Babel, or how to use next generation JavaScript nowadays?

Babel

От моя гледна точка Babel най-лесно се изполва от NPM.
По тази причина се налага първо да обясним какво е NPM.
NPM е подразбиращият се package manager на JavaScript runtime environment-а- NodeJS.
NodeJS по-принцип се използва за разработка на backend web приложения, но по-голямата си популярност придобива чрез екосистемата от инструменти, развити около неговия package manager - NPM.

scaffolding tools

Изключително голяма популярност имат например т.нар. scaffolding tools.

Това са инструменти за бърза генерация на скелет на проект на типично приложение.

Един от тях е например yeoman:

http://yeoman.io/ (http://yeoman.io/)

Той поддържа изключително голям списък от генератори, които можете да видите тук:
http://yeoman.io/generators/ (http://yeoman.io/generators/)

Освен генерация на стартовото приложение, тези генератори поддържат и полезни инструменти за по-нататъшната разработка на приложението.
Например, както може да се види тук:

https://github.com/yeoman/generator-angular#readme

Генераторът на AngularJS поддържа генериране на нови контролери, директиви, филтри, view-та и така нататък.
Нещата отиват и доста по-далеч. Още с началото на проекта вие вече имате организиран билд процес +
минификация на приложението.

Пример за малко нещо, направено с yeoman, можете да видите тук:
http://gonaumov.github.io/easterEggs/ (http://gonaumov.github.io/easterEggs/)

На който му е интересно, може да разгледа кода ето тук:
https://github.com/gonaumov/easterEggs (https://github.com/gonaumov/easterEggs)

Разработката ми отне около два часа. Бяха ми спестени - сваляне на библиотеки и на структурата на приложението – контролери, директиви, сървиси и тн.

Тенденция при command line tools.

Има тенденция все повече инструменти да стават npm пакети.
Един пример за тази тенденция е, например jpm. Jpm е инструмент от командния ред, поддържан от Мозила, служи за разработване,
тестване и пакетиране на браузерни добавки на Mozilla Firefox:

https://developer.mozilla.org/en-US/Add-ons/SDK/Tools/jpm (https://developer.mozilla.org/en-US/Add-ons/SDK/Tools/jpm)

jpm се явява наследник на cfx инструмента, който си вървеше със старите версии на Mozila FireFox add-on SDK.

Зависимости в npm

Npm държи списъка от зависимости във .json файл, наречен package.json. Ето пример за типичен package.json:

```
{
  "name": "babelpresentation",
  "version": "1.0.0",
  "description": "babel presentation",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Georgi Naumov",
  "license": "ISC",
  "devDependencies": {
    "babel-cli": "^6.11.4"
  }
}
```

Къде се съдържат модулите?

npm модулите се съдържат обикновено във директорията node_modules в същия проект. Ако я няма командата npm install, ще ви свали нужните модули и ще създаде директорията.

Смешната страна на npm.

Тези дни имаше един забавен случай, когато се оказа, че има asci art във babel-core-а, но той е премахнат вече. Нека разгледаме какво има в този PR:
https://github.com/babel/babel/pull/3656 (https://github.com/babel/babel/pull/3656)
![image](./guy_fieri_ascii_art-1471349356494.png =100x20)

Защо ни е нужен Babel?

Babel е де факто конвъртър от EcmaScript 6 към EcmaScript 5 с чиято помощ можем да ползваме EcmaScript 6 стандарта, преди да бъде поддържан на определената среда.

Трансформатори на синтаксис.

Babel поддържа текущата версия на EcmaScript – EcmaScript 2015 с помощта на трансформатори на синтаксис. Това са плъгини, които ти позволяват да ползваш текущия синтаксис на EcmaScript 2015 точно сега – без да очакваш поддръжка в браузерите.
Babel ни осигурява това чрез babel-preset-es2015 плъгина, който обединява няколко неща.

Arrows и Lexical This

Arrow function изразите са по-кратък синтактис в сравнение със expression функциите и bind-ват текущия this към контекста на
функцията. Де факто са по-кратък и бърз начин за дефиниране на анонимна функция.

```
var a = () => {};
var a = (b) => b;

const double = [1,2,3].map((num) => num * 2);
console.log(double); // [2,4,6]

var bob = {
  _name: "Bob",
  _friends: ["Sally", "Tom"],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};
console.log(bob.printFriends());
```

Класове

В ES2015 класовете са просто sugar syntax около прототипно-базираното
ООП. Тъй като имат по-просто декларативна форма, те са по-лесни за употреба.
Класовете поддържат прототипно базирано наследяване, извикване на базовия клас със super, инстанциране, статични методи и конструктори.

Пример за клас.

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

Подобрени object literals

Object literals са подобрени да поддържат задаване на простотипа по време на създаване, по-кратък начин на foo: foo задаване, дефиниция на методи и викане на функция от парента (super call).

```
var obj = {
    // Sets the prototype. "__proto__" or '__proto__' would also work.
    __proto__: theProtoObj,
    // Computed property name does not set prototype or trigger early error for
    // duplicate __proto__ properties.
    ['__proto__']: somethingElse,
    // Shorthand for ‘handler: handler’
    handler,
    // Methods
    toString() {
     // Super calls
     return "d " + super.toString();
    },
    // Computed (dynamic) property names
    [ "prop_" + (() => 42)() ]: 42
};
```

Template стрингове

Template стринговете осигуряват syntactic sugar за създаване на стрингове. Това е подобно
на стринговата интерполация в Perl, Python и други. По желание може да бъде добавен таг,
което ни позволява конструирането на стринга да бъде по-custom, избягване на

инжекшън
атака или сглобявана на по-високо ниво структура от данни от съдържанието на стринга.

```
// Basic literal string creation
`This is a pretty little template string.`

// Multiline strings
`In ES5 this is
 not legal.`

// Interpolate variable bindings
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`

// Unescaped template strings
String.raw`In ES5 "\n" is a line-feed.`

// Construct an HTTP request prefix is used to interpret the replacements and
construction
GET`http://foo.org/bar?a=${a}&b=${b}
    Content-Type: application/json
    X-Credentials: ${credentials}
    { "foo": ${foo},
      "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

Destructing

Destructing синтаксисът представлява JavaScript израз, който прави възможно да екстрактваме данни от масиви или обекти в различни променливи.
Това се постига с т.нар pattern matching.

Пример за destructing

```
// list matching
var [a, ,b] = [1,2,3];
a === 1;
b === 3;

// object matching
var { op: a, lhs: { op: b }, rhs: c }
       = getASTNode()
```

Още примери за destructing

```
// object matching shorthand
// binds `op`, `lhs` and `rhs` in scope
var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position
function g({name: x}) {
  console.log(x);
}
g({name: 5})
```

Още примери за destructing

```
// Fail-soft destructuring
var [a] = [];
a === undefined;

// Fail-soft destructuring with defaults
var [a = 1] = [];
a === 1;

// Destructuring + defaults arguments
function r({x, y, w = 10, h = 10}) {
  return x + y + w + h;
}
r({x:1, y:2}) === 23
```

Default + Rest + Spread

EcmaScript 2015 ни позволява да имаме параметри по подразбиране - да третираме последователни аргументи като масив и да
превръщаме масив в последователни аргументи при извикване на функция.

```
function f(x, y=12) {
  // y is 12 if not passed (or passed as undefined)
  return x + y;
}
f(3) == 15
```

```
function f(x, ...y) {
  // y is an Array
  return x * y.length;
}
f(3, "hello", true) == 6
```

```
function f(x, y, z) {
  return x + y + z;
}
// Pass each elem of array as argument
f(...[1,2,3]) == 6
```

Let + Const

Scope на ниво control flow structures. let е новата версия на var const е константа.

```
function foo(flag) {
    if (flag) {
        let a = 10;
    }
    return a;           // ReferenceError: a is not defined
}
console.log(foo(true));
```

Iterators + For..Of

Iterator обектите позволяват итерация с подобно for..in с custom for..of
Няма нужда от от масив, позволявайки ни lazy design.

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Generators

Декларация на function* (function ключовата дума, следвана от звездичка) дефинира функция генератор. Генератори са функции, които могат да бъдат прекъснати и след това
да бъдат стартирани отново. Техният контекст ще бъде запазен между извикванията.

Прост пример за функция генератор.

```
function* idMaker(){
  var index = 0;
  while(index < 3)
    yield index++;
}

var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // undefined
// ...
```

Модули

Поддръжка на модули за дефиниране на компоненти. Имплементира patterns от популярните JavaScript module loadres (AMD, CommonJS).
Безусловно асинхронен модел - кодът не се изпълнява, преди изиксваните модули да са достъпни и обработени.

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;

// app.js
import * as math from "lib/math";
console.log("2π = " + math.sum(math.pi, math.pi));

// otherApp.js
import {sum, pi} from "lib/math";
console.log("2π = " + sum(pi, pi));
```

Map + Set + WeakMap + WeakSet

По-добри структури от данни. WeakMaps осигурява leak-free обект към ключ таблица.

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;

// Weak Maps
var wm = new WeakMap();
wm.set(s, { extra: 42 });
wm.size === undefined

// Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });
// Because the added object has no other references, it will not be held in the
set
```

Как да инсталираме babel от npm?

```
npm install --save-dev babel-core
```

Използване

```
require("babel-core").transform("code", options);
```

За да използваме babel-preset-es2015 трябва да ги инсталираме.

```
npm install babel-preset-es2015 --save-dev
```

Доста по-удобно е обаче да използваме gulp-babel. Това е gulp плъгин за babel.
https://www.npmjs.com/package/gulp-babel (https://www.npmjs.com/package/gulp-babel)

.babelrc

.babelrc файла е конфигурационният файл на babel. babel го търси в root-а на проекта.

Чрез .babelrc файла се разрешават инсталираните плъгини и се сетват редица опции.

Например, ако имате инсталиран
ecma2015 preset трябва да го разрешите от конфигурационния файл така:

```
{
  "presets": ["es2015"]
}
```

gulp-babel

gulp-babel е gulp плъгин за по-лесна работа с babel.
Ето пример за неговата употреба:

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('default', () => {
    return gulp.src('app.js')
        .pipe(babel({
            presets: ['es2015']
        }))
        .pipe(gulp.dest('dist'));
});
```

Въпроси

Q & A