нетология

JS, Ajax, REST, JSON, CORS

Григорий
Вахмистров

Григорий Вахмистров

Backend Developer в Tennisi.bet

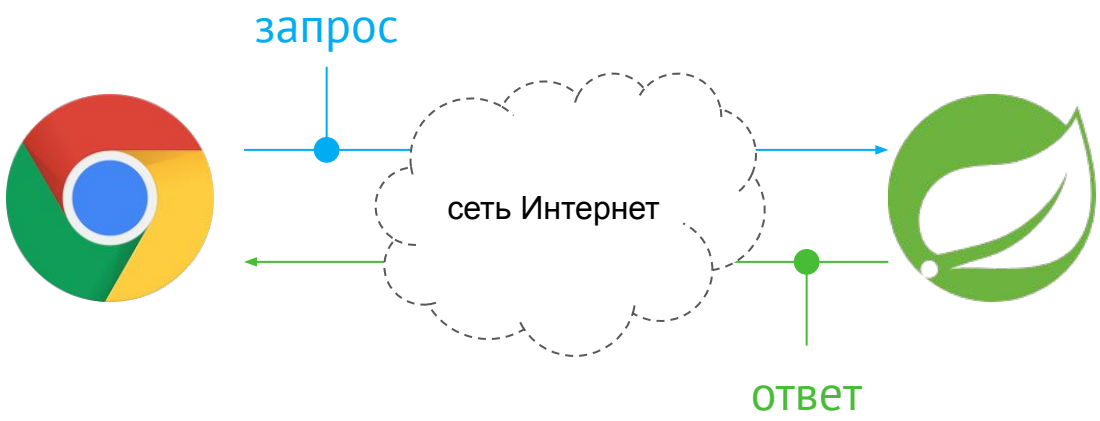
План занятия

1. Предисловие
2. JS
3. Grizzly
4. XHR & AJAX
5. REST
6. SOP & CORS
7. Итоги

Предисловие

Предисловие

На прошлой лекции мы разобрали, как происходит взаимодействие (методы и форматы передачи) при работе в Web:



Задача

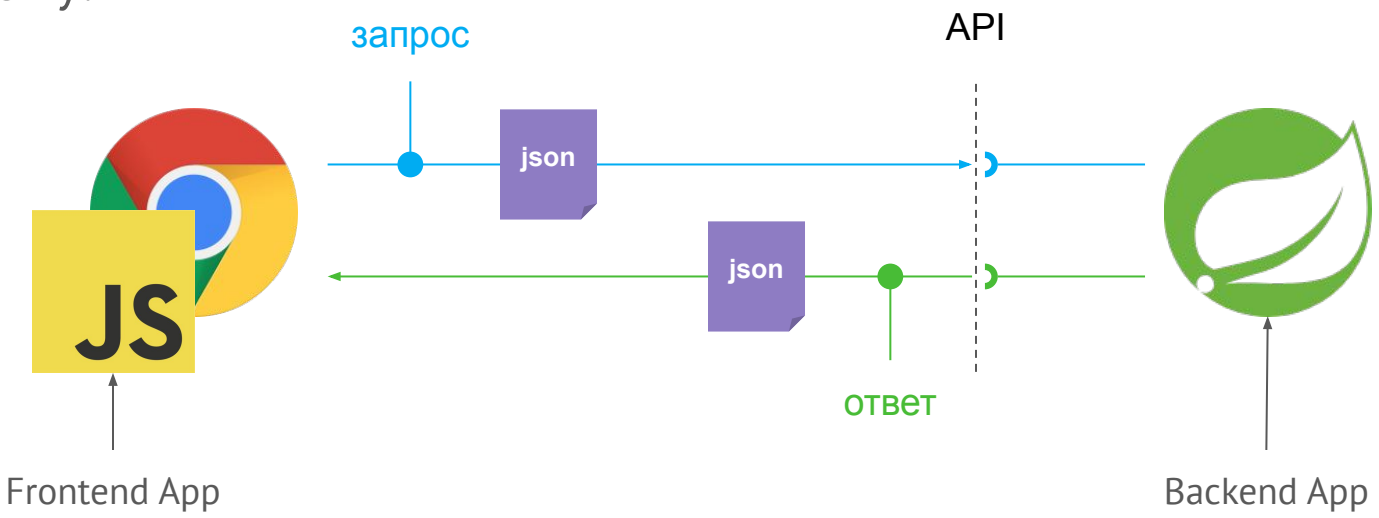
На сегодняшнюю лекцию у нас две задачи:

1. Получить общее представление о JS
2. Посмотреть на API промышленного http-сервера

JS

JS

Современные приложения, конечно, же выглядят немного по-другому:



Причём не обязательно используется HTTP, достаточно широко используются и другие технологии (например, WebSockets), но HTTP до сих пор остаётся самым распространённым.

