



HTTP



ВЛАДИМИР ЯЗЫКОВ



ВЛАДИМИР ЯЗЫКОВ

Основатель UsefulWeb



neizerth@gmail.com



[@neizerth](https://t.me/neizerth)



ПЛАН ЗАНЯТИЯ

1. [HTTP](#)
2. [Server](#)
3. [XMLHttpRequest](#)
4. [Same-Origin Policy & CORS](#)
5. [Heroku](#)



HTTP

ЗАДАЧА

Мы делаем лендинг-страницу и перед нами стоит задача отправлять форму контактов на сервер:

Имя

Телефон

Записаться

КЛЮЧЕВЫЕ МОМЕНТЫ

Для лучшего понимания того, что происходит, начиная с этой лекции мы с вами также будем писать серверную часть для своих приложений:

- HTTP Server
- REST API
- SSE
- WebSockets

Но мы будем делать это в максимально упрощённой форме (quick and dirty), т.е. без баз данных, валидации, безопасности, правильной организации кода и т.д.

РАЗМЕТКА

```
1 <div data-widget="subscribe">
2   <form data-id="subscribe-form">
3     <input data-id="name" name="name" required>
4     <input data-id="phone" name="phone" required>
5     <button>Записаться</button>
6   </form>
7 </div>
```

Получать значения из формы, валидировать её и оформлять вы уже умеете, поэтому мы для простоты опустим эти моменты.

PREVENT DEFAULT

Момент первый: если не отменить поведение по умолчанию, то это приведёт к отправке формы и последующей перезагрузке страницы, чего мы хотим избежать.

```
1  const subscribeWidget = document.querySelector('[data-widget=subscribe]');
2  const subscribeForm = subscribeWidget.querySelector('[data-id=subscribe-form]');
3  const nameInput = subscribeWidget.querySelector('[data-id=name]');
4  const phoneInput = subscribeWidget.querySelector('[data-id=phone]');
5
6  subscribeForm.addEventListener('submit', (evt) => {
7    evt.preventDefault();
8  });
```




SERVER

SERVER

Момент второй: куда нам отправлять данные? Мы, конечно, можем их отправлять на какой-нибудь «демо-сервер» вроде httpbin.org, но это не серьёзно, нам нужен собственный сервер.

Напишем его на базе стандартной библиотеки Node.js:

```
1  const http = require('http');
2  const server = http.createServer((req, res) => {
3    console.log(req);
4    res.end('server response');
5  });
6
7  const port = 7070;
8  // слушаем определённый порт
9  server.listen(port, (err) => {
10    if (err) {
11      console.log('Error occurred:', error);
12      return;
13    }
14    console.log(`server is listening on ${port}`);
15  });
```

SERVER

Итак, мы написали простой http-server, который работает на порту 7070 и отвечает на все HTTP-запросы одной строкой 'server response' (можем проверить это в браузере).

Т.е. фактически мы используем предоставляемое Node.js API, и всё, что мы сделали — это написали callback, который принимает два объекта:

- `req` — запрос
- `res` — ответ

Метод `res.end(data)` позволяет отправить данные клиенту.

Детальное описание модуля `http` можно найти на [странице документации](#).

SERVER

Создадим отдельный проект (`npm init`) и установим пакет `forever`, который будет автоматически перезапускать сервер при изменении в файлах:

```
npm init
npm install forever // не Dev
```

```
1  "scripts": {
2    "start": "forever server.js",
3    "watch": "forever -w server.js",
4  }
```

А также создадим файл `.foreverignore`, чтобы указать, что не нужно следить за каталогом `node_modules`:

```
node_modules
```



HTTP 1.1

Протокол, работающий в формате запрос–ответ с двумя участниками общения:

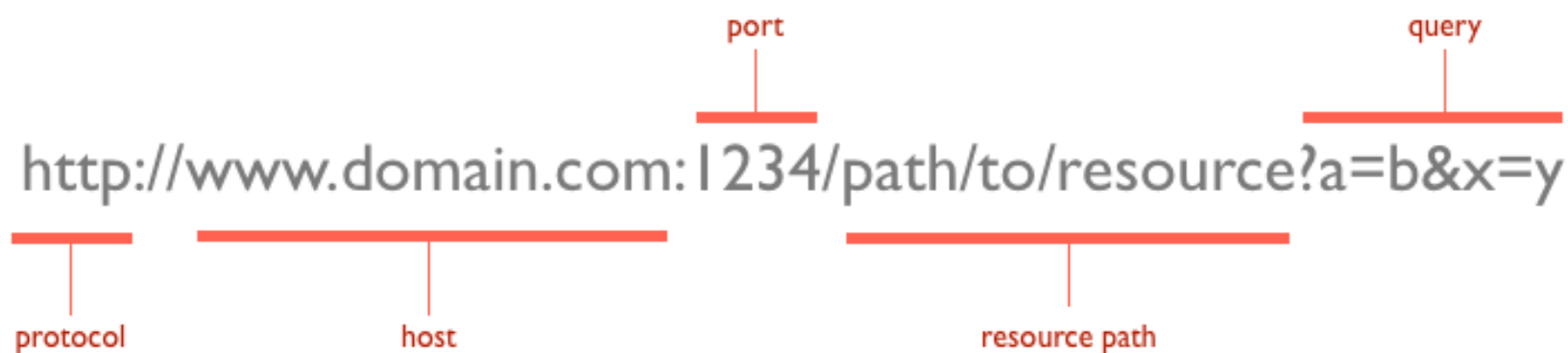
- клиент (формирует запросы, обрабатывает ответы)
- сервер (обрабатывает запросы, формирует ответы)

HTTP 1.1

Запрос и ответ представляют из себя текстовые запросы, состоящие из заголовков и тела:

Name	× Headers Preview Response Cookies Timing
localhost	▼ General
favicon.ico	Request URL: http://localhost:7070/ Request Method: GET Status Code: 200 OK Remote Address: [::1]:7070 Referrer Policy: no-referrer-when-downgrade
	▼ Response Headers view parsed
	HTTP/1.1 200 OK Date: Wed, 20 Mar 2019 19:46:16 GMT Connection: keep-alive Content-Length: 15
	▼ Request Headers view parsed
	GET / HTTP/1.1 Host: localhost:7070 Connection: keep-alive Pragma: no-cache Cache-Control: no-cache

URL



Изображение с сайта [Wikimedia.org](https://www.wikimedia.org)

ОТПРАВКА ЗАПРОСОВ

Всё отлично, давайте посмотрим, как отправится наша форма, если мы не будем делать `preventDefault`.

Для этого в атрибутах формы нужно установить `action` с URL'ом нашего сервера (допустим, `http://localhost:7070/subscribe`):

▼ Request Headers [view parsed](#)

```
GET /subscribe?name=Vasya&phone=%2B79000000000 HTTP/1.1
```

```
Host: localhost:7070
```

```
Connection: keep-alive
```

```
Pragma: no-cache
```

```
Cache-Control: no-cache
```


GET, POST, PUT И ДРУГИЕ

По умолчанию форма отправляется методом `GET` (если иное не указано в атрибуте `method`). Кроме того, форма, отправляемая средствами HTML может отправляться только методами `GET / POST`, тогда так сам протокол HTTP определяет гораздо больше методов, например, `PUT`, `HEAD`, `OPTIONS`, `DELETE`.

GET, POST, PUT И ДРУГИЕ

Отправка формы методом **POST** :

▼ Request Headers [view parsed](#)

```
POST /subscribe HTTP/1.1
Host: localhost:7070
Connection: keep-alive
Content-Length: 31
Pragma: no-cache
Cache-Control: no-cache
Origin: http://localhost:8080
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
```

▼ Form Data [view source](#) [view decoded](#)

```
name: Vasya
phone: %2B790000000000
```

Обратите внимание, теперь данные не передаются в URL'е, а передаются в теле запроса, также устанавливается заголовок **Content-Type** .



ЗАГОЛОВКИ

Q: Зачем нужны эти заголовки?

A: Поскольку сервер и клиент могут быть написаны на разных языках, именно заголовки используются как «служебная информация», помогающая им понять друг друга.

ЧИТАЕМ ПАРАМЕТРЫ В URL

Попробуем прочесть параметры в URL (если отправляется GET-запрос):

```
1  const server = http.createServer((req, res) => {  
2    console.log(req.url);  
3    res.end('server response');  
4  });
```

Данные содержат путь + ещё и закодированы:

```
/subscirbe?name=Vasya&phone=%2B790000000000
```

ЧИТАЕМ ТЕЛО ЗАПРОСА

Попробуем прочесть тело запроса на сервере (если отправляется POST-запрос):

```
1  const server = http.createServer((req, res) => {  
2    const buffer = [];  
3    req.on('data', (chunk) => {  
4      buffer.push(chunk);  
5    });  
6    req.on('end', () => {  
7      const body = Buffer.concat(buffer).toString();  
8      console.log(body);  
9    });  
10   res.end('server response');  
11 });
```

Теперь URL нет, но данные пришли в закодированном формате:

```
name=Vasya&phone=%2B790000000000
```

ЭКСКУРС В NODE.JS

В Node.js используется понятие `EventEmitter` – объект, «эмитирующий» события (в отличие от браузера, где есть `EventTarget`).

Подписка на события объекта осуществляется с помощью метода `on`:

```
eventEmitter.on('event', callback);
```

УДОБСТВО

Естественно, парсить вручную неудобно, подбирать отдельно библиотеки, наверное, тоже не очень.

Поэтому мы с вами возьмём фреймворк с экосистемой плагинов, которые позволят нам перевесить эти рутинные задачи на его плечи.

В качестве такого мы возьмём один из самых популярных — [koa](#).

```
npm install koa
```

KOA

```
1  const http = require('http');
2  const Koa = require('koa');
3  const app = new Koa();
4
5  app.use(async (ctx) => {
6    ctx.response.body = 'server response';
7  });
8
9  const server = http.createServer(app.callback()).listen(7070);
```

Т.е. koa в нашем случае использует существующий http-сервер, для того чтобы мы могли надстроить туда свою логику.

KOA MIDDLEWARE

`app.use(middleware)` позволяет установить middleware-функцию, т.е. функцию, через которую будет проходить обработка запроса и формирование ответа.

По факту всё приложение koa представляет из себя каскад middleware-функций:

```
app.use(async (ctx, next) => {  
  // do something  
  await next(); // передача контроля следующему middleware  
  // do something  
})
```

KOA CONTEXT

`ctx` — это объект, который инкапсулирует в себе и запрос, и ответ:

- `ctx.request` — запрос
- `ctx.response` — ответ

`ctx` использует для удобства сокращения, например, `ctx.body` -> `ctx.response.body`, но мы для упрощения восприятия их использовать не будем.

QUERYSTRING

Koa уже включает в себя необходимую функциональность по обработке queryString, поэтому:

```
1  const http = require('http');
2  const Koa = require('koa');
3  const app = new Koa();
4
5  app.use(async ctx => {
6    console.log(ctx.request.query);
7    ctx.response.body = 'server response';
8  });
```

KOA BODY

А вот для обработки форм, отправляемых методом POST нам понадобится дополнительный пакет:

```
npm install koa-body
```

```
1  const http = require('http');
2  const Koa = require('koa');
3  const koaBody = require('koa-body');
4  const app = new Koa();
5
6  app.use(koaBody({
7    urlencoded: true,
8  }));
9
10 app.use(async ctx => {
11   console.log(ctx.request.querystring);
12   console.log(ctx.request.body);
13   ctx.response.body = 'server response';
14 });
15
16 const server = http.createServer(app.callback()).listen(7070);
```

ОТПРАВКА ФОРМЫ СРЕДСТВАМИ JS

Вернём `preventDefault` и попробуем самостоятельно отправить форму. Для этого нам нужен инструмент, позволяющий отправлять HTTP-запросы из браузера.

Их на данный момент в браузере у нас целых два:

- [XMLHttpRequest](#)
- [fetch](#)

Не считая библиотек (`axios`, `superagent`), которые являются надстройками над этими двумя + реализуют возможности в среде Node.js



XMLHTTPREQUEST

XMLHttpRequest

Типовой формат использования:

```
1  const xhr = new XMLHttpRequest(); // создание объекта
2
3  xhr.open('<method>', '<url>', '<async>'); // подготовка запроса
4
5  xhr.setRequestHeader('<name>', '<value>'); // установка заголовков
6  xhr.setRequestHeader('<name>', '<value>');
7
8  xhr.addEventListener('readystatechange', (evt) => {
9      if (xhr.readyState === 4) {
10         if (xhr.status === 200) {
11             console.log(xhr.response);
12         }
13     }
14 });
15
16 xhr.send('<body>'); // отправка запроса
```

READYSTATE

`readystatechange` — событие, возникающее при изменении состояния запроса:

- `UNSENT` (0) — метод `open` не вызывался
- `OPENED` (1) — был вызван `open`
- `HEADERS_RECEIVED` (2) — был вызван `send`, получены заголовки и статус ответа
- `LOADING` (3) — загрузка тела ответа
- `DONE` (4) — загрузка завершена

STATUS CODES

Для сообщения клиенту об итогах выполнения операций определены статус-коды:

- 1xx – informational
- 2xx – success
- 3xx – redirection
- 4xx – client error
- 5xx – server error

Код 200 – означает OK.

СИНХРОННОСТЬ

Раньше допускалось при использовании метода `open` последний флаг устанавливать в `false`, что приводило к выполнению синхронного запроса (т.е. блокировке до получения ответа).

Эта возможность по-прежнему может быть доступна, но является `deprecated`, и мы настоятельно не рекомендуем её использовать.

ОТПРАВЛЯЕМ ЗАПРОС

```
1  const queryString = Array.from(subscribeForm.elements)
2    .filter(({ name }) => name)
3    .map(({ name, value }) => `${name}=${encodeURIComponent(value)}`)
4    .join('&');
5  const url = `http://localhost:7070/?${queryString}`;
6  const xhr = new XMLHttpRequest();
7  xhr.open('GET', url, true);
8  // event listener here
9  xhr.send();
```

ЗАПРОС

Во вкладке **Network** есть специальный фильтр, который позволяет посмотреть только **XHR** запросы:

The screenshot shows the Chrome DevTools Network tab. The top toolbar includes tabs for Elements, Console, Sources, Network (selected), Performance, Memory, Application, Security, and Audits. Below the tabs, there are icons for a red circle, a crossed-out circle, a video camera, a funnel (filter), and a magnifying glass. The 'View' section shows a list icon, a blue wavy line icon, and checkboxes for 'Group by frame', 'Preserve log', 'Disable cache' (checked), 'Offline', and 'Online'. A 'Filter' input field is present, followed by a 'Hide data URLs' checkbox and a list of filter categories: 'All', 'XHR' (selected), 'JS', 'CSS', 'Img', 'Media', 'Font', 'Doc', 'WS', 'Manifest', and 'Other'. Below the filter bar is a timeline with markers at 5000 ms, 10000 ms, 15000 ms, 20000 ms, 25000 ms, 30000 ms, and 35000 ms. The main list of network requests shows a single entry with a square icon and the URL '?name=Vasya&phone=%2B7900000000...'. The details panel for this request is open, showing the 'Headers' tab. The 'General' section displays the following information: 'Request URL: http://localhost:7070/?name=Vasya&phone=%2B790000000000', 'Request Method: GET', 'Status Code: 200 OK' (with a green status icon), 'Remote Address: [::1]:7070', and 'Referrer Policy: no-referrer-when-downgrade'. The 'Query String Parameters' section is also visible, showing 'name: Vasya' and 'phone: %2B790000000000'.

Name	Headers	Preview	Response	Timing
<input type="checkbox"/> ?name=Vasya&phone=%2B7900000000...	General Request URL: http://localhost:7070/?name=Vasya&phone=%2B790000000000 Request Method: GET Status Code: 200 OK Remote Address: [::1]:7070 Referrer Policy: no-referrer-when-downgrade Query String Parameters name: Vasya phone: %2B790000000000			

CORS POLICY

На сервер данные мы отправляем, но при этом в консоли видим ошибку и наш `eventListener` не отрабатывает:

```
✖ Access to XMLHttpRequest at 'http://localhost:7070/?name=Vasya&phone=%2B790000000000' from origin 'http://localhost:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

Вернёмся к этому вопросу чуть позже, посмотрим, как отправлять данные с помощью POST-запроса.

ОТПРАВЛЯЕМ ЗАПРОС

```
1  const params = Array.from(subscribeForm.elements)
2    .filter(({ name }) => name)
3    .map(({ name, value }) => `${name}=${encodeURIComponent(value)}`)
4    .join('&');
5  const url = 'http://localhost:7070';
6  const xhr = new XMLHttpRequest();
7  xhr.open('POST', url, true);
8  xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
9  // event listener here
10 xhr.send(params);
```

OLD STYLE

Те подходы, которые мы с вами рассмотрели, достаточно стары и сейчас можно писать чуть проще.

В части событий XHR поддерживает:

- `load` — загрузка завершена успешно (но статус-код не обязательно 200)
- `error` — ошибка (ответ от сервера не получен)
- `abort` — запрос прерван
- `loadend` — запрос завершился (неважно, с ошибкой, успешно или прерван)

Есть новые объекты `URLSearchParams` и `FormData`, позволяющие проще работать с формами и данными.

EVENTS

```
1  xhr.addEventListener('load', () => {
2    if (xhr.status === 200) {
3      console.log(xhr.responseText);
4    } else {
5      // TODO: handle other status
6    }
7  });
8  xhr.addEventListener('error', () => {
9    // TODO: handle error
10  });
11  xhr.addEventListener('loadend', () => {
12    // TODO: request finished
13  });
```


URLSEARCHPARAMS

Объект, позволяющий нам избавиться от необходимости кодировать всё вручную:

```
1  const params = new URLSearchParams();
2  Array.from(subscribeForm.elements)
3    .filter(({ name }) => name)
4    .forEach(({ name, value }) => params.append(name, value));
5  const xhr = new XMLHttpRequest();
6  xhr.open('GET', `http://localhost:7070/?${params}`);
7  xhr.send();
```

```
1  const params = new URLSearchParams();
2  Array.from(subscribeForm.elements)
3    .filter(({ name }) => name)
4    .forEach(({ name, value }) => params.append(name, value));
5  const xhr = new XMLHttpRequest();
6  xhr.open('POST', 'http://localhost:7070');
7  xhr.send(params);
```

При этом для **POST** автоматически выставляется **Content-Type**.

ОБРАБОТКА НА СЕРВЕРЕ

Научимся получать ответ от сервера, например, если всё ОК, то сервер будет возвращать код 200, а если такая запись уже есть, то пусть возвращает код 400 и в теле ответа какое-нибудь сообщение:

```
1  app.use(async ctx => {
2    const { name, phone } = ctx.request.querystring;
3    // const { name, phone } = ctx.request.body;
4    if (subscriptions.has(phone)) {
5      ctx.response.status = 400
6      ctx.response.body = 'You already subscribed'
7      return;
8    }
9
10   subscriptions.set(phone, name);
11   ctx.response.body = 'Ok';
12 }));
```

Сервер работает корректно, но мы по-прежнему не получаем ничего на клиенте. Давайте разбираться.

SAME ORIGIN-POLICY & CORS

SOP

The Same-Origin Policy (SOP) is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin.

Перевод: SOP - механизм безопасности, определяющий правила, по которым документ или скрипт, загруженный из одного источника (origin), может взаимодействовать с ресурсом из другого источника (origin).

Определение с сайта [MDN](#).

Origin – это набор из схемы (например, http/https), хоста (или домена) и порта.



SOP

Но при этом мы без проблем отправляем форму (не из JS).
Можем включать изображения, CSS и JS с других доменов и много чего другого.

CORS

Далее мы будем говорить только о XMLHttpRequest и fetch.

Для них правила в простейшем случае следующие: мы можем отправить запрос, но браузер не отдаст нам ответ, пока сервер не выставит заголовки Access-Allow-Control-*.

Это уже Cross-Origin Resource Sharing: Cross-Origin Resource Sharing (CORS) is a mechanism that uses additional HTTP headers to tell a browser to let a web application running at one origin (domain) have permission to access selected resources from a server at a different origin.

Определение с сайта [MDN](#).



SOP & CORS

Q: Почему это важно для нас?

A: Потому что вы должны уметь определять, что на сервере не выставляются нужные заголовки и сообщать об этом разработчику серверной части (в противном случае вы, не применяя ухищрений вроде проху или JSONP, не сможете получить данные с сервера).

ЗАГОЛОВКИ

```
1 Access-Control-Allow-Origin: http://localhost (или * - для всех)
2 Access-Control-Allow-Methods: POST, GET (* - использовать нельзя)
3 Access-Control-Allow-Headers: X-Secret, Content-Type (* - использовать нельзя)
```

Выставим их через коа:

```
1 app.use(async ctx => {
2   const { name, phone } = ctx.request.querystring;
3   // const { name, phone } = ctx.request.body;
4
5   ctx.response.set({
6     'Access-Control-Allow-Origin': '*',
7   });
8   if (subscriptions.has(phone)) {
9     ctx.response.status = 400
10    ctx.response.body = 'You already subscribed'
11    return;
12  }
13
14  subscriptions.set(phone, name);
15  ctx.response.body = 'Ok';
16 });
```


PREFLIGHT

Так всё работает, но на самом деле — не совсем. CORS устроен немного сложнее, в частности, тот механизм, который мы рассмотрели, называется Simple Request:

- Метод запроса `GET`, `POST`, `HEAD`
- `Content-Type`:
 - `application/x-www-form-urlencoded`
 - `multipart/form-data`
 - `text/plain`
- Не используются заголовки кроме некоторых predefined + [ещё несколько правил](#)

PREFLIGHT

Все остальные запросы идут по схеме:

1. Сначала браузер делает запрос `OPTIONS` на тот же URL
2. Если в ответе на этот запрос есть заголовки `Access-Control-Allow-*`, то браузер выполняет оригинальный запрос, если же нет — то не выполняет

CORS

Напишем простой middleware, который обрабатывает эту ситуацию (вы можете использовать готовые для коа). Логика будем строить на основании того, что браузер отправляет заголовок `Origin` (а при Preflight ещё и `Access-Control-Request-Method`, а также может отправить `Access-Control-Request-Headers`):

CORS

```
1 app.use(async (ctx, next) => {
2   const origin = ctx.request.get('Origin');
3   if (!origin) {
4     return await next();
5   }
6
7   const headers = { 'Access-Control-Allow-Origin': '*', };
8
9   if (ctx.request.method !== 'OPTIONS') {
10    ctx.response.set({...headers});
11    try {
12      return await next();
13    } catch (e) {
14      e.headers = {...e.headers, ...headers};
15      throw e;
16    }
17  }
18
19  if (ctx.request.get('Access-Control-Request-Method')) {
20    ctx.response.set({
21      ...headers,
22      'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, PATCH',
23    });
24
25    if (ctx.request.get('Access-Control-Request-Headers')) {
26      ctx.response.set('Access-Control-Allow-Headers', ctx.request.get('Access-Control-Allow-Request-Headers'));
27    }
28
29    ctx.response.status = 204; // No content
30  }
31 });
```

ПОЛУЧЕНИЕ ДАННЫХ

Отлично, мы научились отправлять данные и получать их.

Задача интерпретации этих данных целиком лежит на нас, сервер нам может прислать что угодно — от JSON, до HTML-разметки и файлов.

Здесь стоит отметить следующий важный момент: в `xhr` можно указать свойство `responseType`:

- `text` (по умолчанию)
- `arraybuffer`
- `blob`
- `document`

Это позволит нам получить в нужном формате данные (а работать с `Blob` 'ами и `ArrayBuffer` вы уже умеете).

ОТПРАВКА ФАЙЛОВ

Новая задача стоит перед нами — организовать хранилище картинок, где пользователь может загружать картинку на сервер, а сервер возвращает в ответе URL загруженной картинки.

Имея URL картинки мы без труда её отобразим.

Разметка:

```
1 <form data-id="upload-form">
2   <input type="file" name="file" accept="image/*">
3   <input type="text" name="name" value="My Photo">
4   <button>Upload</button>
5 </form>
```

MULTIPART

Для отправки файлов в обычных формах используется

`enctype="multipart/form-data"` с методом `POST`.

Задачу формирования такого запроса вручную мы оставим эстетам, сами же воспользуемся объектом `FormData`:

```
1 document.querySelector('[data-id=upload-form]').addEventListener('submit', (evt) => {  
2   evt.preventDefault();  
3   const formData = new FormData(evt.currentTarget);  
4  
5   const xhr = new XMLHttpRequest();  
6   xhr.open('POST', 'http://localhost:7070');  
7   // TODO: subscribe to response  
8   xhr.send(formData);  
9 })
```

`FormData` — специальный объект, который упаковывает данные нашей формы для отправки.

Если мы посмотрим запросы, то `Content-Type` отправляется уже `multipart/form-data`.

SERVER

С сервером несколько нюансов:

1. Неудобно обрабатывать все запросы в одном обработчике
2. Наш сервер не понимает `multipart/form-data`

Включим поддержку обработки multipart:

```
1 app.use(koaBody({  
2   urlencoded: true,  
3   multipart: true,  
4 }));
```


ЗАГРУЗКА ФАЙЛОВ

Установим пакет `uuid` для генерации имён:

```
npm install uuid
```

Подключим необходимые модули и определим каталог для загрузки:

```
1  const path = require('path');  
2  const fs = require('fs');  
3  const uuid = require('uuid');  
4  
5  const public = path.join(__dirname, '/public')
```

Примечание*: не забудьте создать каталог `public` (лучше в него ещё положить файл `.gitkeep`), прописать `public` в `.foreverignore` и перезапустить `forever`.

ОТОБРАЖЕНИЕ ФАЙЛА

```
1  xhr.addEventListener('load', () => {  
2    if (xhr.status >= 200 && xhr.status < 300) {  
3      const img = document.createElement('img');  
4      img.src = `http://localhost:7070/${xhr.response}`;  
5      console.log(xhr.response);  
6      document.body.appendChild(img);  
7    }  
8  });
```

ОТДАЧА СТАТИКИ

Файлы загружаются, но как заставить сервер отдавать их? Установка `img.src` приводит к GET-запросу, на который сервер отвечает 500 ошибкой.

Установим специальный middleware для обработки статики:

```
npm install koa-static
```

```
const koaStatic = require('koa-static');  
app.use(koaStatic(public));
```

FORMDATA

На самом деле не обязательно создавать `FormData` на базе имеющейся формы, вполне допустимо создавать этот объект с помощью конструктора `FormData`:

```
const formData = new FormData();  
formData.append(<key>, <value>);
```

`value` может быть как строкой, так и `Blob`.

XMLHttpRequest

Помимо `FormData` сам метод `send` умеет отправлять `Blob` и `ArrayBuffer`, но для сервера нужно будет писать отдельный `middleware`, который сможет обработать это.



HEROKU

HEROKU

Для дальнейшего прохождения курса вам потребуется серверная часть (backend), с которой вы и будете взаимодействовать. В качестве основы мы предлагаем вам использовать [Koa](#), а в качестве бесплатного облачного сервиса для размещения серверной части — [Heroku](#).

Таким образом, у вас получится связка:

1. Frontend на GitHub Pages
2. Backend на Heroku

Примечание*: конечно, вы можете использовать `koa-static` и положить собранный frontend на Heroku.

HEROKU

B `package.json`:

```
1  "scripts": {  
2    "prestart": "npm install",  
3    "start": "forever server.js",  
4    "watch": "forever -w server.js"  
5  },  
6  "engines": {  
7    "node": "11.x"  
8  },
```

B `server.js`:

```
const port = process.env.PORT || 7070;  
const server = http.createServer(app.callback()).listen(port)
```




HEROKU + GITHUB

Поскольку мы всё-таки Frontend-разработчики, мы не будем настраивать CI/CD для серверной части, а пойдём по самому простому пути — просто подключим репозиторий GitHub с кодом backend к Heroku.

Каждый раз, когда вы будете делать push в этот репозиторий, Heroku будет разворачивать новую версию вашего сервера.

HEROKU + GITHUB

Зайдите в Dashboard Heroku и выберите опцию **Create New App**:

App name

coursar-heroku



coursar-heroku is available

Choose a region



Europe



Add to pipeline...

Create app

HEROKU + GITHUB

После создания приложения вы попадёте на вкладку **Deploy**, где нужно выбрать **GitHub**:

The screenshot shows the Heroku 'Deploy' tab. At the top, there's a navigation bar with 'Overview', 'Resources', 'Deploy' (selected), 'Metrics', 'Activity', 'Access', and 'Settings'. Below this, the main content area is divided into two sections. The left section, titled 'Add this app to a pipeline', instructs the user to 'Create a new pipeline or choose an existing one and add this app to a stage in it.' The right section, titled 'Add this app to a stage in a pipeline to enable additional features', provides information about connecting to GitHub and enabling review apps. It includes a diagram showing a pipeline with stages and a 'Choose a pipeline' dropdown menu. At the bottom, the 'Deployment method' section offers three options: 'Heroku Git' (Use Heroku CLI), 'GitHub' (Connect to GitHub), and 'Container Registry' (Use Heroku CLI).

Overview Resources **Deploy** Metrics Activity Access Settings

Add this app to a pipeline

Create a new pipeline or choose an existing one and add this app to a stage in it.

Add this app to a stage in a pipeline to enable additional features

Pipelines let you connect multiple apps together and **promote code** between them. [Learn more.](#)

Pipelines connected to GitHub can enable **review apps**, and create apps for new pull requests. [Learn more.](#)

Choose a pipeline

Deployment method

- Heroku Git**
Use Heroku CLI
- GitHub**
Connect to GitHub
- Container Registry**
Use Heroku CLI

HEROKU + GITHUB

И нажать `Connect to GitHub`:

View your code diffs on GitHub

Connect your app to a GitHub repository to see commit diffs in the activity log.

Deploy changes with GitHub

Connecting to a repository will allow you to deploy a branch to your app.

Automatic deploys from GitHub

Select a branch to deploy automatically whenever it is pushed to.

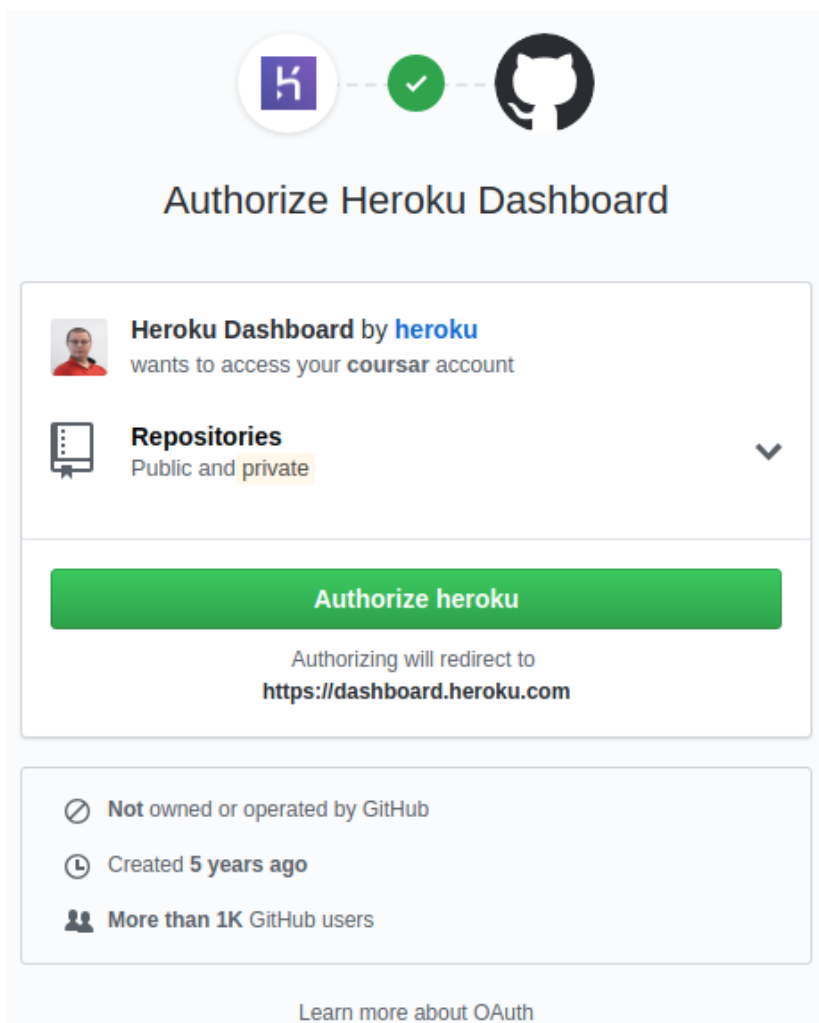
Create review apps in pipelines

Pipelines connected to GitHub can enable **review apps**, and create apps for new pull requests. [Learn more](#).

Connect to GitHub

HEROKU + GITHUB

Авторизовать Heroku:



The screenshot shows the GitHub authorization page for Heroku Dashboard. At the top, there are three icons: the Heroku logo (a purple 'H' in a circle), a green checkmark in a circle, and the GitHub logo (a black octocat in a circle). Below these icons is the text "Authorize Heroku Dashboard".

The main content area is a white box with a light gray border. It contains a profile picture of a man with glasses and a red shirt, followed by the text "Heroku Dashboard by heroku" and "wants to access your coursar account". Below this is a section titled "Repositories" with a sub-label "Public and private" and a dropdown arrow. A large green button labeled "Authorize heroku" is prominently displayed. Below the button, it says "Authorizing will redirect to" followed by the URL "https://dashboard.heroku.com".


At the bottom of the white box, there is a light gray section with three items: a warning icon and the text "Not owned or operated by GitHub", a clock icon and the text "Created 5 years ago", and a group of people icon and the text "More than 1K GitHub users".

At the very bottom of the page, there is a link that says "Learn more about OAuth".

HEROKU + GITHUB

Найти нужный репозиторий и подключить его:

Search for a repository to connect to

 coursar

heroku-nodejs

Search

Missing a GitHub organization? [Ensure Heroku Dashboard has team access.](#)

 coursar/heroku-nodejs

Connect

HEROKU + GITHUB

Включить Automatic Deploy:

Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

Enable automatic deploys from GitHub

Every push to the branch you specify here will deploy a new version of this app. **Deploys happen automatically**; be sure that this branch is always in a deployable state and any tests have passed before you push. [Learn more](#).

Choose a branch to deploy

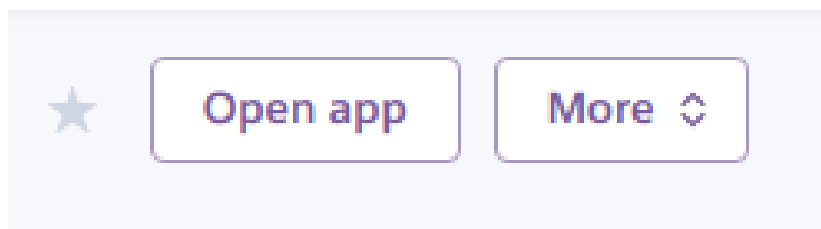
☐ Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

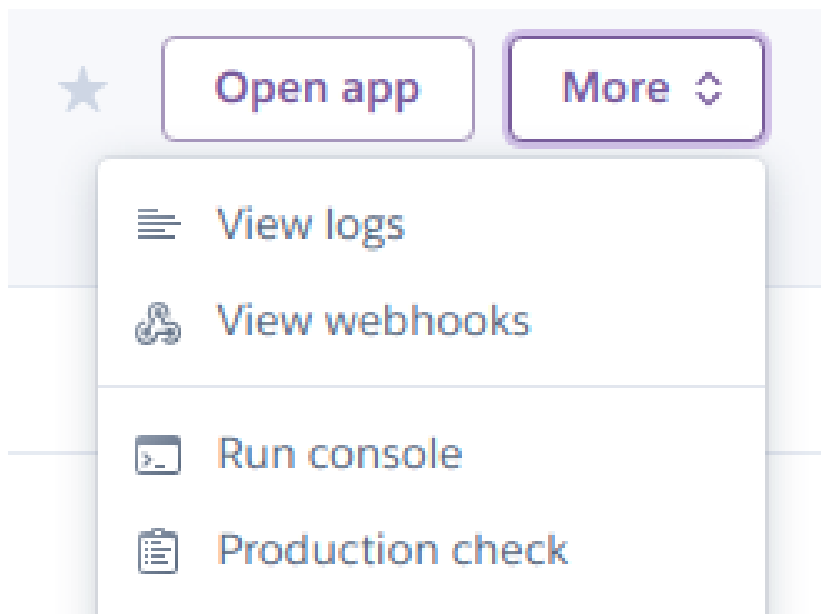
Enable Automatic Deploys

FINAL

После push'a открыть ваше приложение:



Либо, если что-то пошло не так, логи:





ИТОГИ

Сегодня у нас была достаточно большая лекция:

- мы обсудили работу с HTTP (как с точки зрения клиента, так и с точки зрения сервера)
- рассмотрели SOP и CORS
- обсудили основы REST
- посмотрели, как разворачивать сервер на Heroku



Спасибо за внимание!

Время задавать вопросы ☐

ВЛАДИМИР ЯЗЫКОВ



neizerth@gmail.com



[@neizerth](https://t.me/neizerth)