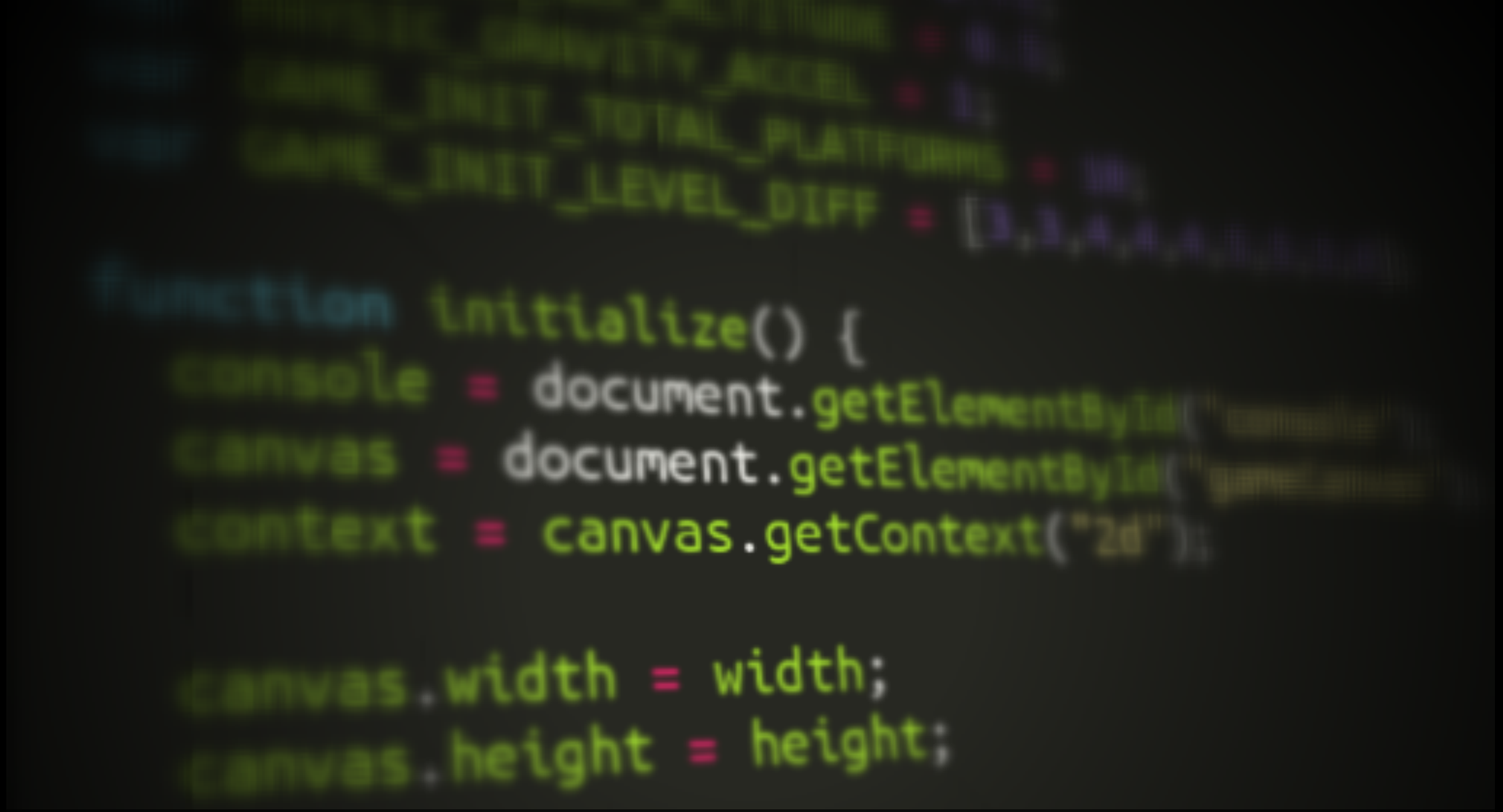


JavaScript Basics

JavaScript

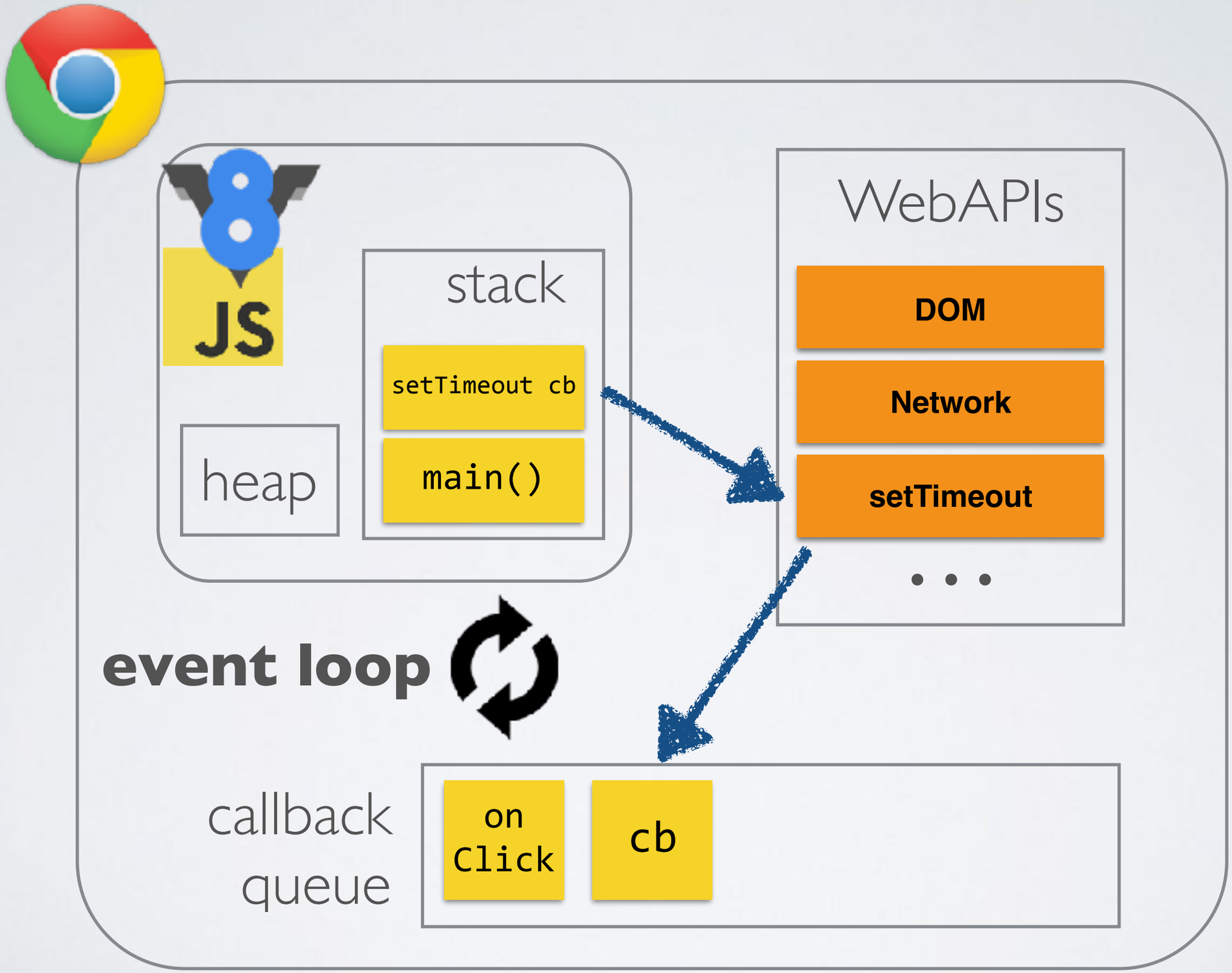
Language (Syntax)

Runtime

JavaScript Runtime Built-In Objects

Browser Built-In Objects & APIs (DOM, AJAX ...)

Event Loop



What the heck is the event loop anyway? https://www.youtube.com/watch?v=8aGhZQkoFbQ

The Variable `this`

- `this` is the context of a function. The value of this is determined by *how a function is called*:
    1. If the function is called as a constructor with the `new` keyword, `this` is a new object
    2. If the method is called via call() or apply(), `this` is explicitly passed
    3. If the function is called as a method of an object, `this` points to that object
    4. If none of the above are used, `this` is the global object (or `undefined` in strict mode).

- Especially for callbacks it might be tricky to find out the value of `this.`

Higher Order Functions

- Higher order functions operate on functions
  - functions as parameters
  - functions as return values
- Callback Functions
- Partial Application

```
$("#myBtn").on('click', function() {
  alert("Button Clicked!");
});
```

```
function arrayForEach(array, func) {
  for (var i = 0; i < array.length; i++) {
    func(array[i]);
  }
};
arrayForEach([1,2,3,4,5],
     function(msg){console.log(msg)});
```

```
var add = function(first, second) {
  return first + second;
};

var splitCall = function(first, func){
    return function(second){
        return func(first, second);
    }
}

var addOne = splitCall(1, add);
addOne(22); // -> 23
```

ES5: bind()

```
function print(){console.log(this.message);};
var controller = {message: 'Hello!', greet: print};
setTimeout(controller.greet); // Does not work!
```

Manual explicit binding:

```
function bind(fn, object){
  return function(){
    fn.apply(object)
  }
}
```

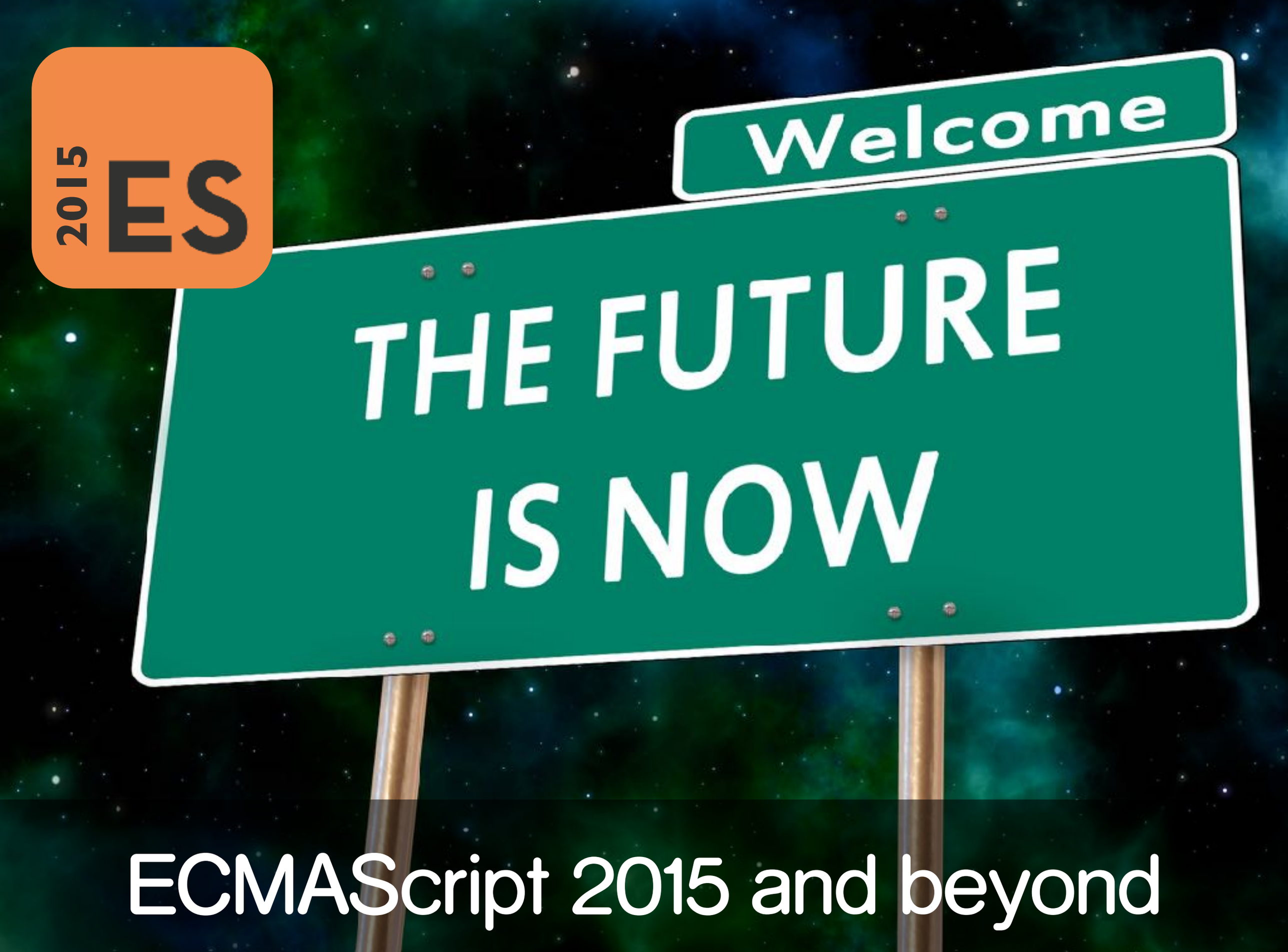
```
setTimeout(bind(print, controller));
```

ES5 provides the **`bind`** method:

```
setTimeout(print.bind(controller));
```

Demo: 11-This

2015 ES



ECMAScript 2015 and beyond

Versions

- JavaScript: Initial Release in Netscape 2, 1995
- ECMAScript 1: 1997
- ECMAScript 2: 1998
- ECMAScript 3: 1999
- ~~ECMAScript 4~~: abandoned
- ECMAScript 5: 2009
- ECMAScript 2015: June 2015
- ECMAScript 2016: June 2016
- ECMAScript 2017: June 2017

lead to ActionScript 3

also called ES6

also called ES7

also called ES8

http://www.ecma-international.org/ecma-262/8.0/

ES 2015 Features

`const & let`

for…of & iterators

Template Strings

Arrow Functions

Default Parameters

Rest Parameters

Enhanced Object Literals

Destructuring

Spread

Classes

Modules

Generators

Promises

Set, Map, WeakSet, WeakMap

Proxies

Symbols

Reflect API

Language Feature (syntax)

Built-In Objects (runtime)

(list is not complete …)

https://babeljs.io/docs/learn-es2015/

https://github.com/lukehoban/es6features

Variable Declarations with `let` and `const`

Variables declared with `var` have function scope and are hoisted.

ES5 JS

```
function assign(code){
  if (code === 1) var x = 42;
  else var x = 43;

  return x;
}
```

Variables declared with `let` or `const` have block scope. They are still hoisted but not initialized…

2015 ES

```
function assign(code){
  let x = 42;
  if (code === 1) let x = 43;

  return x;
}
```

https://rainsoft.io/variables-lifecycle-and-why-let-is-not-hoisted/

String Templates

Terse syntax for string creation

```
const person = {name: 'John Smith'};
const tpl1 = `My name is ${person.name}.`;
```

- `${}` can contain any JavaScript expression, which is evaluated against the current scope.
- `…` strings can contain multiple lines

Classes

```
class Counter {

  constructor(){
   this.count = 0;
  }

  increase(){
    this.count++;
  }
}

let counter = new Counter();
counter.increase()
console.log(counter.count)
```

Classes are syntactic sugar on top of constructor functions and prototypal inheritance.

ES2015 introduced classes.
Every modern browser supports this today.

Arrow Functions: ()=>{...}

ES5 JS

```
var square = function(x){
   return x * x;
}

var add = function(a, b){
   console.log('Adding ...');
   return a + b;
}

var pi = function(){
   return 3.1415;
}

var obj = function(){
   return {props: '!!'};
}
```

2015 ES

```
const square = x => x * x;
const add = (a, b) => a + b;
const pi = () => 3.1415;

const add2 = (a, b) => {
   console.log('Adding ...');
   return a + b;
}

const obj = () => ({prop: '!!'});
```

Arrow Functions

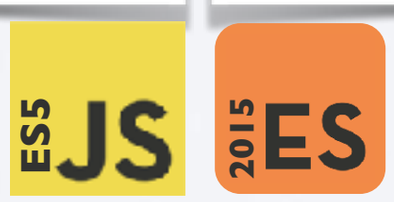
Arrow function do not explicit bind **`this`** (~lexical binding):

The value of **`this`** inside the function body is resolved along the scope chain.

This means that the **`this`** value of the enclosing execution context is used.

```
function Controller1(){
    this.count = 0;
    this.increment = function(){
        this.count++;
        console.log(this.count);
    };
}
var ctrl = new Controller1();
setTimeout(ctrl.increment, 0);
// -> NaN
```

```
function Controller2(){
    this.count = 0;
    this.increment = () => {
        this.count++;
        console.log(this.count);
    };
}
var ctrl2 = new Controller2();
setTimeout(ctrl2.increment, 0);
// -> 1
```



Other differences:

- calling with **`new`** not allowed
- no **`prototype`** property
- no binding of **`arguments`**
- no this binding with **`call`** or **`apply`**

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Rest and Spread

```
function multiGreet(message, ...names){
    names.forEach((name) =>
        console.log(message + ' ' + name));
}

multiGreet('Hi', 'John', 'Jane', 'Alice');
```

```
function addThreeThings(a, b, c) {
    return a + b + c;
}
console.log(addThreeThings(...[1,2,3]));
```

There is a proposal for object spread in ESnext:
https://github.com/sebmarkbage/ecmascript-rest-spread

Destructuring

Break-up an object into variables

```
const obj = {three: 3, four: 4};
const {three, four, other = 42} = obj;
const {three:five, four:six} = obj; // renaming
console.log(three);
console.log(six);
```

Break-up an array into variables

```
const array = [1,2];
const [one, two, three = '33'] = array;
console.log(one);
console.log(two);
```

Destructuring Examples

Selecting paramters from a parameter object:

```
function doSomething({param1, param3 = 42}){
     console.log('params: ', param1, param3);
}

const params = {param1: 1, param2: 2, param3: 3};
doSomething(params);
```

Selecting values from a return object:

```
function getValue(){
     return {val1: 1, val2: 2, val3: 3};
}
const {val1, val3 = 42} = getValue();
console.log('values: ', val1, val3);
```

Nested destructuring:

```
const {val2: {b = 42}} = {val1: 1, val2: {a:8, b:9}, val3: 3};
console.log(b);
```

Modules

In ES5 sharing constructs between files was only possible via the global namespace. Module- and Namespace Patterns helped to restrict what is exposed. Dependencies were managed implicitly.

`library.js`

```
window.doSomething
          = function(){…};
```

ES5 JS

`program.js`

```
window.doSomething();
```

ES6 introduces modules to manage dependences explicitly.

```
export function
          doSomething(){
  …
};
```

2015 ES

```
import {doSomething}
       from './library.js';

doSomething();
```

Modules: Characteristics

- Modules are executed when imported
- A modules is only executed once (even when imported several times)
- Modules are always executed in strict mode
- Modules have a top-level scope that is not the global scope
- Modules can `export` bindings to variables, functions or classes
- Modules can `import` bindings from other modules
- `import` bindings are readonly
- `import` statements are hoisted and can't be dynamic

Modules: Syntax

There are different ways to export & import

`library.js`

```
export function
       doSomething(){...};

export let dataObj = {...};

export default () => {...};

export {fun1 as fun2, obj};
```

`program.js`

```
import './library1.js';

import {doSomething, dataObj}
       from './library2.js';

import {doSomething as doIt}
       from './library3.js';

import * as lib3
       from './library4.js';

import bla
       from './library5.js';
```

Modules

```
import React from 'react';

class Greeter extends React.Component {
    render() {
        return <h1>Hello World!</h1>;
    }
}

export default Greeter;
```

Modules have their own scope. Sharing constructs is explicit with import & export.

The specification for modules is finished, but only most recent browsers support modules.
To use modules today a module bundler/loader is needed.

ES

Beyond ECMAScript 2015

ES2017: async functions

```
async function main(){
 var result = await doItAsnc();
 console.log('Finished waiting ...);
 ...
}
console.log('I am not blocking');
```

Already supported in the latest version of
all modern browsers (not IE).
Support via TypeScript and Babel.

https://github.com/lukehoban/ecmascript-asyncawait

ESnext: Object Spread

Object spread allows to copy & merge objects in an elegant way.

```
const sourceObject1 = {a: 1, b:2, c:3};
const sourceObject2 = {c: 5, d:6, e:7};
const mergedObject =
        {...sourceObject1, ...sourceObject2, f:9};
console.log(mergedObject);
```

object rest/spread is currently a stage-4 proposal and will be part of ES2018

object rest/spread is supported in TypeScript and Babel

https://github.com/tc39/proposal-object-rest-spread
https://babeljs.io/docs/plugins/transform-object-rest-spread/

ESnext: Class Fields

```
class Counter {

  count = 0;

  increase = () => {
    this.count++;
  }
}

let counter = new Counter();
counter.increase()
console.log(counter.count)
```

Declaring properties of a class increases readability.

Declaring methods as arrow functions avoids issues with `this` binding.

But the function does not exist on the prototype and can't be called via `super` in a derived class.

The specification class fields is not yet finished.
Browsers do not yet supports class fields natively.

class fields are supported in
TypeScript and Babel

https://github.com/tc39/proposal-unified-class-features

ESnext: Decorators

```
@Component({
   selector: 'app-counter',
   template: 'counter.component.html'
})
class CounterComponent {
  constructor(){
    this.count = 0;
  }
  increase(){
    this.count++;
  }
}
```

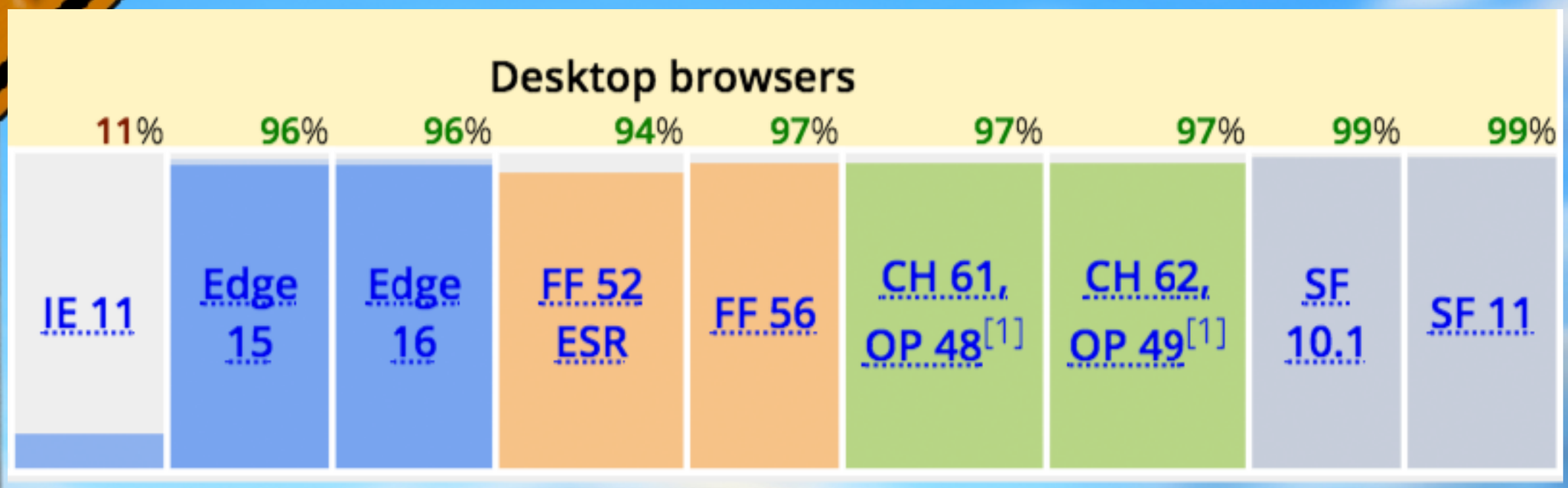
Decorators can be used to attach metadata to classes.
Frameworks can then use this metadata.

The specification for decorators is not yet finished.
Browsers do not yet supports decorators natively.
To use decorators today, they have to be compiled to JavaScript.

decorators are supported in TypeScript and Babel

https://github.com/tc39/proposal-unified-class-features

ES2015 browser support:



https://kangax.github.io/compat-table/es6/

ES2015 Modules

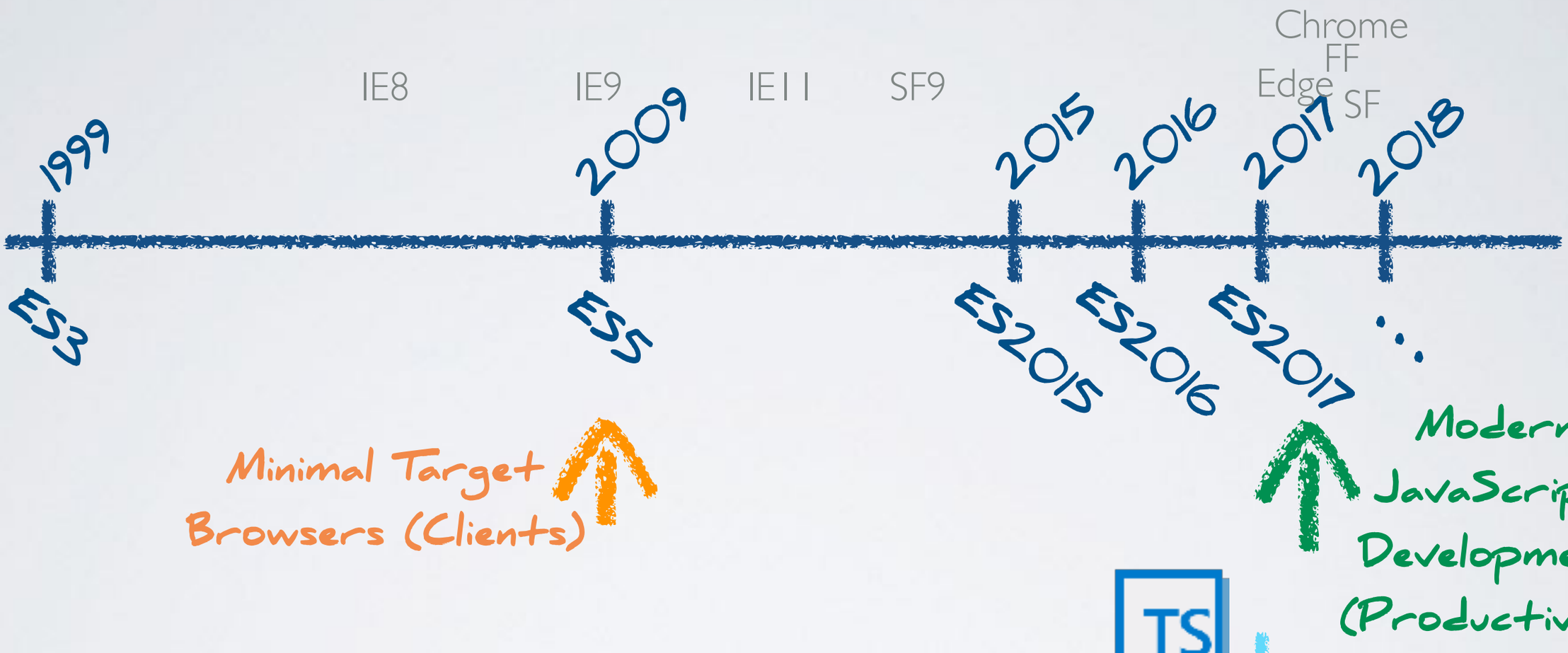
- New JavaScript syntax for `import` and `export` is part of the ES2015 spec.
- Module loading is not part of the ES2015 spec. There is a separate WhatWG loader spec.
- Native browser support for modules is still lacking
  - Official support in Safari 10.1, Chrome 61, Edge 16
  - Preview in Firefox 54+,

http://caniuse.com/#feat=es6-module

https://medium.com/dev-channel/es6-modules-in-chrome-canary-m60-ba588dfb8ab7

Addressing the Feature Gap

JavaScript has evolved rapidly in the past few years.



TypeScript is a language that compiles down to JavaScript.

Babel is a modular JavaScript compiler that can compile future features to a current ES version.

Transpiler

A transpiler is a scource-code to source-code compiler.

TypeScript language features and most ES2015+ features can be written in plain ES5.

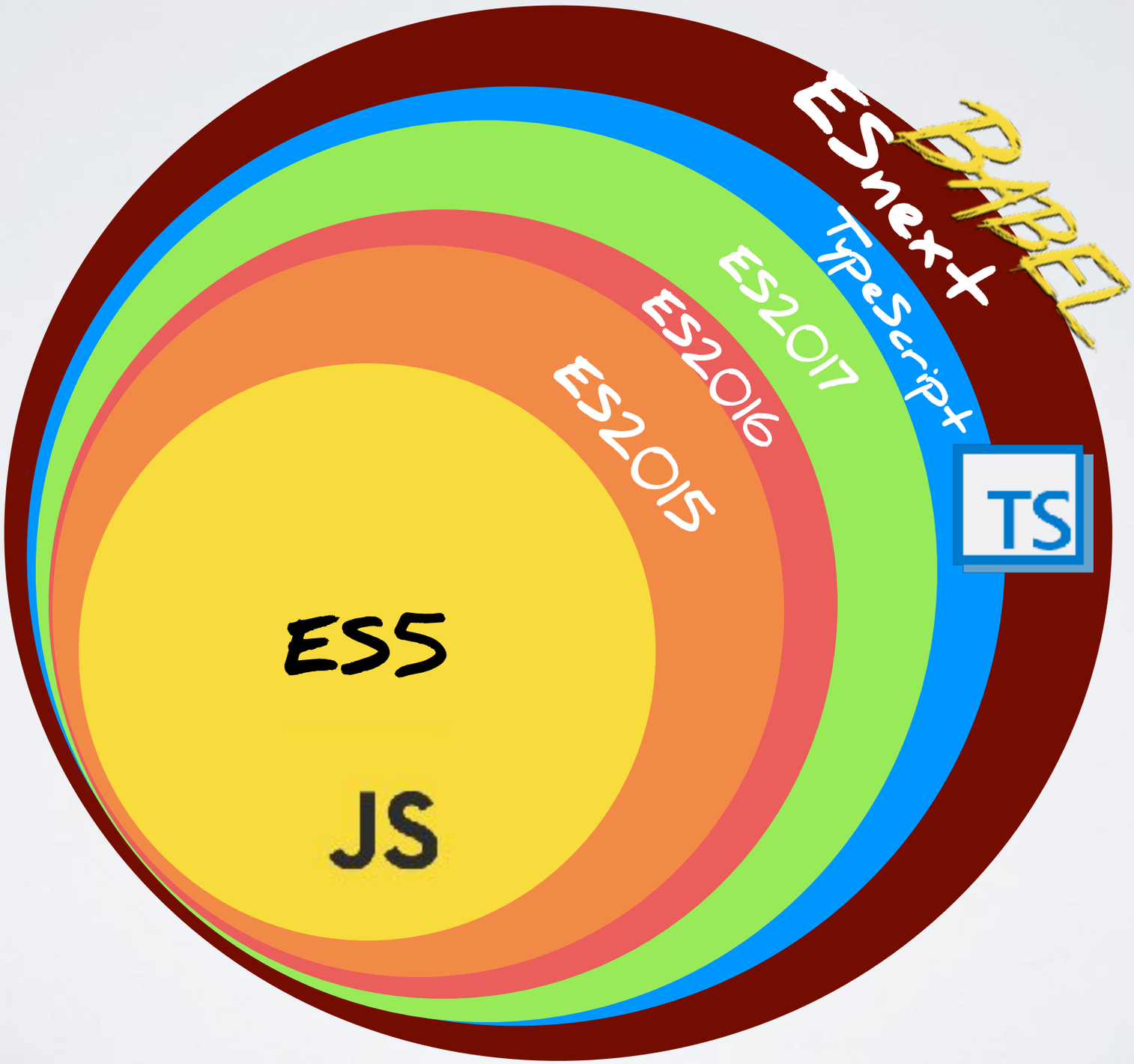
```
class Person {
  private name: string;
  constructor(name: string){
    this.name = name;
  }
}

const pers: Person = new
Person('John');
```

```
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    return Person;
}());
var pers = new Person();
```

https://babeljs.io/repl/
https://www.typescriptlang.org/play/index.html

Supporting JavaScript Features with a Transpiler

Each version is a superset of the previous versions of the language.



JavaScript

Language (Syntax)

Runtime

JavaScript Runtime Built-In Objects

Browser Built-In Objects & APIs (DOM, AJAX ...)

ES2015 Today: Transpiler & Polyfills

New Language Features (Syntax)

transpiling to ES5 for older browsers

TypeScript

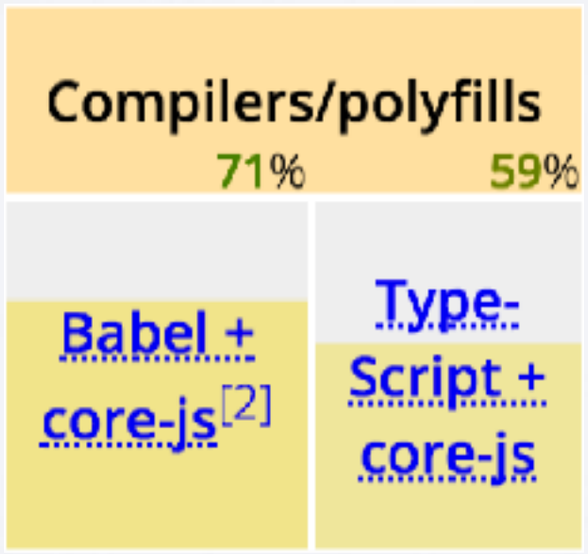
BABEL

New Built-In Functionality (Runtime Objects)

polyfills for older browsers

A polyfill is a a JavaScript library that implements a missing browser feature.

core-js
babel-polyfill
polyfill.io



https://kangax.github.io/compat-table/es6/

https://github.com/zloirock/core-js
https://polyfill.io/v2/docs/examples


JS
15
CHROME VANADIUM
17
CHROME VANADIUM
13
CHROME VANADIUM
CHROME VANADIUM
8

Toolset

Dependency Management

Declare & Resolve project dependencies.
Including transitive dependencies.

npm · yarn · Bower · jspm.io · ovo — A NEXT-GENERATION PACKAGE MANAGER FOR THE FRONT-END.

Build Automation

Infrastructure to implement and run build steps.
Orchestrate other tools.

npm · Gulp · GRUNT · Brunch · Broccoli

Module Bundling

Build one or several javascript files for deployment.
Resolve module dependencies.
Concatenate and minify JavaScript files.

webpack MODULE BUNDLER · browserify · jspm.io · PARCEL · REQUIRE.JS · rollup.js · splittable · FUSEBOX

Transpilation

Transform development sources (ES2015+ / TS / JSX) into ES5.

BABEL · TypeScript · tc

Linting / Static Type-System

Static code analysis.
Compile-time checked static type-system

ESLint · TSLint · JSHint · JSLint · TypeScript · flow

Libraries / Frameworks

Language

IDE

dependency management

Write Code

Toolset

compile

package

Build

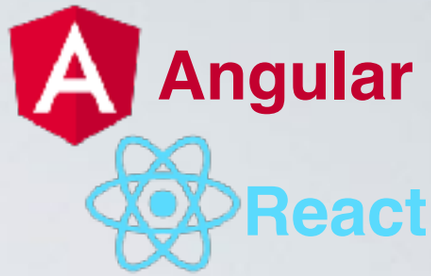
Test

Deploy

Test-Frameworks

IDE

Language



Libraries / Frameworks

dependency management

Write Code

Toolset

compile

package

Build

Test

Deploy

Test-Frameworks

Node Package Manager

npm is installed together with Node

npm offers:

- Dependency Management (global and local packages)
- Simple Build Tool

Node Package Manger

- Packages can be local (for the current project) or global
- `package.json` describes a package or project including it's dependencies
- packages are stored in `node_modules`
- hierarchical dependencies: dependencies can include their own dependencies
  (you can have several versions of a package in your project)
- Dependencies are versioned according to semantic versioning (https://semver.npmjs.com/)
- Starting from npm 5, exact versions are listed in `package-lock.json`
- Repository: npmjs.org (or private)
- Config: `.npmrc`

Tip: `npm config set save-exact true`

See: https://semver.npmjs.com/

Typical commands:

```
npm help
npm search
npm info
npm install
npm uninstall
npm list
npm update
npm init
npm root
npm config
```

Flags:

```
--global / -g
--help
--save / -S
--save-dev / -D
```

Enterprise Concerns

There are options for a private npm registry:

- Nexus
  http://books.sonatype.com/nexus-book/2.10/reference/npm.html
  (Nexus 2 does not support scoped packages, i.e. `@angular/core`)
- Artifactory
  http://www.jfrog.com/confluence/display/RTF/Npm+Repositories
- Visual Studio Team System:
  https://docs.microsoft.com/en-us/vsts/package/overview
- NPM Enterprise: https://www.npmjs.com/enterprise
- Gemfury: https://gemfury.com/
- verdaccio: http://www.verdaccio.org/

```
npm config list
npm config set registry <registry url>
```

Config files:
`<project>/.npmrc`
`$HOME/.npmrc`

yarn

…a better npm client.

npm@5 also introduced locking and speed similar to yarn

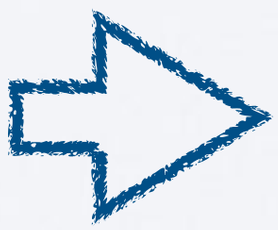
https://yarnpkg.com/

Yarn is faster and provides a reliable dependency management by pinning all dependencies (including transitive dependencies) with a lockfile (`yarn.lock`).

Recommended installation via installer.

deprecated: `npm install --global yarn`

```
npm install
npm install --save [package]
npm install --save-dev [package]
npm run [script]
```



```
yarn
yarn add [package]
yarn add [package] --dev
yarn run [task]
```

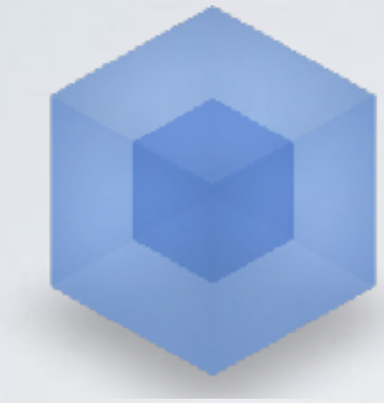
Using yarn on CI
https://yarnpkg.com/blog/2016/11/24/offline-mirror/

https://yarnpkg.com/en/docs/migrating-from-npm
https://shift.infinite.red/npm-vs-yarn-cheat-sheet-8755b092e5cc

BUNDLING

(Development Time Build Toolchain)

- Resources are optimized
  - Code is minimized
  - Bundles are coarse grained, network overhead is minimzed
- Cache-Busting mitigates caching problems
- ES2015 modules prevent polluting the global namespace
- Bonus: Modern language constructs (ES2015+) can be used



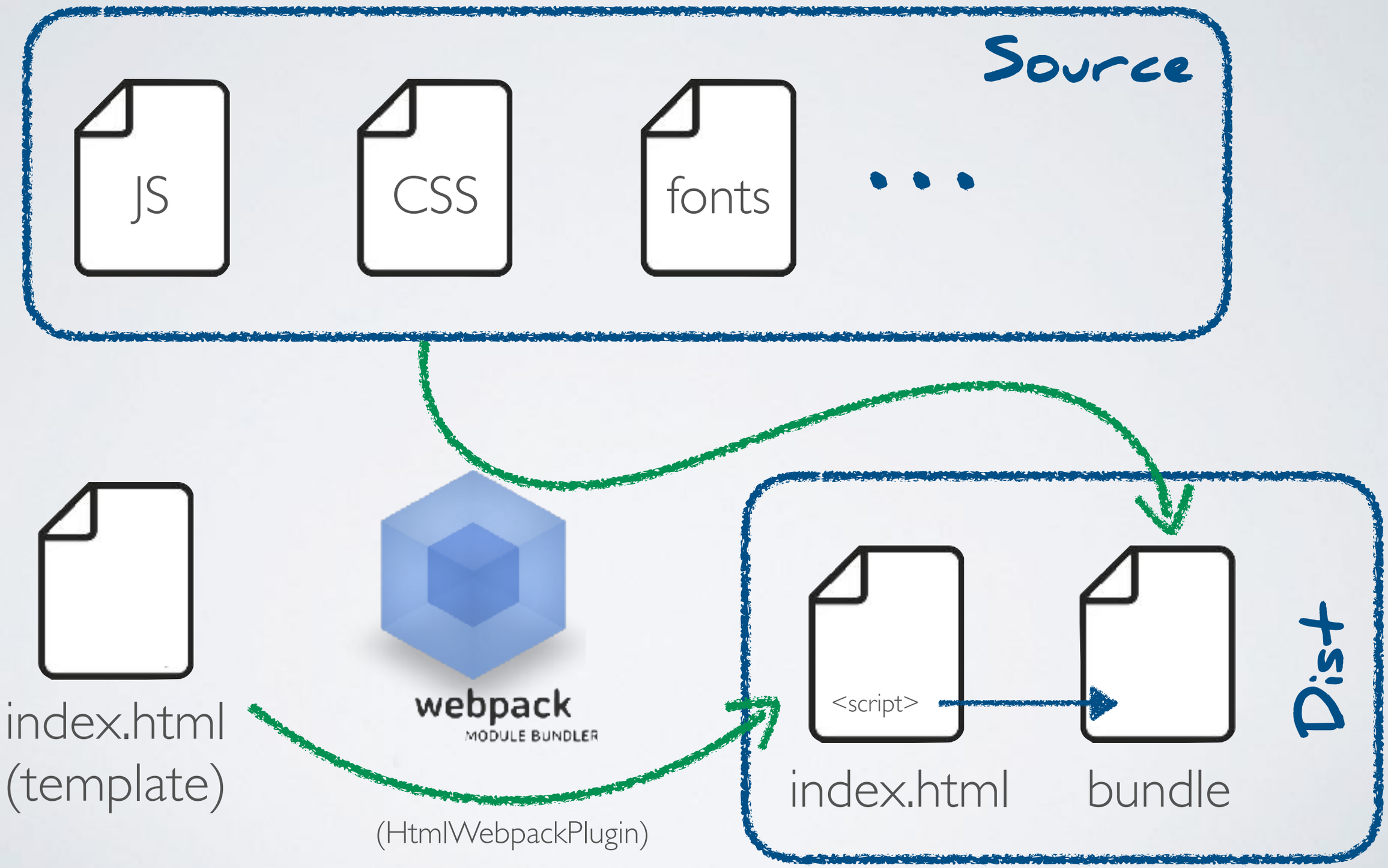
WebPack

Webpack is a bundler that bundles the fine grained assets of the application (primarily JavaScript and CSS) into one or several corse grained bundles at build time.
The contents of a bundle are defined by building a dependency graph (following imports).

Bundles are JavaScript files that are loaded by the browser at runtime.

https://webpack.js.org/

WebPack Build



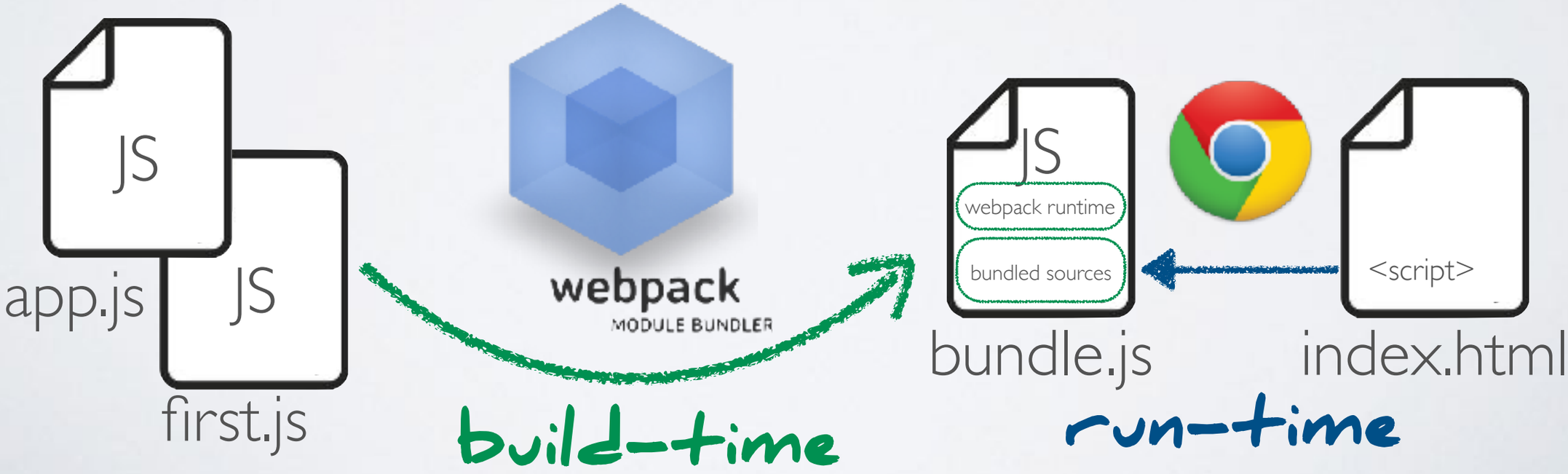
ES2015 Modules with WebPack

index.html

```
<script src="bundle.js"></script>
```

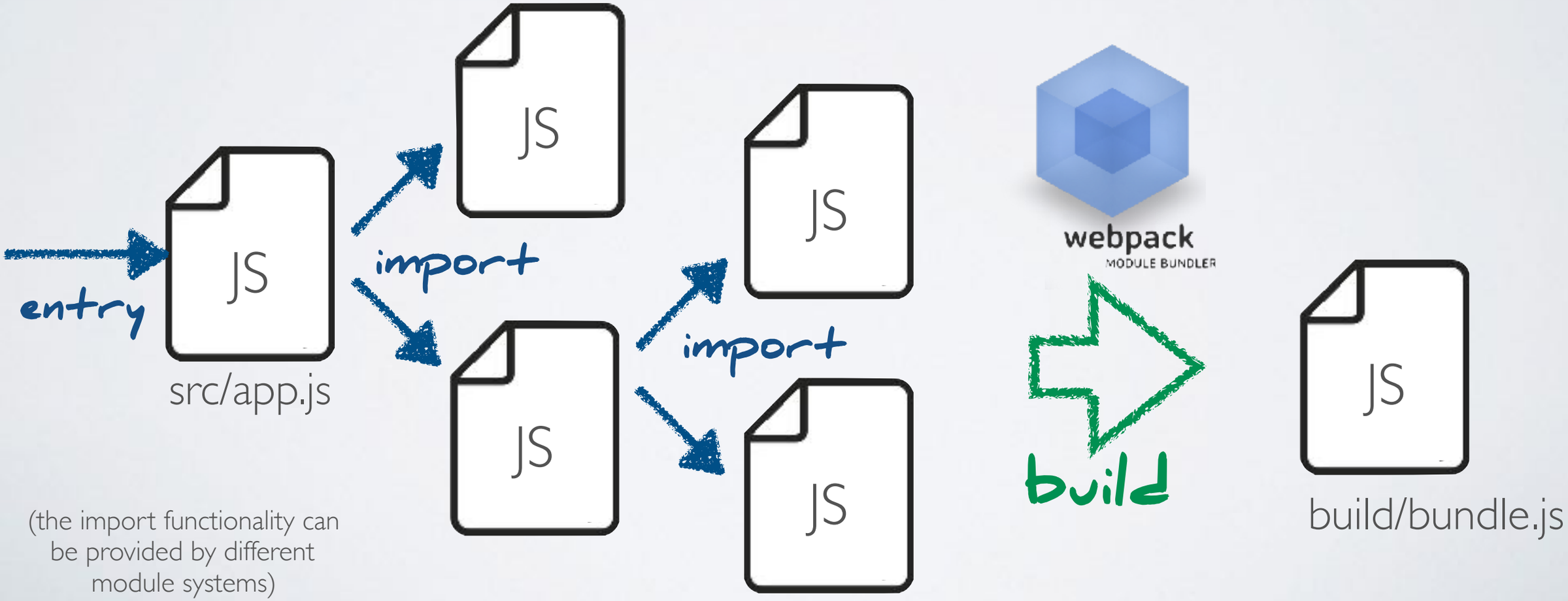
app.js

```
import './modules/first';
console.log("Hello from ES6 modules");
```



A minimal WebPack config

```
module.exports = {
    entry: './src/app/app.js',
    output: {
        filename: 'build/bundle.js'
    },
    devtool: 'source-map'
};
```



(the import functionality can be provided by different module systems)

TypeScript

JavaScript that scales!

TypeScript is a language for application-scale JavaScript development.

"Any browser, any host, any OS"

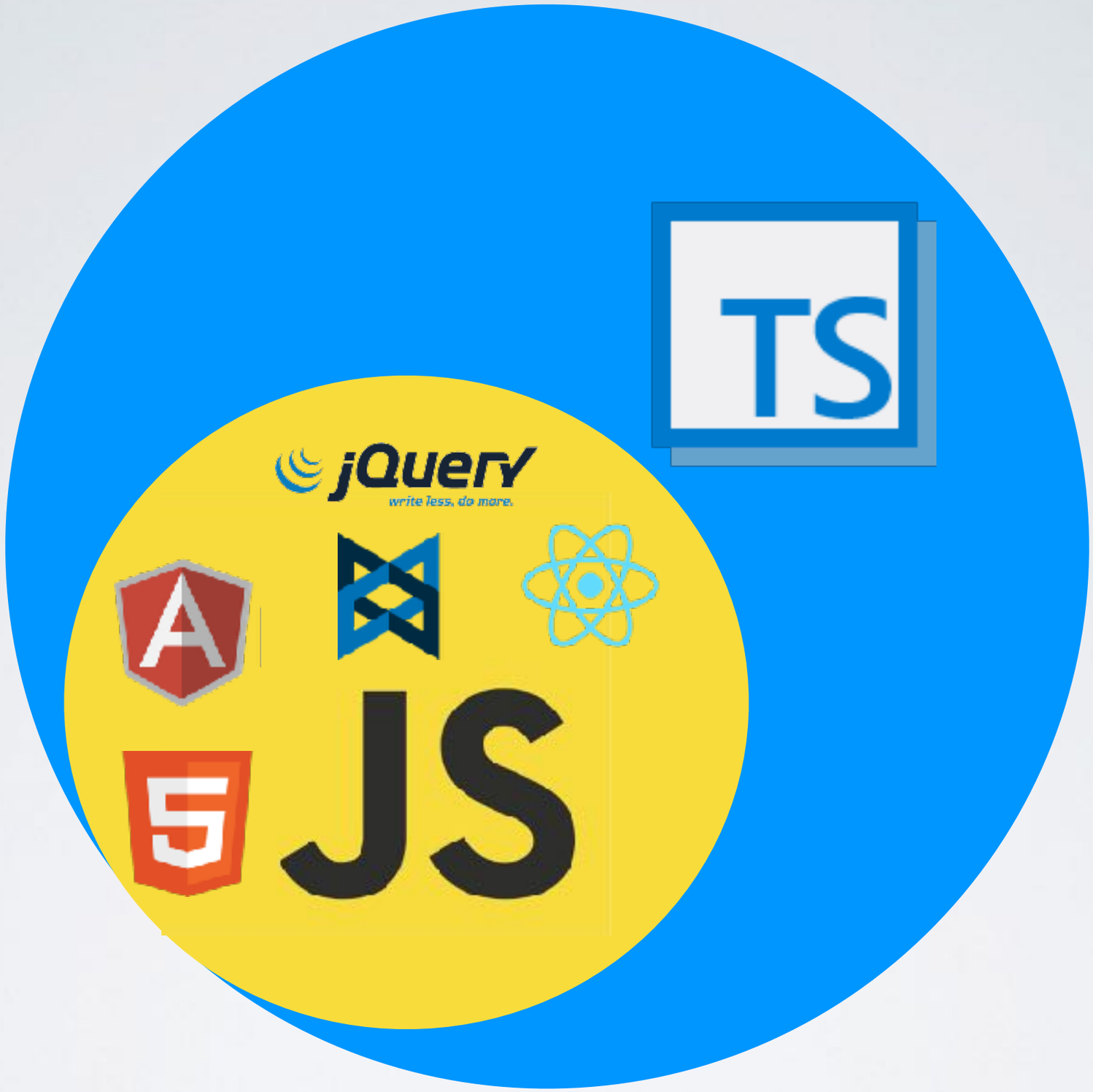
http://www.typescriptlang.org/

TS is a superset of JS. Start with your existing JS and "enhance" it ...

It starts with JavaScript.

It ends with JavaScript.

After transpilation it's pure JS again. At runtime there is nothing left of TS!



TypeScript is a superset of JavaScript

Angular 2: Why TypeScript?: https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8#.qf7up1nj7

The original goal of TypeScript

on top of ES5

new (future)

language features

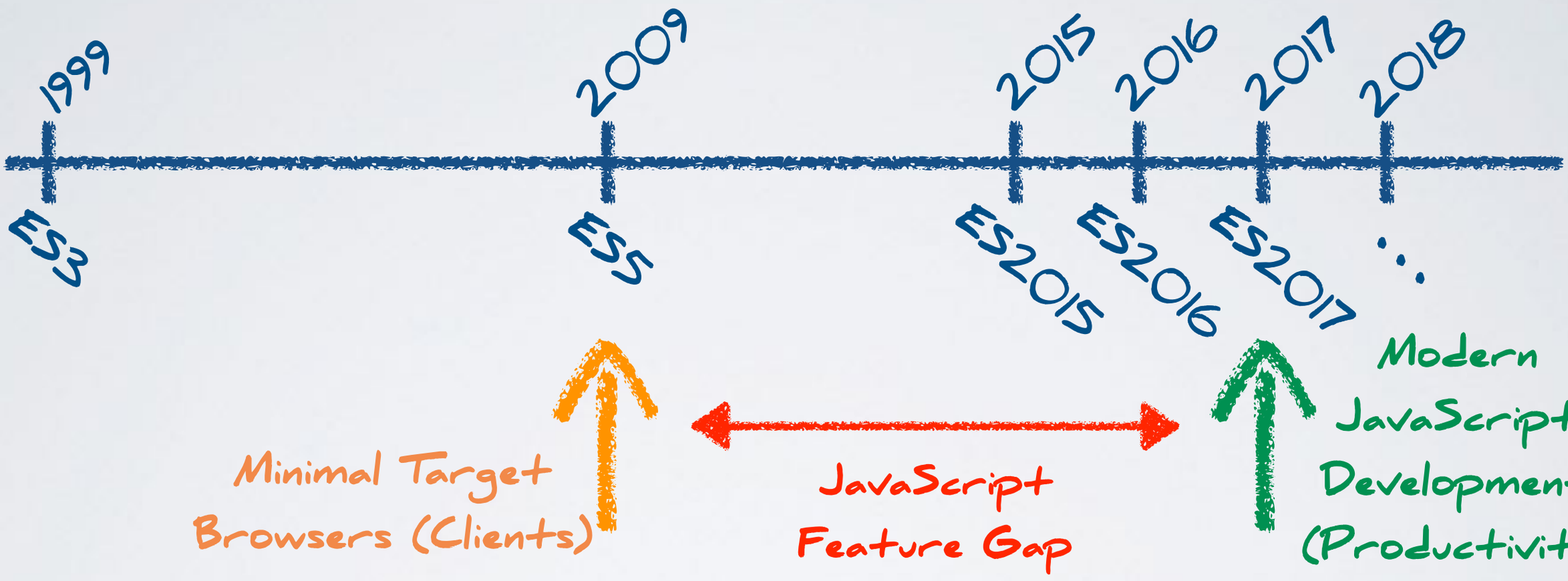
modules, classes, ...

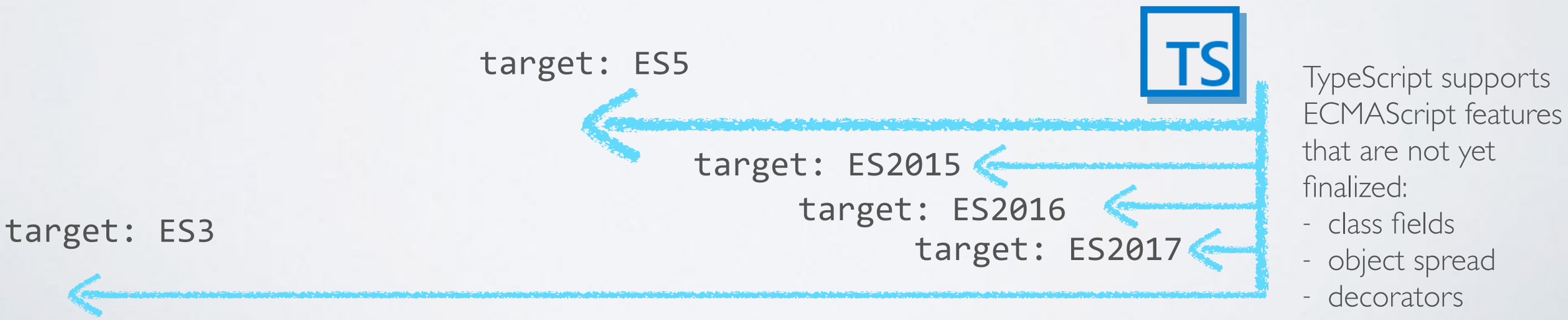
static types

enables tools

Transpilation

TypeScript is a language that compiles down to JavaScript.





TypeScript supports ECMAScript features that are not yet finalized:

- class fields
- object spread
- decorators

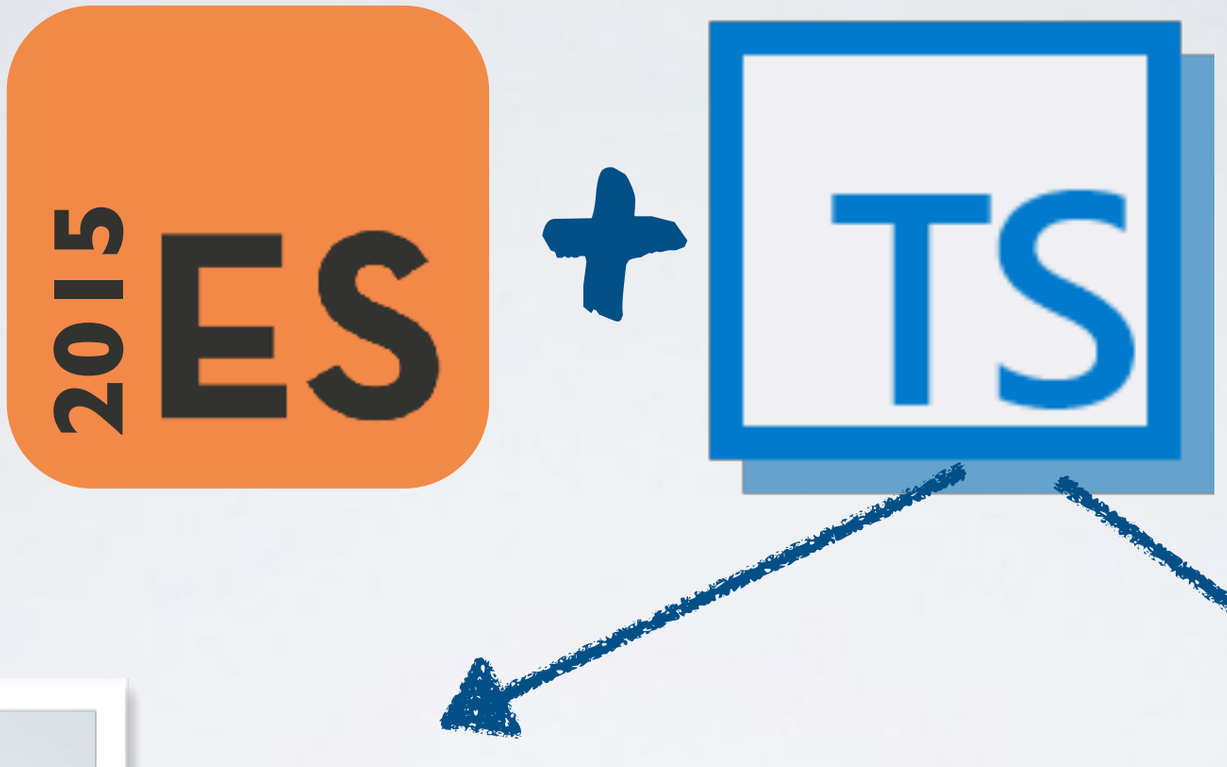
https://www.typescriptlang.org/docs/handbook/compiler-options.html

Static Typing & Access Modifiers

```
class Counter {

  private count = 0;

  increase(amount: number){
    this.count += amount;
  }
  getValue(): number {
    return this.count;
  }
}

let counter = new Counter();
counter.increase(2)
console.log(counter.getValue())
```

Static typing and access modifiers are not part of JavaScript. The compiler removes this information at build time.

TypeScript Today



new (future) language features

modules, classes, ...

static types

enables tools

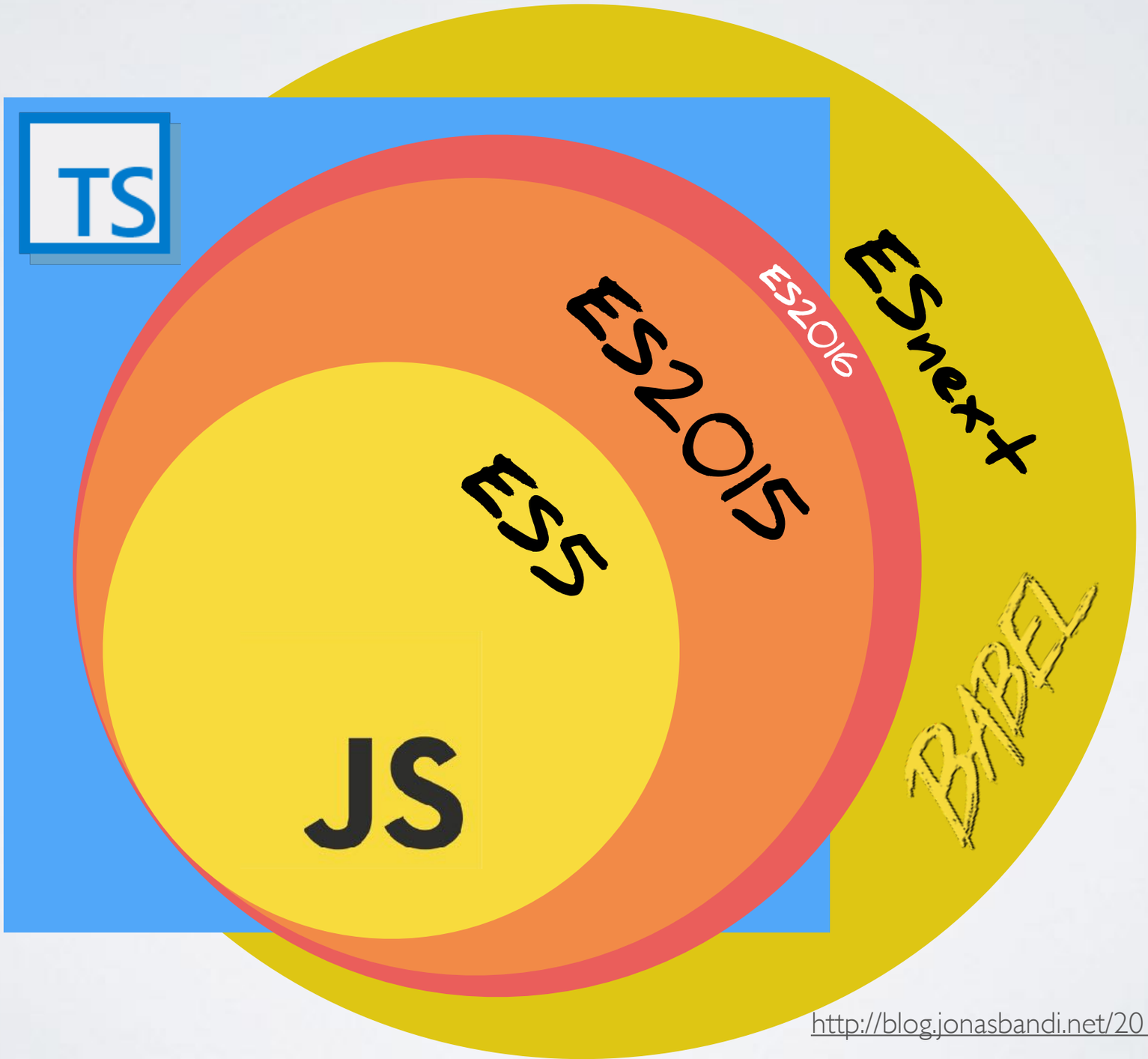
TypeScript still provides some future language features on top of ES2015...

... but it provides less future language features compared with Babel.

Superset ?

TypeScript has compile-time language features that are not valid ECMAScript (enums, generics ...)

TypeScript has an optional compile-time static type system on top of ECMAScript and the browser APIs.



http://blog.jonasbandi.net/2016/10/myth-of-superset.html

Installing 3rd-Party Type Definitions

In TypeScript 2.0 type definitions are distributed via NPM:

```
npm install @types/jquery
```

(`@types` is called a "scope" in npm)

Before TypeScript 2.0 other package managers were used: initially `tsd` (typescript definition manager) and later `typings`:

```
typings init
typings install dt~jquery --global --save
```

obsolete!

Why not to use TypeScript?

In many smaller-scale use cases, introducing a type system may result in more overhead than productivity gain.

- Vue.js Documentation

I worry about building up a large codebase using TypeScript, only to have the ECMAScript spec introduce conflicting keywords and type features.

– Eric Elliot, The Shocking Secret About Static Types

https://vuejs.org/v2/guide/comparison.html#TypeScript

https://medium.com/javascript-scene/the-shocking-secret-about-static-types-514d39bf30a3#.i1xembose

Flow is a static type checker for JavaScript.

Flow was created by Facebook and has a strong user base in the React ecosystem.

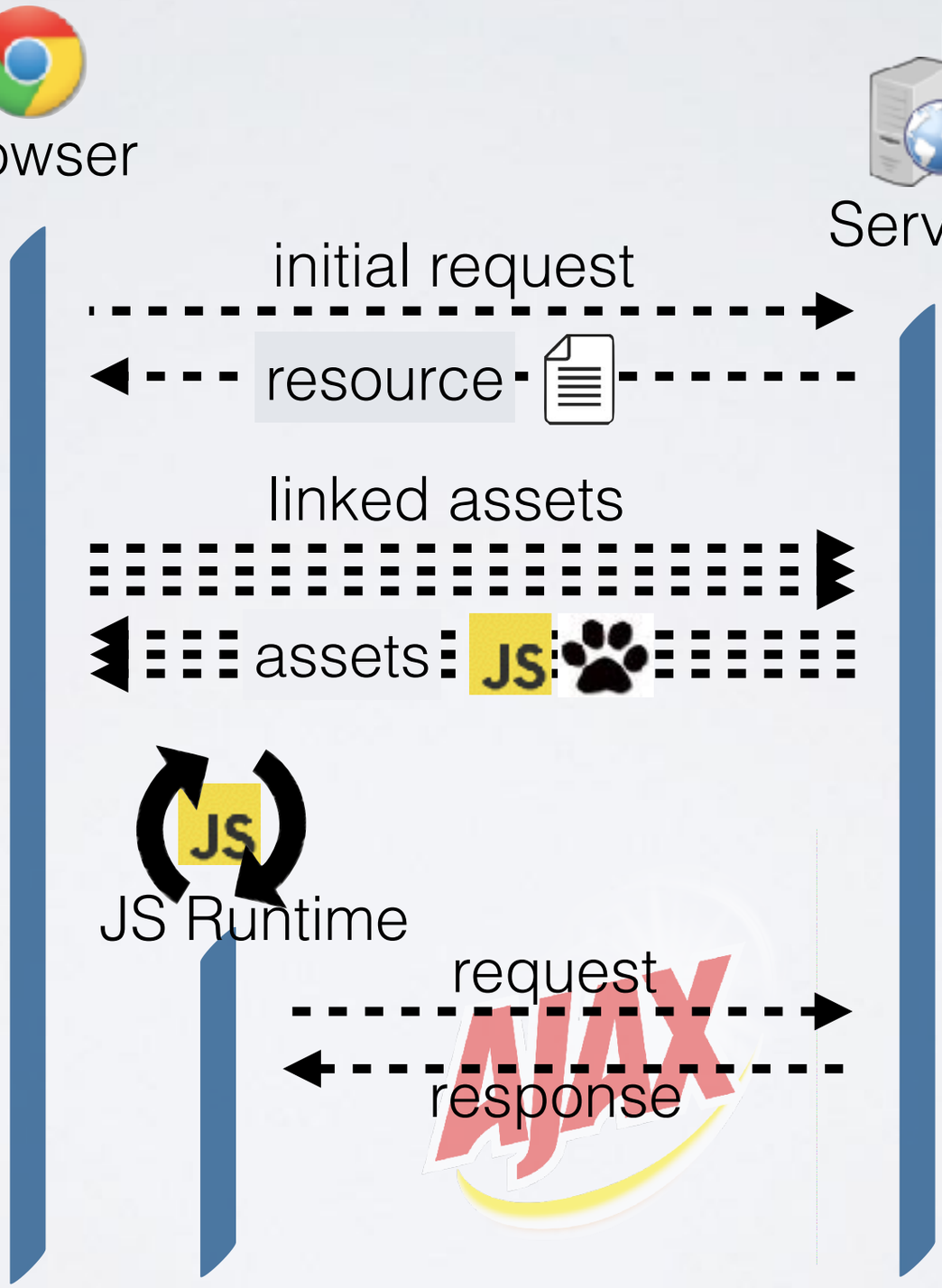
Flow integrates well with the Babel ecosystem.

https://flow.org/

AJAX & Async JavaScript

Browsers & Network Access

Browsers can perform network requests when loading resources or when submitting forms.



The JavaScript runtime can perform network requests triggered by scripts.
These are called AJAX requests.

XMLHttpRequest

The underlying technology for AJAX.

```
var req = new XMLHttpRequest();
req.addEventListener("load", done);
req.addEventListener("error", failed); // only fired on network-level events
req.addEventListener("readystatechange", stateChanged);
req.open("GET", "http://localhost:3001/comments");
req.send();

function stateChanged(){
    if (this.readyState === 4) {
        if (this.status === 200) {
            console.log('success state', this.responseText)
        } else {
            console.log('error state', this.statusText);
        }
    }
}

function done() {
    console.log('DONE', this.responseText);
}

function failed() {
    console.log('ERROR', this.statusText);
}
```

The API is low-level, complicated and verbose.

https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest

AJAX with Callbacks

Using jQuery

```
$.get('http://localhost:3001/comments', function (data) {
    console.log(data);
});
```

```
$.post('http://localhost:3001/comments',
       {text: 'test - ' + new Date()},
       function () {
          console.log('POST!');
});
```

fetch: a modern AJAX alternative

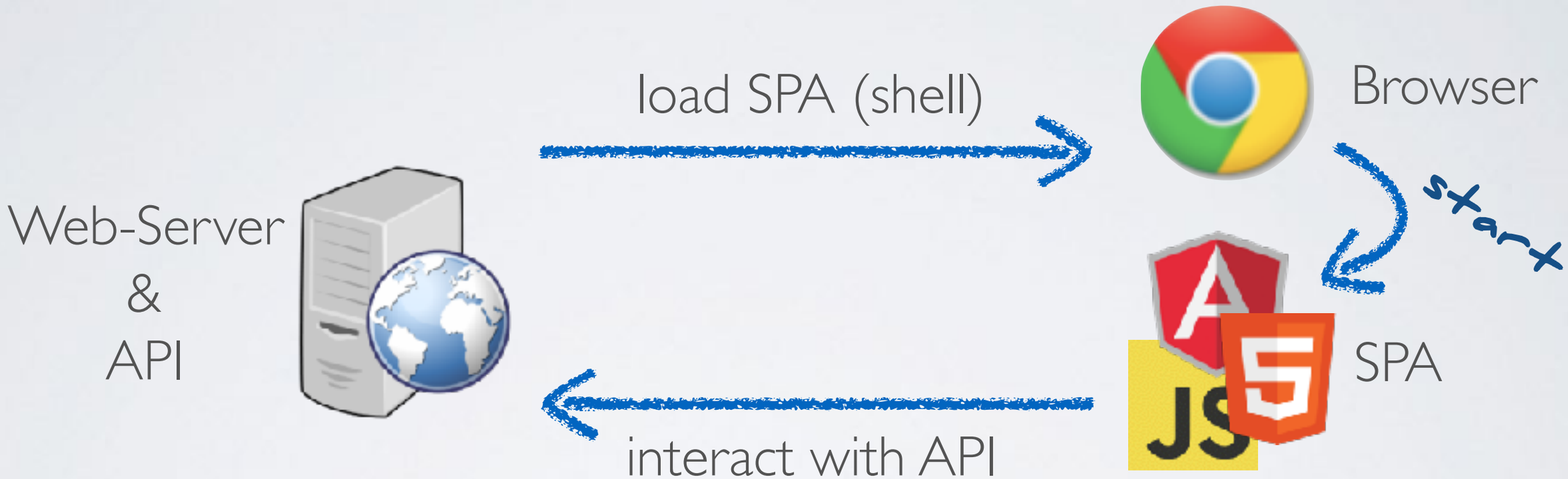
`fetch` is a new API available in modern browsers that is more elegant than **`XMLHttpRequest`**.

```
fetch('http://localhost:3001/comments2')
    .then(response => response.json())
    .then(data => console.log('SUCCESS', data))
    .catch(error => console.log('ERROR:', error));
```

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
https://fetch.spec.whatwg.org/#fetch-api
https://jakearchibald.com/2015/thats-so-fetch/
Polyfill: https://github.com/github/fetch

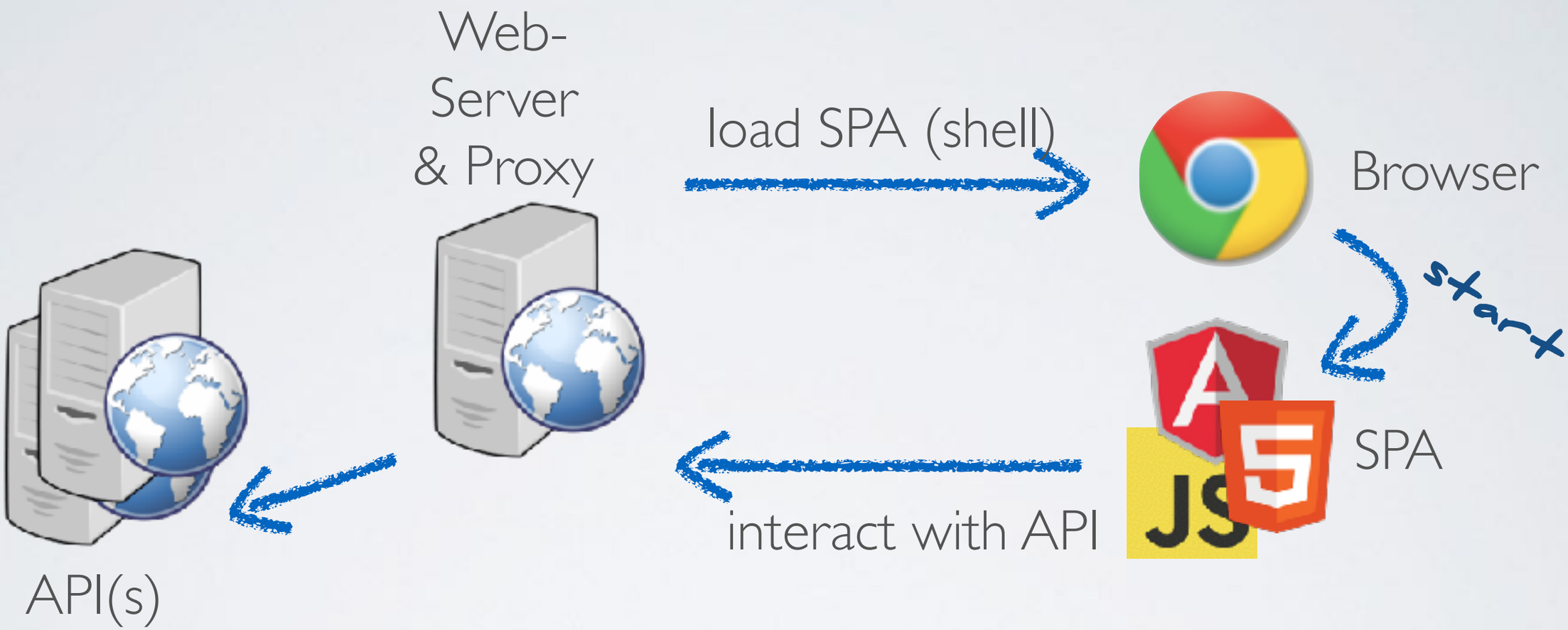
Browser support:
http://caniuse.com/#search=fetch

"Traditional Web-App"



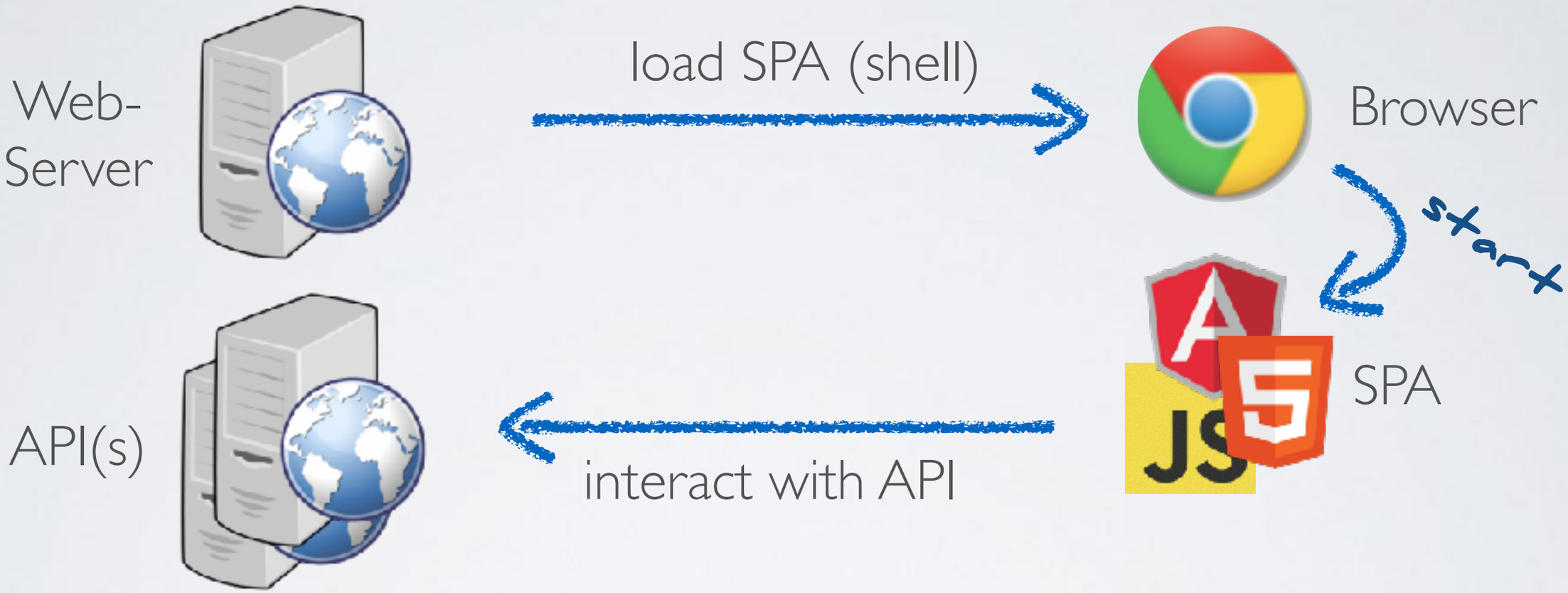
- Shell can be static or dynamic (rendered on the server)
- Same origin for all resources (shell & API)
- Often combined technologies on the Server (MVC & WebAPI, JSF & JAX-RS)
- "Monolithic", Advantages: Authentication/Authorization, Deployment ...

Web + Proxy + API-Backend



- One central entry point for the application
- Same origin for all resources (shell & API)
- Distributed / independent services ("SOA", "Microservices" ...)
- Advantages: logging, transactions, authentication

Web + API-Backend



- Shell is typically static (simple web-server or CDN)
- Different origin for shell & API resources
- Distributed / independent services ("SOA", "Microservices" ...)

Same Origin Policy (SOP)

The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin.

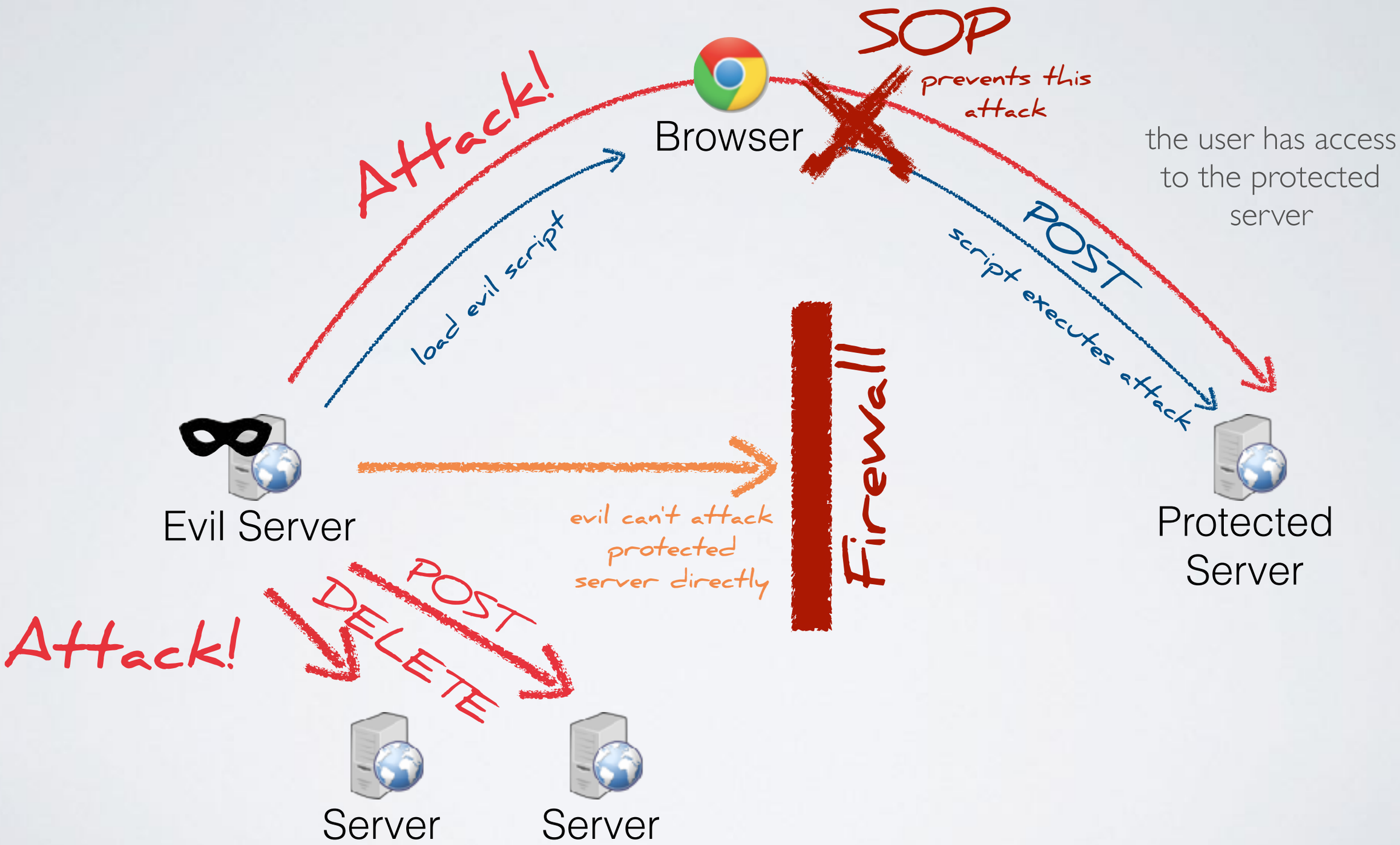
- A page can't access data on another page
- A script can only make AJAX calls to resources from the origin it was loaded from
- Example: A script from "example.com" can't read or write your emails from "gmail.com"

The SOP protects the client not the server!

The SOP prevents "distributed APIs"!

https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

SOP



https://annevankesteren.nl/2015/02/same-origin-policy

Cross Origin Resource Sharing (CORS)

CORS allows to circumvent the SOP to implement "distributed APIs".

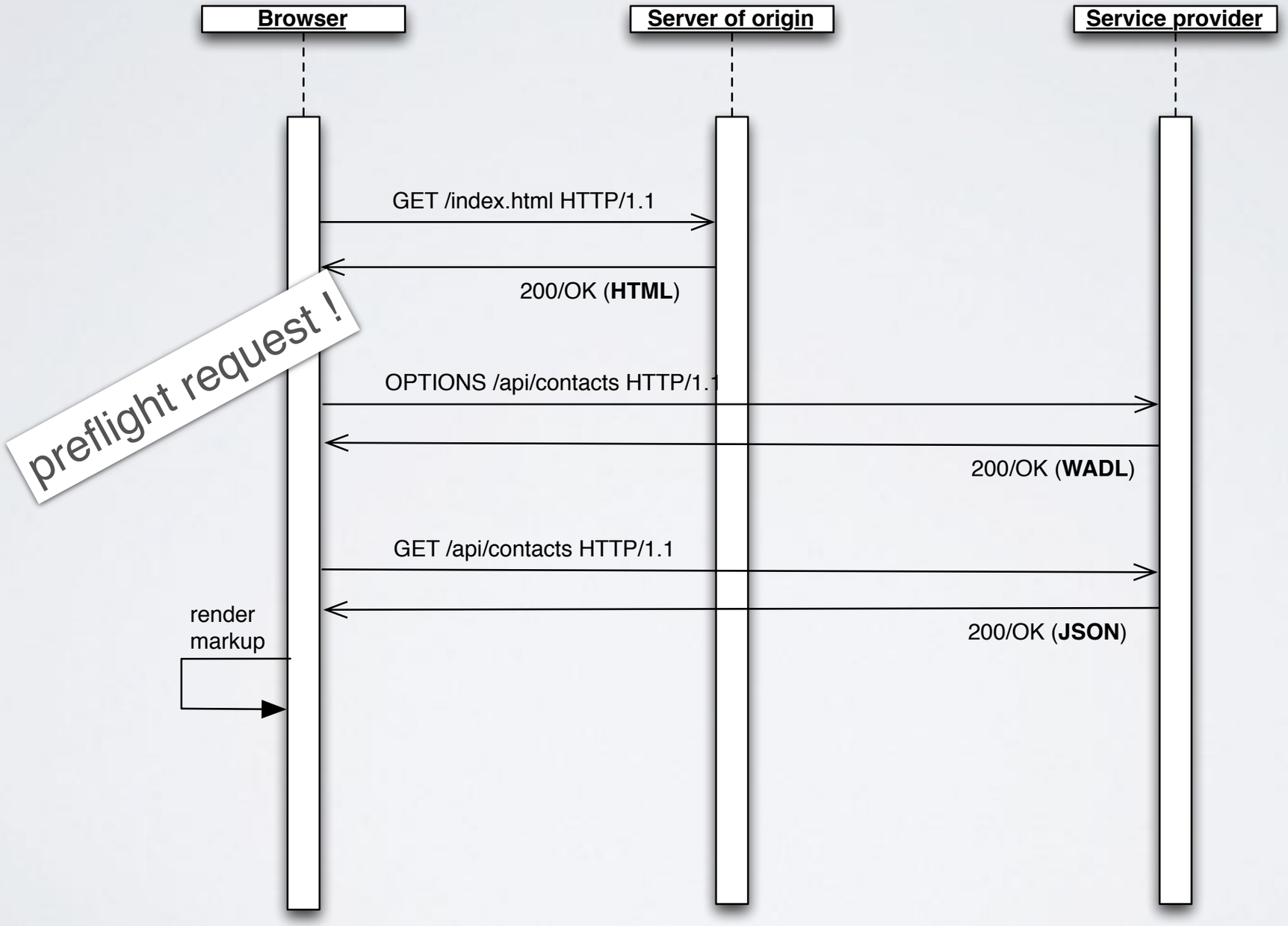
- A server API can define that its resources can be accessed from other origins.
- Browsers then allow scripts to make AJAX calls to that API.

CORS is implemented on the server but affects the behaviour of the browsers.

CORS only affects browser clients. Other clients (i.e. mobile) are not affected.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

CORS: Preflight Request



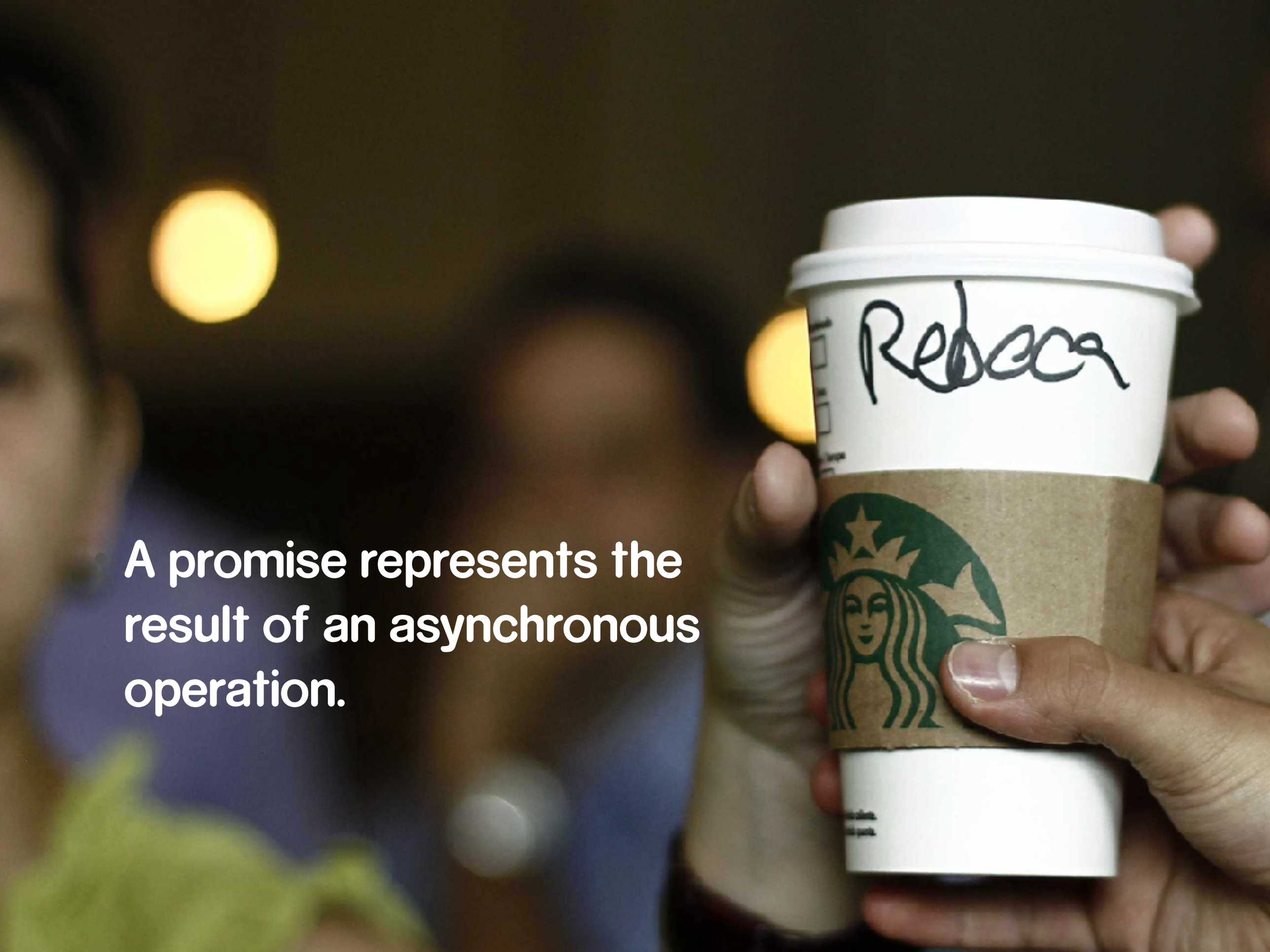
https://www.w3.org/TR/cors/#cross-origin-request-with-preflight-0

Setting CORS Headers

JEE WebFilter

```
@WebFilter(filterName = "CorsFilter", urlPatterns = {"/rest/ratings-cors/*"})
public class CorsFilter implements Filter{

@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                     FilterChain filterChain) throws IOException, ServletException {
    final HttpServletRequest httpRequest = (HttpServletRequest)servletRequest;
    final HttpServletResponse httpResponse = (HttpServletResponse)servletResponse;

    httpResponse.addHeader("Access-Control-Allow-Origin", "*");
    httpResponse.addHeader("Access-Control-Allow-Methods",
                                  "GET, POST, PUT, DELETE, OPTIONS");
    httpResponse.addHeader("Access-Control-Allow-Headers",
       "content-type, x-requested-with, accept, origin, authorization, X-PINGOTHER");
...
}
```

Promises



A promise represents the result of an asynchronous operation.

Promises

- A promise represents the result of an asynchronous operation.
  - Counterparts: Future<> in Java or Task<> in .NET (TPL)
- Promises enable another programming model for asynchronous functions than the callback programming model.
  - Asynchronous functions can return a value without blocking (so they look much more like synchronous functions)
  - The return value is a promise that represents the final value that is possibly not yet known

Using Promises

Consume a promise:

```
var promise = startAsyncOperation();
promise.then(function (value) {
        // success handler
    })
    .catch(function (error) {
      // error handler
    });
```

Provide a promise (ECMAScript 2015):

```
function startAsyncOperation(){
  var promise = new Promise(function (resolve, reject) {
     // resolve the promise later
      setTimeout(function () { resolve("Success!");}, 1000);
  });
  return promise;
}
```

Advantages of Promises: Chaining Async Operations

Callback-Hell:
(aka Pyramid of Doom)

```
api(function(result){
    api2(function(result2){
        api3(function(result3){
            // do work
        });
    });
});
```

Promise Chaining:

If a "then-callback" returns another promise, the next "then" waits until it is settled.

```
api().then(function(result){
    return api2();
}).then(function(result2){
    return api3();
}).then(function(result3){
     // do work
}).catch(function(error) {
     //handle any error
});
```

```
api().then(api2)
      .then(api3)
      .then(/* do work */);
```

Advantages of Promises: Combining Multiple Actions

```
var firstData = null, secondData = null;

var responseCallback = function() {
 if (!firstData || !secondData) return;
 // do something
}

getData("http://url/first", function(data) {
  firstData = data;
  responseCallback();
});

getData("http://url/second", function(data) {
  secondData = data;
  responseCallback();
});
```

```
var promise1 = getData("http://url/first");
var promise2 = getData("http://url/second");

Promise.all([promise1, promise2])
    .then(function(arrayOfResults) {…})
    .catch(function(errOfPromise) {…});

Promise.race([promise1, promise2])
    .then(function(valOfPromise) {…})
    .catch(function(errOfPromise) {…});
```

Promise Chaining

```
doSomething().then(function () {
  return doSomethingElse();
}).then(finalHandler);
```

```
doSomething().then(function () {
  doSomethingElse();
}).then(finalHandler);
```

```
doSomething()
   .then(doSomethingElse())
   .then(finalHandler);
```

```
doSomething()
   .then(doSomethingElse)
   .then(finalHandler);
```

https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html

EXERCISES

Exercise 5 - Promises

async / await

Async / Await

```
function sleep(millis){
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), millis);
  });
}

async function start(){
  await sleep(1000);
  console.log('finished!');
}

start();
```

An **`async`** function (implicitly) returns a promise, which is resolved with the return value.

An **`async`** function can contain an **`await`** expression, that pauses the execution of the **`async`** function and waits for the passed **`Promise`**'s resolution, and then resumes the **`async`** function's execution and returns the resolved value.

async / await is a language feature of ES2017. It is already natively supported in all modern browsers.

https://caniuse.com/#feat=async-functions

async / await is still promises, but with a really nice syntax!

```
function sleep(millis){
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), millis);
  });
}

async function start(){
  await sleep(1000).then(() => console.log('waking up!'));
  console.log('finished!');
}

start();
```

Demo:

Async exercise with ES2017 async functions

Alternative: Generators & Promises

```
function sleep(millis){
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(), millis);
  });
}

function *start(){
  yield sleep(1000);
  console.log('finished sleeping');
  yield;
}

let iterator = start();
iterator.next();
iterator.next();
console.log('finished!');
```

Demo:

Async exercise with ES2015 generators.