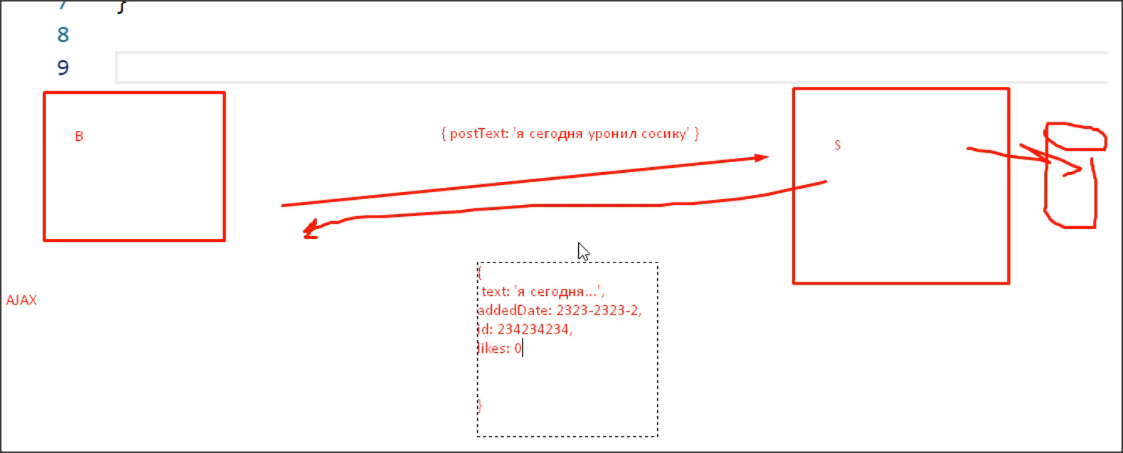
AJAX:

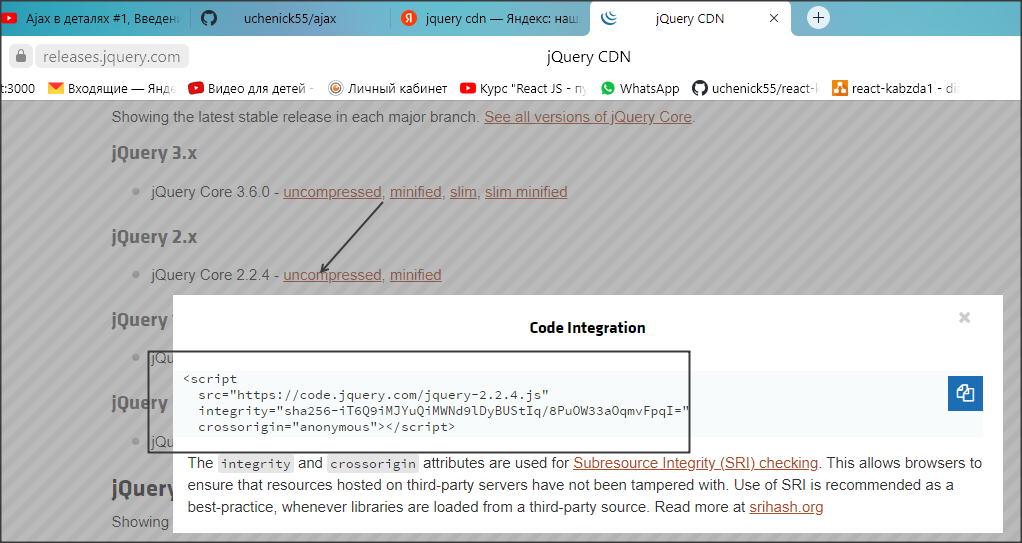
1. позволяет получать периодически или по запросу данные с сервера, не появилось ли что нового на сервере?
2. Отправлять новые данные на сервер и получать ответ



API – Application programming interface – интерфейс программирования приложений

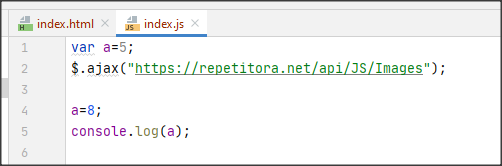
Endpoint – (конечная точка) – то к чему мы обращаемся.