JS

JS (JavaScript) - скриптовой язык общего назначения, соответствующий спецификации ECMAScript 262.

Сейчас это полноценный язык, который позволяет писать приложения, работающие в браузере*.

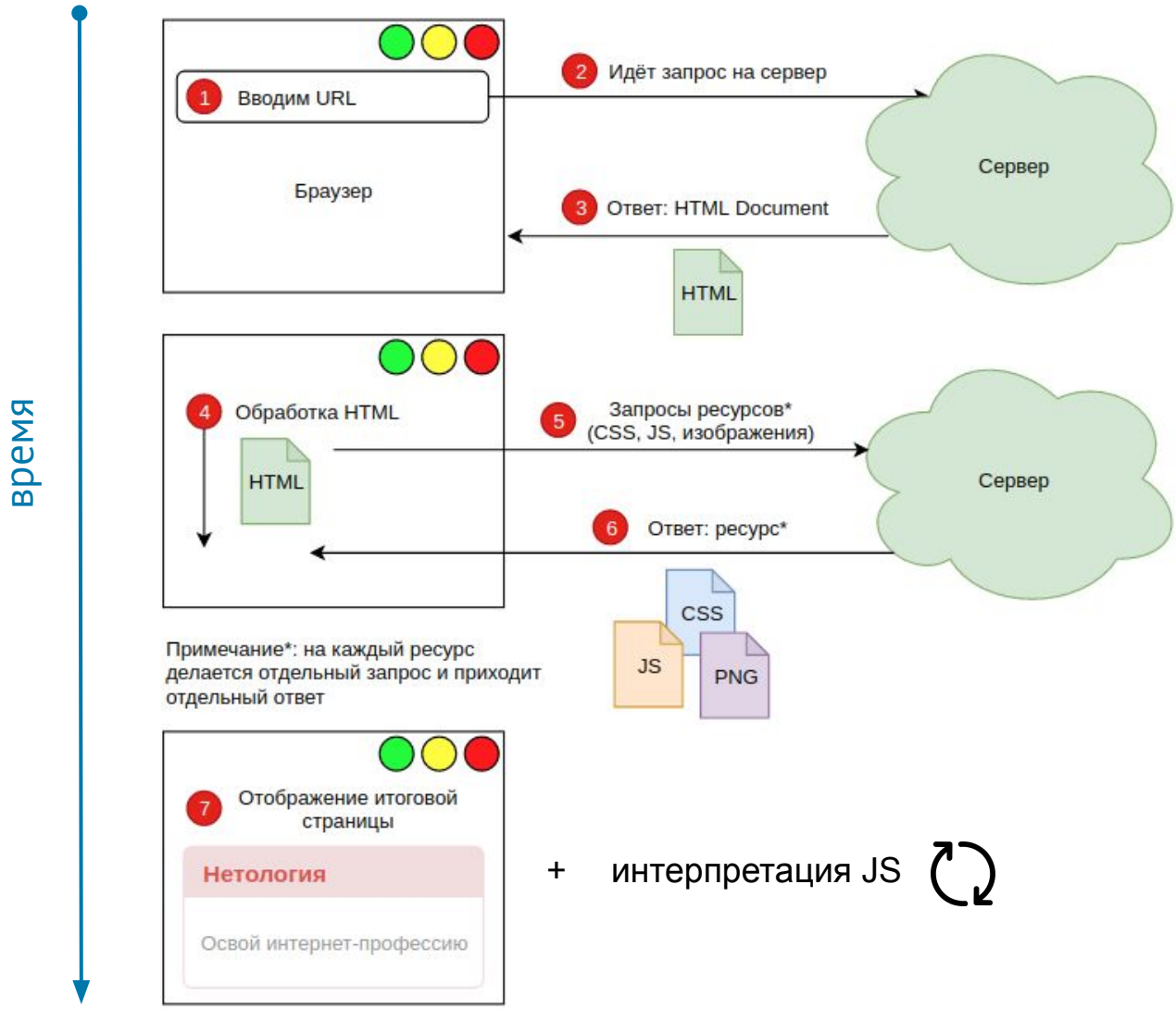
Примечание*: на самом деле, JS можно запускать уже везде.

JS

В чём суть:

1. Браузер содержит реализацию движка JS (как JVM для Java)
2. Браузер предоставляет Web API - набор интерфейсов, позволяющих выстраивать полноценные приложения

Обработка страницы



\+ интерпретация JS

Как всё устроено

1. Вы вводите адрес страницы или кликаете на ссылку
2. Браузер загружает страницу по протоколу HTTP и начинает обрабатывать её (парсинг)
3. Для каждого элемента, встречаемого на веб-странице, создаётся объект в памяти, к которому затем можно получить доступ
4. После того, как дерево объектов построено (поскольку это UI, то он организуется иерархически), браузер обрабатывает JS
5. Далее JS может "управлять" страницей, взаимодействуя с объектами

обычный порядок

Обычный порядок

Стоит отметить, что это самый часто встречающийся порядок:

- сначала создаются объекты страницы,
- затем уже запускается JS.

