GeekBrains

Курс «Веб-технологии: уязвимости и безопасность»

Same Origin Policy

Что такое Origin, как его посмотреть и как он наследуется. Правила Same Origin Policy для DOM и XHR-запросов. Что такое Web Storage, правила SOP для него и для Cookie.

Оглавление

Итоги урока

Ссылки к уроку

Введение

В этом уроке мы разберем Same Origin Policy подробно для различных аспектов браузера.

План урока:

1) Что такое Origin, как его можно посмотреть и как он наследуется.

2) Правило Same Origin Policy для DOM и XHR-запросов.

3) Что такое Web Storage, какие правила SOP есть для него и для Cookie.

К концу урока вы будете понимать, как работает Same Origin Policy для DOM, XHR, Web storage и Cookie, что такое Origin, как в разных стандартных ситуациях у страниц наследуется Origin.

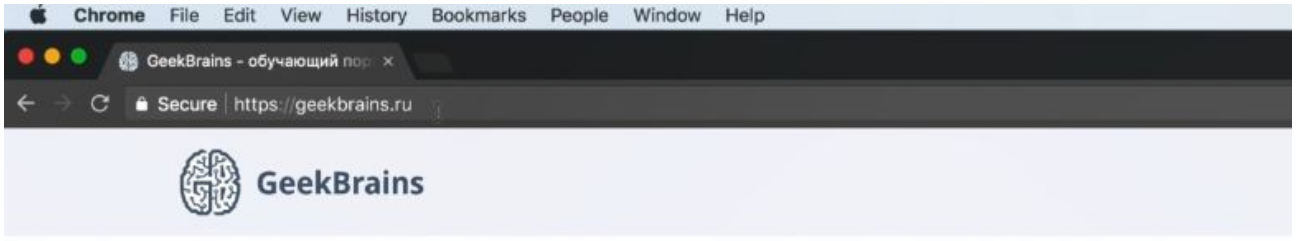
Видеоурок 1

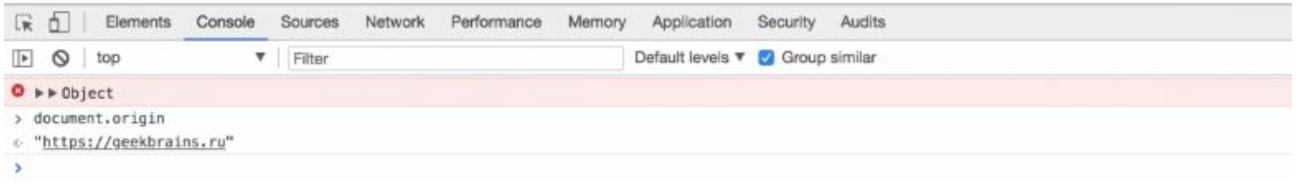
Что такое Origin

До этого момента мы говорили, что один ресурс от другого отличается условно тем, что на одном находится один домен, а на другом — другой. На самом деле понятие **Origin** включает в себя больше, чем просто домен: **Origin — схема + имя хоста + порт**.

Эта тройка и есть **origin**, например https://geekbrains.ru:443 — это один **origin**, а http://geekbrains.ru:80 — совсем другой, так как у них разные порты - 443(HTTPS) и 80(HTTP).

Чтобы посмотреть **origin** в браузере, откроем Chrome и зайдём на https://geekbrains.ru. Затем откроем инструменты разработчика (F12), перейдём в консоль, введём **document.origin** и нажмём Enter:

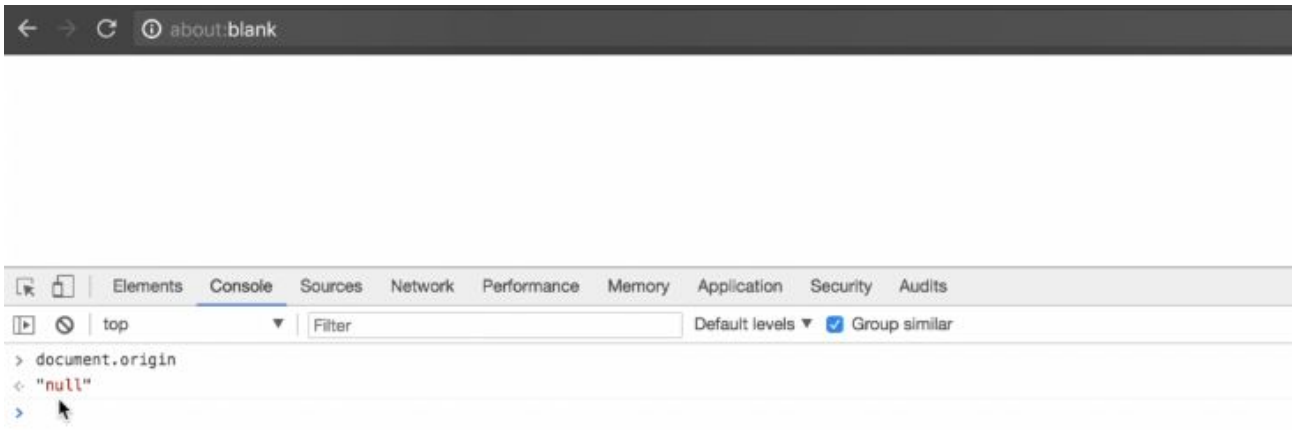




Его **origin** равен https://geekbrains.ru.

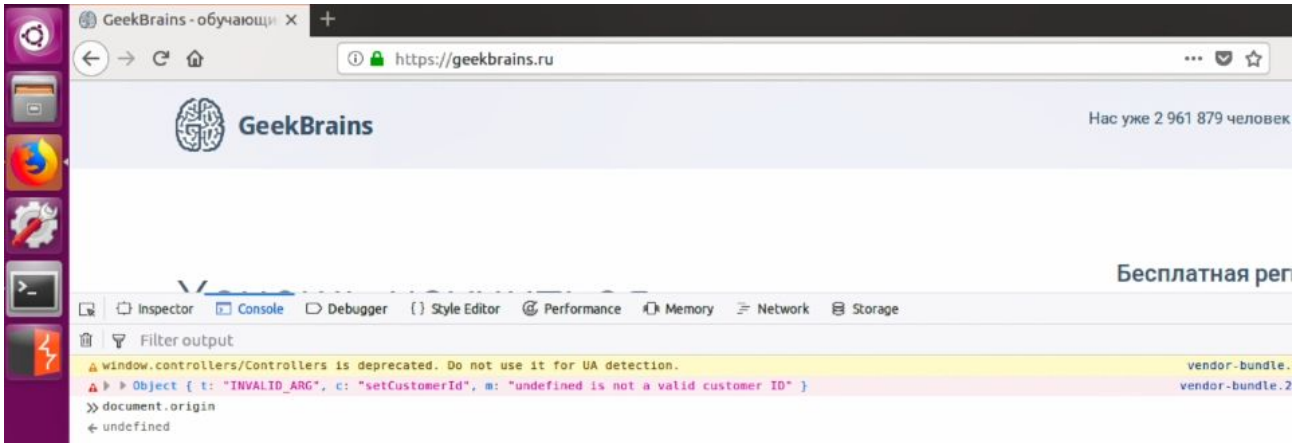
Origin = “null”

Также существует **origin=**“**null**”, например у пустой страницы **about:blank**:

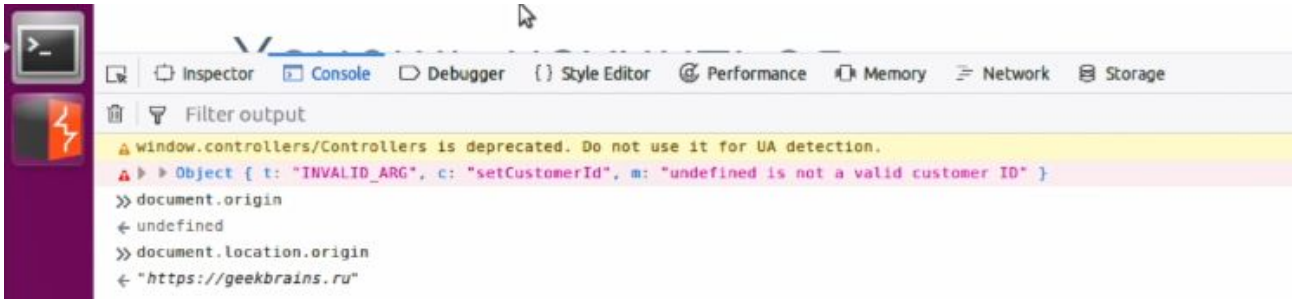


Он применяется, когда **origin** страницы невозможно определить, например если мы открываем пустую страницу. Также он используется, чтобы безопасно разграничить одну страницу от другой, когда мы применяем атрибут sandbox к iframe — во фрейме ставится origin “**null**” и фрейм не может сделать запрос к родительскому окну.

В Firefox **origin** смотрится немного по-другому. Откроем инструменты разработчика в Firefox и введем в консоли **document.origin**:



Выведется **undefined** — потому, что в Firefox origin находится в **document.location.origin**. Видим, что его **origin** также равен https://geekbrains.ru:



Практика

Посмотрим, какой **origin** будет у iframe. Откроем и создадим следующие файлы:

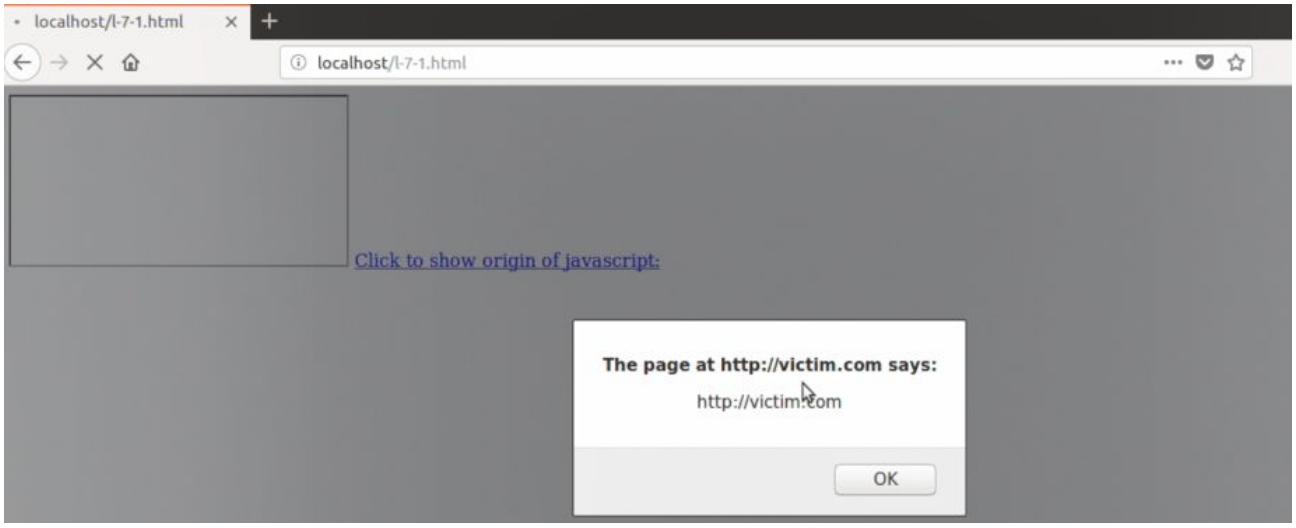
```
cd /var/www/html && sudo nano l-7-1.html
```

```
<iframe src="http://victim.com/showorigin.html"></iframe>
<iframe src="javascript:alert(document.location.origin)"></iframe>
<a href="javascript:alert(document.location.origin)">Click to show origin of
javascript:</a>
```

```
sudo nano showorigin.html
```

```
<script>
    alert(document.location.origin);
</script>
```

Сохраним все файлы. Откроем файл урока http://localhost/l-7-1.html в браузере:



Первым отработал первый фрейм, который включает http://victim.com/showorigin.html. У него естественным образом оказался **origin** victim.com, несмотря на то, что мы зашли на http://localhost.

Дальше нажмем на ссылку Click to show origin of javascript:

JavaScript-схема наследует **origin** от основной страницы. На самом деле это очень опасное поведение: если мы способны внедрить на страницу вредоносный код, например javascript:.. и дальше наш JavaScript-код, эти скрипты будут выполняться в контексте нашего домена, а это, как вы уже понимаете, большая проблема безопасности.

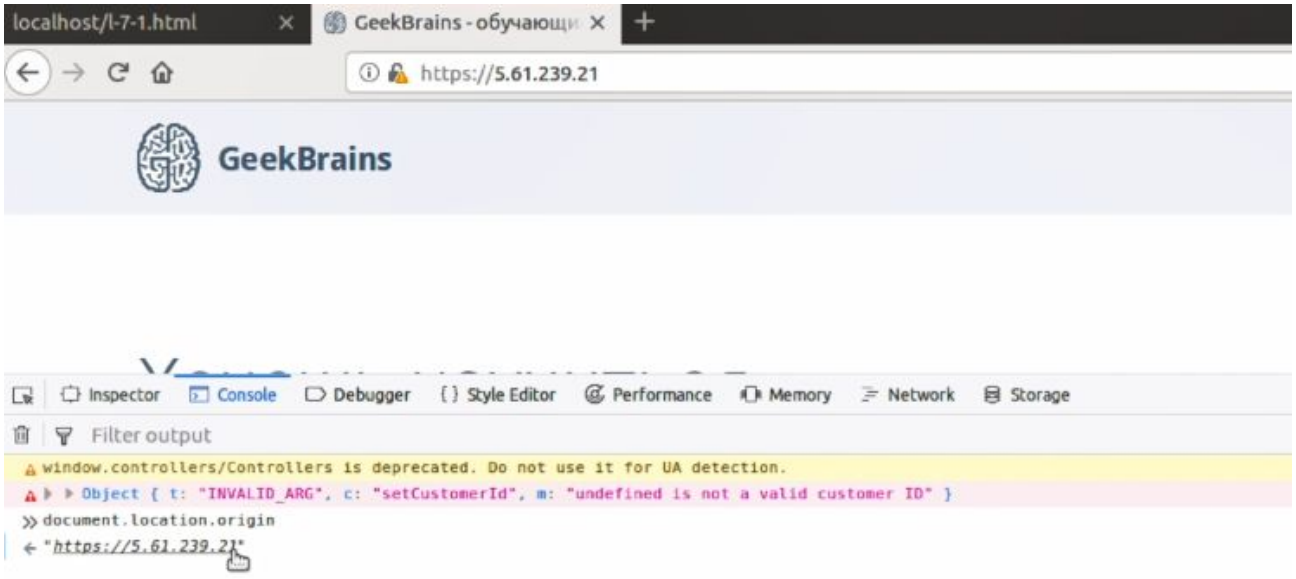
IP-адреса и Origin

Рассмотрим еще несколько пограничных случаев, где не всегда очевидно, как проставить **origin**.

Один из них — указание IP-адреса вместо доменного имени. Зачастую сервера настроены так, что по умолчанию отдается главная страница (с каким бы заголовком `Host` мы ни пришли на сайт).

Правильно будет отдавать только ту страницу, на которую пришел пользователь, Если пользователь пришёл на https://geekbrains.ru, отдавать нужно именно её, а если запросил страницу https://example.com, но при этом сделал запрос на https://geekbrains.ru, нужно ответить, что такой страницы нет на сервере, а не возвращать по умолчанию https://geekbrains.ru — это порождает проблемы с Same Origin Policy и множество других.

Вернемся к IP-адресам в имени хоста. Проверим, какой **origin** будет у URL `https://5.61.239.21`:



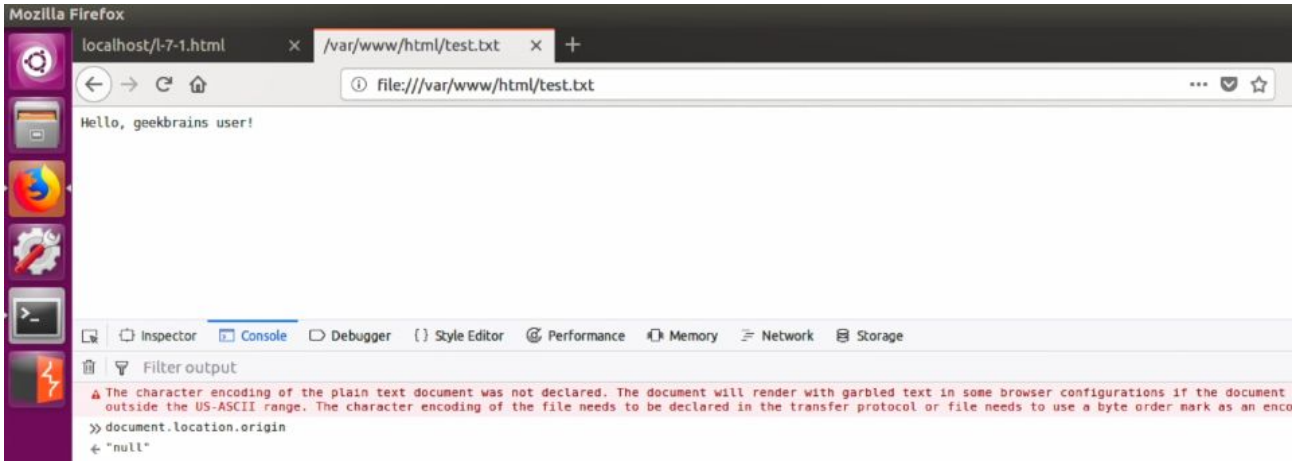
Origin равен `https://5.61.239.21`. Возникает такой момент: Cookie с IP-адреса **5.62.*** можно поставить на IP-адреса ***.239.21**, так как браузер воспринимает эти адреса как домены. Но, конечно же, на самом деле это не домен.

Рассмотрим имя домена geekbrains.ru: мы не можем получить up.geekbrains.ru так как не являемся представителем geekbrains.ru.

А вот IP-адреса, заканчивающиеся на ***.239.21**, например **5.62.239.21**, можно зарегистрировать и получить спокойно, т. к. это другие IP-адреса. Если рассматривать IP-адрес как доменное имя, IP-адрес **5.62.239.21** будет поддоменом IP-адреса ***..239.21** (и IP-адреса **5.61.239.21** домена geekbrains.ru). Вывод — обращение по IP-адресам лучше не разрешать.

Origin и схема file

Ещё один опасный случай — открытие файла с помощью схемы **file**, например file:///home/r/index.html. Проверим и посмотрим, какой **origin** будет у страницы file:///var/www/html/test.txt:



Введем **document.location.origin** — у схемы **file** origin=”**null**”.

Какое-то время назад это было не так: схема **file** позволяла JavaScript получить доступ вообще в любой файл на компьютере, то есть как будто бы для неё не существовало Same Origin Policy. И если злоумышленник попросил бы вас скачать файл, и вы его скачали и открыли, он имел бы доступ ко всему вашему браузеру, как будто **Same Origin Policy** не существовало.

Сейчас все крупные производители браузеров уже давно приняли, что у схемы **file://** origin="**null**" и поэтому JavaScript отсюда никуда не может достучаться. В браузерах это варьируется, то есть, например, в Firefox середины 2018 года можно было выйти за пределы этого файла, но нельзя прочитать документы из других доменов и других **origin**.

Итоги

Подведем итоги. Мы узнали:

1) Что такое **origin**, из чего он состоит, зачем нужен.
2) Посмотрели правила наследования **origin** для **iframe**, для схемы **javascript:** и для схемы **file://**.
3) Рассмотрели на практике различные опасные ситуации и научились выяснять **origin** документа в браузере.

Видеоурок 2

На этом уроке мы разберем **Same Origin Policy** для **DOM** и **XHR**-запросов.

1. Рассмотрим правила **Same Origin Policy** для **DOM**.
2. Разберем правила **SOP** для **XHR**-запросов.
3. Узнаем, как правильно ослаблять правила для **XHR**.

SOP для DOM

Рассмотрим, что может делать JavaScript-код с **DOM** другого документа. Под другим документом мы подразумеваем документ с другим **origin.**

Приведем пару примеров:

https://geekbrains.ru/a	OK — у них одинаковый **origin**, т. к. совпадает схема+домен+порт.
https://geekbrains.ru/	

https://geekbrains.ru/a	Не OK — у этих документов будут разные **origin**, потому что у них разные схемы https и http.
http://geekbrains.ru/	

http://geekbrains.ru:8080/a	Не ОК — это тоже разные **origin**, потому что различается порт.
http://geekbrains.ru:80/	

И это нужно помнить, потому что тройка — схема, хост и порт — должна совпадать в точности, чтобы **origin** был один и тот же.

SOP и document.domain

С **DOM** другого документа и другим **origin** ничего нельзя сделать, т. к. у JavaScript ограничен доступ к **DOM** другого документа — он может получить доступ только к **DOM** своего документа. Это основное правило **SOP**, которое позволяет обезопасить пользователя в браузере.

Ограничение возможно обойти с помощью корректировки **document.domain** — это, как можно догадаться из названия, текущий домен документа и переменная, соответственно, в эту переменную мы можем вносить необходимые изменения.

Представим, что у нас есть домены **login.example.com** и **payments.example.com**, оба принадлежат домену **example.com** и он может разрешить им общаться между собой.

Оба домена могут выставить общий домен через **document.domain=”example.com”**, тогда у них будут одинаковые **origin**, при условии, что схема и порт при этом совпадут). После этого **login.example.com** и **payments.example.com** получат доступ к DOM-дереву друг друга.

Однако тут есть особенность: если **login.example.com** и **example.com** хотят общаться, **example.com**, несмотря на то, что его **origin** равен **example.com**, должен явно проставить **document.domain=”example.com”**, чтобы стала возможна коммуникация между этими DOM.

Кросс-доменные взаимодействия через **document.domain** уже не делают, потому что есть куда более совершенные, безопасные и понятные технологии.

SOP для XHR

Перейдём к безопасности **XHR** и работе **Same Origin Policy** для **XHR**.

XHR-запросы можно делать только на тот же самый **origin**. Мы не можем отправить JavaScript на другой домен и загрузить оттуда документ. Разработчикам это очень часто неудобно и поэтому специально для них придумали небезопасную схему. Сначала мы рассмотрим, как делать нельзя, а потом — как можно.

XHR через JSONP

Опасная схема заключается в **JSONP**.

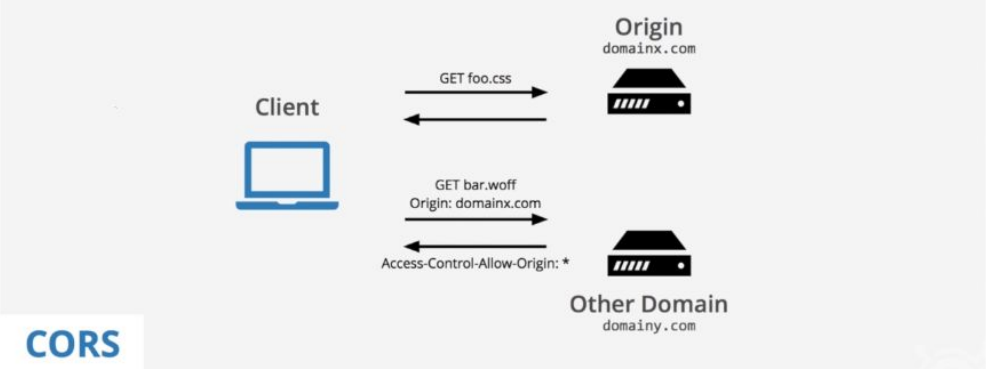
Сначала был **JSON** — строка, в которой сериализованы, то есть переведены в строку, объекты JavaScript. Затем к нему добавили **Padding** (padding в переводе с английского — подкладка, обёртка) и стало возможно обернуть **JSON** в функцию:

Когда мы обращаемся к **JSONP,** мы запрашиваем не данные, которые **Same Origin Policy** запретит нам получать с разных доменов, а JavaScript-код. А кросс-доменный JavaScript-код подгружать можно — мы так уже делали директивой **script src** и это можно, потому что в JavaScript обычно просто код, он открытый и в нем не должно содержаться данных пользователей.

JSONP грубо нарушает это правило и записывает данные пользователей в JavaScript-код. Это чревато тем, что абсолютно любой домен, который дернет ручку **JSONP**, сможет получить данные пользователей, потому что с каждым запросом в браузере уходит Cookie пользователя. Именно поэтому **JSONP** небезопасен. Есть случаи, когда **JSONP** безопасно используют, но сейчас это не нужно — это только ещё больше запутывает и добавляет угрозы безопасности в перспективе.

CORS

У нас есть хороший способ взаимодействия между доменами — **CORS** (**Cross Origin Resource Sharing**). Это технология, которая позволяет разным **origin** общаться между собой, посылать запросы и читать ответы — а значит, она ослабляет политики **Same Origin Policy**. Посмотрим, как работает **CORS**.



Сначала клиент может сделать запрос на domainx.com и получить ответ, но браузер его не воспримет, потому что **Same Origin Policy** запрещает делать запросы с разных доменов. Когда мы делаем запрос на domainy.com, он отвечает нам **CORS**-заголовком **Access-Control-Allow-Origin** и в нем указывается, какому **origin** разрешено читать запрос. Domainy.com отвечает заголовками **CORS** и браузер, если они разрешают этому **origin** читать ответы, разрешает ему это сделать.

Практика

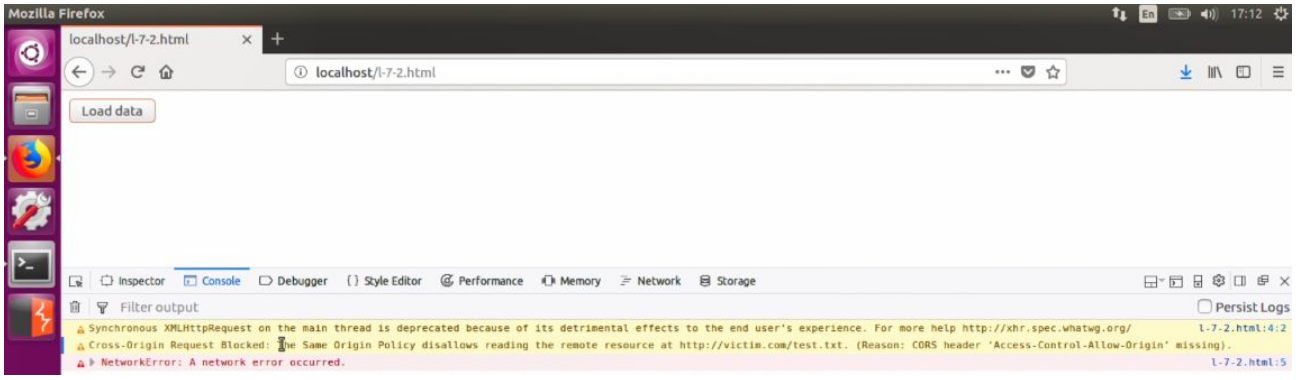
Посмотрим, как отработает без заголовков **CORS**-файл. Скопируем файл с предыдущих уроков и отредактируем его. Здесь делается **XHR**-запрос, давайте сделаем его кроссдоменным:

```
cd /var/www/html && sudo cp 1-5-4.html 1-7-2.html && sudo nano 1-7-2.html
```

```
<script>
function xhrTest(){
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://victim.com/test.txt", false);
  xhr.send();

  if (xhr.status != 200) {
     alert(xhr.status + ': ' + xhr.statusText);
  } else {
     alert(xhr.responseText);
  }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Зайдем в браузер и откроем инструменты разработчика, чтобы увидеть, что происходит.

Нажимаем кнопку **Load data**:



Cross-Origin Request Blocked — значит, наш запрос к ресурсу заблокирован.

Заголовок Access-Control-Allow-Origin

Включим заголовок **Access-Control-Allow-Origin** в конфигурации веб-сервера:

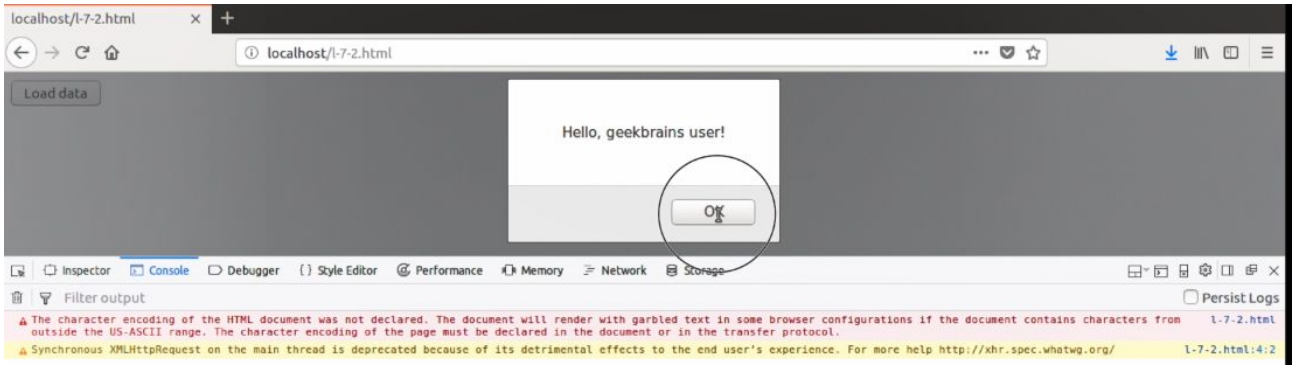
```
cd /etc/nginx/sites-enabled && sudo nano default
```

```
        root /var/www/html;

        # Add index.php to the list if you are using PHP
        index index.html index.htm index.nginx-debian.html;

        server_name _;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
                Add_header Access-Control-Allow-Origin "http://localhost";
                try_files $uri $uri/ =404;
        }
```

Сохраним его и перезагрузим **Nginx**:

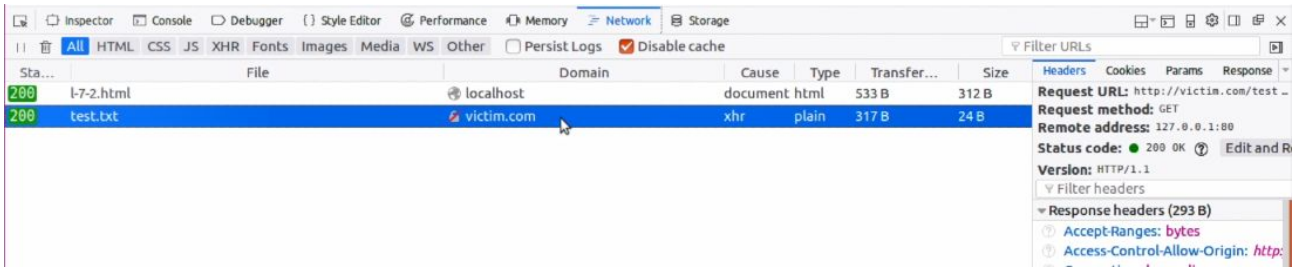
```
sudo nginx -s reload
```

Заново загрузим страницу, нажимаем **Load data**:



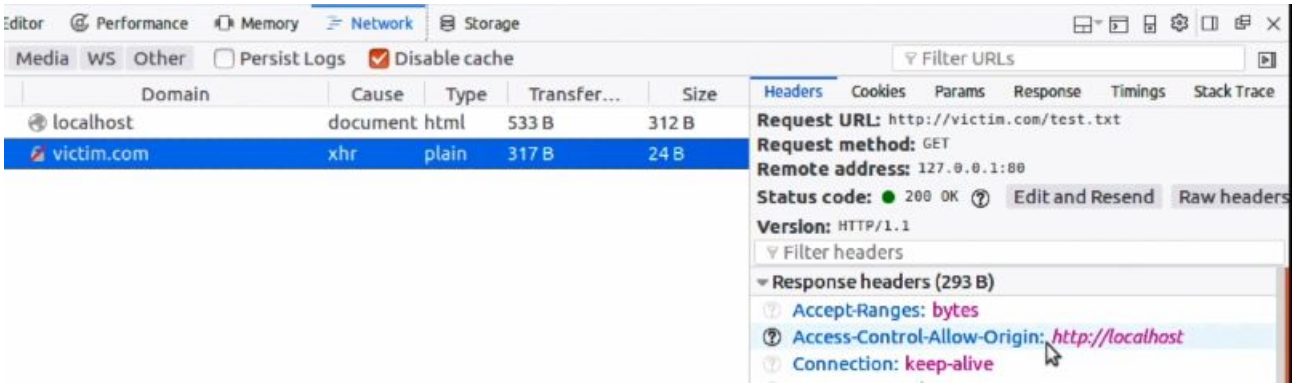
В этот раз данные успешно загрузились.

Во вкладке **Network** посмотрим на запрос за **test.txt**:



Он ушел на другой домен.

Это возможно благодаря заголовку **Access-Control-Allow-Origin**:



Он означает в буквальном смысле разрешить читать ответы **origin** http://localhost, так как **origin** — http://localhost, ответы разрешено читать.

Также сюда можно указать звездочку (*) - это будет означать, что любой домен может читать ответы. Но нужно быть аккуратным, потому что не всегда нужно, чтобы любой домен смог прочитать ответ.

Заголовок Access-Control-Allow-Credentials

```
cd /etc/nginx/sites-enabled && sudo nano default
```

```
        root /var/www/html;

        # Add index.php to the list if you are using PHP
        index index.html index.htm index.nginx-debian.html;

        server_name _;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to displaying a 404.
                add_header Access-Control-Allow-Origin "http://localhost";
                add_header Access-Control-Allow-Credentials true;
                try_files $uri $uri/ =404;
        }
```

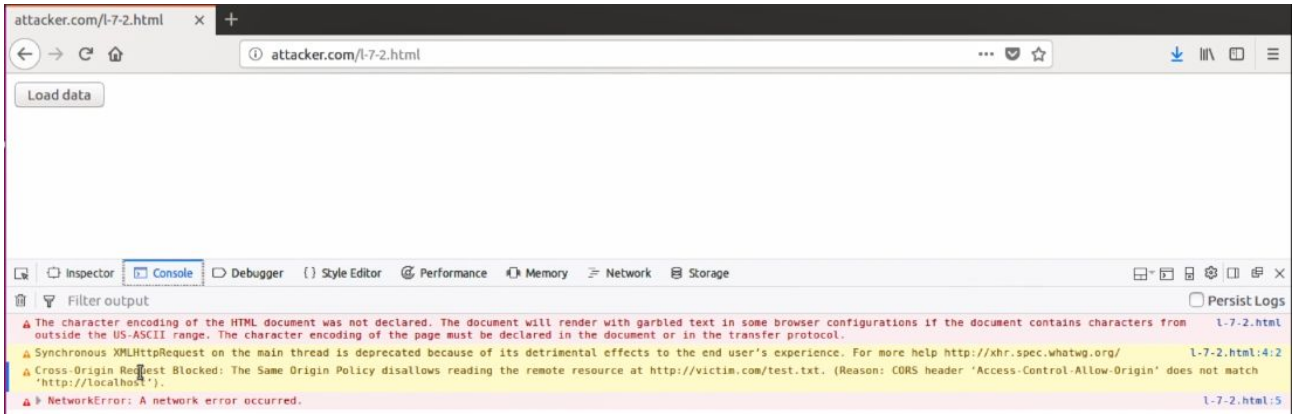
Заголовок `Access-Control-Allow-Credentials` принимает два значения — **true** и **false**. Этот заголовок говорит браузеру разрешено ли JavaScript-у читать данный ответ, когда запрос был выполнен с параметром `xhr.withCredentials = true`.

JavaScript не может прочитать ответ от сервера в случае, когда `xhr.withCredentials = true` и `Access-Control-Allow-Credentials: false`. Во трех других случаях браузеру разрешено читать ответ.

Возникает критичная ситуация, более подверженная уязвимости. Когда мы делаем запрос с Cookie, мы подразумеваем, что вернутся данные, привязанные к пользователю, а их потеря всегда очень страшна. Поэтому мы не хотим это разрешать кому попало и должны четко следить за списком доменов, которые попадают в заголовок **Access-Control-Allow-Origin**.

Если мы укажем звездочку в **Access-Control-Allow-Origin**, то есть разрешим любому домену обходить **SOP**, и при этом **Access-Control-Allow-Credentials** установлен в **true**, Cookies не будут отправляться. Когда разработчики **CORS** делали стандарт, они посчитали, что никто не разрешит отключать **Same Origin Policy** для своего домена: если любой домен сможет ходить с куками к другому и читать ответ при этом, по сути не работает **Same Origin Policy**. И они ввели правило, что если **Access-Control-Allow-Origin** — звездочка, **Access-Control-Allow-Credentials** игнорируется.

Введём http://attacker.com/l-7-2.html и нажмём **Load data**:



Запрос заблокируется. Это произошло потому, что в заголовке **Access-Control-Allow-Origin** стоит http://localhost, а **origin** у localhost и attacker.com совершенно разные — и браузер запретил чтение ответа.

Итоги

Подведём итоги. Вы узнали:

1) Правила **Same Origin Policy** для DOM.
2) Правила **Same Origin Policy** для **XHR**.
3) Как ослабить **Same Origin Policy** для **XHR** с помощью заголовков **CORS**.

4) Как не нужно ослаблять SOP с помощью **JSONP** и почему это небезопасно.

Видеоурок 3

В этом видео мы узнаем:

1) Что такое **Web storage** и зачем он нужен.
2) Какие правила **Same Origin Policy есть** для **Web storage**.
3) **Same Origin Policy** для **Cookie**.

Web storage

Web storage нужен, чтобы сохранять данные между запусками браузера. Допустим, JavaScript хочет сохранять данные о пользователе, но при этом не отправлять их с каждым запросом. Поэтому Cookie тут не помогут, но **LocalStorage** подходит идеально, он был разработан специально для этого.

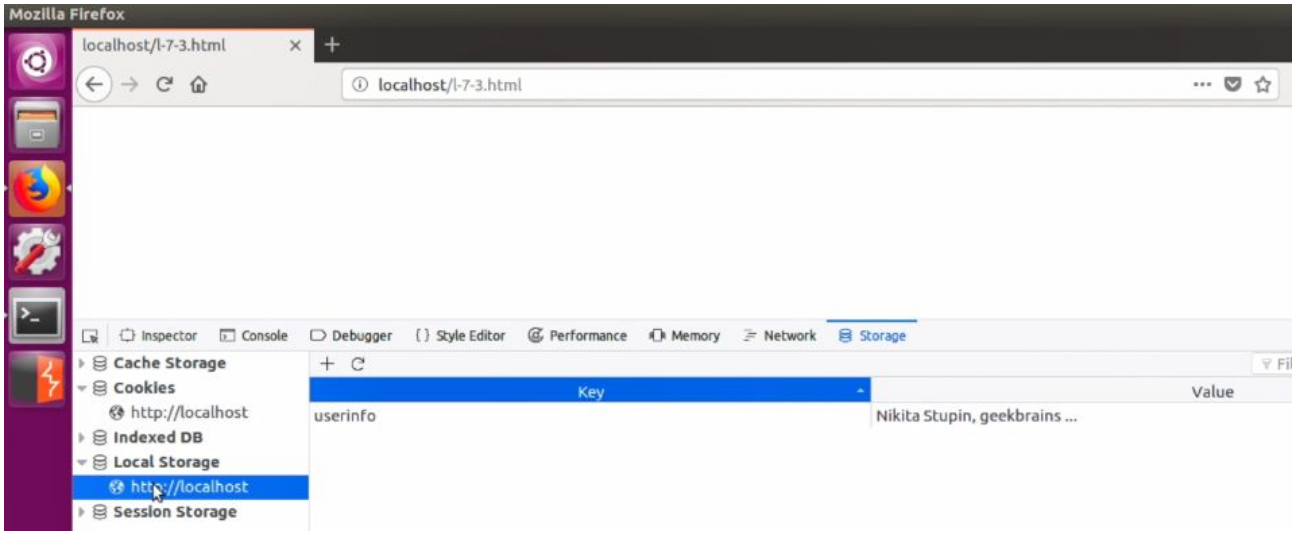
LocalStorage — хранилище, в которое JavaScript может записать данные; они сохраняются в пределах одной либо нескольких сессий браузера. Когда мы закроем браузер или вкладку, данные сохранятся и будут доступны при повторном открытии.

Чтобы взаимодействовать с persistent **localStorage** (persistent — постоянный; это данные, которые сохраняются между сессиями браузера), есть специальный объект **localStorage**. У него есть метод **setItem()** и мы можем добавить **item**, то есть элемент, который станет парой «ключ-значение». И это будет, допустим, ключ — **userinfo**, а значение — информация о пользователе. Откроем файл и напишем скрипт:

```
cd /var/www/html && sudo nano 1-7-3.html
```

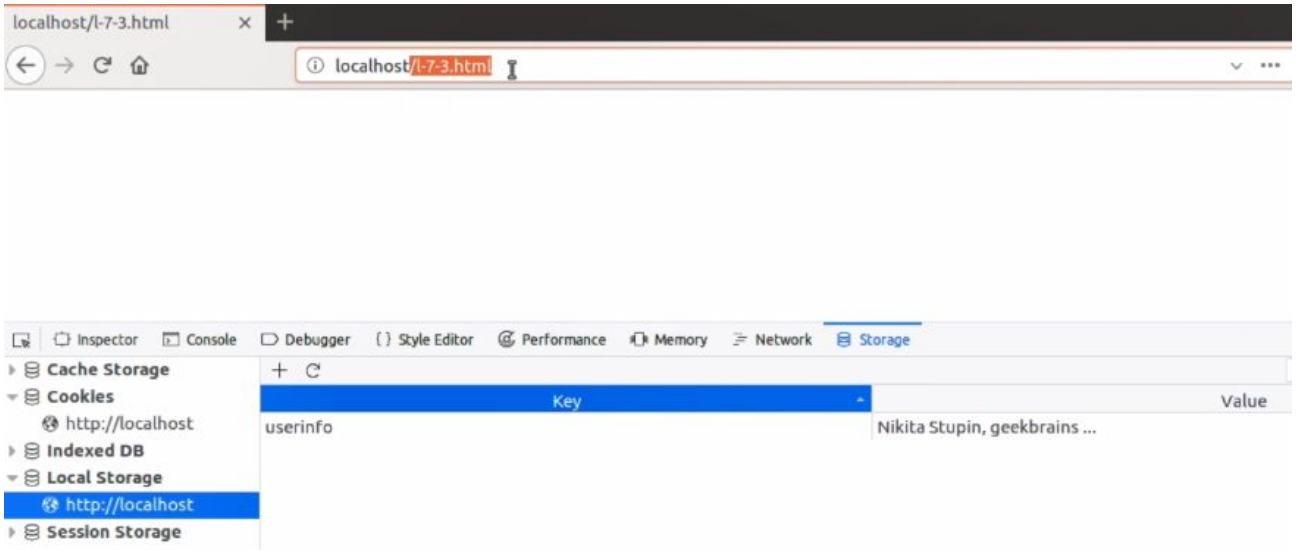
```
<script>
  localStorage.setItem("userinfo", "Pavel Statsenko, geekbrains ...");
</script>
```

После этого данные сохранятся. Мы убедимся в этом, если откроем браузер и перейдем в консоль разработчика. **Storage** — **Local Storage** — и видим записанные ранее данные:



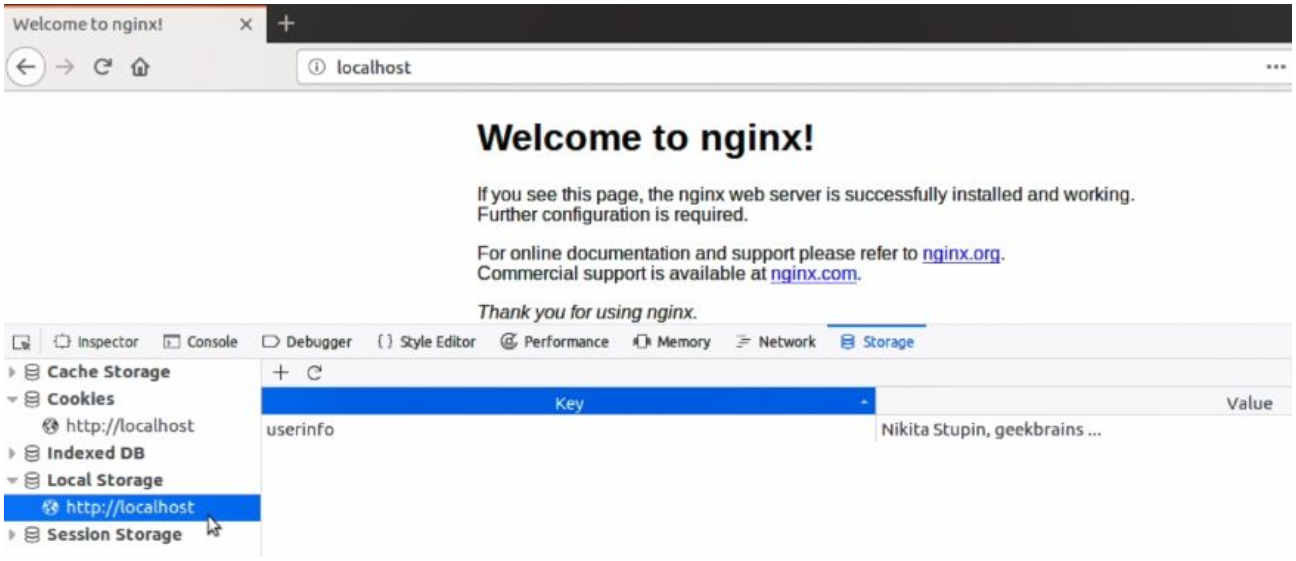
В **Local Storage** сохранились данные, которые мы туда записали.

Теперь закроем браузер и убедимся, что они остались:



После переоткрытия браузера эти данные остались, причем они установлены для всего **origin**.

Если мы перейдем на http://localhost, увидим, что для него они тоже установлены:

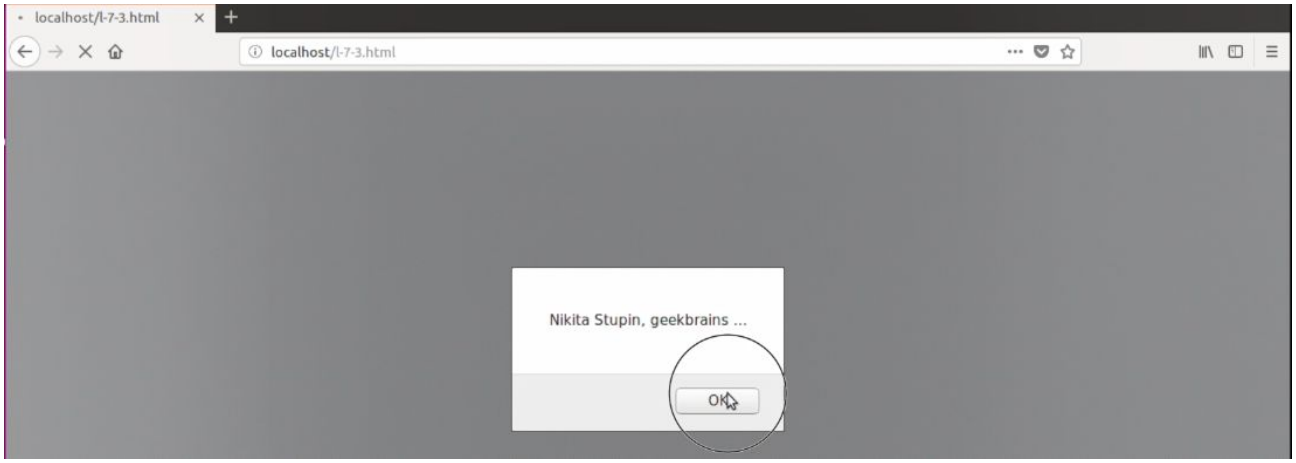


Попробуем прочитать что-то из **localStorage** и, например, удалим их после этого:

```
cd /var/www/html && sudo nano l-7-3.html
```

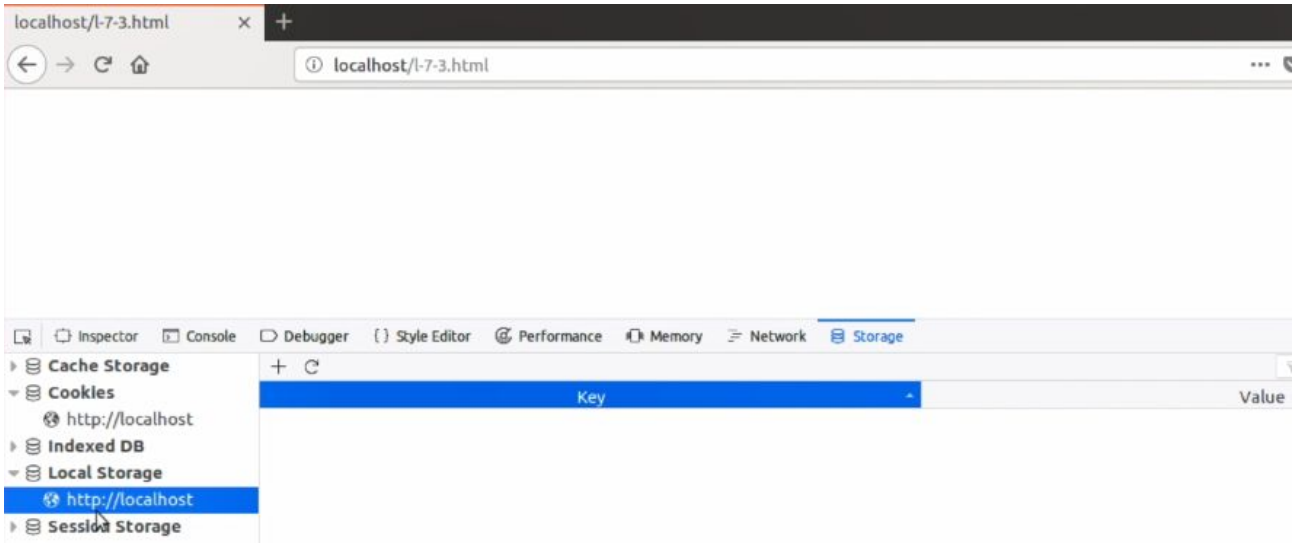
```
<script>
  localStorage.setItem("userinfo", "Pavel Statsenko, geekbrains ...");
  alert(localStorage.getItem("userinfo"));
  localStorage.removeItem("userinfo");
</script>
```

Проверим, что все работает: откроем Firefox и перейдем на страницу:



Alert() высвечивает сохраненные данные.

Проверим, удалились ли они из **localStorage**:



В **localStorage** нет ключей.

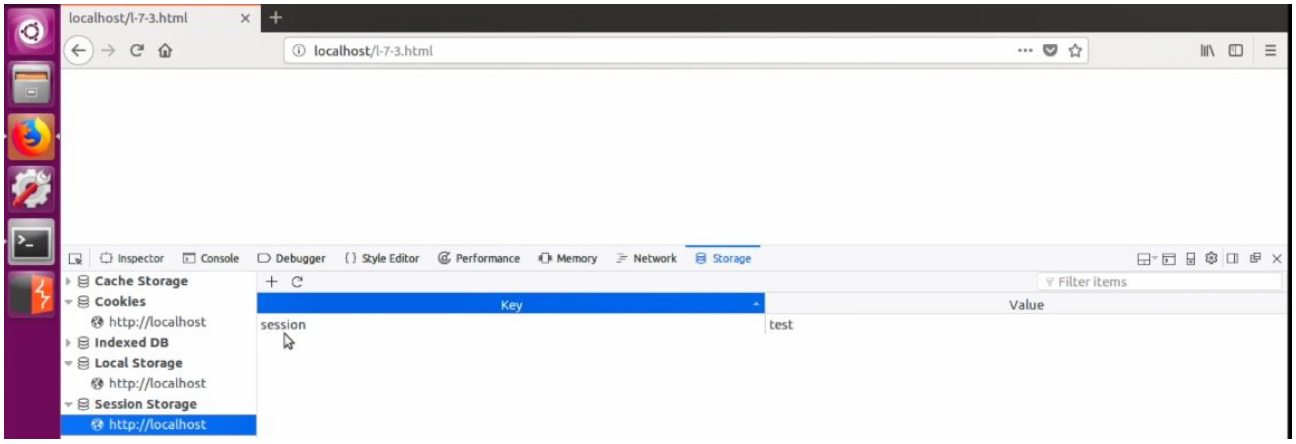
Посмотрим, как будет работать **sessionStorage**, чтобы сохранять данные только на эту сессию браузера (после закрытия браузера или вкладки они будут удаляться).

Воспользуемся **sessionStorage**, интерфейс точно такой же, как и у **localStorage**:

```
cd /var/www/html && sudo nano l-7-3.html
```

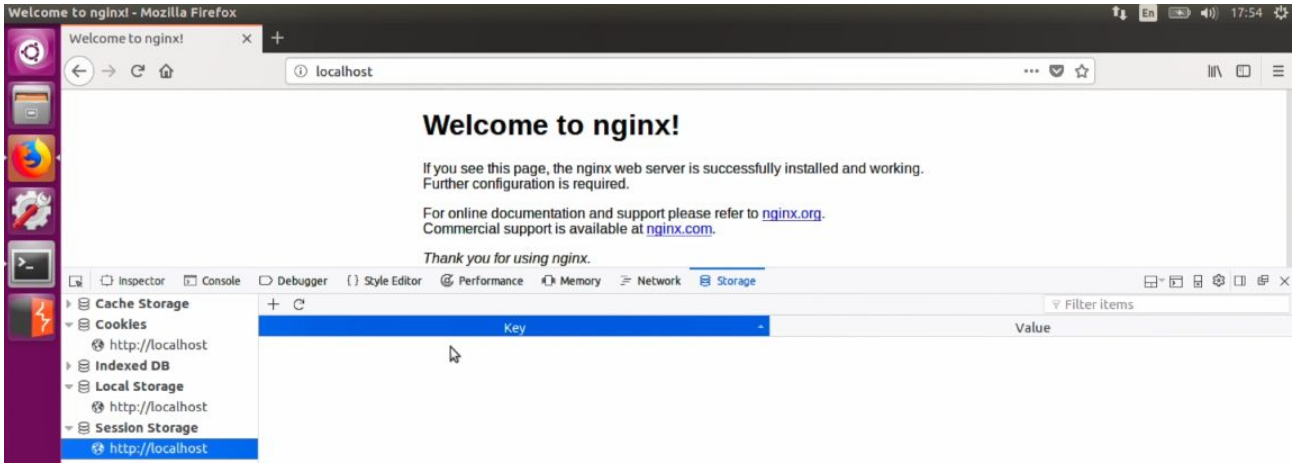
```
<script>
  localStorage.setItem("userinfo", "Pavel Statsenko, geekbrains ...");
  sessionStorage.setItem("session", "test");
</script>
```

Перезагрузим страницу:



В **Session Storage** тоже появился ключ.

Закроем браузер и откроем заново:



В **localStorage** остался ключ, а в значении **sessionStorage** ничего не осталось — результат такой, как и ожидался.

В целом, что касается правил **Same Origin Policy** для **Web Storage**: JavaScript не сможет записать или прочитать данные из **Web Storage** другого **origin** — правила такие же, как для **DOM**. Нужно обязательно это знать, потому что **Web Storage** активно используют JavaScript-разработчики.

SOP для Cookie

SOP для Cookie заключается в том что, например, домен a.com не может поставить Cookie на домен b.com, потому что у них разные **origin**. Но с Cookie не все просто — например, дочерний домен может поставлять их на родительский, это не возбраняется. **Same Origin Policy** для Cookie немного другой:

Cookie set at *foo.example.com*, *domain* parameter is:	**Scope of the resulting cookie**	
	Non–IE browsers	**Internet Explorer**
(value omitted)	*foo.example.com* (exact)	**.foo.example.com*
bar.foo.example.com	Cookie not set: domain more specific than origin	
foo.example.com	**.foo.example.com*	
baz.example.com	Cookie not set: domain mismatch	
example.com	**.example.com*	
ample.com	Cookie not set: domain mismatch	
.com	Cookie not set: domain too broad, security risk	

В этой таблице варианты по установленному параметру **domain** и две колонки по типу браузера: не Internet Explorer и старая версия Internet Explorer. Например, мы начали поставлять куки на foo.example.com. Если параметр **domain**, допустим пустой (первая строчка в таблице — value omitted), туда проставляются Cookie в зависимости от типа браузера: для старого IE — `*.foo.example.com`, для других — `foo.example.com`. Разберем по порядку:

1) Мы зашли на foo.example.com — Cookie проставляются с пустым доменом, значит, в обычных браузерах они проставятся на foo.example.com, а в IE — еще и на все поддомены *.foo.example.com.

2) Если значение параметра **domain** на Cookie равно bar.foo.example.com, то на страничке foo.example.com мы не сможем проставить куки, потому что родительский домен не может сделать это на дочерние, т. к. это запрещено **Same Origin Policy**.

3) Если параметр домена **domain** на Cookie равен foo.example.com, то есть будет в точности совпадать, Cookie проставятся на этот домен и его поддомены.

4) Если домен третьего уровня или ниже будет отличаться — **origin** не совпадает и мы не можем проставить Cookie, даже несмотря на то, что домен 2-го уровня совпадает.

5) Если мы укажем родительский домен example.com, Cookie проставятся на все поддомены родительского домена.

6) Если мы укажем совсем другой домен, **Same Origin Policy** скажет, что так нельзя делать.

7) При указании домена верхнего уровня браузеры отказываются проставлять куки.

Итоги

Подведем итоги:

1) Узнали, что такое **Web Storage** и зачем он нужен.

2) Узнали о двух видах **Web Storage**:

    a) **Local Storage**, который сохраняет данные на протяжении нескольких запусков браузера, пока мы полностью не очистим данные браузера, либо пока JavaScript не удалит отдельные элементы (item), добавленные в **Local Storage**.

    b) **Session Storage**, который хранит данные в пределах одной сессии браузера, то есть пока браузер или текущая вкладка открыты.

3) Рассмотрели правила **Same Origin Policy** для **Web Storage**.

4) Разобрали правила **Same Origin Policy** для **Cookie**, поняли, почему с одного домена они не могут проставляться на другой, почему с родительского мы не можем проставить Cookie на дочерний и почему возможно это сделать в обратную сторону.

Итоги урока

В следующем уроке нас ждет работа с современными ClientSide-технологиями:

1) Познакомимся глубже с **CORS**, узнаем, какие уязвимости у него бывают, как от них защищаться, как сделать безопасный **CORS**.

2) Разберем **postMessage** — это технология общения между браузерными вкладками или окнами, например между двумя вкладками в браузере или между окном и iframe. Изучим основные уязвимости, связанные с **postMessage**.

3) Разберём протокол **WebSocket** — он используется для постоянного соединения клиента и сервера, через которое без дополнительной нагрузки будут передаваться произвольные данные.

4) Подведём итоги курса и познакомимся с основными уязвимостями веба: удаленным выполнением кода RCE, SQL-инъекциями, CSRF, переполнением буфера и другими.

Ссылки к уроку

1. Кроссдоменные ограничения и их обход.
2. Same-origin policy.
3. Небезопасный cross-origin resource sharing.
4. Cross-Origin Resource Sharing (CORS).
5. LocalStorage, sessionStorage.