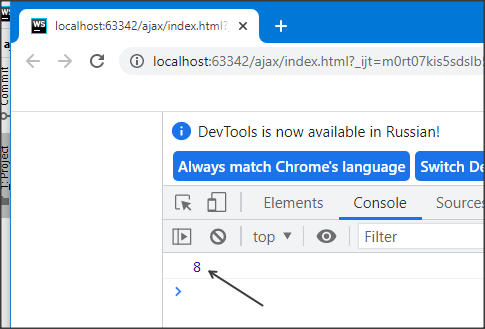
Создадим новый проект и подключим jquery cdn

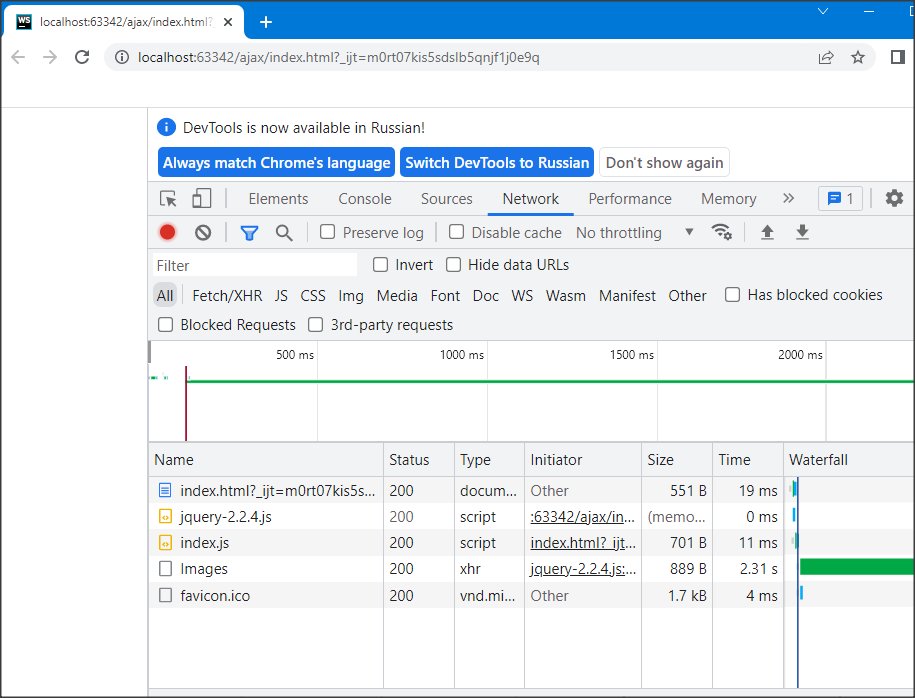


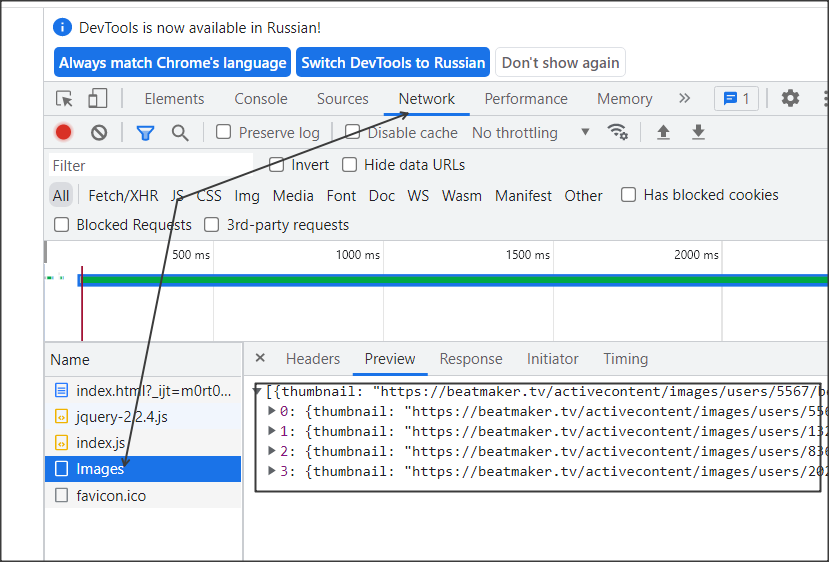


Сделаем первый запрос на https://repetitora.net/api/JS/Images









Git (новый проект ajax)

Ajax в деталях #2, JavaScript, success, callback

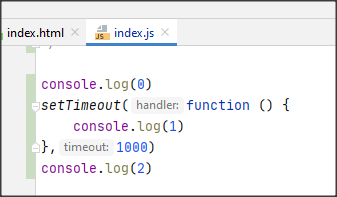
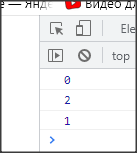
Callback используется часто в асинхронном коде.

Синхронный код – все выполнилось последовательно и закончилось.

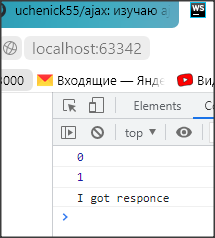
Асинхронный код – выполнение каких либо действий по завершению после выполнения каких либо условий, которые мы не знаем когда закончатся.

Асинхронный код – например нажатие кнопки, ввод текста, задержка по времени, успешный ответ от сервера и т.д. То есть некоторые действия мы запустим только после успешного завершения какого-либо другого действия, которое мы не знаем заранее, когда выполнится.

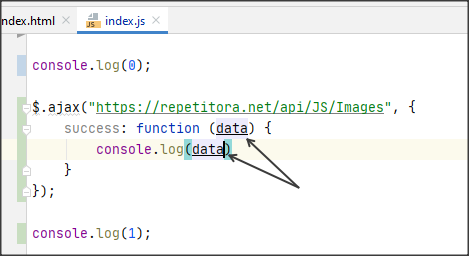
Например, здесь выведется 0, 2, 1 (1 выводится после секундной задержки)

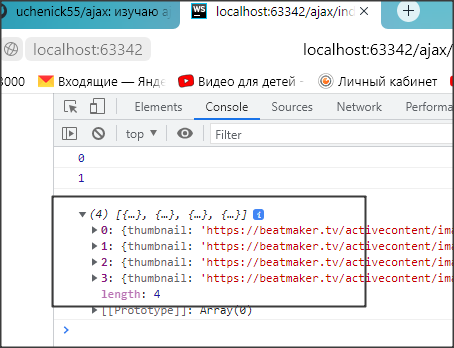
 

Аналогично делаем вызов coтыщду log, а после ответа от ajax выводим сообщение:

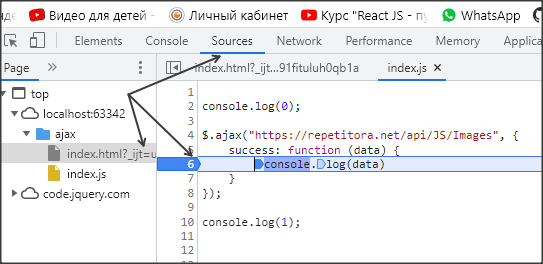


Ajax в ответе присылает данные (пропсы), которые мы можем так же вывести, и далее использовать:





Вместо console.log старайся использовать точки остановок



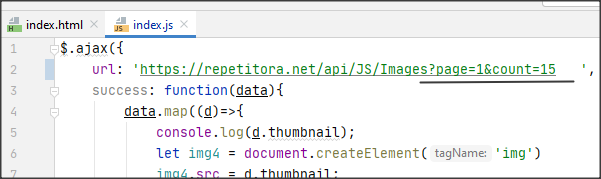
1. Отправляем запрос
2. В случае успеха мапим данные
3. Создаем элемент img и его src присваиваем элемент данных
4. Добавляем в конец боди каждую картинку

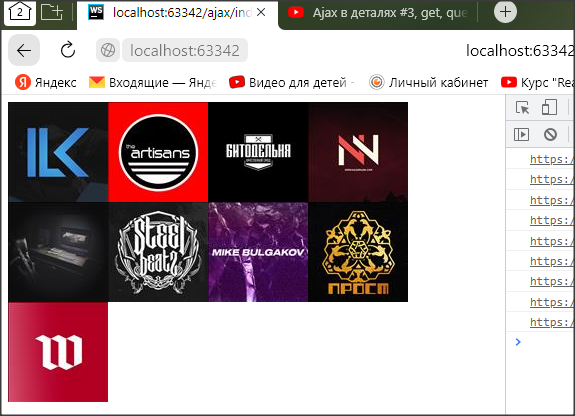


Git

Ajax в деталях #3, get, query params, get-параметры, query string

В запросах можно отправлять доп. данные





Конкатена́ция — операция склеивания объектов линейной структуры, обычно строк. Например, конкатенация слов «микро» и «мир» даст слово «микромир».

document.createElement('img') – создание элемента img (картинки)

HTML: <div id="result"></div>

JS: document.querySelector("#result") – привязать элемент (переменную) к выбранному id (независимо от тега div)

HTML: <button id="click-me">Click Me2</button>

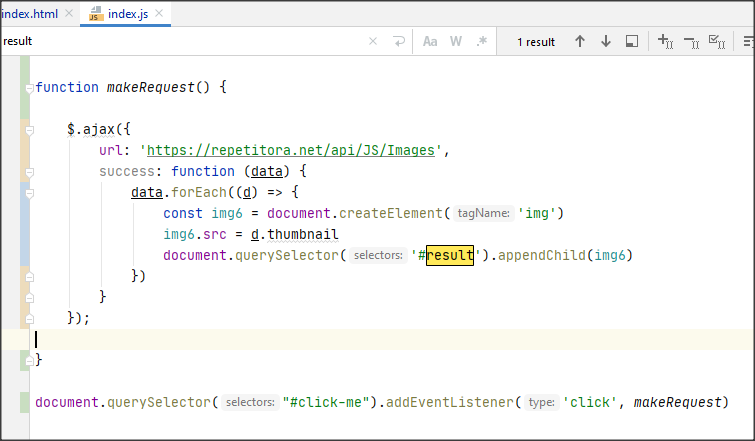
JS: document.querySelector("#click-me").addEventListener('click', *makeRequest*) запускает *makeRequest по клику в HTML теге* <button> *с id* click-me

*makeRequest () {…}*

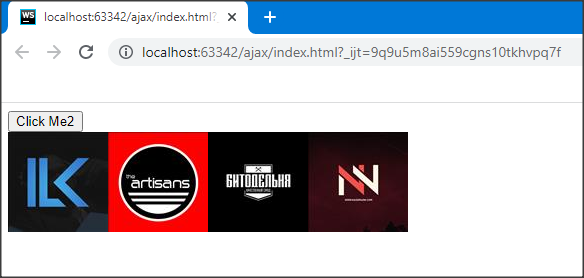
Задаем кнопку и тег вывода div



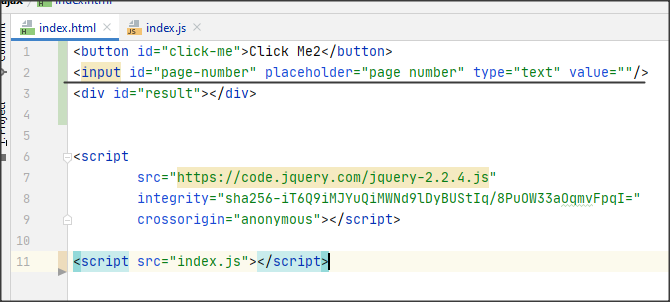
Запускаем функцию по клику



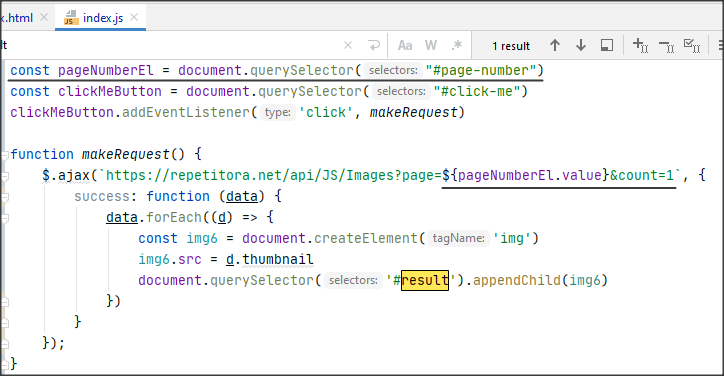
Вывод 4 изображений по умолчанию

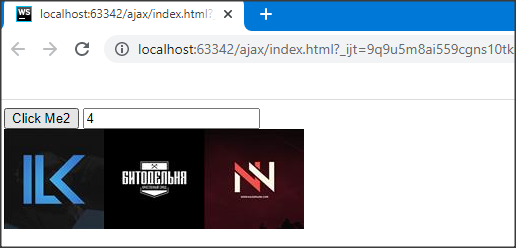


Добавим поле ввода номера страниц



Добавим отрисовку введенной страницы по 1 картинке



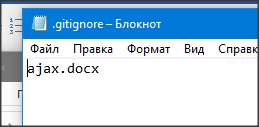


Git

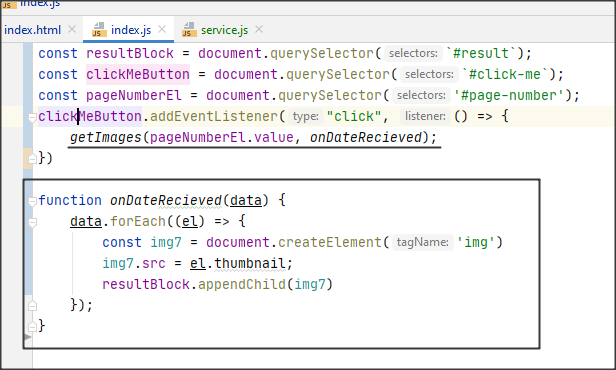
Ajax в деталях #4, ui-dal (data access layer)

Проблема – код не разделен на слои. Index js и обращается к северу, и обрабатывает DOM элементы. Разделим обязанности (single responsibility)

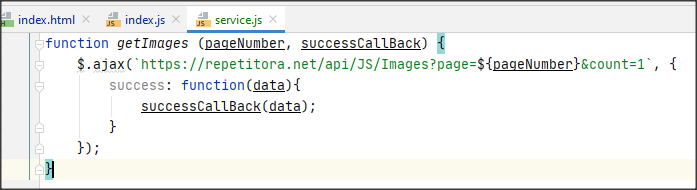
Добавим в git ignore ajax.docx



Создадим отдельный файл service js, который в пропсах будет передавать страницы, вводимые с кнопки PageNumber, и отдельную функцию onDataRecieved, обрабатывающую данные в DOM:



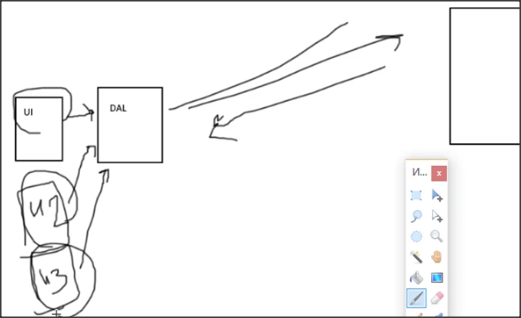
detImages при этом будет заниматься только отправкой и получением данных с сервака



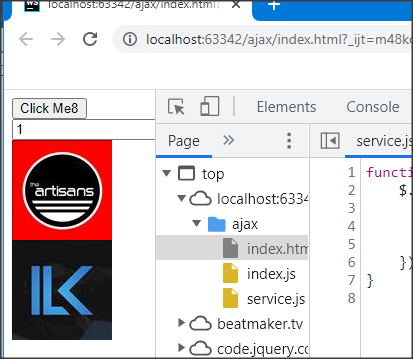
Подключаем service.js к index.html



Мы создали разделение на UI (index.js), и DAL (Data access layer service.js) каждая часть отвечает за свою функцию, и может быть заменена на другой аналогичный инструмент безболезненно



Работает



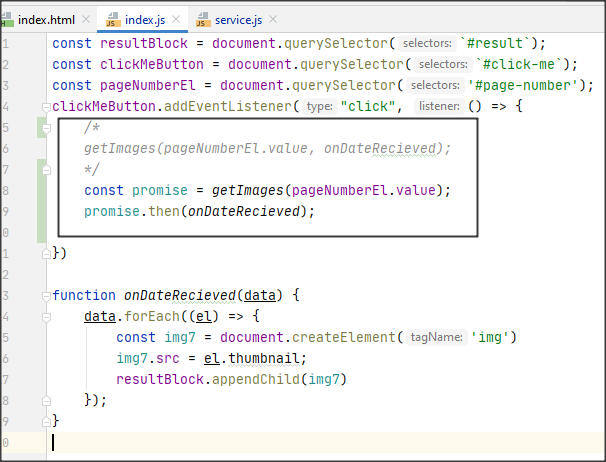
Git

Ajax в деталях #5, promise, axios

Заменим колбеки на промисы



Так же в index.js уберем колбек из пропсов и добавим промисы



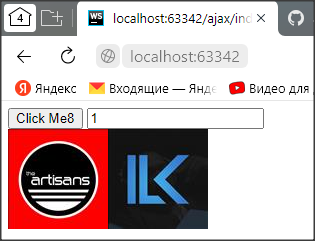
Все работает.

