AJAX

**AJAX: Asynchronous Javascript and XML** **– технология обращения к серверу без перезагрузки страницы. Как делаются AJAX запросы?**

**Устаревшие методы: XMLHttpRequest, jQuery**

**Сейчас можно использовать fetch API, axios**

**Создадим файл, подключим jQuery, создадим AJAX запрос.**

**Для простейшего AJAX запроса нужен будет только URL**

$.ajax('https://repetitora.net/api/JS/Images')

**В итоге в devtools dj вкладе мы будем работать в этом окне, пользуемся вкладками!**

****

Чтобы обработать запрос от сервера нужен колл-бек. Это функция, которую мы отдам кому-то, чтобы кто-то другой ее запустил когда настанет время это сделать.

Например, **setTimeout(function() {}, 1000).** Ей передаем колл-бек, который она запустит через секунду.

Чтобы передать функцию-коллбек AJAX, нужно создать в запросе объект с функцией:

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* () {

    console.log("I've got response")

  }

})

**Колл-беки очень часто используются, когда речь идет про АСИНХРОННЫЙ КОД**

**Когда библиотека Jquery запускает метод ajax и получает ответ, в функцию-колл-бек можно передать объект data от сервера:**

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* (*data*) {

    console.log(*data*)

  }

})

**Если работаем из браузера, то во вкладе Sources можно использовать точки останова, чтобы дебажить и смотреть данные!**

**Итак, напишем код, который будет подтягивать картинки с сервера по запросу:**

$.ajax('https://repetitora.net/api/JS/Images', {

  success: *function* (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

      img.src = *element*.thumbnail

      document.querySelector('body').appendChild(img)

    })

  }

})

Есть бесплатные сервер API по запросу в нете: publick get api for testing

**ТИПЫ ЗАПРОСОВ (REQUEST METHOD)**

Все AJAX запросы являются HTTP запросами. Вообще можно любым типом запроса выполнить любую задачу. Но есть договоренность использовать характерные запросы.

**GET** - используем, когда хотим что-то взять с сервера. Когда мы пишем адрес в адресной строке браузеры мы по сути делаем GET запрос на конкретную страницу.

Технически при выполнении GET запроса мы должны также отправлять инфо на сервер (для более точного понимания сервером, что мы хотим от него получить, например). Делается это с помощью **GET-параметра ключ-значение. Их мы будем запихивать в строку запроса.** Например:

https://repetitora.net/api/JS/Images**?page=2**

**?page=2 - это GET-параметр. Если их несколько, используем &:**

<https://repetitora.net/api/JS/Images?page=2&otherParam=3&name=james&age=32>

**В качестве параметра, технически можно передать даже файл (перед этим его нужно закодировать)**

**Какие параметры мы можем передавать на сервер?** Это решает серверный разработчик, какие параметры прописаны в серверной программе.

Параметры могут быть **обязательными и опциональными**



Допустим, есть некий URL:

https://repetitora.net/api/JS/Images

Возьмем из таблицы get-параметры, которые мы можем передать на сервер и добавим их к строке:

https://repetitora.net/api/JS/Images?page=1&count=20

Сервер выдаст 20 картинок

**UI - DAL**

Желательно разделить код программы на **UI** - это User Interface и **DAL** - Data Access Layer

В нашем случае ГШ это будут команды, касающиеся только интерфейса

*const* resultBlock = document.querySelector('#result')

*const* pageNumber = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

  getImages()

})

ajax запрос это будет DAL, этот код работает только с сервером и не знает ни о разметке ни о логике. Принцип single responsibility. Поэтому код ниже неправильный, из него нужно убрать все, что касается UI.

*function* getImages () {

  $.ajax(

    `https://repetitora.net/api/JS/Images?page=${pageNumber.value}&count=2`,

    {

      success: *function* (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

          img.src = *element*.thumbnail

          document.querySelector('#img\_container').appendChild(img)

        })

      }

    }

  )

}

После рефакторинга получим следующее (мы разнесли по разным файлам UI и DAL):

*///////////UI*

*const* resultBlock = document.querySelector('#result')

*const* pageNumber = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

  getImages(pageNumber.value, onDataReceived)

})

*function* onDataReceived (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

    img.src = *element*.thumbnail

    document.querySelector('#img\_container').appendChild(img)

  })

}

*///// DAL*

*function* getImages (*pageNumber*, *successCallback*) {

  $.ajax(`https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`, {

//callback сработает, когда придет ответ от сервера

    success: *successCallback*

  })

}

Иногда в программах может появиться очень много колл-беков, поэтому на помощь приходят ПРОМИСЫ.

PROMISE, AXIOS

Передача напрямую колл-беков нарушает принцип разделения UI и DAL, поэтому сделаем свой промис, убираем колл-беки

метод $.ajax() на самом деле возвращает ПРОМИС.

