

ПЛАТФОРМЫ: БРАУЗЕР VS NODE.JS



ИЛЬЯ МЕДЖИДОВ



ИЛЬЯ МЕДЖИДОВ

CEO в RageMarket

 medzhidov@ragemarket.ru



ПЛАН ЗАНЯТИЯ

1. [Платформы](#)
2. [Общие подходы](#)
3. [Глобальный объект](#)
4. [Браузер](#)
5. [Node.js](#)



ПЛАТФОРМЫ



JS

JS настолько распространённый язык на сегодняшний день, что используется практически во всех сферах:

- браузерные приложения
- серверные приложения
- desktop приложения
- мобильные приложения
- IOT
- и т.д.



JS

Возможность выучить один язык и использовать его на разных платформах - всегда была достаточно заманчивой идеей.

И если у других языков это получалось только частично (в браузерах альтернатив нет), то у JS получается достаточно неплохо.

Ключевое: в большей части случаев не получится "переиспользовать" код на разных платформах, но возможность использовать один язык - уже хорошо.



ПОПУЛЯРНОСТЬ

Мы оставим в стороне вопрос популярности на тех или иных платформах (споры вроде Node.js vs Go vs Java).

Наша задача понять общие принципы, которые вам позволят работать с этим языком на разных платформах (ведь и сами платформы различаются).



ОБЩИЕ ПОДХОДЫ



ОБЩИЕ ПОДХОДЫ

Ключевых подходов для исполнения JS на разных платформах два:

1. WebView - интеграция браузера в приложение
2. VM - фактически, подход без WebView




WEBVIEW

Основная идея сводится к следующему: в программную оболочку в виде отдельного компонента интегрируется целый веб-браузер.


Этот самый веб-браузер может показываться как часть какого-то экрана приложения, так и в виде самостоятельного экрана:



Оплата картой




Номер карты



Срок действия

/

CVV Код 

Сумма платежа

Email



WEBVIEW

То, какой браузер встраивается, целиком зависит от платформы: например, на iOS это скорее всего будет Safari, а на Android - Chrome.



WEBVIEW

WebView - это самая частая опция при интеграции платёжных систем, но иногда может использоваться и для других целей.

Например, мобильное приложение - всего лишь оболочка к сервису с веб-интерфейсом, у которого нет API. Т.е. просто загружаются адаптивные веб-странички с JS.

WEBVIEW

В такой схеме есть одна очень интересная возможность: можно организовать коммуникацию между нативным кодом и кодом на странице.

Особенности такой коммуникации целиком зависят от платформы, но, например, [в Android](#), это можно организовать следующим образом:

```
// определяем класс для объекта, который будем "пробрасывать в JS"
class WebAppIntegration(private val context: Context) {
    @JavascriptInterface
    fun showToast(toast: String) {
        Toast.makeText(context, toast, Toast.LENGTH_SHORT).show()
    }
}
```

WEBVIEW

Сам проброс осуществляется следующим образом:

```
webView.addJavascriptInterface(WebAppIntegration(this), "Android")
```

После чего мы можем прямо из JS вызывать функции указанного объекта:

```
document.getElementById('show-toast').addEventListener('click', () => {  
  Android.showToast('JS Works!');  
});
```




WINDOW

Вопрос к аудитории: что вы знаете об объекте `window`?

Вопрос к аудитории: что вы знаете о `глобальном объекте`?

ES

Чтобы понять, как и почему это работает, давайте обратимся к спецификации ECMAScript.

В ней написано следующее (частичные вырезки):

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax and semantics described in this specification.

A conforming implementation of ECMAScript may provide additional types, values, objects, properties, and functions beyond those described in this specification.

... the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific objects, whose description and behaviour are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

ВОЛЬНЫЙ ПЕРЕВОД

1. Реализация должна предоставлять все типы, значения, объекты и т.д., описанные в данной спецификации (включая семантику и синтаксис)
2. Реализация может предоставлять все типы, значения, объекты и т.д., выходящие за рамки (не описанные) в этой спецификации
3. ... помимо объектов и других возможностей, описанных в спецификации, программное окружение предоставляет объекты, которые зависят от окружения и доступ к которым предоставляется из ES

Таким образом, мы получаем, что на самом деле, спецификация ES - это всего лишь часть того, с чем вам предстоит работать.



ENVIRONMENT

И чаще всего, гораздо более важными* являются объекты, которые предоставляются вне рамок спецификации (как например, мы рассмотрели выше).

Примечание*: конечно же при условии, что вы уже знаете и умеете работать с самим языком и стандартными объектами.

GLOBAL OBJECT

В спецификации описан глобальный объект `global object`, который и предоставляет доступ ко всем описанным в спецификации типам (и не только):

*... may have host defined properties in addition to the properties defined in this specification.
This may include a property whose value is the global object itself.*

На самом деле, если в примере с Android мы будем использовать `window.Android`, а не просто `Android`, то никакой разницы не будет.

Но прежде чем двинуться дальше, давайте рассмотрим второй вариант.

VM

Второй вариант заключается в том, что встраивают не целиком браузер, а только "часть" его, которая называется Engine, Interpreter, VM и т.д.*

Т.е. условно, внутри каждого браузера есть компонент, обрабатывающий JS: у Chrome - [V8](#), у Firefox - [SpiderMonkey](#), в Safari - [JavaScriptCore](#).

Этот самый компонент можно использовать "отдельно" от браузера, предоставив программистам возможность писать на JS, но не иметь браузерного окружения.

Помимо компонентов из браузера, есть совершенно отдельные реализации, которые вообще никогда в составе браузеров не использовались: например, [GraalVM](#).

Примечание*: мы не будем "закапываться" в терминологию.



VM

При этом некоторые браузерные реализации (V8 для Node.js, JavaScriptCore для React Native) используются отдельно, чтобы предоставить вам доступ к программному окружению, не описанному в спецификации.

Ключевое: что в эти реализации так же можно "пробрасывать" объекты, не описанные в спецификации, которые позволяют вам, в первую очередь, взаимодействовать с окружением, на котором работает ваш скрипт (в большинстве случаев - конкретным устройством).

ЗАЧЕМ ЭТО ВСЁ?

На сегодняшний день сфера применения JS не ограничена приложениями в браузере, и даже не ограничена приложениями Node.js.

Вы можете использовать React Native или Native Script и даже (экзотика) писать [скрипты для Photoshop](#).

Конечное же, наиболее частыми сценариями (в порядке убывания) будут:

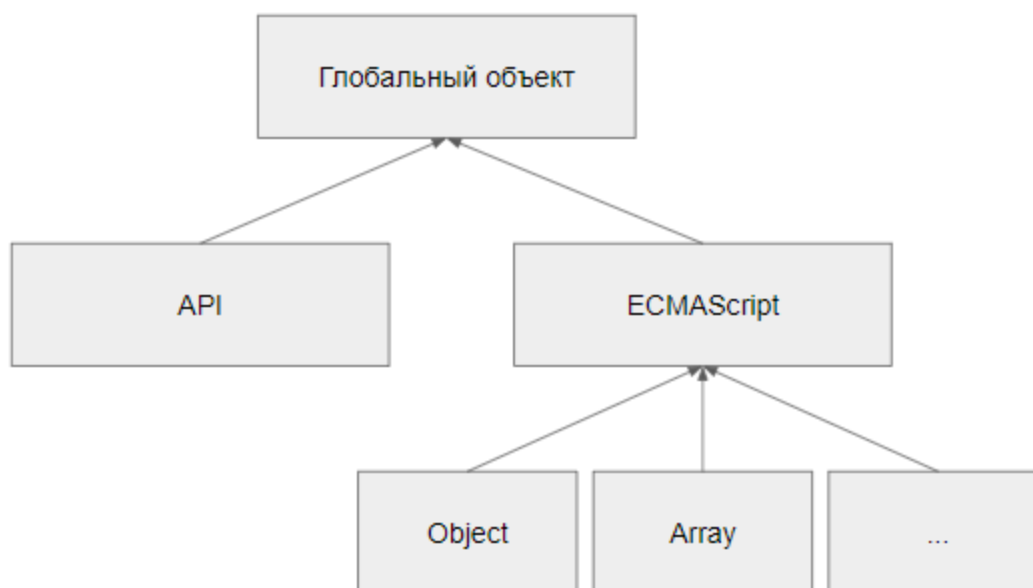
1. Веб-приложения
2. Node.js
3. Гибридные приложения
4. React Native, Native Script
5. Скриптинг



ГЛОБАЛЬНЫЙ ОБЪЕКТ

ГЛОБАЛЬНЫЙ ОБЪЕКТ

Полезно запомнить следующую картинку:



Она имеет смысл в рамках стандарта ЕСМА, заявляя, что все "нелокальные" имена должны являться свойствами глобального объекта*.

Примечание*: пока не говорим о модулях.

ГЛОБАЛЬНЫЙ ОБЪЕКТ

Что именно это значит? Когда мы с вами берём код:

```
const id = Number(inputEl.value);
```

Откуда берётся имя `Number`?

Или:

```
const audio = new Audio('/audio/new-message.mp3');  
audio.play();
```

Откуда берётся имя `Audio`?



ГЛОБАЛЬНЫЙ ОБЪЕКТ

JS устроен так, что любое имя сначала ищется в локальной области видимости, потом выше и выше, пока области видимости не закончатся.

Если объект при этом поиске найден не будет, то берётся глобальный объект и имя используется в качестве имени поиска для свойства.



ГЛОБАЛЬНЫЙ ОБЪЕКТ

Здесь стоит отметить, что как именно он реализован средой выполнения не принципиально, главное, что он (или его имитация) имеются.

ГЛОБАЛЬНЫЙ ОБЪЕКТ

Т.е., вспоминая про Android, получается что имя `Android` было добавлено в глобальный объект (об этом мы уже говорили).

Этому принципу следуют большинство реализаций.

При этом объекты, определённые в рамках спецификации, тоже являются свойствами глобального объекта:

```
> window.Number === Number
```

```
< true
```

```
> |
```

ГЛОБАЛЬНЫЙ ОБЪЕКТ

В разных окружениях глобальный объект называется по-разному:

- `window` - в браузере
- `global` - в Node.js

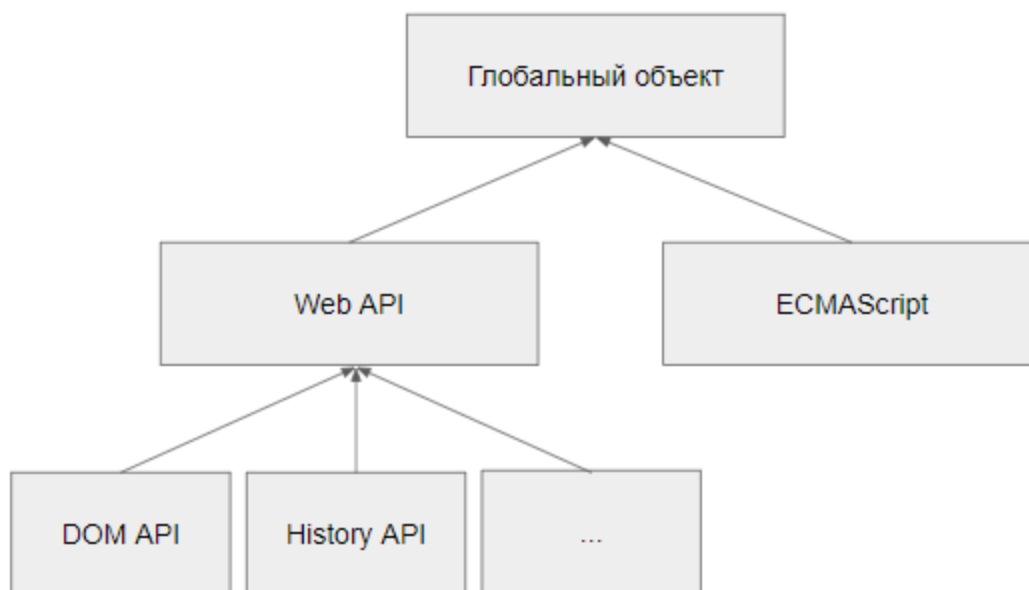
Но в самых новых версиях спецификации ему дали общее имя - `globalThis`.



БРАУЗЕР

БРАУЗЕР

В браузере глобальный объект представлен объектом `window`:



Здесь стоит отметить, что если вы попытаетесь вывести в консоль объект `window`, то получите "гигантскую портянку", просто потому, что объём API, реализуемых в дополнение к спецификации [достаточно велик](#).

DOM API

И самое важное, что [DOM API](#) - всего лишь одно из API.

Так же, как и [Console API](#), которое вам предоставляет объект `console`.

Ключевое: вы должны понимать, откуда берутся те типы, которые вы используете.



NODE.JS



NODE.JS

[Node.js](#) - это среда выполнения, построенная на базе V8.

Ключевых применений два:

1. Построение серверных приложений (backend)
2. Разработка инструментов для frontend'a



FRONTEND TOOLS

Те инструменты, которые мы с вами использовали (ESLint, Babel, Webpack) работают благодаря Node.js.

Общую схему Node.js можно представить так же, как и все предыдущие (вам даются ECMAScript объекты + API), но она немного хитрее.

GLOBAL

Глобальный объект в Node.js называется `global`.

Напишем небольшой скрипт и запустим его на исполнение:

```
// файл app.js  
console.log(global);
```

Запуск осуществляется с помощью команды `node app.js`

GLOBAL

При распечатке в консоли мы получим немного:

```
Object [global] {  
  global: [Circular],  
  clearInterval: [Function: clearInterval],  
  clearTimeout: [Function: clearTimeout],  
  setInterval: [Function: setInterval],  
  setTimeout: [Function: setTimeout] {  
    [Symbol(nodejs.util.promisify.custom)]: [Function]  
  },  
  queueMicrotask: [Function: queueMicrotask],  
  clearImmediate: [Function: clearImmediate],  
  setImmediate: [Function: setImmediate] {  
    [Symbol(nodejs.util.promisify.custom)]: [Function]  
  }  
}
```

GLOBAL

Но на самом деле, полное описание его есть в документации, из которого выясняются важные вещи:

1. `__dirname`, `__filename`, `exports`, `module` и `require()` всего лишь имена, "вживляемые" в каждый модуль и доступные в контексте модуля
2. `process` - объект, предоставляющий доступ к процессу, в том числе его переменным окружения

Вы часто будете видеть в коде инструментов обращение к `process.env`.

MOMENT

А теперь давайте вернёмся к прошлой лекции и посмотрим ещё раз на код Moment.js:

```
1  ;(function (global, factory) {  
2      typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory() :  
3      typeof define === 'function' && define.amd ? define(factory) :  
4      global.moment = factory()  
5  })(this, (function () { ... }));
```

Теперь вы должны без проблем прочитать вторую и четвертую строки.

MOMENT

А теперь давайте вернёмся к прошлой лекции и посмотрим ещё раз на код Moment.js:

```
1 | ;(function (global, factory) {  
2 |   typeof exports === 'object' && typeof module !== 'undefined' ? module.exports = factory() :  
3 |   typeof define === 'function' && define.amd ? define(factory) :  
4 |   global.moment = factory()  
5 | })(this, (function () { ... }));
```

Фактически, вторая строка определяет: работаем ли мы в CommonJS (CJS), если да - то использует `module.exports`.

А на четвёртой строке библиотека добавляет свой ключевой "объект" в качестве свойства `global`.

Ради полноты картины следует сказать, что третья строка отвечает за систему `amd`, которая уже мало используется.

GLOBAL

Почему мы заострили внимание именно на этом?

Потому что во frontend-разработке Node.js будет окружать вас везде и вы будете конфигурировать инструменты, на которых основана ваша работа именно основываясь на том, как работает Node.js.

Вы должны понимать, что несмотря на то, что и в браузере (Chrome), и в Node.js используется V8, это ни разу не значит, что среды идентичны:

- набор предоставляемого API разный
- версия V8 (а соответственно и уровень реализации ES) могут быть разные



ИТОГИ



ЧЕМУ МЫ НАУЧИЛИСЬ

Мы с вами обсудили достаточно фундаментальную вещь, позволяющую вам понять, что JS живёт далеко за пределами спецификации ES*.

И вы должны понимать, как ищутся объекты, откуда они берутся и какие в этом плане есть различия между платформами.

Конечно же, в первую очередь вы будете использовать браузер и Node.js, но вполне можете столкнуться и с другими.

Примечание*: строго говоря, сама спецификация об этом говорит.



Задавайте вопросы и напишите отзыв о лекции!

ИЛЬЯ МЕДЖИДОВ

 medzhidov@ragemarket.ru