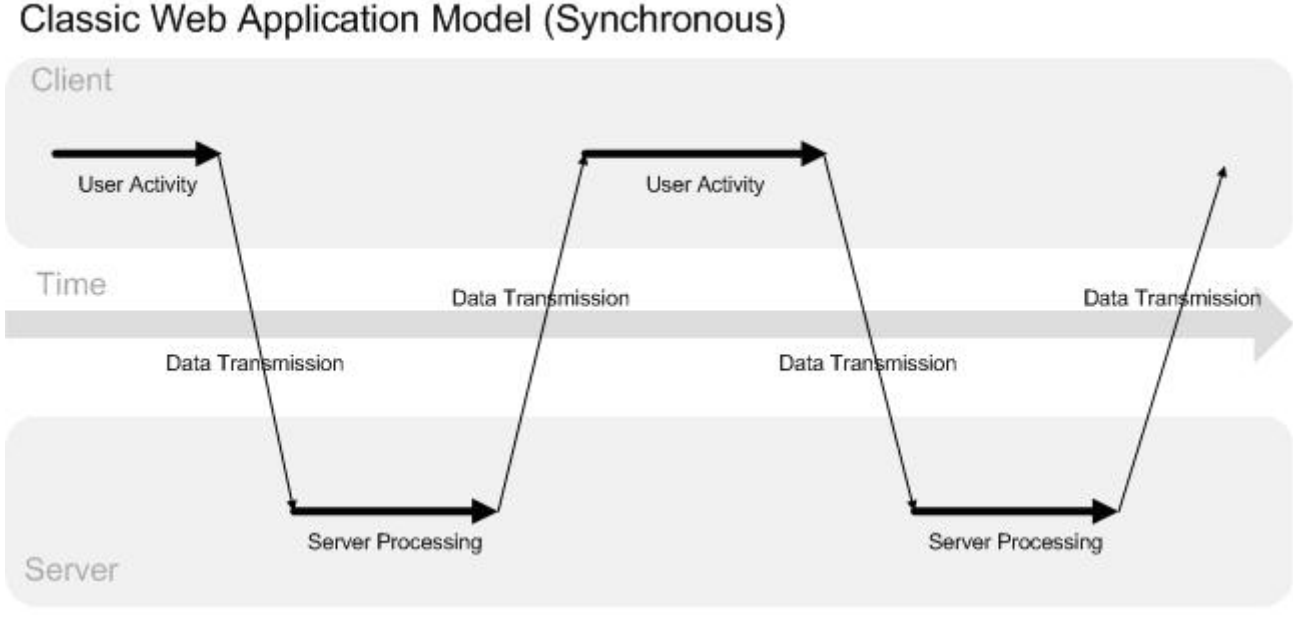
JS в браузере

Эволюция web-приложений: long time ago

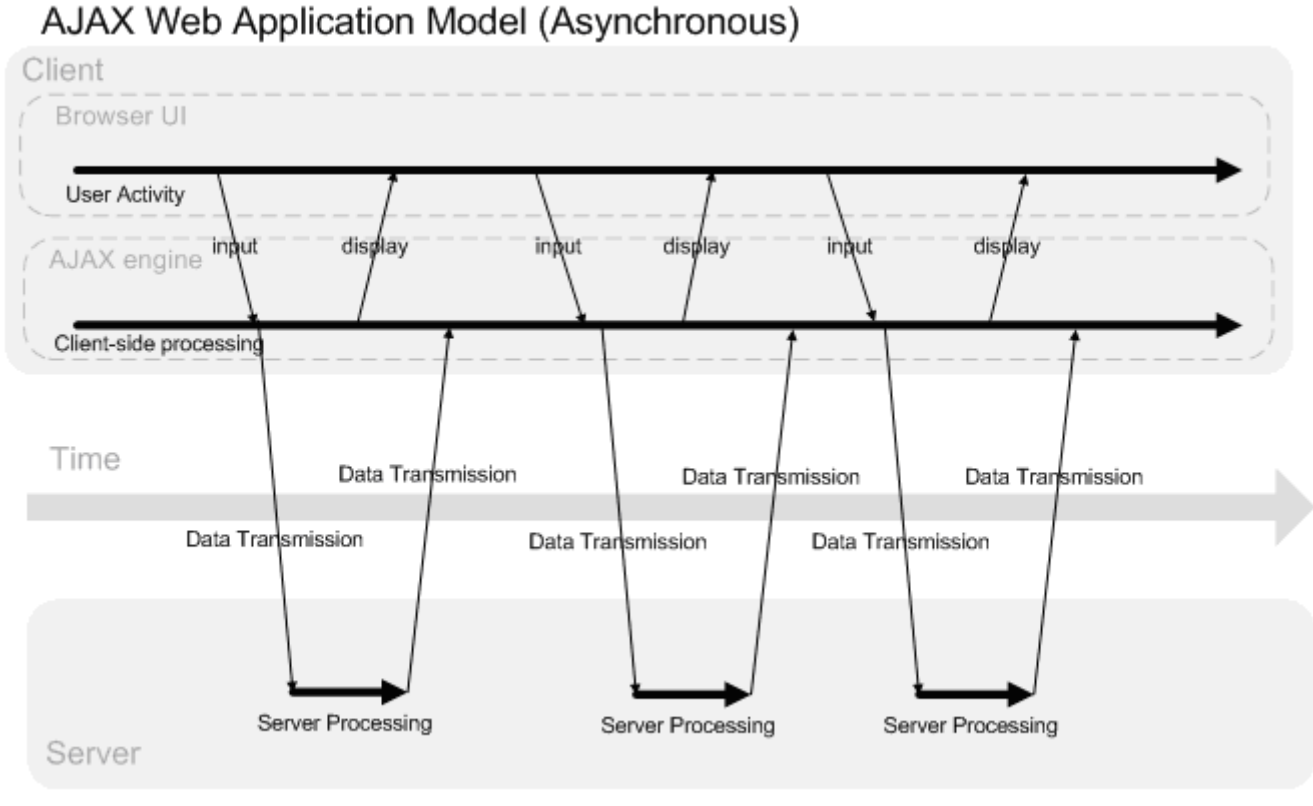
- Классическая схема web-приложения не содержит JS вовсе
- (1) Браузер делает HTTP-запрос на сервер
- (2) Сервер генерирует HTML + CSS для отображения в браузере
- Pros: простота, не требует JS
- Cons: в каждом ответе дублируется часть разметки (header, footer)
- сервер отвечает за генерацию HTML (серверу это не нравится)



Эволюция web-приложений: AJAX

- Async Javascript and XML - технология общения с сервером без перезагрузки страницы
- клиентский JS запрашивает отдельные данные или части страницы
- Pros:
  - уменьшает нагрузку на сервер
  - уменьшает передаваемый объем данных (при длительной работе)
  - улучшает UI/UX ("отзывчивый интерфейс")
  - может использовать любые методы и способы передачи данных HTTP
- Cons:
  - начальная загрузка страницы занимает больше времени
  - код клиента и сервера усложняется
  - код отображения разметки дублируется на сервере и клиенте

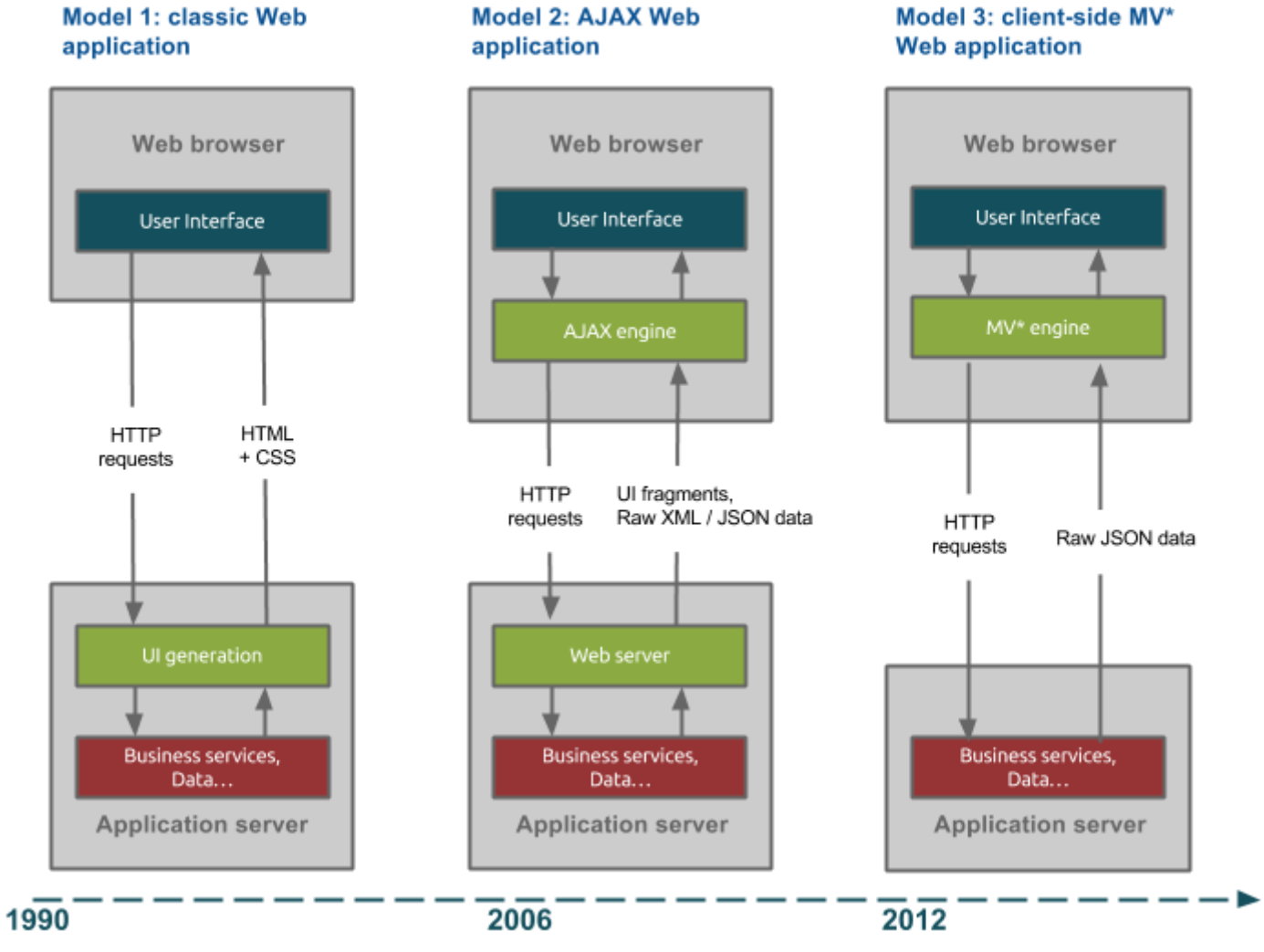
Эволюция web-приложений: AJAX



Эволюция web-приложений: SPA

- Single Page Application (SPA) - web-приложение, в котором вся логика генерации и отображения HTML содержится в JS
- сервер отвечает только за хранение и обработку данных
- страница никогда не перезагружается
- данные передаются в формате JSON
- Pros:
  - сервер освобождается от задачи генерации HTML
  - один сервер может работать с любыми клиентами
  - наилучший UI/UX из возможных
- Cons:
  - проблемы с индексацией в поисковиках
  - не работают с отключенным JS

Эволюция web-приложений



XMLHttpRequest

Синхронный запрос (не повторяйте этого дома!)

```
let xhr = new XMLHttpRequest();
xhr.open('GET', 'http://www.yoursite.com/',
    false);

// посылаем AJAX запрос
xhr.send();

// браузер висит и ждет ответа от сервера

// получили ответ от сервера
if (xhr.status === 200) {
    // запрос удачно завершен
    console.log(xhr.responseText);
} else {
    // обработка ошибки
}
```

Асинхронный запрос (а вот это повторяйте)

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://www.yoursite.com/',
    true);

// Событие изменения состояния запроса
xhr.addEventListener('readystatechange',
    function(e) {
    // интересует только завершенный запрос
    if (!xhr.readyState === 4) return;

    if(xhr.status === 200) {
        console.log(xhr.responseText);
    } else {
        // обработка ошибки
    }
});

// посылаем AJAX запрос
xhr.send();
```

События XMLHttpRequest

Действие	Событие	xhr.readyState
XMLHttpRequest создан	-	0
Вызов xhr.open()	-	1
Вызов xhr.send()	loadstart	1
Получены заголовки ответа	-	2
Получен пакет данных	progress	3
Ошибка	abort, error, timeout	4, xhr.status !== 200
Успешный запрос	load	4, xhr.status === 200
Запрос завершен	loadend	4

XMLHttpRequest API

`xhr.open(method, url, async)`	Создание нового XMLHttpRequest
`xhr.setRequestHeader(name, data)`	Установка заголовков запроса
`xhr.send([body])`	Посылка AJAX-запроса
`xhr.readyState`	Статус объекта XHR
`xhr.status/statusText`	Статус ответа сервера
`xhr.responseText/responseXML`	Ответ сервера, строка/DOM-документ
`xhr.responseType`	Ожидаемый тип данных (text|json|document|blob etc.)
`xhr.response`	Ответ сервера с учетом responseType

Отправка данных

- HTTP - текстовый протокол, и мы можем вручную задать все части запроса
- HTTP позволяет отправлять данные в URL (query string) и в теле запроса

```
POST /path/script.cgi?key1=value1&key2=value2 HTTP/2.0
Host: mysite.com
. . .
Content-Type: application/x-www-form-urlencoded
Content-Length: 23

key3=value3&key4=value4
```

Отправка данных в query string

- данные в query string должны быть закодированы через urlencoded
- urlencoded: строка вида `'key1=value1&key2=value2&...'`
- всё кроме латинских букв и некоторых спецсимволов заменяется на цифровой код в UTF-8 со знаком %

```
var params = 'param1=' + encodeURIComponent(value1) + '&'
             'param2=' + encodeURIComponent(value2);
xhr.open('GET', 'http://tinyurl.com/4poyc6x?' + params, true);
// http://tinyurl.com/4poyc6x?param1=%D0%A6%20%D0%A6&param2=hello
```

Отправка данных в теле запроса

- отправить данные в теле запроса можно в любой кодировке
- однако лучше использовать одну из стандартных:

application/x-www-form-urlencoded

```
Content-Type: application/x-www-form-urlencoded

foo=bar&baz=The+first+line.%0D%0AThe+second+line.%0D%0A
```

text/plain

```
Content-Type: text/plain

foo=bar
baz=The first line.
The second line.
```

Отправка данных в теле запроса

multipart/form-data

```
Content-Type: multipart/form-data; boundary=---------------------------314911788813839

-----------------------------314911788813839
Content-Disposition: form-data; name="foo"

bar
-----------------------------314911788813839
Content-Disposition: form-data; name="baz"

The first line.
The second line.

-----------------------------314911788813839--
```

```
let formData = new FormData(), xhr = new XMLHttpRequest();

formData.append("username", "Groucho");
formData.append("accountnum", 123456); // коневертируется в строку
// Выбранный пользователем файл из input type="file"
formData.append("userfile", fileInputElement.files[0]);

xhr.open("POST", "http://foo.com/submitform.php");
xhr.send(formData); // multipart/
```

FormData

- API, позволяющий добавлять пары "ключ-значение" и передать их в XHR
- может содержать файлы и объекты Blob (JS-строка с mime-типом)
- при передаче в XHR данные автоматически кодируются как `multipart/form-data`
- `FormData` также может принять аргументом обычную HTML-форму

```
let formElement = document.querySelector("form"),
    xhr = new XMLHttpRequest(),
    form = new FormData(formElement);

let content = '<a id="a"><b id="b">hey!</b></a>',
    blob = new Blob([content], { type: "text/xml"});
form.append("webmasterfile", blob);

xhr.open("POST", "submitform.php");
xhr.send(form);
```

Отправка данных в теле запроса

application/json

```
Content-Type: application/json;

{"key1":"value1","key2":"value2","number1":1234,"arr":[1,2,"a"]}


var xhr = new XMLHttpRequest();
xhr.open("POST", "/json-handler");
xhr.setRequestHeader("Content-Type", "application/json");

xhr.send('{"key1":"value1","key2":"value2","number1":1234,"arr":[1,2,"a"]}' );
```

Формат JSON

- Предназначен для передачи и хранения данных
- Содержит минимально необходимый набор типов данных:
  - JS-объект (ключ-значение); ключи обязательно в двойных кавычках
  - JS-массив
  - строки в двойных кавычках | числа | `true` | `false` | `null`
- `JSON.stringify(JSON [, replacer])`- преобразует JS-объект в строку
  - При сериализации объекта пытается вызвать его метод toJSON
  - `replacer` - фильтр полей для сериализации; массив|функция
- `JSON.parse(JSONString [, reviver])`- извлекает JS-данные из строки

```
let obj = {a: 1, b: 2, c: 3},
    xhr = new XmlHttpRequest();

xhr.open("POST", "/json-handler");
xhr.setRequestHeader("Content-Type", "application/json");

xhr.send(JSON.serialize(obj));
```

Непрерывное получение данных (COMET)

- Polling (частые опросы)
  - Клиент посылает запросы обновлений с некоторой периодичностью
  - Задержка между возникновением события и его отправкой
  - Pros: Простота реализации
  - Cons: Дополнительная нагрузка на сервер
- Long polling (длинные опросы)
  - Клиент посылает запрос обновлений, который не закрывается до тех пор, пока не возникнут обновления; после получения обновлений клиент сразу же открывает новый запрос
  - Pros: Как только появляются данные, происходит их отправка
  - Cons: Сервер должен уметь работать с большим числом "висящих" соединений

Websockets

- Протокол, работающий над HTTP; требует специальной поддержки сервером
- События: `open`, `close`, `message`, `error`

```
let socket = new WebSocket("ws://javascript.ru/ws");
socket.onmessage = function(event) {
    console.log(event.data);
}
socket.send('hello');
```

`sessionStorage`/`localStorage`

- Позволяют хранить данные в виде "ключ-значение"
- `sessionStorage` очищается при завершении сессии
- `localStorage` хранит данные постоянно
- `Storage`: `length`, `setItem`, `getItem`, `removeItem`, `clear`
- можно хранить только строки

```
localStorage.setItem('userSettings', JSON.serialize(userSettings));
```