**ПРОМИС** - это обещание, что когда функция асинхронная закончится, к результату операции можно достучаться. **Это жирый чувак, который следит за асинхронной операцией.**

//

// UI

//

*const* resultBlock = document.querySelector('#result')

*const* pageNumberEl = document.querySelector('#page-number')

*const* clickMeButton = document.querySelector('#click-me')

clickMeButton.addEventListener('click', () *=>* {

*const* promise = getImages(pageNumberEl.value)

  promise.then(onDataReceived)

})

*function* onDataReceived (*data*) {

*data*.forEach(*element* *=>* {

*const* img = document.createElement('img')

    img.src = *element*.thumbnail

    document.querySelector('#img\_container').appendChild(img)

  })

}

//

// DAL

//

*function* getImages (*pageNumber*) {

*const* promise = $.ajax(

    `https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`

  )

  return promise

}

У промиса есть метод **THEN**. Он дает возможность выполнить функцию, когда промис зарезолвится. **Каждый THEN возвращает ПРОМИС, поэтому можно делать цепочку THEN! promise.then(…).then(…).then(…)**

**Подключим AXIOS g html. Будем использовать вместо Jquery.**

За счет того, что мы четко разделили два layerm нам достаточно поменять только функцию getImages:

*function* getImages (*pageNumber*) {

*const* promise = axios.get(

    `https://repetitora.net/api/JS/Images?page=${*pageNumber*}&count=2`

  )

  return promise

}

ВНИМАНИЕ! У каждой библиотеки свой интерфейс респонза!!! Но в любом случа возвращается объект!

TODOLIST-API, GET, POST

Что добавить инфо на сервер будем использовать метод **POST** для добавления тасок

*function* createTask (*title*) {

*const* promise = axios.post(

    `https://repetitora.net/api/JS/Tasks?widgetId=125&title=${*title*}`

  )

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

вызов функции сделаем прямо в файле

createTask('LearnJSd').then(*data* *=>* {

  console.log(*data*)

})

REQUEST PAYLOAD, POST, отправка данных на сервер

В примере ранее мы передавали данные прямо в строке методом POST. Но у браузера есть ограничения по длине, поэтому лучше использовать **payloader**

У POST запроса есть еще тело запроса **(body).** Либа axios вторым параметром принимает JSON объект с данными. В DevTools их можно увидеть в **Request Payload**

*function* createTask (*title*) {

*const* promise = axios.post(`https://repetitora.net/api/JS/Tasks`, {

    widgetId: 127,

    title: *title*

  })

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

ИДЕМПОТЕНТНОСТЬ, rest api, put, delete

В примере ранее мы передавали данные прямо в строке.

Идемпотентность GET запроса заключается в том, что сколько бы раз мы не отправили GET запрос, мы всегда получим один и тот же ответ. Другими словами, GET запрос ничего не меняет на сервере.

POST запрос НЕ идемпотентный.

**Принцип REST API предусматривает:**

- одинаковые URL для всех типов запросов

- запросы GET POST PUT DELETE

**PUT** используется для обновления данных.

У метода DELETE как и у GET запроса нет body, n.t. мы не можем передавать объект payloader

*function* deleteTask (*id*) {

*const* promise = axios.delete(

    `https://repetitora.net/api/JS/Tasks?widgetId=128&taskId=${*id*}`

  )

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

deleteTask('23d2f0ce-5c47-4011-af91-6952f1e9f4cd')

Также запишем метод PUT (в нем есть возможность отправить PayLoad)

*function* updateTask (*title*, *taskId*) {

*const* promise = axios.put(`https://repetitora.net/api/JS/Tasks`, {

    widgetId: 128,

    title: *title*,

    taskId: *taskId*

  })

  return promise.then(*response* *=>* {

    return *response*.data

  })

}

updateTask('New title', '4348ea01-2aed-4525-9b4b-21b5519f04b2')

CORS, options, content-type, accept-type

Перед тем как отправить «опасные» запросы на сервер, браузер отправляет метод **OPTIONS,** чтобы спросить у сервера, ты знаешь, что к тебе хочет обратиться кто-то, кому разрешаешь к тебе обращаться? В ответ приходит данные, которые можно увидеть в HEADERS закладке, в том числе объект {status: success}

Cross-Origin Resource Sharing ([CORS](https://developer.mozilla.org/ru/docs/Glossary/CORS)) — механизм, использующий дополнительные [HTTP](https://developer.mozilla.org/ru/docs/Glossary/HTTP)-заголовки, чтобы дать возможность [агенту пользователя](https://developer.mozilla.org/ru/docs/Glossary/User_agent) получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент. Говорят, что агент пользователя делает запрос с другого источника **(cross-origin HTTP request),** если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.