DOM API

DOM (Document Object Model) - специальное API, которое браузер предоставляет для доступа к созданным объектам из различных языков программирования (в первую очередь из JS).

Каждый созданный объект соответствует определённому интерфейсу.

DOM API

```
<button data-action="inc">Click me</button>
```

превращается в объект, соответствующий

DOM interface:

IDL

```
[Exposed=Window]
interface HTMLButtonElement : HTMLElement {
  [HTMLConstructor] constructor();

  [CEReactions] attribute boolean disabled;
  readonly attribute HTMLFormElement? form;
  [CEReactions] attribute USVString formAction;
  [CEReactions] attribute DOMString formEnctype;
  [CEReactions] attribute DOMString formMethod;
  [CEReactions] attribute boolean formNoValidate;
  [CEReactions] attribute DOMString formTarget;
  [CEReactions] attribute DOMString name;
  [CEReactions] attribute DOMString type;
  [CEReactions] attribute DOMString value;

  readonly attribute boolean willValidate;
  readonly attribute ValidityState validity;
  readonly attribute DOMString validationMessage;
  boolean checkValidity();
  boolean reportValidity();
  undefined setCustomValidity(DOMString error);

  readonly attribute NodeList labels;
};
```

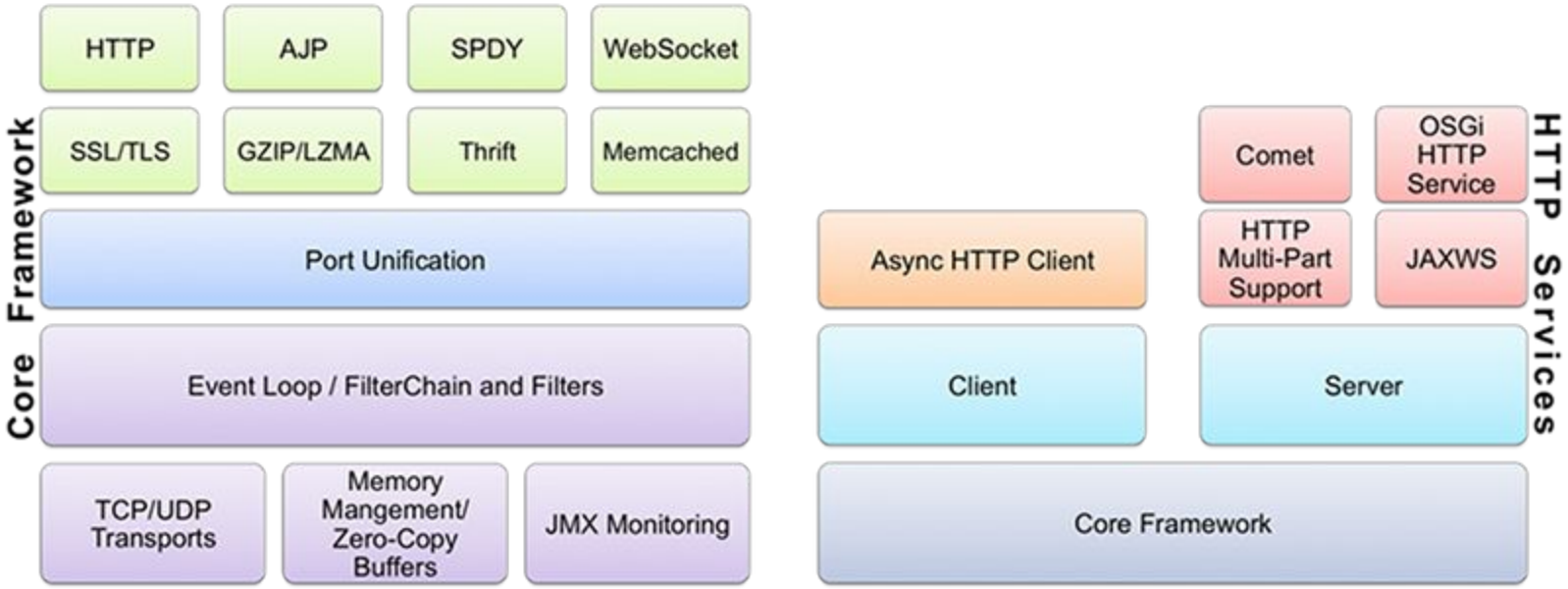
Сам интерфейс описывается в нотации Web IDL.

Важно: интерфейс описывает не только методы, но и атрибуты.

Grizzly

Сервер

Поскольку мы уже потренировались с самописным сервером, пришло время использовать более-менее промышленный вариант. В качестве такого мы возьмём Grizzly:



Сервер

```
<dependencies>
    <dependency>
        <groupId>org.glassfish.grizzly</groupId>
        <artifactId>grizzly-http-server-core</artifactId>
        <version>3.0.0-M1</version>
    </dependency>
</dependencies>
```

demo + debugger

Сервер

```
public class Main {
  public static void main(String[] args) throws IOException, InterruptedException {
    final var server :HttpServer = HttpServer.createSimpleServer( docRoot: "static",  port: 9999);
    server.getServerConfiguration().addHttpHandler(new HttpHandler() {
      @Override
      public void service(Request request, Response response) throws Exception {
        response.getWriter().write( str: "Ok");
      }
    }, ...mappings: "/api");

    server.start();
    Runtime.getRuntime().addShutdownHook(new Thread(server::shutdown));
    Thread.currentThread().join();
  }
}
```

Ключевые моменты

1. Mapping - привязка обработчика к URL'у:

```
server.getServerConfiguration().addHttpHandler(new HttpHandler() {
  @Override
  public void service(Request request, Response response) throws Exception {
    response.getWriter().write( str: "Ok");
  }
}, ...mappings: "/api");
```

2. Runtime.getRuntime().addShutdownHook() - регистрируемся на завершение работы JVM, чтобы завершить работу сервера
3. Thread.currentThread().join() - блокирование main thread'а (т.к. сервер не блокирует main thread и приложение завершится)

Ключевые моменты

4. DocRoot - возможность указать каталог, из которого будут браться статичные файлы (т.е. отдаются файлы в том же виде, в котором хранятся на диске):

```
HttpServer.createSimpleServer( docRoot: "static",  port: 9999);
```

Зачем нам эти детали?

Мы специально обращаем ваше внимание на эти детали, поскольку они фактически будут повторяться в большинстве решений, которые мы будем использовать (в том числе Spring).

HTML страница

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
        content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <form id="form">
    <input name="value">
    <input name="avatar" type="file" accept="image/*">
    <button>Submit</button>
  </form>
  <script src="js/app.js"></script>
</body>
</html>
```

подключение JS-кода

JS код

метод

```
console.log('js executed');
```

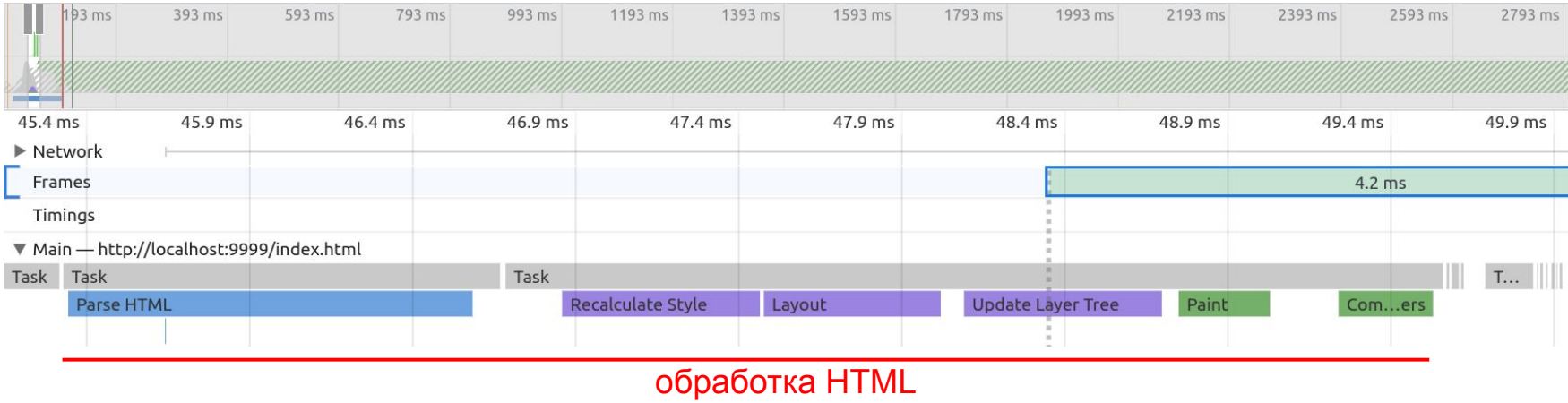
объект

По факту, выглядит почти как вызов Java-кода, но: ни создания переменных, ни объектов, ничего нет.

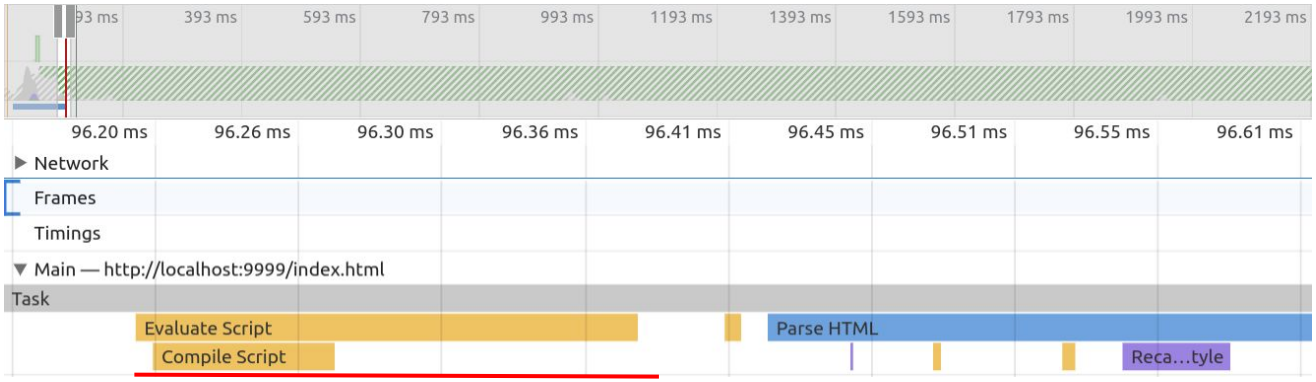
На самом деле, это пример использования объекта, предоставляемого браузер (т.е. он создан самим браузер и предоставляется в качестве API для использования).

Обработка

Если смотреть на реальную картину обработки, то:



обработка HTML



выполнение JS

document

Помимо объекта console, браузер предоставляет объект document, который позволяет взаимодействовать с веб-страницей:

```
const form = document.getElementById( elementId: 'form');
form.addEventListener( type: 'submit',  listener: (evt : Event ) => {
  evt.preventDefault();
});
```

const - аналог final var в Java

(evt) => {} аналог lambda

События

Основная идея: событийно-ориентированная модель: мы подписываемся на события и сам браузер вызывает нашу функцию, когда событие происходит (передавая туда объект события). Это примерно как с Grizzly: мы передаём объект, метод которого вызывается тогда, когда мы получаем HTTP запрос:

```
server.getServerConfiguration().addHttpHandler(new HttpHandler() {
  @Override
  public void service(Request request, Response response) throws Exception {
    response.getWriter().write( str: "Ok");
  }
}, ...mappings: "/api");
```

Default Behaviour

У большинства событий есть поведение по умолчанию: например, при клике на кнопку "Ok" в обычной форме форма отправляется на сервер, что приводит к HTTP-запросу и загрузке новой страницы.

JS даёт переопределить поведение для некоторых событий, вызвав на объекте события метод preventDefault.

Отмена Default Behaviour

Q: зачем это может быть нужно?

A: перезагрузка страницы ведёт к тому, что все* объекты выгружаются из памяти и JS приложение интерпретируется заново с нуля (равносильно тому, что мы рестартуем сервер на Java). Вместо этого, иногда выгоднее не уничтожать все объекты и создавать их заново, а использовать API для эмуляции того поведения, которое было без JS.

---

*На самом деле, можно сохранить состояние в персистентных хранилищах вроде LocalStorage и IndexedDB, а также не выгружать некоторые объекты, используя Service Worker. Но всё это выходит за рамки нашей лекции.

XHR & AJAX

Перезагрузка страницы

Перезагрузка страницы фактически ведёт к рестарту Frontend-приложения. Это достаточно ресурсоёмко и не всегда нужно.

Перезагрузка страницы

Если мы подумаем, то окажется, что нам не нужно перезагружать страницу, нам нужно лишь отправить на сервер данные и получить их обратно. Соответственно, было разработано API, которое позволяет осуществлять HTTP запросы из JS без перезагрузки страницы (AJAX и XHR). При этом следует помнить, что без JS, чтобы отправить HTML форму, мы должны перезагрузить страницу (можно отправить запрос через URL, указав:

).

XHR

В JS для этого существует специальный объект XHR (XMLHTTPRequest). Несмотря на то, что в названии присутствует XML, этот объект в состоянии отправлять практически любые данные.

Давайте убедимся в этом.

Отправка формы через Query

```
{
  // GET с Query URL
  const data = new URLSearchParams();
  Array.from(form.elements)
    .filter((el : Element ) => el.name !== '') // только с атрибутом name
    .forEach((el : Element ) => data.append(el.name, el.value));

  const xhr = new XMLHttpRequest();
  xhr.open( method: 'GET', url: `/api?${data}`);
  xhr.send();
  form.reset(); // очистка формы
}
```

▼ **Query String Parameters** view source

value: My Avatar

avatar: C:\fakepath\avatar.jpg

`/api?${data}` - template literal, позволяющий "вшивать" в строку значение переменной.

При этом перезагрузки страницы не произойдёт.

XHR

На сервере:

```
server.getServerConfiguration().addHttpHandler((request, response) → {
    response.getWriter().write( str: "Ok");  response: Response@1877
}, ...mappings: "/api");
```


```
shutdown));
```

Отправка формы через Body

Автоматически выставляется application/x-www-form-urlencoded:

```
{
  // POST
  const data = new URLSearchParams();
  Array.from(form.elements)
    .filter((el : Element ) => el.name !== '') // только с атрибутом name
    .forEach((el : Element ) => data.append(el.name, el.value));

  const xhr = new XMLHttpRequest();
  xhr.open( method: 'POST',  url: `/api`);
  xhr.send(data);
  form.reset();
}
```

▼ **Form Data** view source view URL encoded

value: My Avatar

avatar: C:\fakepath\avatar.jpg

▼ **Request Headers** view parsed

```
POST /api HTTP/1.1
Host: localhost:9999
Content-Type: application/x-www-form-urlencoded;charset=UTF-8
```

XHR

На сервере:

```
server.getServerConfiguration().addHttpHandler((request, response) → {
    response.getWriter().write( str: "Ok");  response: Response@1877
}, ...mappings: "/api");
```


```
shutdown));
```

Отправка Multipart

FormData (автоматически выставляется multipart/form-data):

```
{
  // Multipart
  const data = new FormData(form);
  const xhr = new XMLHttpRequest();
  xhr.open( method: 'POST',  url: `/api`);
  xhr.send(data);
  form.reset();
}
```

▼ Form Data view source view decoded

value: My Avatar

avatar: (binary)

▼ Request Headers view parsed

```
POST /api HTTP/1.1
Host: localhost:9999
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7jOW68k1A5SWFsBH
```

Отправка Multipart

```
public void service(Request request, Response response) throws Exception {
  response.suspend();
  MultipartScanner.scan(request, multipartEntry -> {
    LOGGER.info(multipartEntry.getContentDisposition().getDispositionParam( paramName: "name"));
    // TODO: handle entry
  }, new EmptyCompletionHandler<>() {
    @Override
    public void completed(Request result) {
      response.resume();
      try {
        response.getWriter().write( str: "ok");
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    @Override
    public void failed(Throwable throwable) {
      response.resume();
    }
  });
}
```

XHR

Отправка бинарных данных:

```
{
  const data = new Blob( blobParts: ["some data"]);
  const xhr = new XMLHttpRequest();
  xhr.open( method: 'POST', url: `/api`);
  xhr.send(data);
}
```

▼ Request Payload

some data

▼ Request Headers view parsed

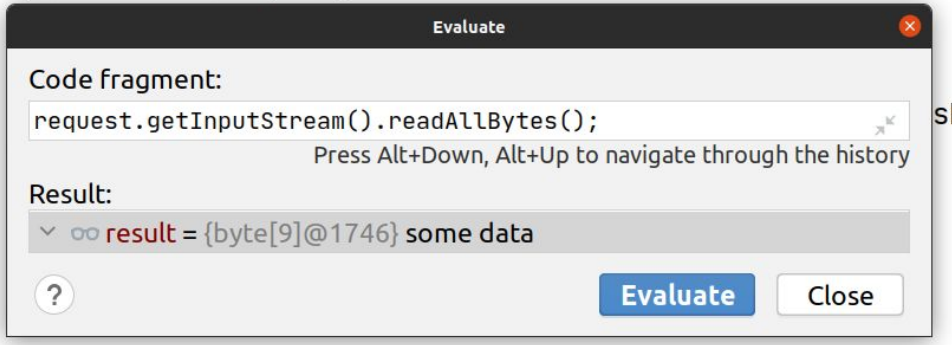
POST /api HTTP/1.1

Host: localhost:9999

XHR

Отправка бинарных данных:

```
server.getServerConfiguration().addHttpHandler(new HttpHandler() {
  @Override
  public void service(Request request, Response response) throws Exception {
    response.getWriter().write( str: "ok");  response: Response@1743
    //...
  }
}, ...mappings: "/api");
```




```
shutdown));
```

XHR

Отправка других данных (имеется возможность выставлять собственный Content-Type):