Теперь мы можем в DAL (Data Access Layer = service.js) подключить например другую библиотеку axios, и она будет без изменений работать с UI обработчиком (index.js). В этом весь смысл – в гибкости:





Все работает, при этом, наш index.js остался без изменений. Чистим нычки



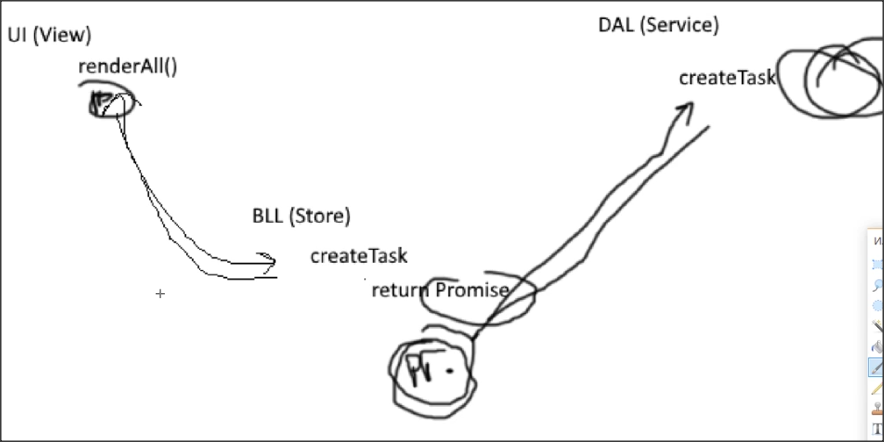
Git

Project 1, Tests for everyone - 003, Промисы, Promise, промис

Отклонимся в сторону, разберем промисы

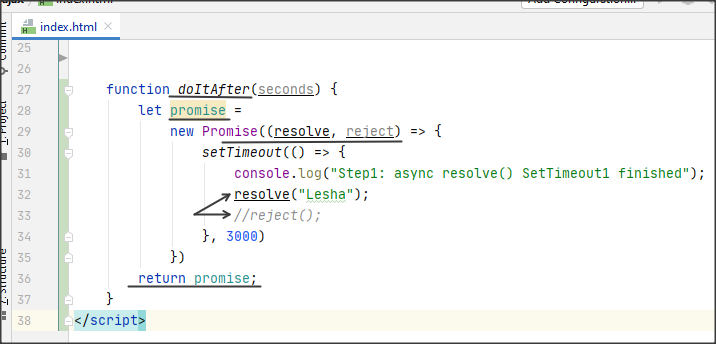
UI через BLL делает запрос на получение данных в DAL

Мы сразу ретурним промисы, не ожидая получения ответа. После его получения, данные последовательно проходят DAL->BLL->UI.



Формируем в BLL функцию doItAfter, в ней создаем переменную promis

Получение данных с сервера (DAL) имитируется setTimeout. Возвращаемый результат resolve и reject



Можно вместо let promise и return promise сразу делать return:



Resolve () соответствует успешному ответу от сервера.

Reject - неуспешному.

В теле UI создаем вызов функции через переменную prom:

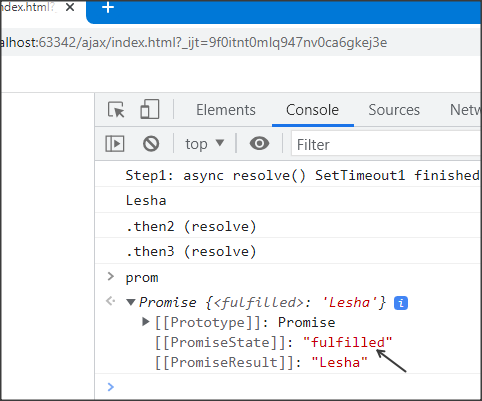


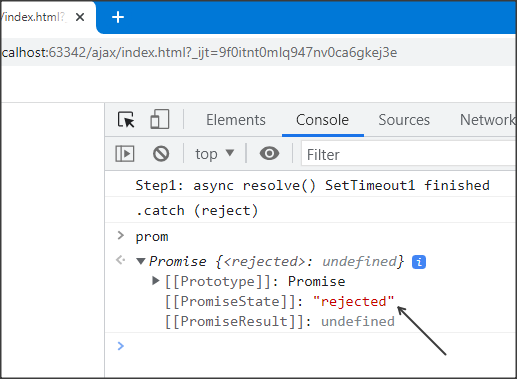
.then соответствует resolve

.catch – reject

Если мы получили ответ resolve, то невозможно в этом же запросе сменить resolve на reject.

Статус prom можно посмотреть в консоли





.then идет за .then последовательно.

Данные, полученные в resolve (или reject) могут быть переданы и выведены в UI:

