Promises

Промис это **объект** (обещание). Этот объект нам возвращает какая то функция вместо конкретного результата, который мы хотим от нее получить. Она хотела бы \ могла бы вернуть нам конкретные данные, но возвращает “обещание” вернуть эти данные когда то позже.

Это происходит потому что в логике функции нет данных, которые она могла бы отдать здесь и сейчас. Либо это запрос на сервер, или в базу данных, либо ещё что то асинхронное. Но когда это асинк логика закончится, нужно эти данные забрать.

У промиса нет свойств, у него есть только методы.

метод then

По сути метод then подписывается на промис, и ожидает изменения его статуса.

метод принимает в качестве аргумента колбэк функцию, которая будет выполнена когда промис выполнится (resolve (fullfield)).

```
const promise = axios.get('https://google.com')
```

```
promise.then( (data) => { console.log(data) } )
```

соответсвенно в data придут данные, которые пришлет сервер google. И мы отобразим их в логе.

Промис может находится в 3 состояниях. pending, resolved, rejected.

pending означает что промис находится в состоянии исполнения, и мы еще знаем, завершится ли он успешно или нет

resolved означает что промис успешно исполнился и мы получили данные, которые запрашивали. После этого мы можем использовать метод .then

rejected означает что промис не выполнился и упала какая либо ошибка и тд

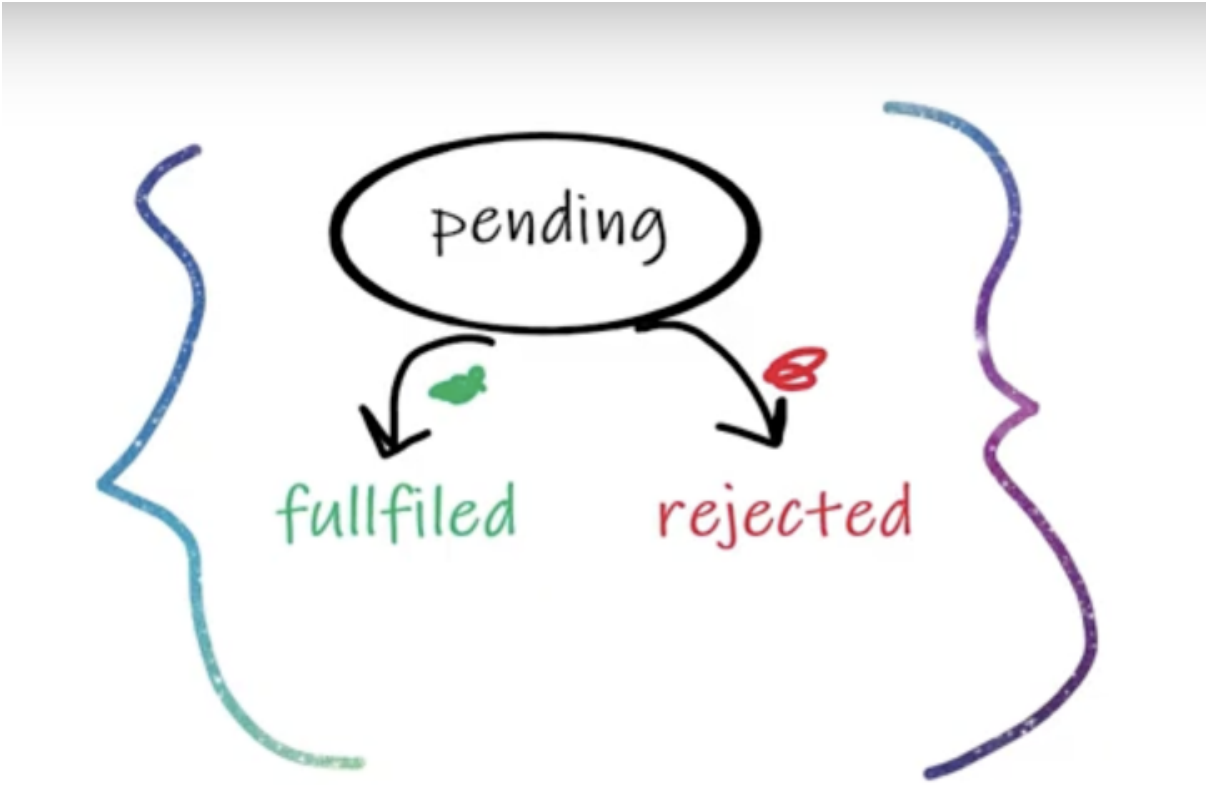
Промис не может поменять своё состояние, он либо зарезолвился, либо зареджектился.

Метод catch

Этот метод позволяет отловить ошибку и обработать её. Такая ошибка не ломает код, он продолжает выполнятся.

Метод finally

отрабатывает ВСЕГДА, не важно, зарезолвился ли промис, или произошел реджект.



Взаимодействие с промисом происходит ТОЛЬКО через методы. Мы не может достучаться к каким то свойствам объекта промис.

В промисе есть “ячейка” которая называется PromiseResult, туда помещается либо ошибка, с которой промис зареджектился, либо данные, которые были запрошены например с сервера, если промис зарезолвился. Вся логика того, что будет доступно в PromiseResult после pending описана в логике создания промиса, например на бэкэнде.

Метод Promise.all()

Это статический метод класса Promise

```
const promise1 = /... some async code .../
const promise2 = /... some async code .../

const finalPromise = Promise.all([promise1, promise2])
```

```
finalPromise.then( ()=> {console.log('all promises resolved' ) } )
```

Метод .all() возвращает промис, когда зарезолвятся промисы, переданные в массиве в его параметрах. В примере выше, метод then отработает только в том случае и когда оба промиса из массива зарезолвятся и в консоли мы увидим сообщение.

Если хотя бы один промис не резолвится, то наш finalPromise тоже не зарезолвится. И точно так же можем отловить ошибку с помощью catch.

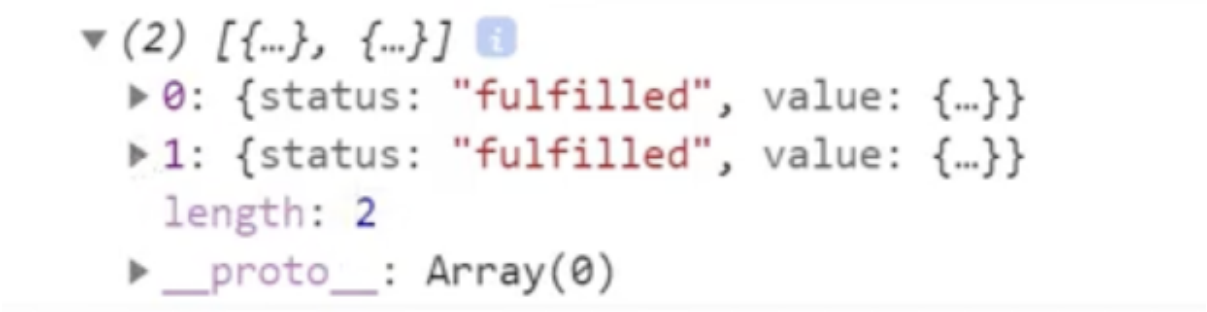
Если оба промиса зарезолвились, то их результат придет как массив результатов, с которым мы можем работать в методе then. Порядок результатов в массиве не определяется тем, какой из промисов быстрее зарезолвился. Они будут находится в том же порядке, в котором были переданы в массиве в параметре .all.

Метод allSettled

Так же статический метод класса Promise. Который не зависимо от того, зарезолвился ли промис или зареджектился, будет иметь метод then() в который придут объекты промисов, которые дальше можно будет обработать.

Они могут либо оба зарезолвится (2 0)-, либо (1 - 1) , либо (0 - 2) как пример.

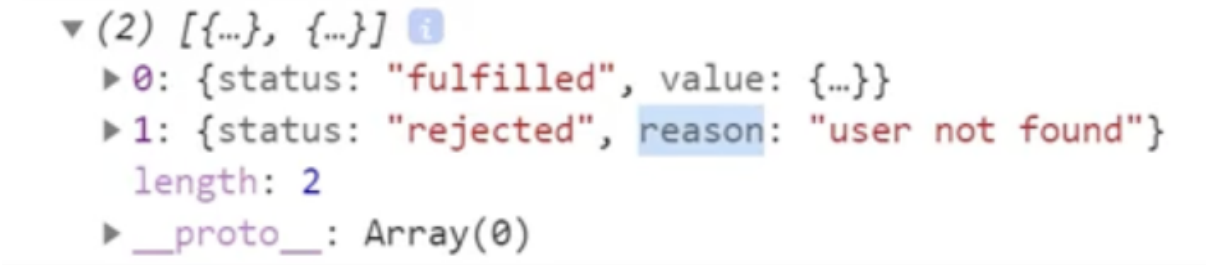
```
const promise1 = /... some async code .../
const promise2 = /... some async code .../

const finalPromise = Promise.allSettled([promise1, promise2])

finalPromise.then( (results )=> {console.log(results) } )
```



В results у нас будет лежать так же массив с объектами, но структура этих объектов немного другая, нежели в методе .all()

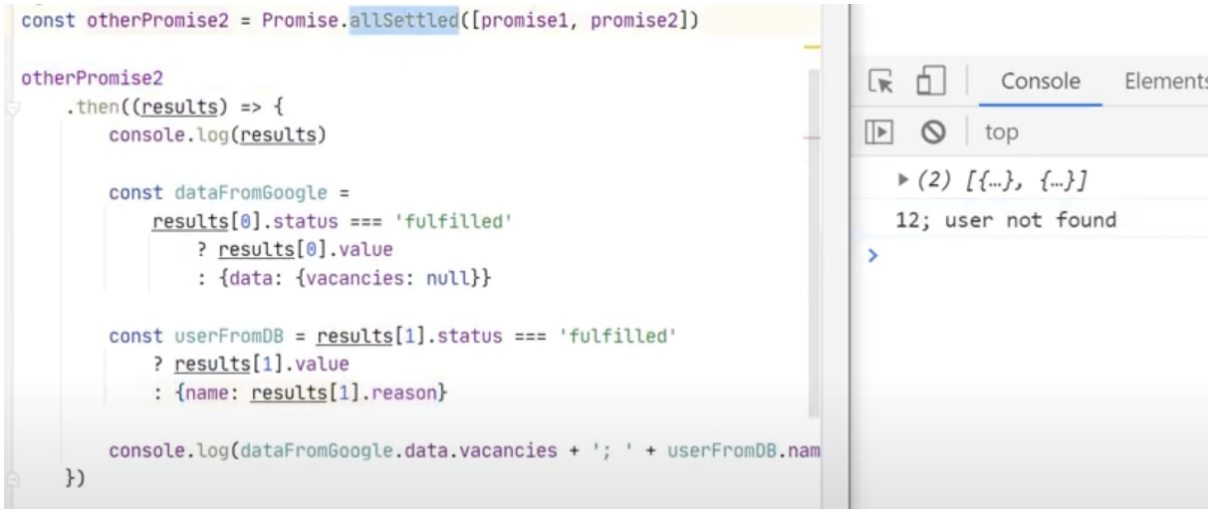
У нас есть 2 свойства: status указывает на статус промиса, а в value лежат сами данные.

Если же один из промисов зареджектился , то структура объекта будет такая.



по ключу reason будет находится ошибка, которая описана в логике при создании промиса.

Пример обработки промисов через allSettled



Статический метод Promise.resolve

Можно сразу создать промис, который моментально зарезолвится значением 100.

Иногда это можно использовать для создания заглушек. Сделать фейк для проверки работы своего кода.

```
const resolvedPromise = Promise.resolve(100)
```

Аналогично можно вернуть reject

```
const rejectedPromise = Promise.reject('Some error')
```

На такой промис можно сразу применить метод catch .

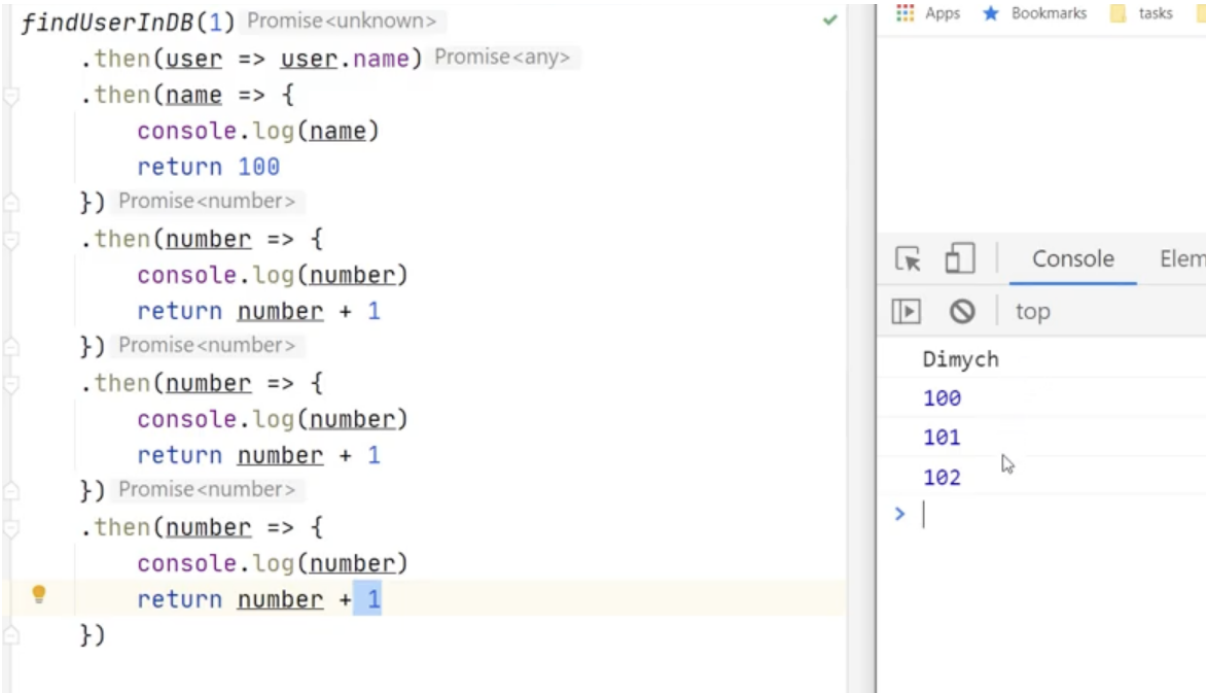
Promises chaining

Каждый вызов then возвращает нам новый промис.

```
const promise2 = findUserInDB() // вызов функции возвратит промис, который
  // запишется в переменную promise2

const promise2_2 = promise2.then(user => //some code)
  // promise2_2  зарезолвится после того, как зарезолвится promise 2  и
  //выполнится колбэк, переданный в then
```

promise2_2 зарезолвится тем значением, которое будет выполнено в колбэке promise2

```
предположим, что в promise1 в качестве user пришел объект, с полем name: 'Sasha'
Но в методе. then мы изменили имя

const promise2_2 = promise2.then(user  => user.name = 'Roman')

и в качестве name в promise2_2 прийдет значение которое мы изменили в колбэке
promise 2

promise2_2.then(name => console.log(name)

в консоли мы увидим 'Roman'
```



```
findUserInDB(1) // находим юзера с айдишкой 1 в базе данных
  .then( user => { // в user сидит объект пользователя из базы данных
    console.log(user.name) // логируем его имя в консоли
    return findUserInDB(user.friendId) //функция ретурнит промис, но в следующий
    //then попадет ТО ЧЕМ ПРОМИС ЗАРЕЗОЛВИЛСЯ. А резолвится он новым
    // пользователем (Другом первого user который первый пришел из базы)
    })
  .then(friendOfFirstUser => {
     console.log(friendOfFirstUser.name)
     return findUserInDB(friendOfFirstUser.friendId)
    })
  .then(friendOfSecondUser => {
     console.log(friendOfSecondUser.name)
    })
```

Async-Await

Мы используем then для того, чтобы выстроить цепочку и дождаться когда предыдущий промис зарезолвится и выполнить далее какие то действия.

async-await это другой вид записи цепочки промисов, проще читается и красивее выглядит. Await можно использовать только в асинхронной функции.

```
async function run () {

  let user = await findUserInDB(1) // в переменную юзер сразу запишется
                                   // выполнения запроса, а не промис
  console.log(user)

  let friend1 = await findUserInDB(user.friendId)
  console.log(friend1)

  let friend2 = await findUserInDB(friend1.friendId)
  console.log(friend2)

}

run()
// в консоли будем видеть объекты пользователей по мере того
// как будет отрабатывать функция и возвращать результат запроса к базе данных
```