```
{
  const data = JSON.stringify( value: {key: 'value'});
  const xhr = new XMLHttpRequest();
  xhr.open( method: 'POST', url: `/api`);
  xhr.setRequestHeader( name: 'Content-Type', value: 'application/json');
  xhr.send(data);
}
```

запись объекта

▼ **Request Payload** view source

▼ {key: "value"}

key: "value"

▼ **Request Headers** view parsed

POST /api HTTP/1.1

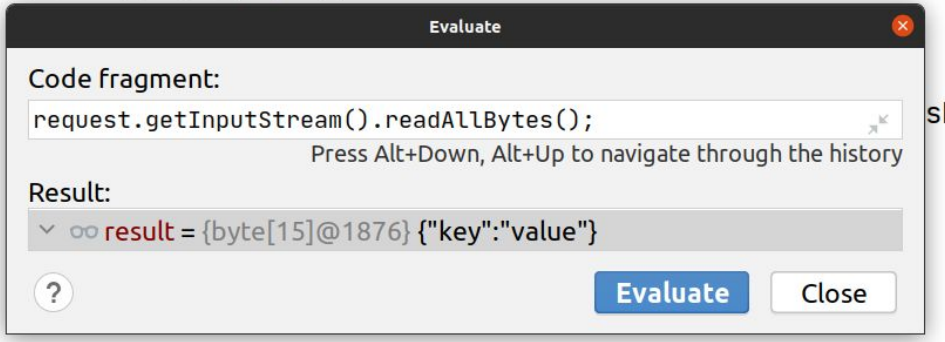
Host: localhost:9999

Content-Type: application/json

XHR

Отправка других данных (имеется возможность выставлять собственный Content-Type):

```
server.getServerConfiguration().addHttpHandler(new HttpHandler() {
  @Override
  public void service(Request request, Response response) throws Exception {
    response.getWriter().write( str: "ok");  response: Response@1873
    //...
  }
},  ...mappings: "/api");
```




```
shutdown));
```

XHR

Таким образом: с помощью XHR мы можем использовать как "традиционные" форматы отправки данных, так и "альтернативные".

REST

REST

Сам термин REST (Representational State Transfer) был введён Roy Fielding в своей диссертации в 2000 году.

В современной интерпретации это выглядит так:

- HTTP как транспорт для передачи сообщений
- статус коды как сигнал успешности выполнения запросов
- HTTP-методы в качестве "осмысленных" операций
- группировка данных в наборы ресурсов с доступным перечнем операций для них
- stateless - клиент должен передавать все необходимые данные для выполнения запроса
- json/xml как формат передачи для большинства запросов (исключение - бинарные данные)

REST

Мы говорим в "современной", потому что изначальный смысл термина (описанный в оригинальной работе) "потерялся" и под REST сейчас понимают именно те пункты, которые мы описали.

демонстрация

REST

Пример: API GitHub.

Ключевое: несмотря на то, что новые подходы к организации API (GraphQL, gRPC и т.д.), набирают обороты, REST по-прежнему остаётся одним из самых распространённых.

REST

List organization repositories

Lists repositories for the specified organization.

`GET` `/orgs/{org}/repos`

Parameters

Name	Type	In	Description
`accept`	string	header	Setting to **`application/vnd.github.v3+json`** is recommended. See preview notices
`org`	string	path	
`type`	string	query	Specifies the types of repositories you want returned. Can be one of **`all`**, **`public`**, **`private`**, **`forks`**, **`sources`**, **`member`**, **`internal`**. Default: **`all`**. If your organization is associated with an enterprise account using GitHub Enterprise Cloud or GitHub Enterprise Server 2.20+, **`type`** can also be **`internal`**.
`sort`	string	query	Can be one of **`created`**, **`updated`**, **`pushed`**, **`full_name`**.
`direction`	string	query	Can be one of **`asc`** or **`desc`**. Default: when using **`full_name`**: **`asc`**, otherwise **`desc`**
`per_page`	integer	query	Results per page (max 100)
`page`	integer	query	Page number of the results to fetch.

REST

Create an organization repository

Creates a new repository in the specified organization. The authenticated user must be a member of the organization.

OAuth scope requirements

When using OAuth, authorizations must include:

- `public_repo` scope or `repo` scope to create a public repository
- `repo` scope to create a private repository

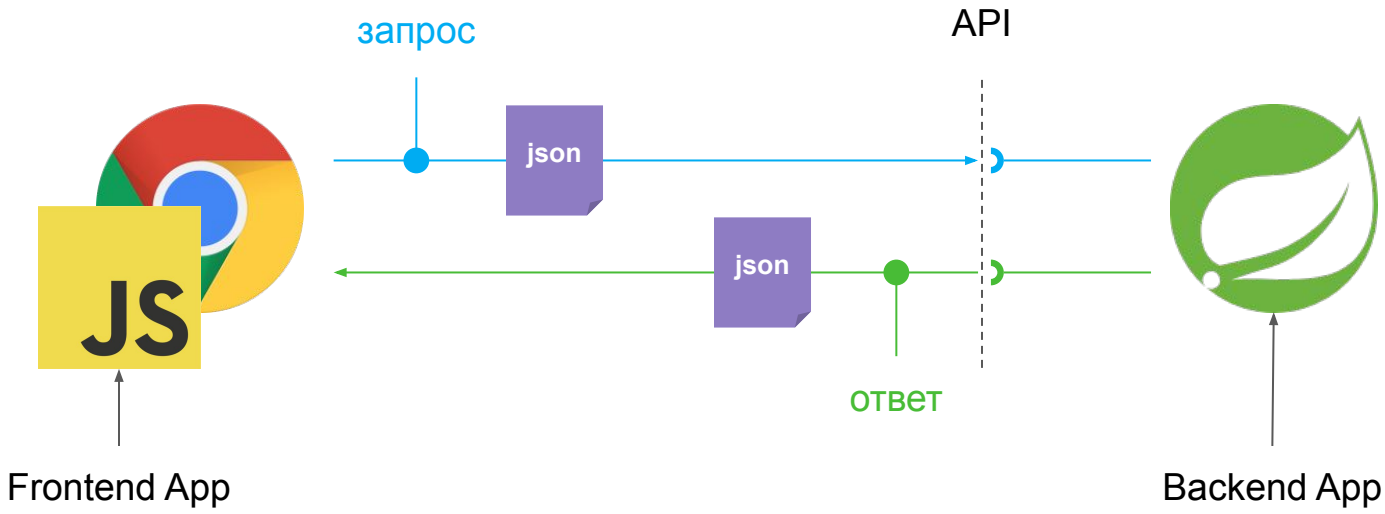
`POST` `/orgs/{org}/repos`

Parameters

Name	Type	In	Description
`accept`	string	header	Setting to **`application/vnd.github.v3+json`** is recommended. See preview notices
`org`	string	path	
`name`	string	body	**Required**. The name of the repository.
`description`	string	body	A short description of the repository.
`homepage`	string	body	A URL with more information about the repository.
`private`	boolean	body	Either **`true`** to create a private repository or **`false`** to create a public one.

REST

При этом REST не отменяет ничего из того, что мы проходили. Эта картинка не означает, что передаётся только JSON. Данные могут передаваться и в Header'ах, и в Path и в Query, а файлы - с помощью Multipart либо в виде бинарного тела.



SOP & CORS

SOP (Same Origin Policy)

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

До сих пор наша веб-страница, с которой мы отправляем данные, и сервер, на который мы отправляем данные, имели одинаковый Origin.

Но стоит нам открыть страницу через встроенный веб-сервер IDEA и попытаться отправить любой запрос:

```
Access to XMLHttpRequest at 'http://localhost:9999/api'    index.html:1
from origin 'http://localhost:63342' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
▶POST http://localhost:9999/api net::ERR_FAILED                app.js:62
```

Причём на сервер запрос придёт.

SOP

При этом формы без JS и ресурсы (если они есть) без проблем будут грузиться с другого Origin'а.

Проще говоря, из JS запрещён доступ к другому Origin'у (в упрощённом представлении).

CORS

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

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.

CORS

Q: почему это важно для нас?

A: потому что вы должны понимать, что от вас будут хотеть Frontend-разработчики, когда будут говорить про CORS.

PREFLIGHT

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 предопределённых + ещё несколько правил.

PREFLIGHT

В нашем же случае, несмотря на то, что в JS прописан метод POST, отправляется запрос типа OPTIONS:

```
{
  // SOP Demo
  const data = JSON.stringify( value: {key: 'value'});
  const xhr = new XMLHttpRequest();
  xhr.open( method: 'POST', url: `http://localhost:9999/api`);
  xhr.setRequestHeader( name: 'Content-Type', value: 'application/json');
  xhr.send(data);
}
```

▼ General

Request URL: http://localhost:9999/api

Request Method: OPTIONS

Status Code: 200 OK

Remote Address: [::1]:9999

Referrer Policy: no-referrer-when-downgrade

PREFLIGHT

Дело в том, что все запросы, которые не подходят под условие Simple Request, считаются Preflight и идут по следующему сценарию:

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

Итоги

Итоги

Сегодня мы посмотрели на основы JS, на то, каким образом могут отправляться данные с помощью него.

Самое главное, что мы увидели:

1. Сохраняются те же форматы, что мы уже видели (form, multipart)
2. Добавляются новые (json)

Именно эти знания нам помогут понять, почему в инструментах, которые мы будем изучать далее выбраны те или иные подходы.

Итоги

Стоит отметить, что мы рассмотрели лишь самые важные возможности JS в части отправки данных с использованием HTTP. Возможностей у современного JS гораздо больше:

- можно персистентно хранить данные на клиенте
- можно использовать SSE, WebSockets
- работа в оффлайн режиме
- обсуждается возможность прямого использования TCP и даже создания серверов внутри браузера

Итоги

Нам, как бэкенд разработчикам, желательно отслеживать эти тенденции. Поскольку чем больше возможностей будет у Frontend'а в плане интеграции, тем больше возможностей будет у нас в части организации Backend'а.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров