую-то работу, независимо от успеха или неудачи
  - Ничего не получает, но пробрасывает результат через себя

```
Promise.resolve('ok').finally(()=> {
  //какая-то работа
}).then((value) => {
  console.log(value)//'ok'
})
```

Хитрости Promise

Цепочки из Promise

- Последовательные операции
- Обработчики неявно возвращают новый промис

```
Promise.resolve()
  .then(() => {
    //return new Promise((resolve) => resolve())
  })
  .then(() => {
  //Мы ничего не вернули из then, но код продолжает работать
})
```

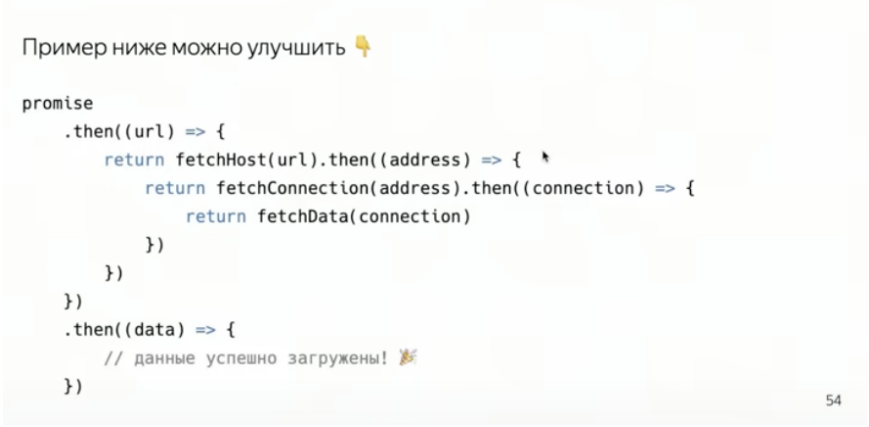
- Любое значение ***кроме промиса*** будет ***завернуто в промис***

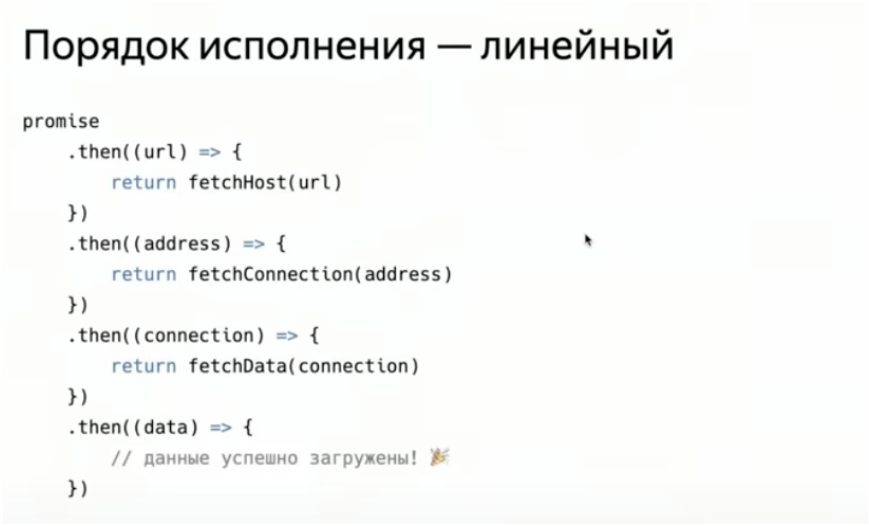
```
Promise.resolve()
  .then(() => {
    return 'Заверните меня, пожалуйста'
  })
  .then((value) => {
    //у String нет метода then, но код продолжает работать
    console.log(value)//'Заверните меня, пожалуйста'
})
```

- Возвращение Promise через return: промис распакуется, а затем значение запакуется в новый промис

```
Promise.resolve()
  .then(() => {
    return Promise.resolve('fire')
  })
  .then((value) => {
    console.log(value)//'fire'
})
```

- Каждый then цепляется к предыдущему





Обработка ошибок

- Проброс отказа

```
Promise.resolve()
  .then(() => {
    return Promise.reject('not ok')
  })
  .then((value) => {
    //все обработчики на выполнение будут пропущены
  })
  .catch((error) => {
    console.log(error) //'not ok'
  })
  .then(()=> {
    //продолжаем выполнять цепочку в штатном режиме
})
```

- Обработчик отказа поймает любые ошибки - встроенный try/catch

```
Promise.resolve()
  .then(() => {
    undefined.toString()
  })
  .catch((error) => {
    console.log(error) //TypeError: cannot read property 'toString' of undefined
  })
```

- В catch тоже работает return

```
Promise.reject('not ok')
  .catch((error) => {
    console.log(error) //'not ok'
    return 'ok'
  })
 .then((value) => {
    console.log(value) //'ok'
  })
```

❗ Нужно ставить catch в конце

- Если ошибка произошла в последнем catch, ее поймает сборщик мусора. На это событие можно подписаться

```
window.addEventListener('unhandledrejection', (event) => {
  console.log(event)//PromiseRejectionEvent
  console.log(event.reason) //not ok
})

Promise.reject('not ok')
```

- Разрыв цепочки

```
Promise.resolve()
  .then(() => {
```

```
    Promise.reject('not ok')
  })
  .catch((error) => {
    //Будет пропущено, потому что не указан return
    //unhandledrejection
  })
```

Thenable объекты

- Объект, у которого есть метод then

```
const thenable = {
  then (fulfill){
    return fulfill('smth')
  }
}
```

- Скорее всего, это полифилл для Promise до ES6
- Совместимость: Promise.resolve, resolve и return распакают значение thenable объекта и обернут в полноценный ES6 Promise

Как Promise.resolve обработает Promise?

- Вернет без изменений

```
const promise = Promise.resolve('apple')
const resolve = Promise.resolve(promise)

console.log(promise === resolved) // true

resolved.then((value) => {
  console.log(value) //apple
})
```

- Но resolve и return создают новый промис

```
const promise = Promise.resolve('apple')

promise === Promise.resolve(promise) //true

promise === new Promise((resolve) => resolve(promise)) //false
```

```
promise === Promise.resolve().then(() => promise) //false
```

Как Promise.reject обработает Promise

- Еще раз обернет в Promise и вернет как причину отказа
- resolve и reject внутри промиса работают также

```
const promise = Promise.resolve('apple')

const rejected = Promise.reject(promise)

promise === rejected //falsee

rejected.catch((value) => {
  console.log(value) //Promise {<resolved>: 'apple'}
})
```

Статические методы Promise

- Promise.all - возвращает массив значение или первый отказ
- Promise.race - возврашет либо первое значение, либо первый отказ (смотря что быстрее)
- Promise.any - возвращает первое значение или массив с причинами отказа
- Promise.allSettled - дожидается всех операций и возвращает массив специальных объектов ({status: ‘fulfilled’, value: ‘ok’})
- Обработка пустого массива:
  - all и allSettled вернут пустой массив
  - any вернет ошибку - all promises were rejected
  - race - никогда не выполнится

Как быть, когда Promise завис?

```
Promise.race([
  fetchLongRequest(),
  new Promise(_, reject) => setTimeout(reject, 3000)),
]).then ((data) => {
  //получили данные
}).catch ((data) => {
  //отказ по таймер
})
```

Промисификация

- Функция-помощник, которая превращает функции обратного вызова в Promise

```
import promisify from 'utils'

function asyncApi (url, callback){
  //выполнить асинхронную функцию
  callback('fire')
}

promisify(asyncApi)('/url')
.then((result) => {
  console.log(result) //'fire'
})
```

Корутины

- Итератор - это корутина (сопрограмма)
- Cooperative concurrently executing routines
- Корутина - это функция, которая

- Приостанавливает работу
- Запоминает текущее состояние
- Имеет несколько точек входа и выхода

Async/await

- Асинхронная функция заворачивает свой результат в Promise

```
async function asyncFunction () {
  return 'fire'
}

asyncFunction().then((value) => {
  console.log(value) //fire
})
```

- Await получает результат из promise
- Await работает только внутри асинхронной функции

```
(async() => {
  const value = await Promise.resolve('fire')
  console.log(value) //'fire'
})
```

- await - это просто оператор

```
console.log(await 42 + await Promise.resolve(42)) //84
```

- Скрытая опасность

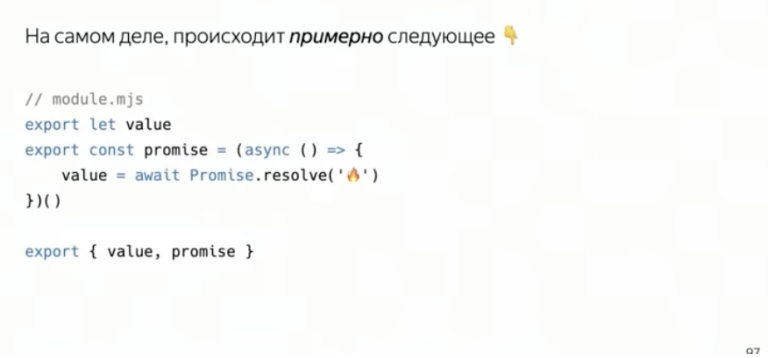
Top level await

- Работает только для ES модулей или в DevTools

```
const connection = await dbConnector()
```

Как работает?

- Делает модуль асинхронным



Не все await одинаково полезны

```
const articles = await fetchArticles()
const pictures = await fetchPictures()


//Запроосы независимы, но выполняются последовательно
//Могли бы независимо грузить картинки и статьи, а так ждем сначала статей



//Делаем запросы параллельными (решение 1)
const articlesPromise = fetchArticles()
const picturesPromise = fetchPictures()

const articles = await articlesPromise;
const pictures = await picturesPromise;


//Лучше вот так:
const [articles, pictures] = await Promise.all([
    fetchArticles(),
    fetchPictures(),
])
```

Обработка ошибок

```
//Вариант 1: try/catch - сможет обработать верхнеуровненый await

async function asyncFunction () {
  try {
    await Promise.reject('not ok')
  } catch (e) {
    //перехватить ошибку
  }
}

asyncFunction().then((value) => {
  //мы в безопасности
})

//Вариант 2: catch

async function asyncFunction () {
  await Promise.reject('not ok')
}

asyncFunction()
  .then((value) => {
      //вдруг ошибка?
  })
  .catch((error) => {
      //Тогда ее поймает catch
  })
```

Async функции и массивы

forEach + async/await

```
const wtf = ['crazy']

wtf.forEach(async (emoji) => {
  console.log(emoji)
})

console.log('It is asynchronous')
```

```
//It is asynchronous
//crazy
```

map + async/await

```
const urls = ['/data', '/meta']

const result = urls.map( async (url) => {
  const response = await fetch(url)
  return await response.json()
})

console.log(result) //[Promise, Promise]


//попытка 2

const urls = ['/data', '/meta']

const promises = urls.map( async (url) => {
  return (await fetch(url)).json())
}

const result = await Promise.all(promises)


console.log(result) //[{...}, {...}]
```

filter + async/await

- Внутренняя функция будет возвращать промисы, то есть объекты, а они всегда true

```
const values = [null, 0, 42]

const wtf = values.filter(async (value) => {
    return Boolean(value)
})

console.log(wtf) //[null, 0, 42]
```

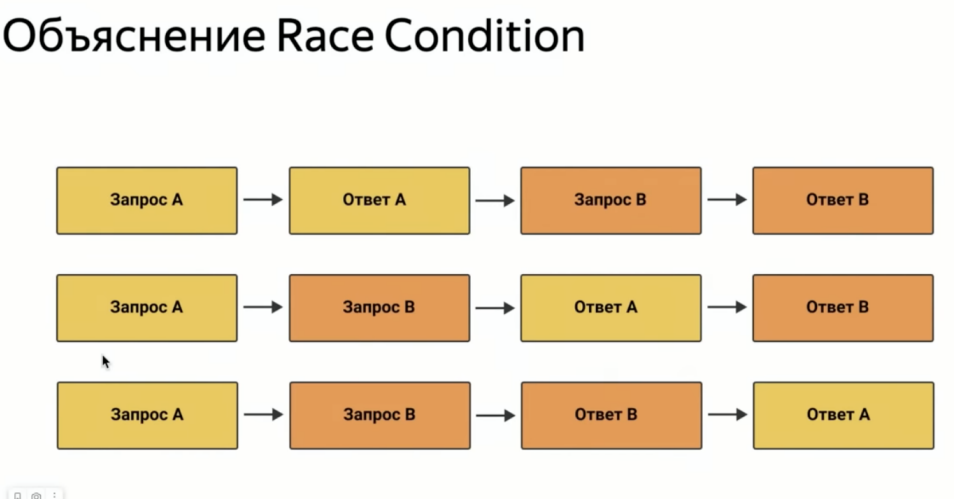
reduce+async/await

```
const numbers = [4, 8, 15, 16, 23, 42]

const result = numbers.reduce( async(sum, number) => {
  return sum + number //складываем промис с числом - перевод в строки и конкатинация
}, 0)

console.log(result) //Promise

console.log(await result) //[object Promise]42



//правильно вот так:

const result = numbers.reduce(async(sum, number) => {
    return (await sum) + number
},0)

console.log(await result)//108
```

Callback vs Promises vs async/await

- Иногда коллбеки необходимы
- Если используем последовательные операции, лучше Promises или async/await

Race Condition



- Решение:
  - При отправке запроса прерываем все предыдущие
  - При отправке запроса игнорируем все предыдущие

Рекомендованная литература

📕 Симпсон - {Вы не знаете JS}Асинхронная обработка и оптимизация

📺 Джейк Арчибальд. В цикле - JSConf.Asia

📰 Полное понимание асинхронности в браузере (хабр)