Frontend-Development with Angular

Mail: jonas.bandi@ivorycode.com

Twitter: @jbandi

ABOUT ME

Jonas Bandi
jonas.bandi@ivorycode.com
Twitter: @jbandi

- Freelancer, in den letzten 5 Jahren vor allem in Projekten im Spannungsfeld zwischen modernen Webentwicklung und traditionellen Geschäftsanwendungen.
- Dozent an der Berner Fachhochschule seit 2007
- In-House Kurse: Web-Technologien im Enterprise
  UBS, Postfinance, Mobiliar, BIT, SBB ...

JavaScript / Angular / React
Schulungen & Coachings,
Project-Setup & Proof-of-Concept:

http://ivorycode.com/#schulung

AGENDA

DAY 1

Intro

Angular CLI

Modern JavaScript Development with ES2015+ and TypeScript

Components

Angular Routing

DAY 2

Template Basics

Services & DI

Component Architecture

More "UI Constructs"

Angular Forms

Async JavaScript

RxJS

DAY 3

Backend Access

Modularization & Routing Part 2

Advanced Topics:

- Project Structure
- State Management
- Dependency injection
- Modularization
- Testing
- ...

Material

Git Repository:
`https://bitbucket.org/jonasbandi/ng-ige-2019`

Initial Checkout:
`git clone https://bitbucket.org/jonasbandi/ng-ige-2019.git`

Update:

```
git pull
```

```
git reset --hard
git clean -dfx
git pull
```

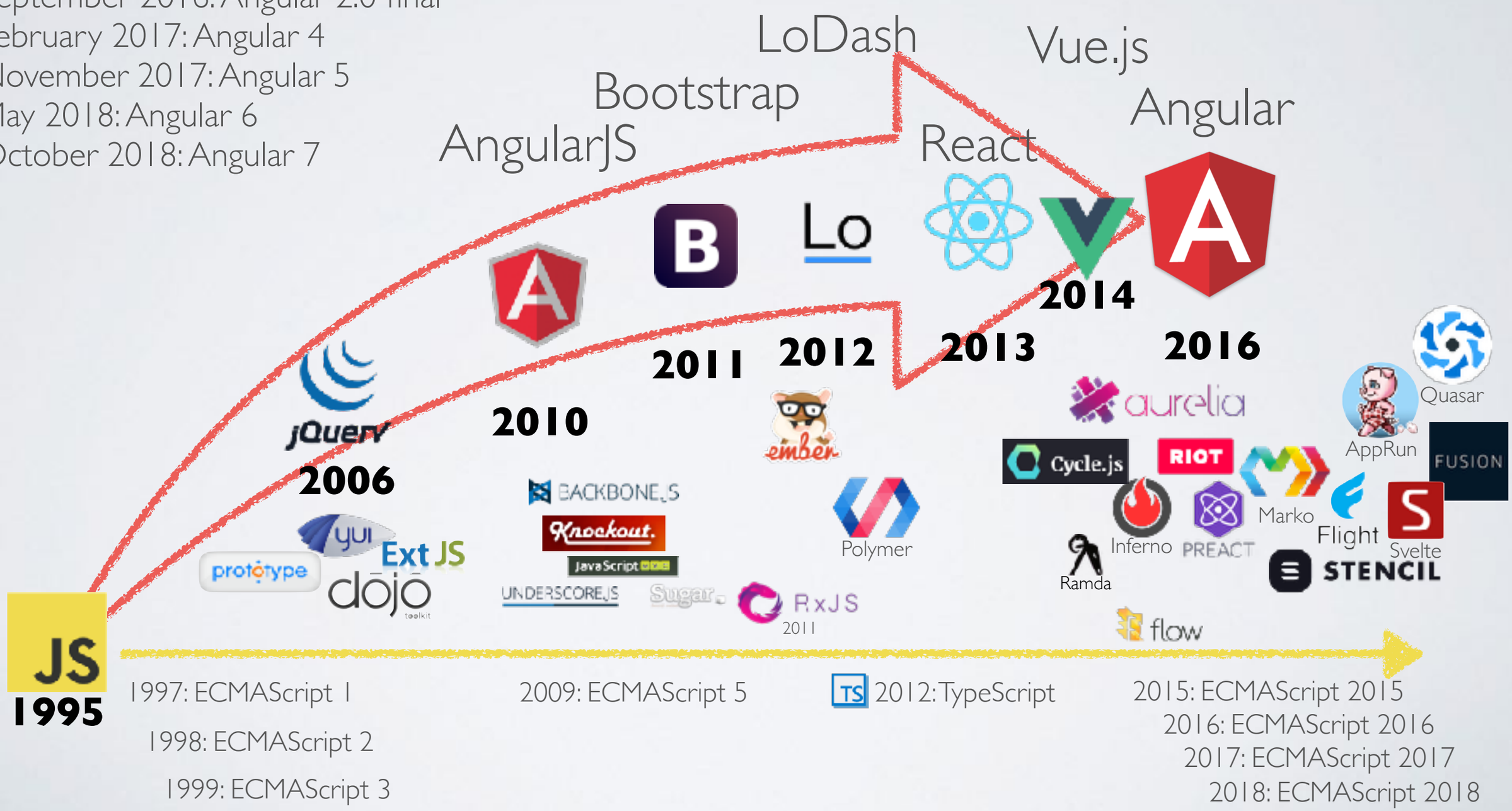
(discard all local changes)

Slides & Exercises: `<checkout>/00-CourseMaterial`

History of Angular

- October 2010: Release of AngularJS
- October 2014: Announcement of Angular 2
- December 2015: Angular 2 beta
- May 2016: Angular 2 release candidates
- September 2016: Angular 2.0 final
- February 2017: Angular 4
- November 2017: Angular 5
- May 2018: Angular 6
- October 2018: Angular 7

AngularJS: latest version 1.7.5 https://angularjs.org/
https://blog.angular.io/stable-angularjs-and-long-term-support-7e077635ee9c



Angular is **built upon** the modern web:

- shadow dom
- web workers

Angular is **built for** the modern web:

- mobile browsers
- modern browsers
- server-side rendering

Angular **improves** over AngularJS:

- faster
- easier to use & learn
- built on proven best practices (i.e. components, unidirectional data flow …)

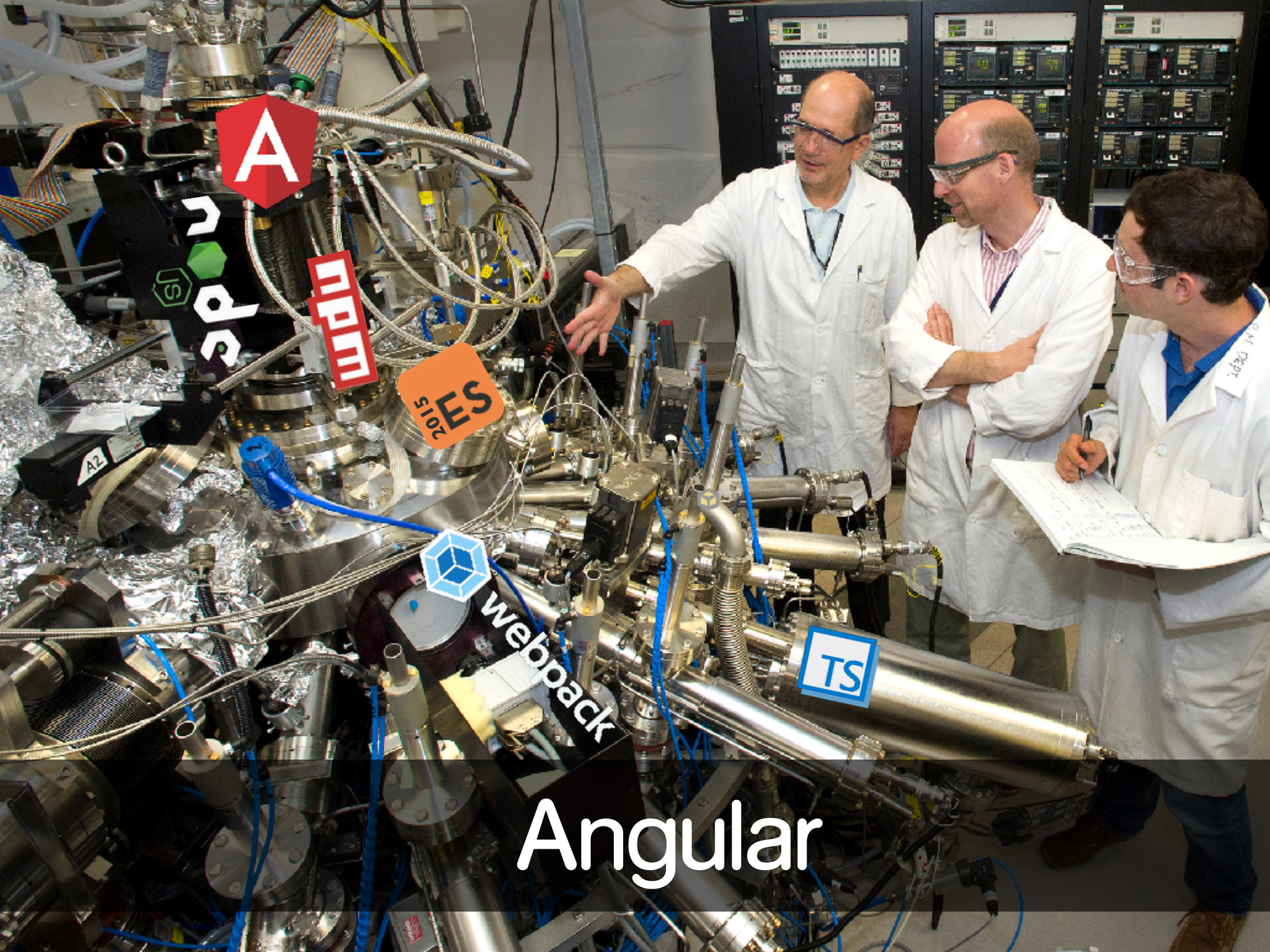
AngularJS

AngularJS

```
(function () {
    'use strict';

    var app = angular.module('todoApp');
    app.controller('todoController', ToDoController);

    ToDoController.$inject = ['toDoService'];
    function ToDoController(todoService) {
        var ctrl = this;

        ctrl.newToDo = new ToDoItem();
        ctrl.todos = todoService.getToDos();

        ctrl.addToDo = function () {
            ctrl.newToDo.created = new Date();
            todoService.addToDo(ctrl.newToDo);
            ctrl.newToDo = new ToDoItem();
        };

        ctrl.removeToDo = function (todo) {
            todoService.removeToDo(todo);
        };
    }
})();
```

Angular

```
import {Component,
  EventEmitter, Output} from '@angular/core';
import {ToDo} from '../../model/todo.model';

@Component({
  selector: 'td-new-todo',
  templateUrl: './new-todo.component.html',
})
export class NewTodoComponent {

  public newToDo: ToDo;

  constructor() {
    this.newToDo = new ToDo();
  }

  @Output() addToDo = new EventEmitter<ToDo>();

  onAddToDo(): void {
    this.addToDo.emit(this.newToDo);
    this.newToDo = new ToDo();
  }
}
```



Angular

Angular aspires to be a platform

"One framework. Mobile & desktop"

classic web-apps for desktops

WebPlatform.org

progressive web-apps for mobile
(web workers, cache, push, offline)

installed mobile apps
(hybrid - web-ui)

installed mobile apps
(hybrid - native UI)

server side rendering
https://angular.io/guide/universal

dev tooling

https://github.com/angular/angular-cli

https://angularconsole.com/

installed desktop apps

https://github.com/NathanWalker/angular-seed-advanced

Angular 5

Released November 2017: Angular 5 and Angular CLI 1.5 released.

https://blog.angular.io/version-5-0-0-of-angular-now-available-37e414935ced

Mainly performance improvements.
RxJS 5.5. New features in Forms.

Angular 6

Released May 2018

https://blog.angular.io/version-6-of-angular-now-available-cc56b0efa7a4

Alignment of project version numbers: Angular, CLI, Material
Dependency upgrades: RxJS 6, WebPack 4
CLI: new structure, update & add, support for libraries
Tree Shakable Services, Angular Elements (initial support)

Angular 7

Released October 2018

https://blog.angular.io/version-7-of-angular-cli-prompts-virtual-scroll-drag-and-drop-and-more-c594e22e7b8c

No relevant changes in the Angular framework.
Updates in Angular CLI & Angular CDK.

Angular 8

Released Mai 2018

https://blog.angular.io/version-8-of-angular-smaller-bundles-cli-apis-and-alignment-with-the-ecosystem-af026112a27

Syntax for lazy loading of routes now uses dynamic `import()`.
`@ViewChild` / `@ContentChild` must specify `static` vs. dynamic.
CLI: differential loading (different bundles for IE vs. modern browsers).
Experimental Ivy opt-in (new rendering engine), experimental Bazel usage (new build system)

Angular 9 (~October 2019)

New rendering engine Ivy.

What is Angular?

- Angular is a framework for dynamic web applications (aka. Single Page Applications)
- A framework to dynamically create & manipulate the DOM and linking it to application functionality.
- Angular is a HTML processor
- Angular lets you extend HTML with domain concepts

Getting started quickly with Angular CLI

Execute the following commands in a terminal:

```
npm install -g @angular/cli
ng new awesome-ng
cd awesome-ng
npm start
```

installs the CLI gobally

uses the global installation of the CLI to create a project … also installs the CLI also locally

uses the local installation of the CLI

https://angular.io/cli/

Using **npx** without installing the cli globally:

```
npx @angular/cli@6 new old-app
cd old-app
npm start
```

(**npx** is available since npm v5.2)

EXERCISES

Exercise 1 – Create an Angular App

Inspecting an Angular CLI App

```
npm start
npm run build -- --prod
npm test
npm e2e
npm run lint
```

App Component:

- /src/app

Bootstrapping:

- main.ts
- index.html
  (`<script>` tags:
  src vs. dev build vs. prod build)

Project Configuration:

- package.json
- angular.json
- tsconfig.json
- browserslist

Angular CLI commands: https://angular.io/cli
Angular CLI build options: https://angular.io/cli/build
Differential Loading: https://angular.io/guide/deployment#differential-builds

Angular CLI Build Setup

ES2015+/TypeScript

Write Code

TypeScript

Compile

TypeScript is compiled to ES5

webpack
MODULE BUNDLER

Bundle

Create coarse-grained & optimised bundles from many files and 3rd-party libraries

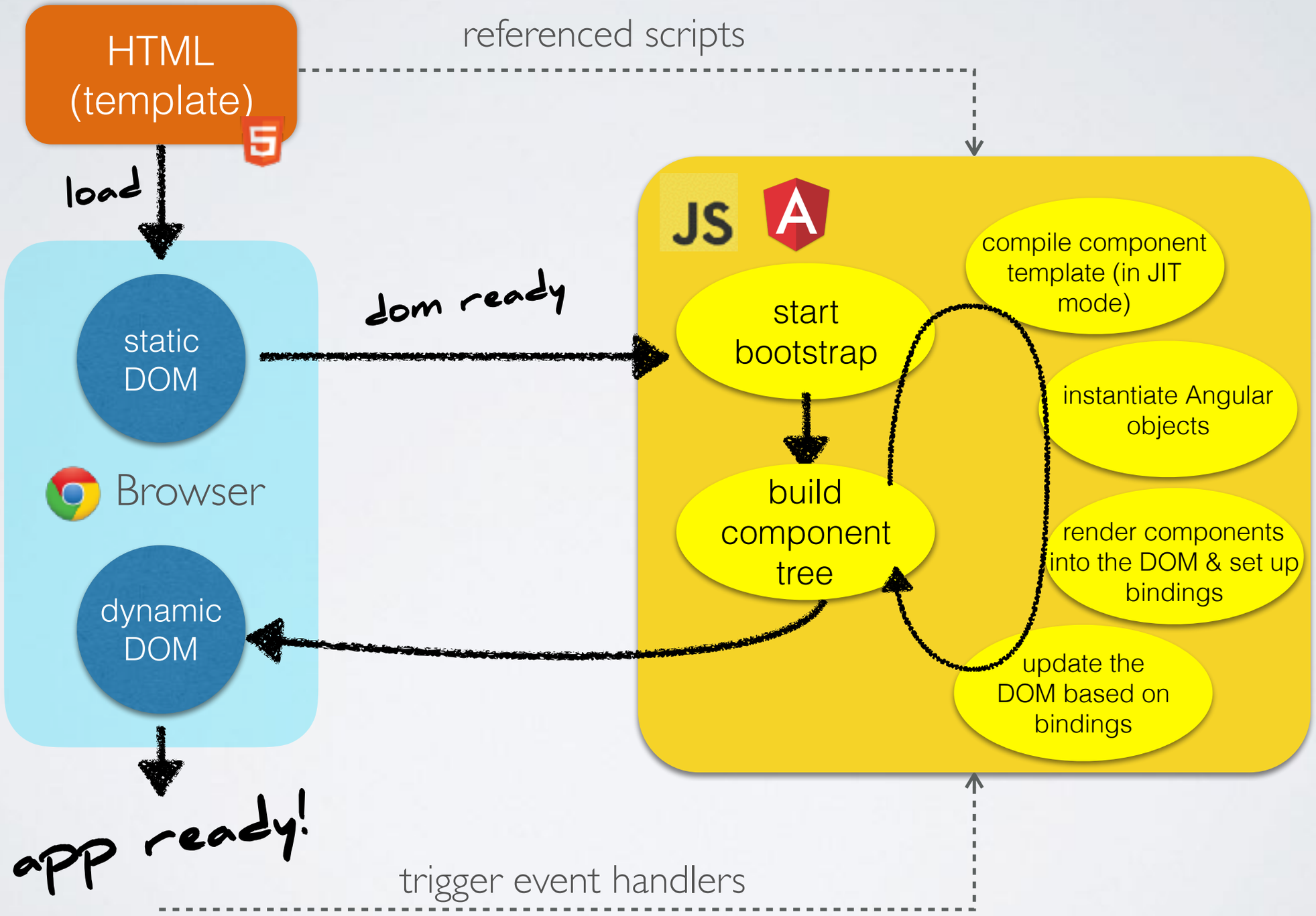
ES5

Run in Browser

Load pure JavaScript bundles.

Modern JavaScript Build: https://medium.com/the-node-js-collection/modern-javascript-explained-for-dinosaurs-f695e9747b70

How Angular Works



How Angular Works

"Angular is a HTML processor"

template

```
<ul class="todo-list" >
    <li *ngFor="let todo of todos">
        {{todo.title}}
        <button class="remove-button"
            (click)="removeToDo(todo)">
            x
        </button>
    </li>
</ul>
```

components
services
...

interact

parse

generate

manipulate

handle events

DOM

Bootstrapping

src/index.html

html shell

```
<body>

  <app-root>
    Loading...
  </app-root>
  <script src=…>
  </script>
```

src/main.ts

bootstrap-script

```
platformBrowserDynamic()
  .bootstrapModule(AppModule)
```

root module

```
@NgModule({
    imports: [BrowserModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule {}
```

src/app/app.module.ts

root component

```
@Component({
    selector: 'app-root',
    template: `Hello world`
})
export class AppComponent {}
```

src/app/app.component.ts

My concerns with the Angular CLI

Setting up an optimized Angular project from scratch is very hard!

I definitely recommend using the Angular CLI!

The CLI is also a risk:
It "hides" the whole toolchain (npm dependencies, webpack ...). If you find out later that the CLI setup does not fit your project, it is very difficult to set up an equivalent toolchain.

Example: https://medium.jonasbandi.net/angular-cli-and-moment-js-a-recipe-for-disaster-and-how-to-fix-it-163a79180173

Documentation of the CLI is sparse. Documentation how to set up an Angular build process without the CLI is even sparser.

Angular CLI used to offer the `ng eject` command to get the config files.
Unfortunately it was disabled in v6 and removed in v7.

ngx-build-plus allows to extend/override the webpack configuration of Angular CLI:
https://github.com/manfredsteyer/ngx-build-plus

Setting up an Angular project without the CLI:
https://medium.freecodecamp.org/how-to-configure-webpack-4-with-angular-7-a-complete-guide-9a23c879f471

Configuring the Angular CLI

Config file: **`angular.json`**

The defaults can be set via **`ng config`**:

```
ng config schematics.@schematics/angular.component.spec false
ng config schematics.@schematics/angular.component.styleext scss

ng config schematics.@schematics/angular.component.inlineTemplate true
ng config schematics.@schematics/angular.component.inlineStyle true
```

(unfortunately no documentation of all the configs, you have too look into the schema of angular.json: https://github.com/angular/angular-cli/wiki/angular-workspace)

CLI: Creating a Production Build

On the commandline using `ng` directly:

```
ng build --prod
```

On the commandline via `npm`:

```
npm run build -- --prod
```

Configuration via `package.json`:

```
"scripts": {
  ...
  "build": "ng build --prod"
}
```

The `dist` directory contains the output of the build.

It contains all assets necessary to deploy and run the SPA.

Serving a prod build in development:

```
ng serve --prod
```

```
npx serve -s dist/awesome-ng
```

```
npx gzip-cli dist/awesome-ng/*.js
npx http-server dist/awesome-ng -g
```

Further configuration of the build:
https://angular.io/cli/build

Relative URLs in Production

Development Setup:

`http://localhost:4200/`first
second
...

Angular CLI
(Webpack Dev Server)

assets
index.html
api

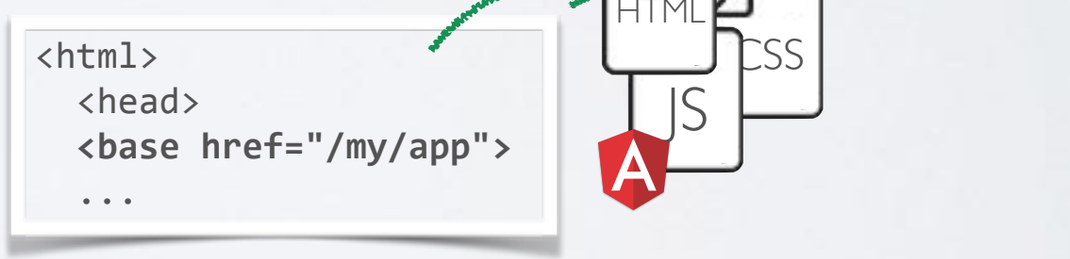

```
<html>
  <head>
  <base href="/">
  ...
```

Production Setup:

`http://www.mysite.com/`**`my/app`**`/`first
second
...

Web Server
(Apache, NgNx, Tomcat ...)

assets
index.html
api




```
<html>
  <head>
  <base href="/my/app">
  ...
```

Building with the CLI:

```
ng build --prod --baseHref=/my/app/
```

The **`base href`** is used by the browser to resolve relative paths and by Angular to create dynamic routes.

There is also the **`--deployUrl`** to explocitly define the path for static assets (js, css)

https://angular.io/guide/deployment#the-base-tag
https://angular.io/cli/build

IE Support:
JS Polyfills & CSS Autoprefixer

Angular (the JS library) assumes a modern browser.
Older browsers have to be patched to appear like a modern browser.
The Angular CLI prepares polyfills to make Angular work in IE.

Angular CLI 8: Differential Loading

Angular CLI 8 introduced "differential loading":

- The CLI generates JavaScript for modern browsers as default (i.e. using ES2015 classes)
- If you configure the project to support older browsers, the production build additionally produces ES5 JavaScript bundles (i.e no classes) and a bundle with the needed polyfills. The appropriate bundles are then referenced in `index.html` via `type="module"` / `nomodule`.

Advantage: smaller bundles for modern browsers.
Disadvantage: testing IE is only possible with the production build.

Configuration:

tsconfig.json:

```
"compilerOptions": {
  "target": "es2015",
  ...
```

browserslist:

```
> 0.5%
last 2 versions
Firefox ESR
not dead
IE 9-11
```

Differential loading can be disabled by setting target to "es5" in tsconfig.json.

Note: `browserslist` also configures autoprefixer for better CSS support.

https://angular.io/guide/deployment#differential-loading

https://angular.io/guide/browser-support
https://angular.io/guide/build#configuring-browser-compatibility
https://github.com/postcss/autoprefixer

IE Support:
Previous Versions of Angular

Angular CLI 7.3:

- Bundles were transpiled to ES5 as default. No differential build of the application bundles.
- Differential build for polyfills:
  If `"es5BrowserSupport": true` in `angular.json` (architect->build->options) then the CLI generated a `es2015-polyfills.js` for older browsers.

For Angular CLI < 7.3:

Polyfills for older browsers had to be included in `src/polyfills.ts`.

(The CLI generated these lines, butt they were commented out)

```
import 'core-js/es6/symbol';
import 'core-js/es6/object';
import 'core-js/es6/function';
import 'core-js/es6/parse-int';
import 'core-js/es6/parse-float';
import 'core-js/es6/number';
import 'core-js/es6/math';
import 'core-js/es6/string';
import 'core-js/es6/date';
import 'core-js/es6/array';
import 'core-js/es6/regexp';
import 'core-js/es6/map';
import 'core-js/es6/weak-map';
import 'core-js/es6/set';
```

CLI: Performance Budgets

angular.json

```
{
  ...
  "configurations": {
      "production": {
          ...
          "budgets": [
            {
              "type": "bundle",
              "name": "main",
              "baseline": "500kb",
              "warning": "50kb",
              "error": "100kb"
            }
          ]
```

With budgets you can control the size and growth of the JavaScript bundles over time.
The build will fail if the error limit is exeeded.

The feature is based on WebPack performance budgets:
https://webpack.js.org/configuration/performance/

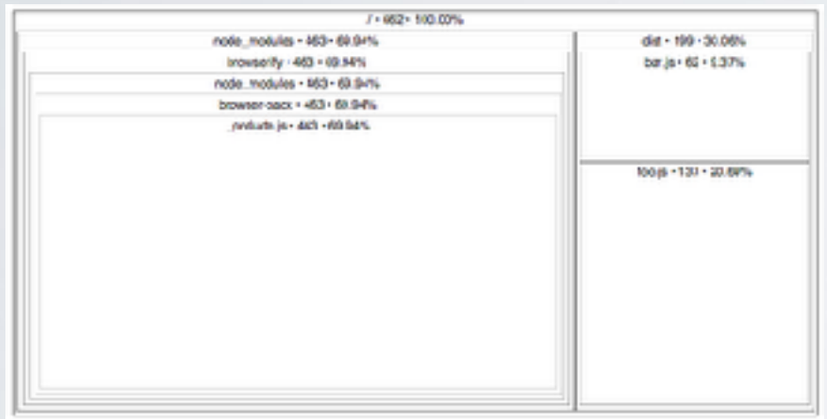
In Angular 7 the CLI generates a performance budget of 2MB (warning) 5MB (error).

The Chrome team at Google suggests a performance budget of 170KB for for the inital JavaScript bundle:
https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/budgets.md
https://medium.com/@tomastrajan/how-did-angular-cli-budgets-save-my-day-and-how-they-can-save-yours-300d534aae7a

Analyzing Bundles

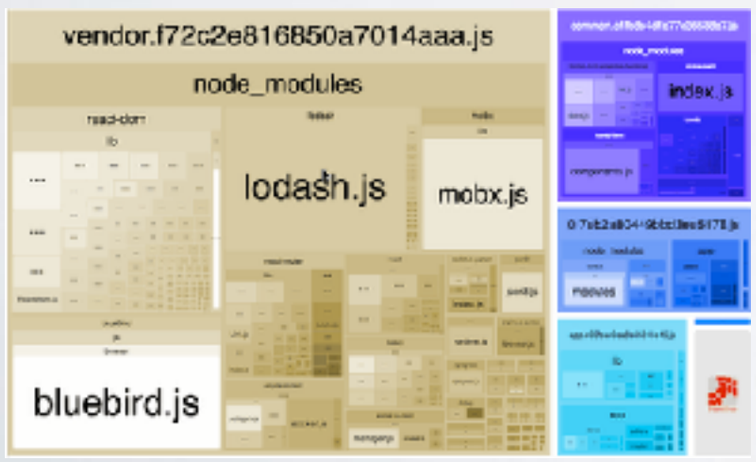
source-map-explorer:
https://www.npmjs.com/package/source-map-explorer




```
npm install -g source-map-explorer
```

```
ng build --prod --source-map
source-map-explorer dist/ng-app/main.XYZ.js
```

webpack-bundle-analyzer:
https://github.com/webpack-contrib/webpack-bundle-analyzer




```
npm install -g webpack-bundle-analyzer
```

```
ng build --prod --stats-json
webpack-bundle-analyzer dist/ng-app/stats.json
```

https://angular.io/guide/deployment

Updating an Angular CLI project

The Angular projects provides detailed instructions
for updating projects between versions:

https://update.angular.io/

The Angular CLI has the **`update`** command to
update an application and it's dependencies:

```
ng update @angular/cli @angular/core
```

https://angular.io/cli/update

Warning: This did not work flawlessly in my experience!

Popular Editors

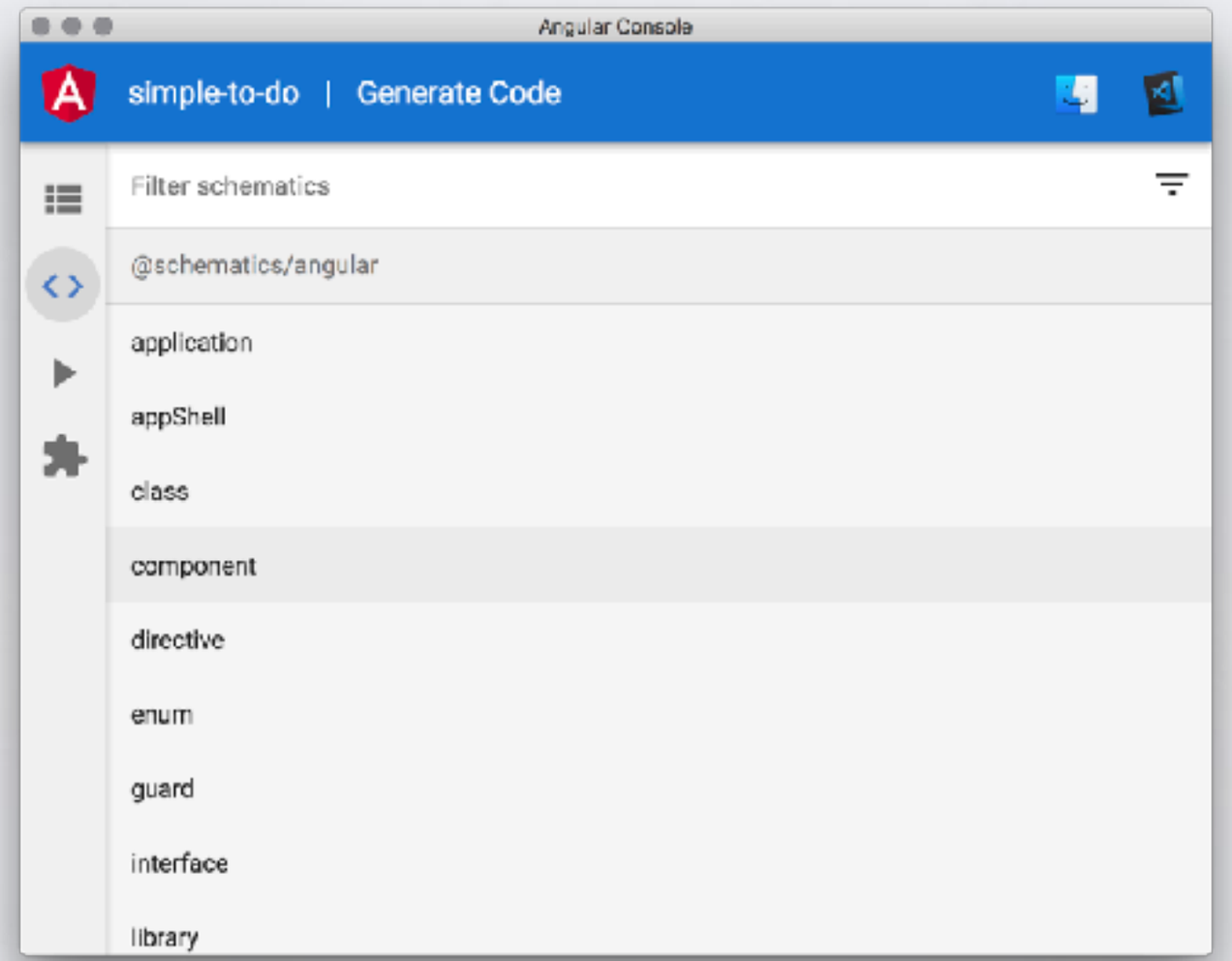
Webstrom & IntelliJ:
Angular 2 TypeScript Live Templates
https://github.com/MrZaYaC/ng2-webstorm-snippets
https://plugins.jetbrains.com/plugin/8395-angular-2-typescript-live-templates

Visual Studio Code:
Angular Essentials
https://github.com/MrZaYaC/ng2-webstorm-snippets

Angular Console

A GUI wrapper around the Angular CLI

https://angularconsole.com/



Debugging

Angular is "just" JavaScript at runtime...

Try the following statements in the console:

```
var helloDomElement = document.getElementsByTagName("app-hello")[0]
var debugElement = ng.probe(helloDomElement)
var helloComponent = debugElement.componentInstance
helloComponent.greetingName = 'Universe'
debugElement.injector.get(ng.coreTokens.ApplicationRef).tick()
```

(the global **ng** is only available in a debug build)

Debugging in Chrome:

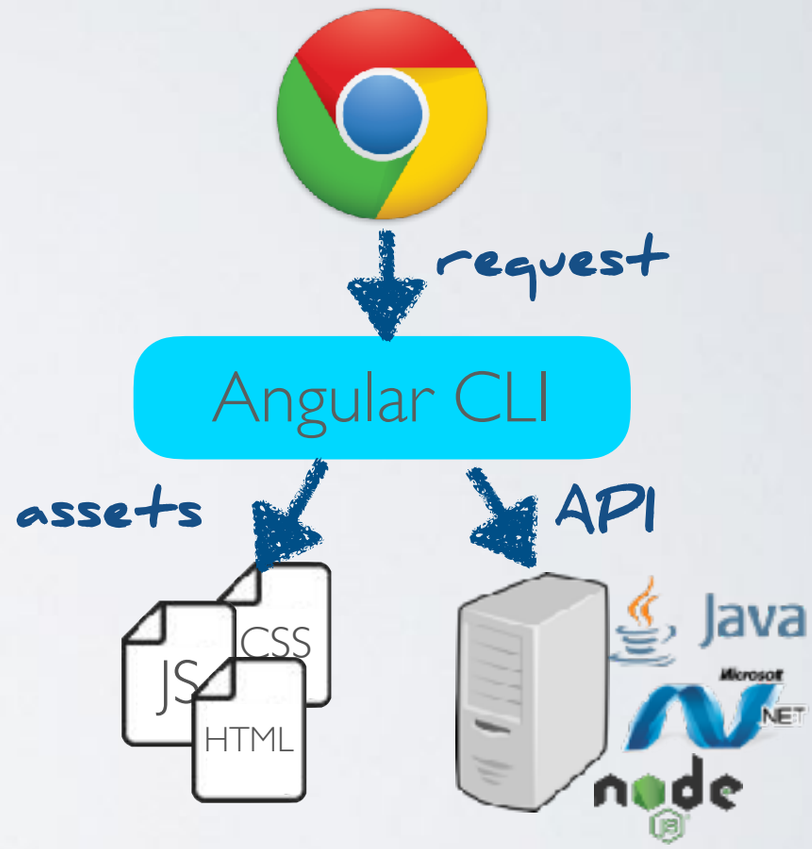
https://augury.angular.io/

CLI: Proxy To Backend

Scenario: Serve your Angular application (static assets) locally during development but access an API which is hosted on another HTTP endpoint.

proxy.conf.json (file must be added manually)

```
{
  "/api": {
    "target": "http://localhost:3456",
    "pathRewrite": {"^/api" : ""},
    "secure": false
  }
}
```



package.json

```
"start": "ng serve --proxy-config proxy.conf.json",
```

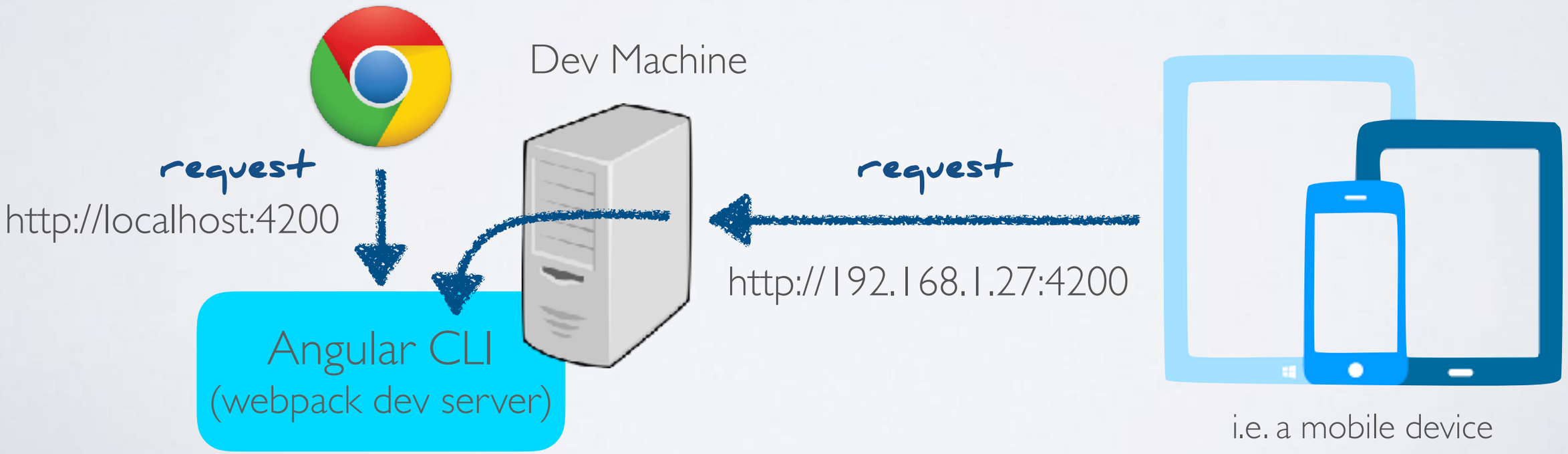
https://github.com/angular/angular-cli/blob/master/docs/documentation/stories/proxy.md

CLI: Access from Other Device

Scenario: Serve your Angular application locally during development but access it with another device.

package.json

```
"start": "ng serve --host 0.0.0.0",
```



CLI: Adding Styling

angular.json

```
{
  ...
  "styles": [
     "src/styles.css",
     "node_modules/bootstrap/dist/css/bootstrap.min.css"
  ]
   ...
}
```

src/styles.css

```
@import '~bootstrap/dist/css/bootstrap.min.css';
```

src/styles.scss

```
@import '~bootstrap/scss/bootstrap.scss';
```

src/app/app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['body {background-color: green}'],
  encapsulation: ViewEncapsulation.None
})
export class AppComponent {}
```

CLI: Adding 3rd Party Components

Installation:

```
npm install @ng-bootstrap/ng-bootstrap
```

Usage:

src/app/app.module.ts

```
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    NgbModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

src/app/app.component.ts

```
<ngb-datepicker></ngb-datepicker>
```

https://ng-bootstrap.github.io/#/home

Toolset

IDE

Language

Libraries / Frameworks

dependency management

Write Code

Toolset

compile

package

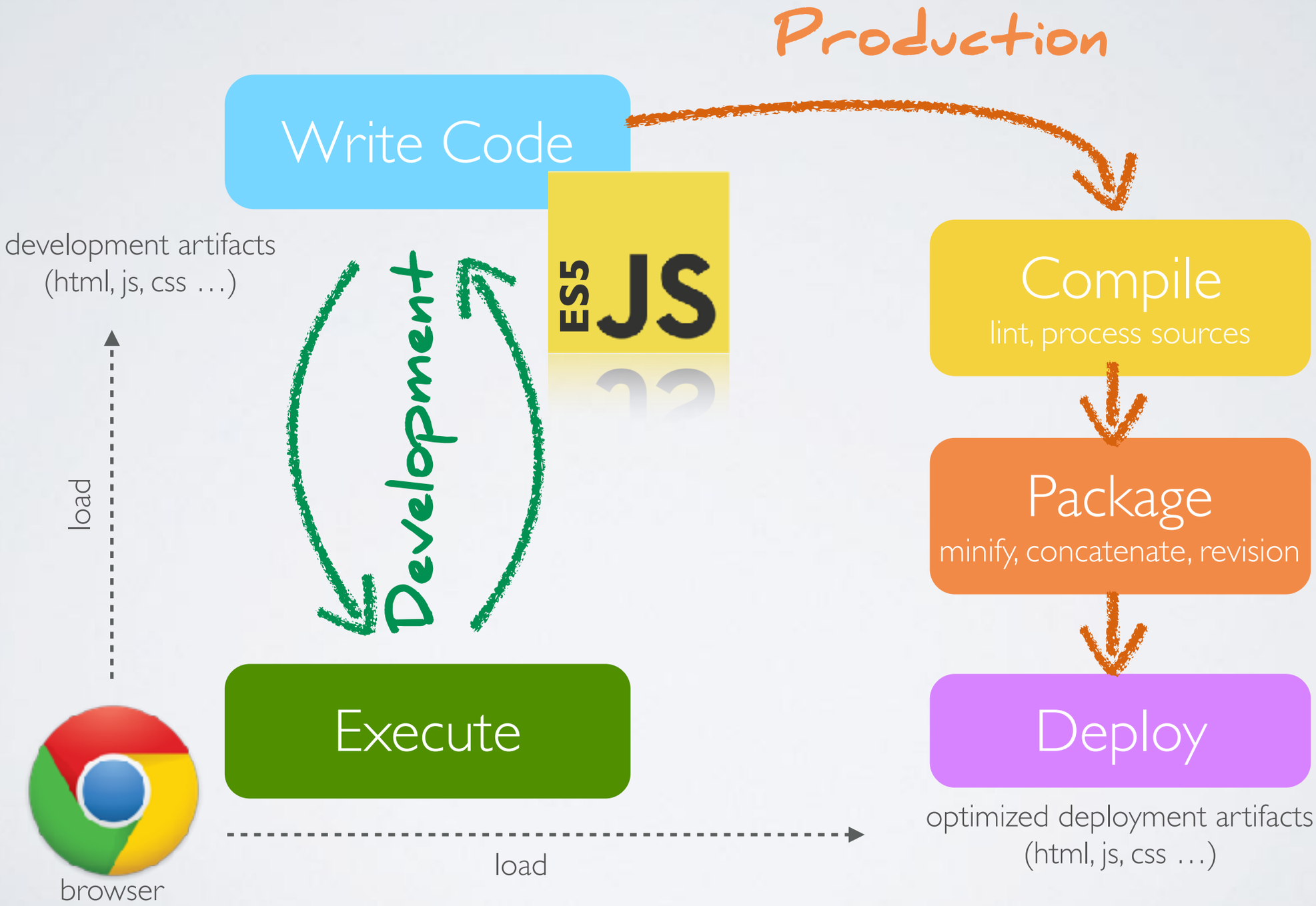
Build

Test

Deploy

Test-Frameworks

Workflow Plain ES5



Modern JavaScript Workflow

write fine-grained development artifacts (typescript, es2015+, sass ...)

Write Code

Production

Development

TypeScript

BABEL

Compile
lint, transpile

transpiled development artifacts

webpack
MODULE BUNDLER

perform module resolution as build step

(what about modern browsers that support ES2015+ and modules natively?)

Package
bundle (minify, revision)

Execute

browser

load

bundled coarse-grained & runnable development artifacts (html, js, css …)

include 3rd party dependencise

load

Deploy

optimized production artifacts (html, js, css …)

http://webpack.js.org/
https://github.com/systemjs/systemjs
http://engineering.khanacademy.org/posts/js-packaging-http2.htm
https://www.contentful.com/blog/2017/04/04/es6-modules-support-lands-in-browsers-is-it-time-to-rethink-bundling/

Dependency Management

Declare & Resolve project dependencies.
Including transitive dependencies.

Build Automation

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

Bundling

Build one or several bundled asset files for deployment.
Resolve module dependencies.
Optimize asset files for production.

Transpilation

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

Static Type-System

Check type correctness of source code with (optional) static types at development time.

Linting

Static code analysis.

npm is installed together with Node

npm offers:

- Dependency Management (global and local packages)
- Scripts: Simple Build Tasks

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`
  (note: `npm install` still upgrades top-level packages if no exact version in `package.json`)
- Public Repository: npmjs.org
- Config: `.npmrc`

Tip: `npm config set save-exact true`

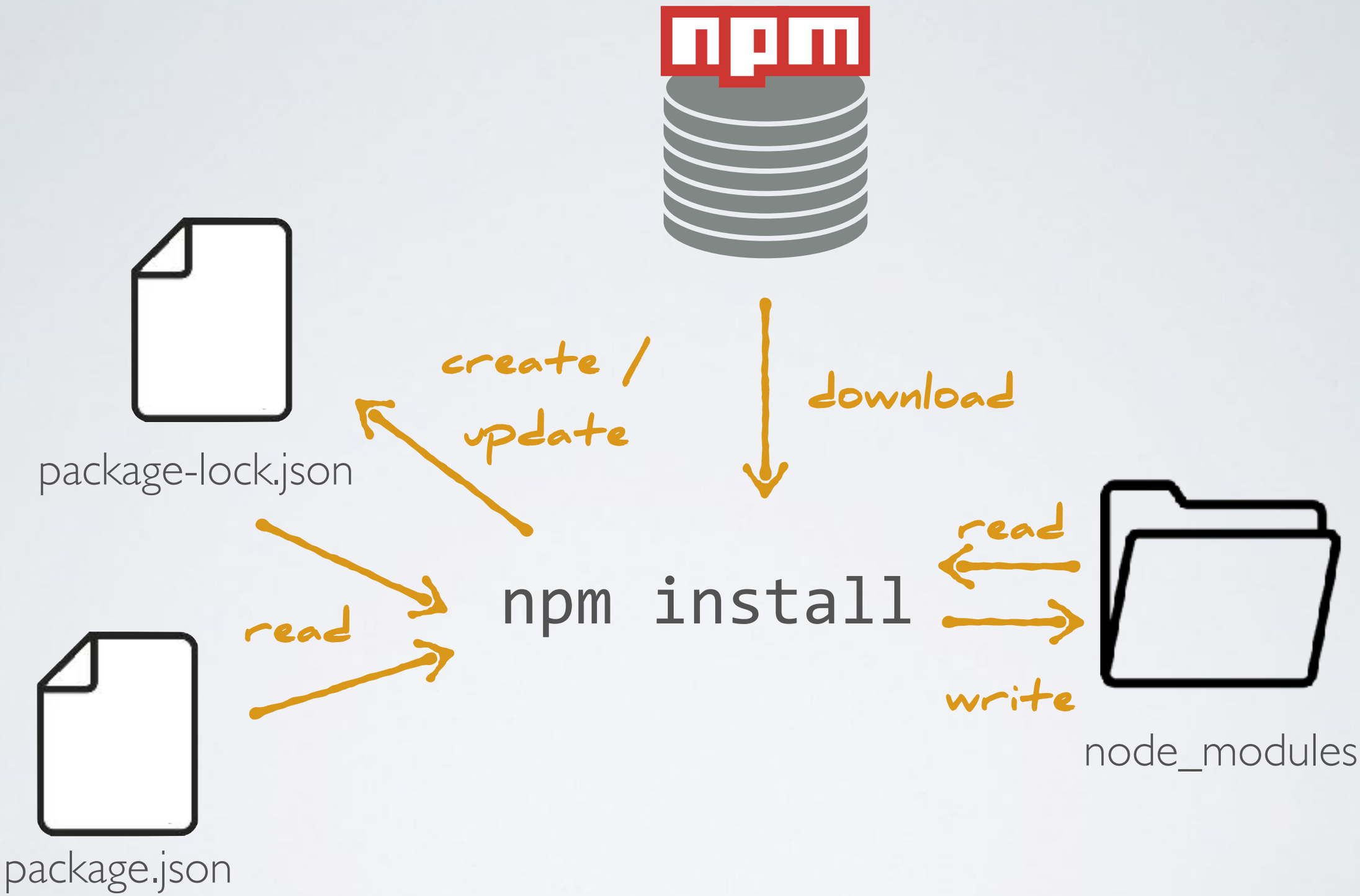
See: https://semver.npmjs.com/

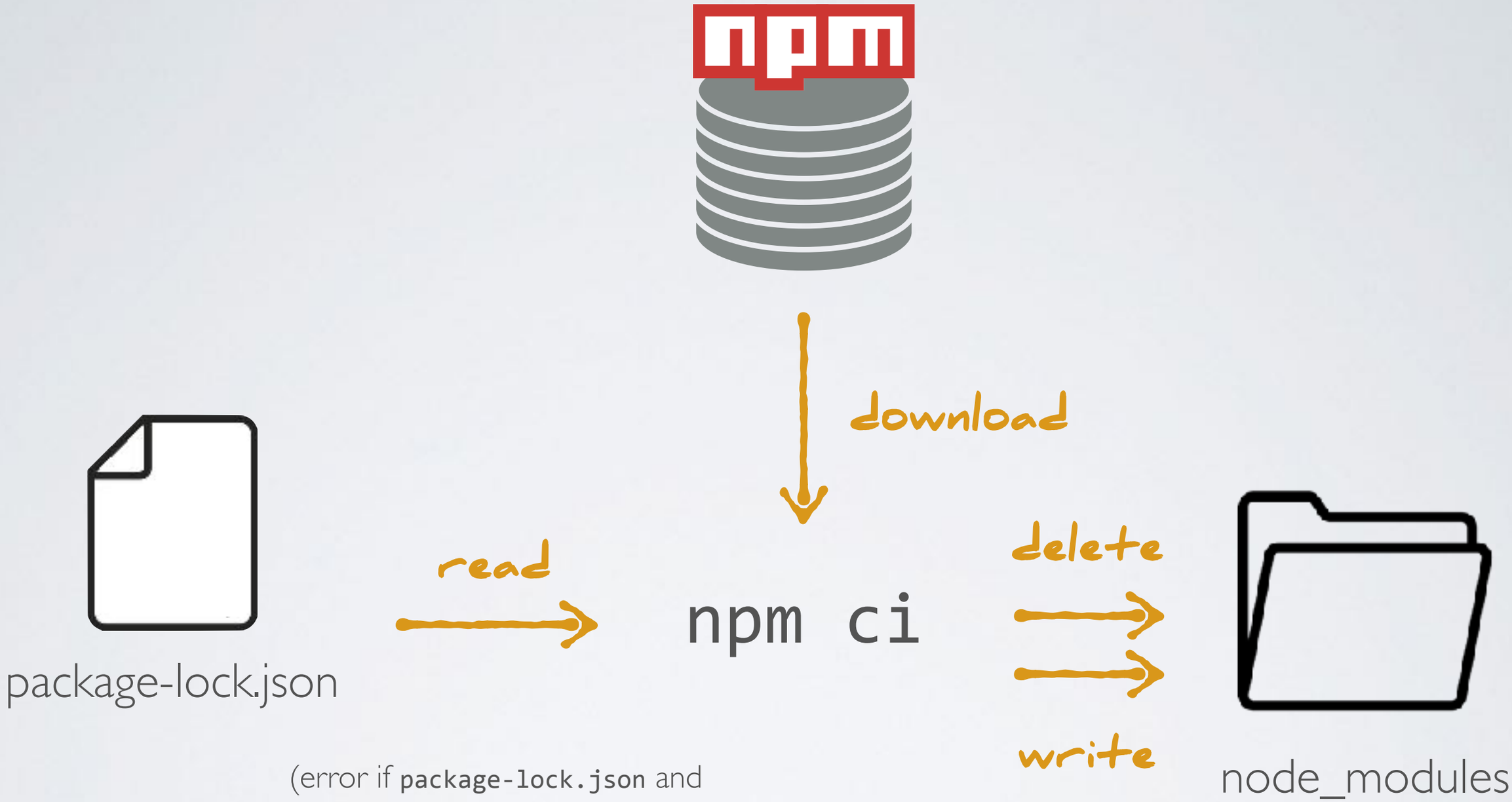
Typical commands:

```
npm search
npm info
npm install
npm uninstall
npm list
npm update
npm init
npm root
npm config
npm ci
npm audit
```

Flags:

```
--global / -g
--help
--save-dev / -D
--save-exact / -E
```


npm
create / update
download
package-lock.json
read
npm install
read
write
node_modules
package.json

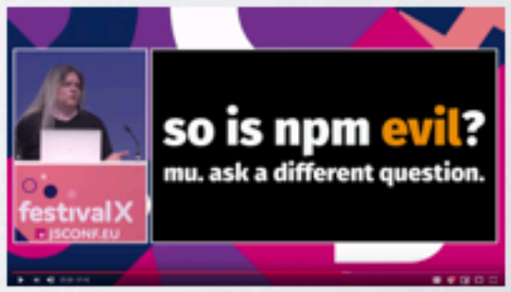


(error if `package-lock.json` and `package.json` do not match)

https://docs.npmjs.com/cli/ci

npm inc. is a private company

- the npm registry is a centralized system owned and operated by a venture capital backed private company
- the npm registry is not open-source



JSConf EU 2019: the economics of open source
https://www.youtube.com/watch?v=MO8hZIgK5zc

Alternatives to npm

- yarn: https://yarnpkg.com
  Initially faster and deterministic compared to npm. Today no big difference any more. Also using the public npm registry.

- pnpn: https://pnpm.js.org/
  A drop-in-replacement which keeps node_modules in a central repository (similar to maven).

- jspm: https://jspm.org/
  A completely different approach.

- entropic: https://github.com/entropic-dev/entropic
  A project atempting to replace the central npm registry by a federated registry.

yarn …a better npm client?

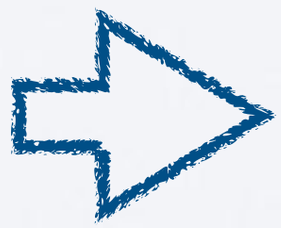
npm@5 also introduced locking and speeds similar to yarn

https://yarnpkg.com/

Yarn was created to be faster than npm and to provide 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
https://github.com/thomaschaaf/npm-vs-yarn

Experimental ...

NPM and Yarn are both experimenting with improved package managers concepts which do mot use the traditional `node_modules`:

Yarn Plug'n'Play:
https://github.com/yarnpkg/yarn/pull/6382
https://github.com/yarnpkg/pnp-sample-app

npm tink:
https://github.com/npm/tink

Updating Dependencies

Get information about outdated npm-packages:

```
npm outdated
npm update
```

```
yarn outdated
yarn upgrade
yarn upgrade-interactive
```

`npm outdated` / `yarn outdated` also show the latest versions available.
`npm update` / `yarn upgrade` only update within the version range specified in `package.json`

Be careful with manually changing `package.json`: remember that `package-lock.json` resp. `yarn.lock` has to be updated too ...

There are also 3rd party tools that can help with the task:

```
npx npm-check -u
```

https://github.com/dylang/npm-check

npm run <script>

execute npm scripts

```
npm start
npm test
npm run build
npm run lint
```

package.json

```
"scripts": {
    "start": "lite-server",
     "lint": "eslint src/**/*.js"
  },
```

In addition to the shell's pre-existing **PATH**, **npm run** adds **node_modules/.bin** to the **PATH** provided to scripts.

Passing arguments to the script command:

```
npm run test -- --grep="pattern"
```

https://docs.npmjs.com/cli/run-script

`npx <command>`

(npm 5.2 or later)

execute npm package binaries

```
npx npm-check -u
```

Executes **`<command>`** either from a local **`node_modules/.bin`**, or from a central cache, installing any packages needed in order for **`<command>`** to run.

`npm init <inizializer>`

(npm 6 or later)

create a project based on an inizializer npm package

```
npm init react-app ./my-react-app
```

i.e. **`create-react-app`** is an npm package

`npm audit` (npm 6 or later)

Npm maintains a databse of known JavaScript package vulnerabilities.

Scan your project for vulnerabilities:

```
npm audit
```

Automatically install any compatible updates to vulnerable dependencies:

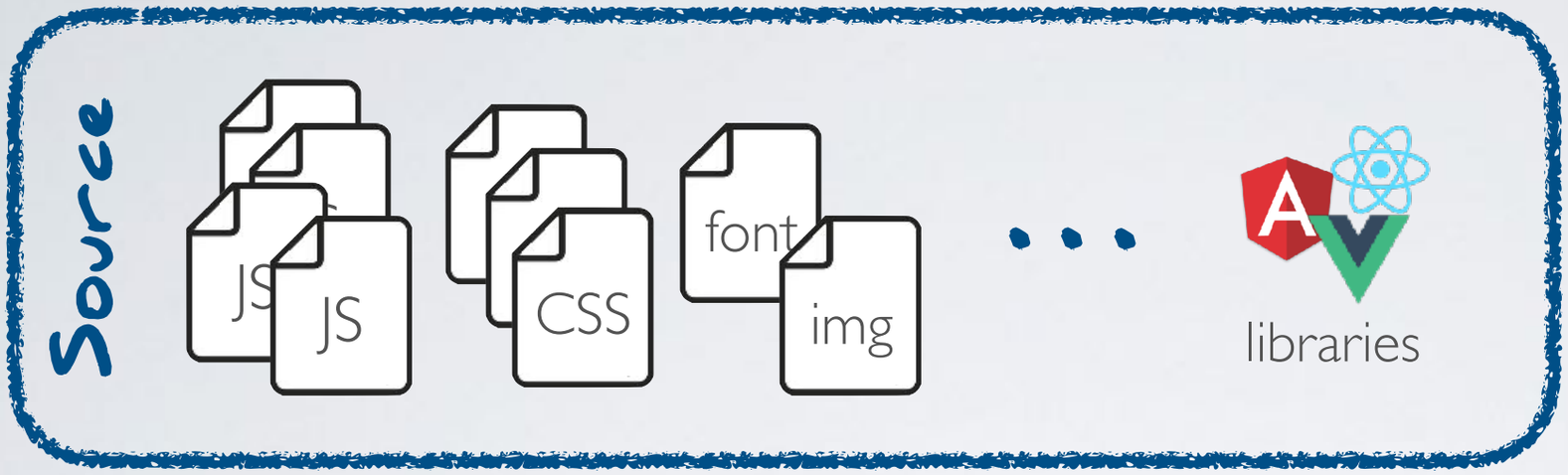
```
npm audit fix
```

https://docs.npmjs.com/cli/audit

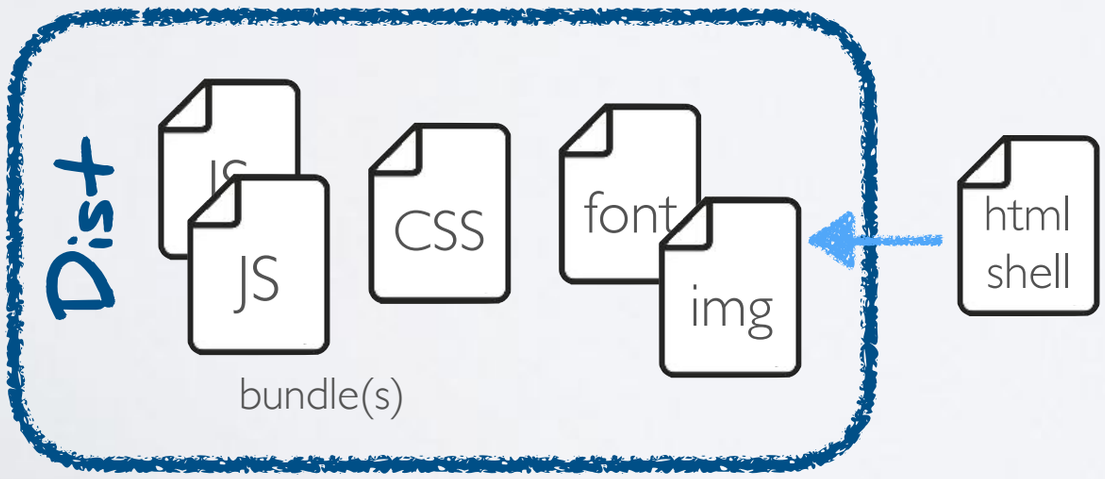
https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5

Bundling

Bundling



Development artifacts:
- fine-grained
- not optimized
- contain unused code

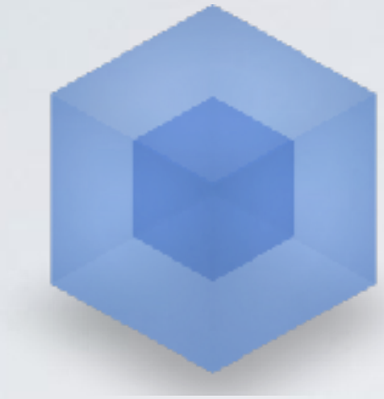


Deployment artifacts:
- coarse-grained
- optimized (size, performance ...)
- unused code is eliminated
- cacheable

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



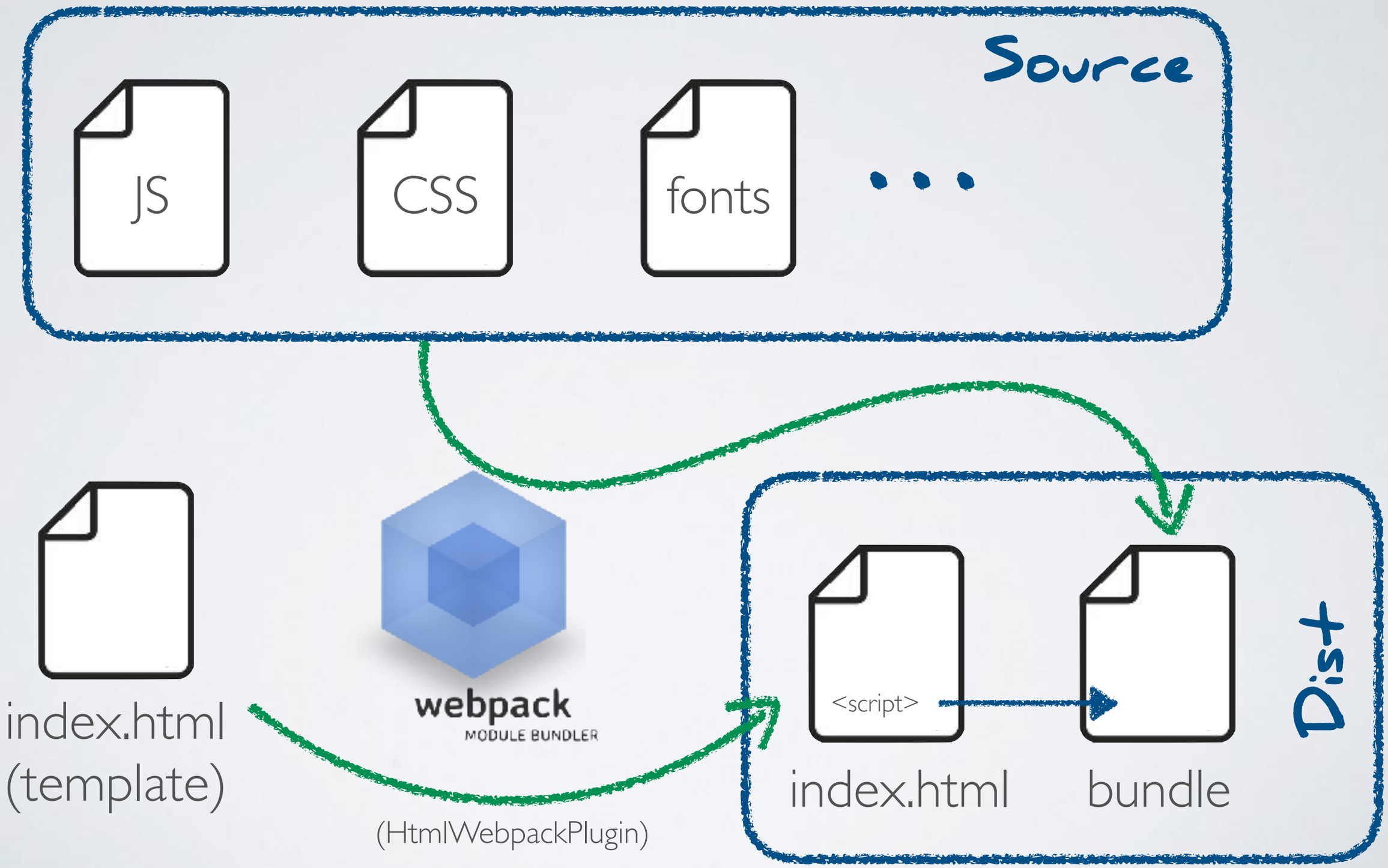
WebPack

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



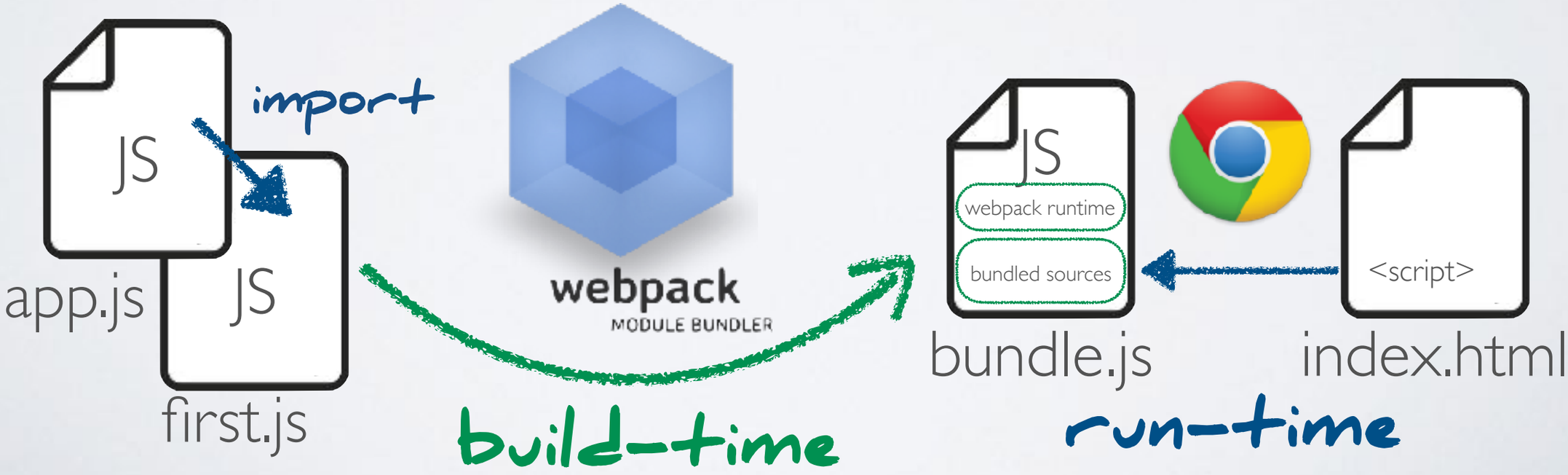
ES2015 Modules with WebPack

index.html

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

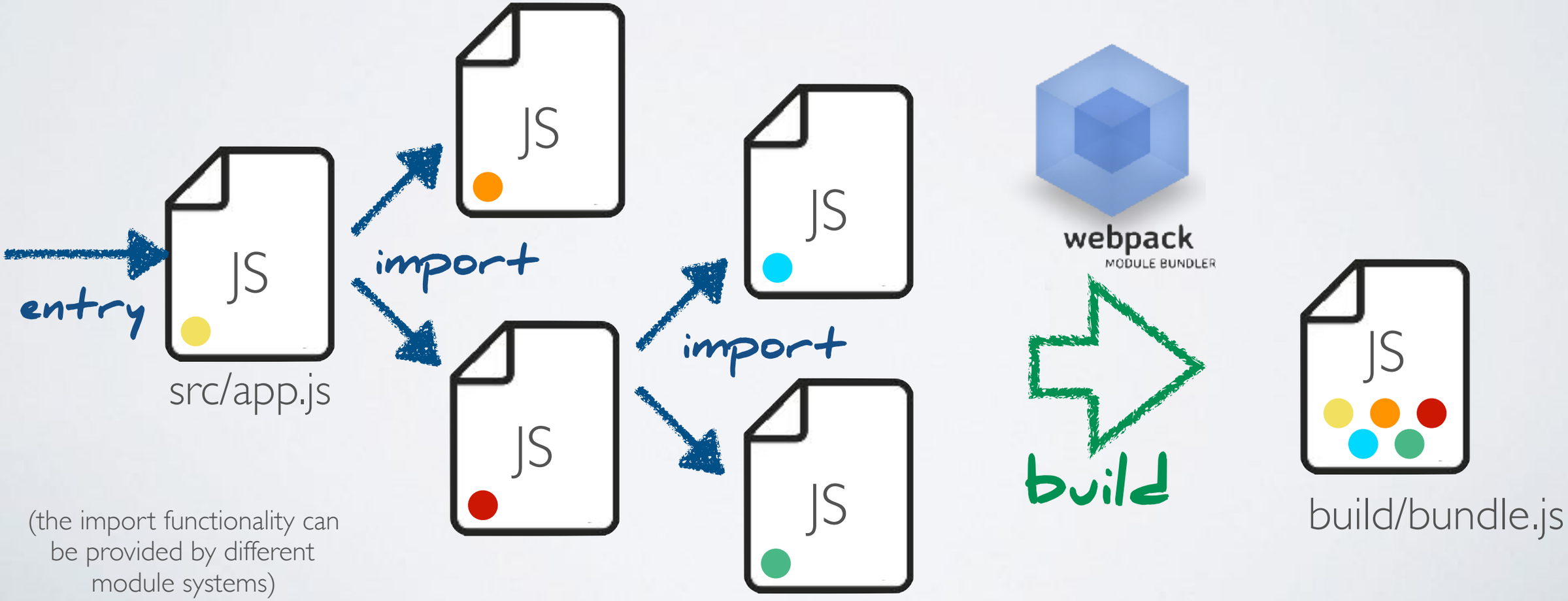
app.js

```
import './modules/first';
console.log("Hello from ES2015 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)

WebPack further Configuration

Automatically insert bundles into HTML-Template:

https://github.com/jantimon/html-webpack-plugin

"Code splitting"- create multiple bundles:

https://webpack.js.org/guides/code-splitting/

Bundling CSS:

https://webpack.js.org/guides/code-splitting-css/

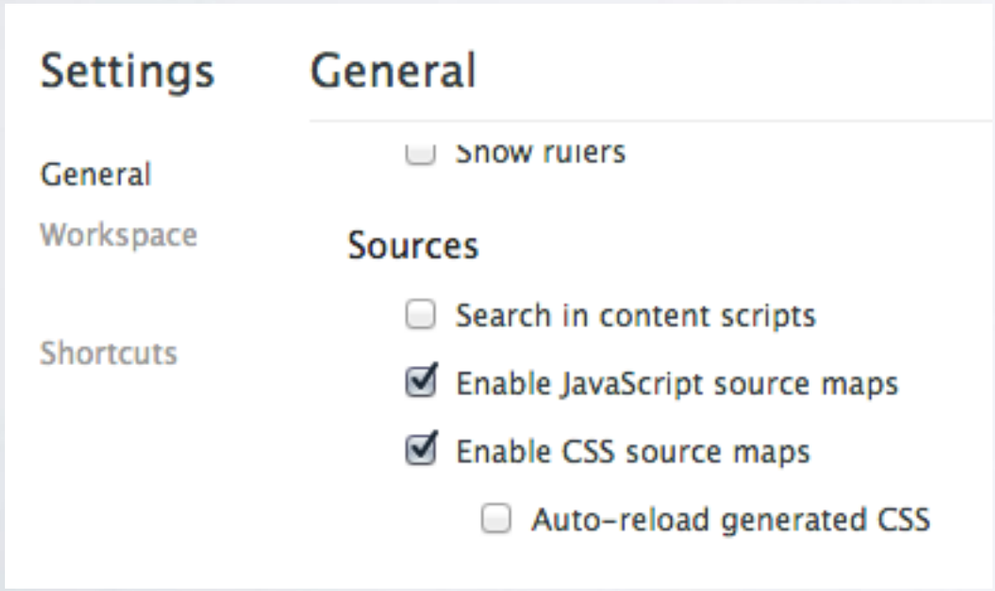
Lazy loading of chunks:

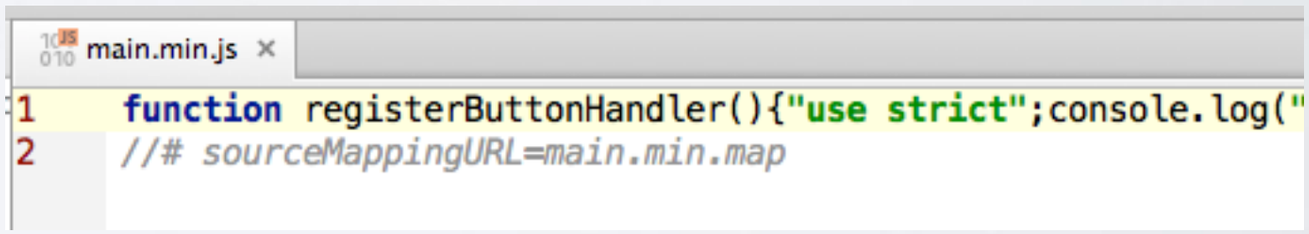
https://webpack.js.org/guides/code-splitting-async/

Sourcemaps

- Build: Tools that transform sources generate a map of from the resulting artefact to the original sources:
  - `jquery.js -> jquery.min.js & jquery.min.map`
  - `main.ts -> main.js & main.min.map`

- Runtime: Browsers perform the mapping when debugging
  => the executed code is mapped to the original code, which is displayed for debugging

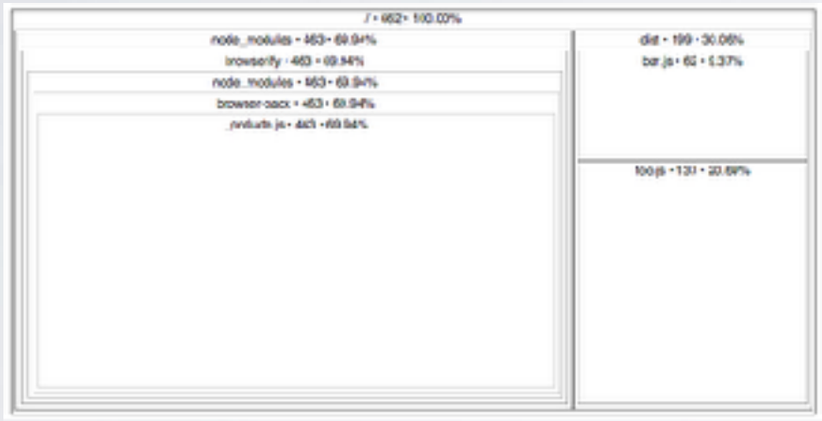




Analyze WebPack Bundles

Sometimes you want to know what your bundles contain and how much space is used by which package …

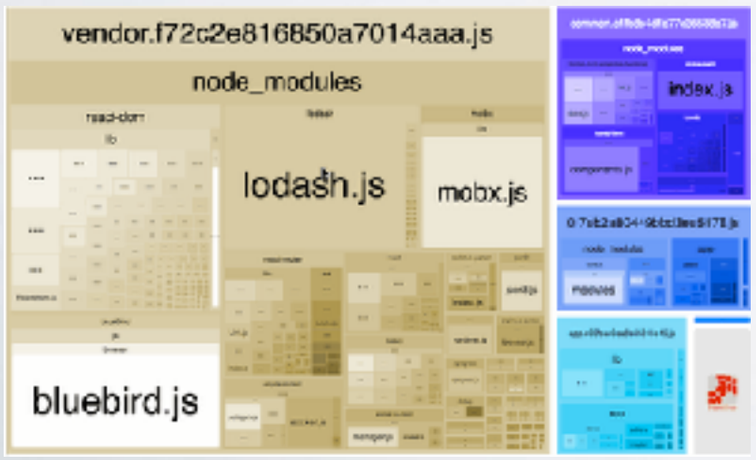
source-map-explorer:
https://www.npmjs.com/package/source-map-explorer




```
npm install -g source-map-explorer
```

```
// configure webpack to generate source maps
source-map-explorer dist/*.js
```

webpack-bundle-analyzer:
https://github.com/webpack-contrib/webpack-bundle-analyzer




```
npm install -g webpack-bundle-analyzer
```

```
webpack --stats
```

```
webpack-bundle-analyzer dist/stats.json
```

ES2015 browser support:

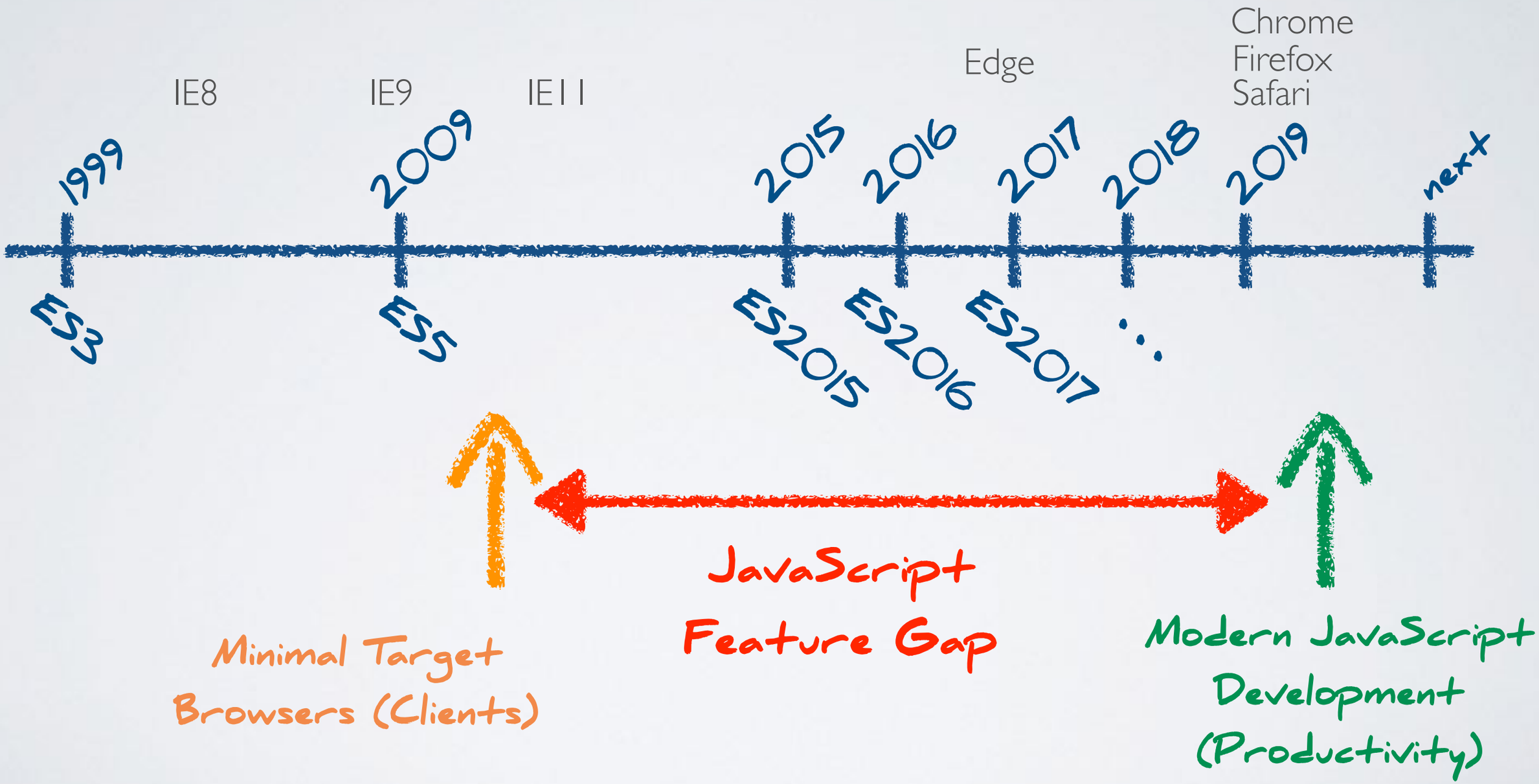
Desktop browsers

IE 11	Edge 17	Edge 18	FF 60 ESR	FF 66	CH 72, OP 59	CH 73, OP 60	SF 12.1
11%	96%	96%	98%	98%	98%	98%	99%

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

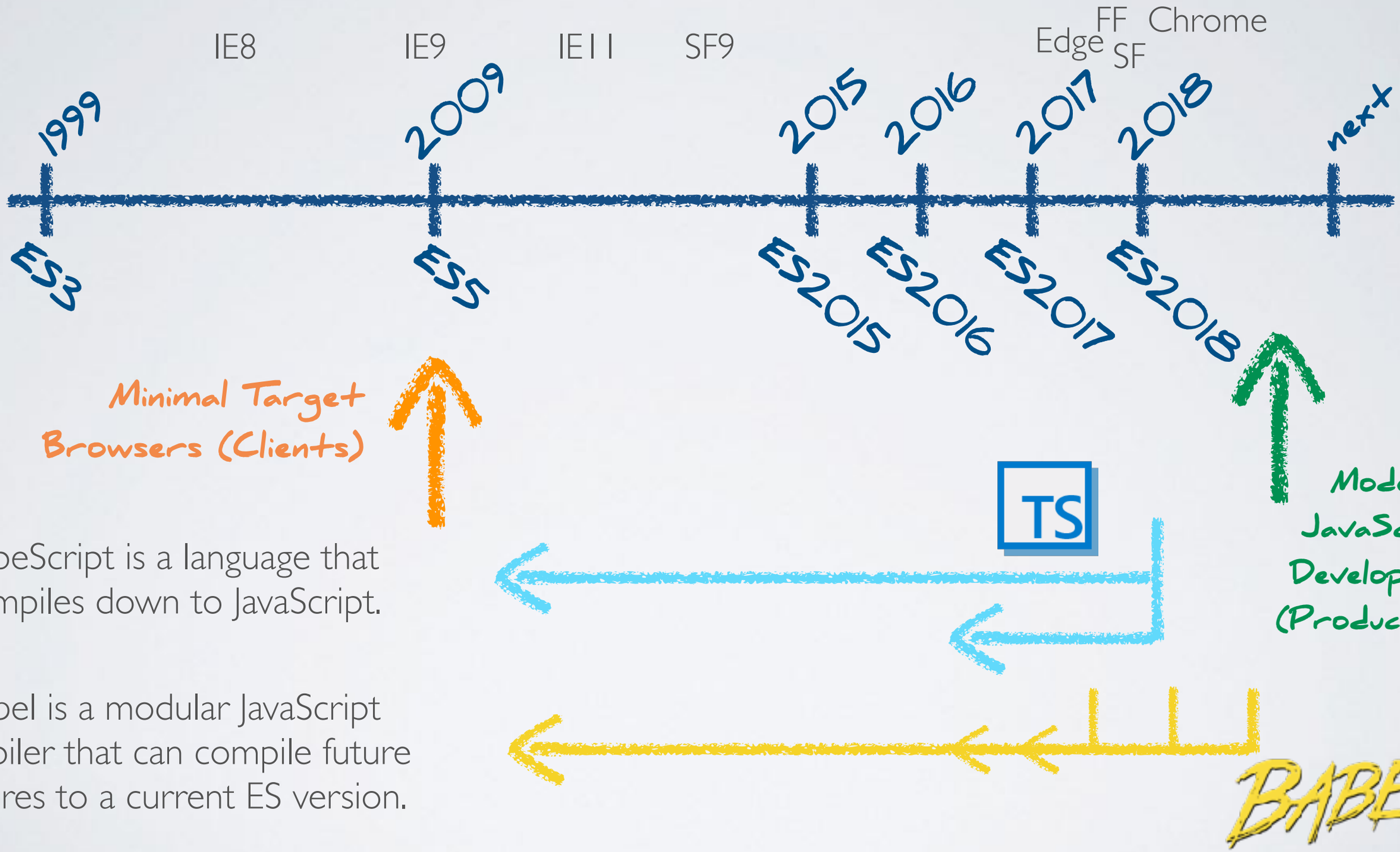
The Feature Gap

JavaScript has evolved rapidly in the past few years.



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

Addressing the Feature Gap



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.

Note: Babel 7 supports the TypeScript syntax for transpilation but it does not perform any type checking.

Transpiler

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

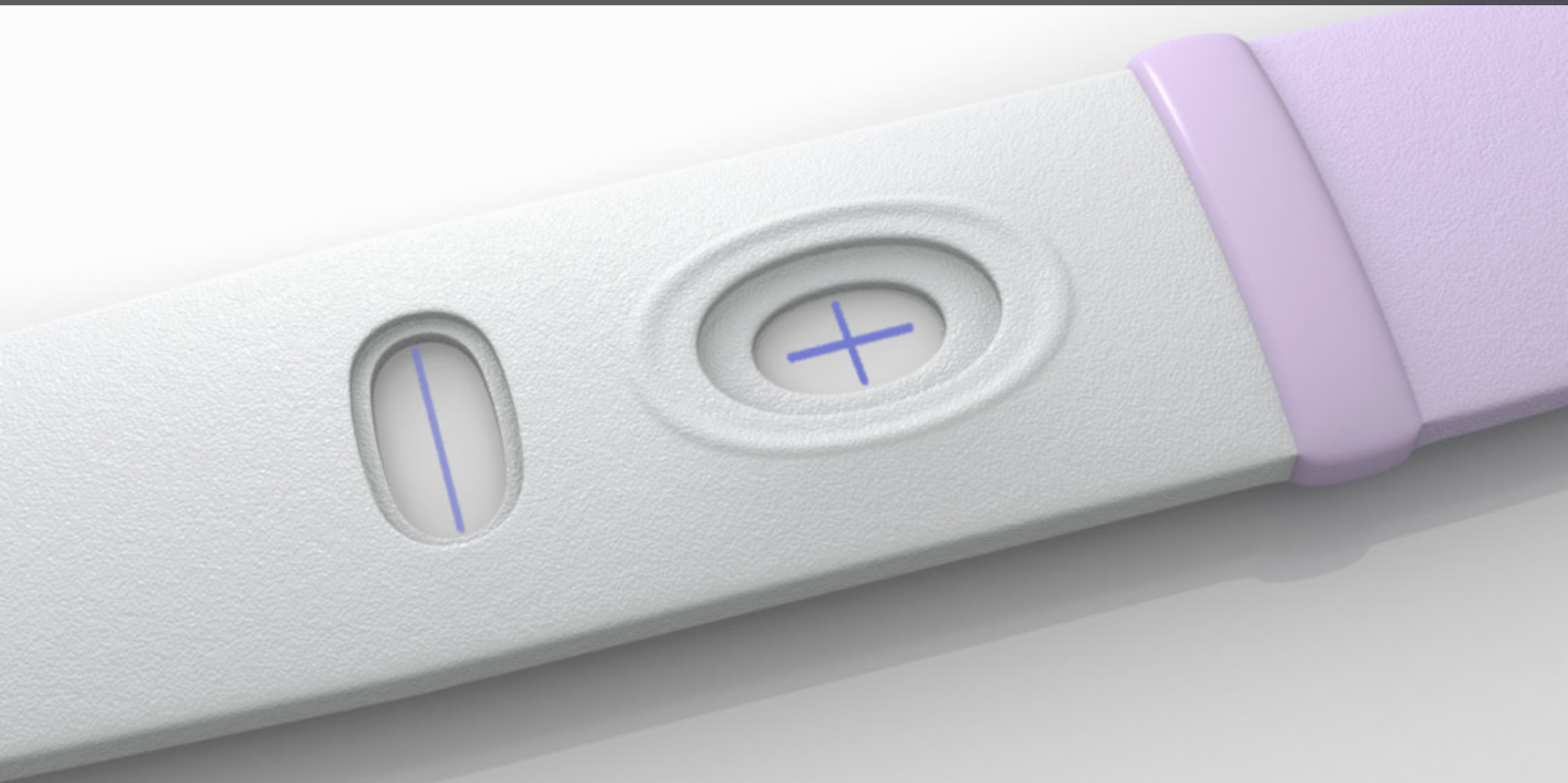
TypeScript language features and most ES2015+ features can be written in plain ES5.

```
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

Testing



JS Testing Scenarios

(where to run tests)

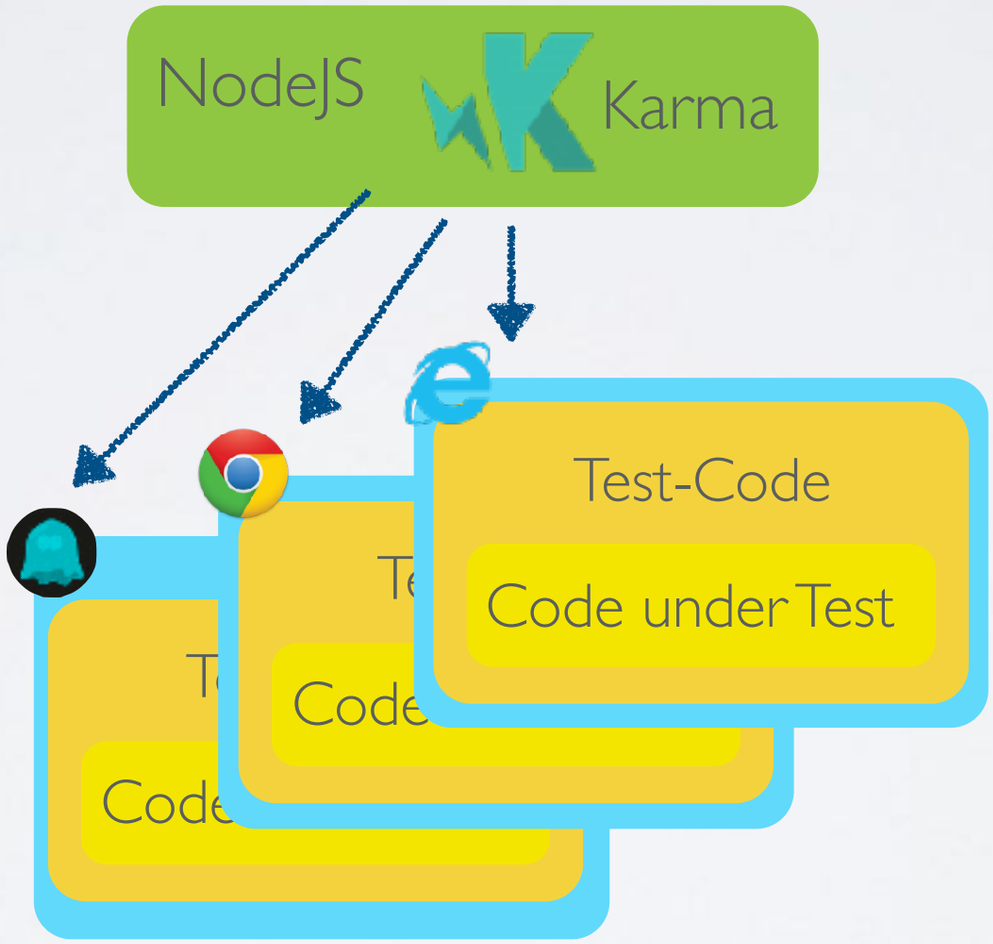
Pure Node-Testing

(unit-tests)

Run tests in node.
Optionaly use jsdom
to fake DOM.

Karma

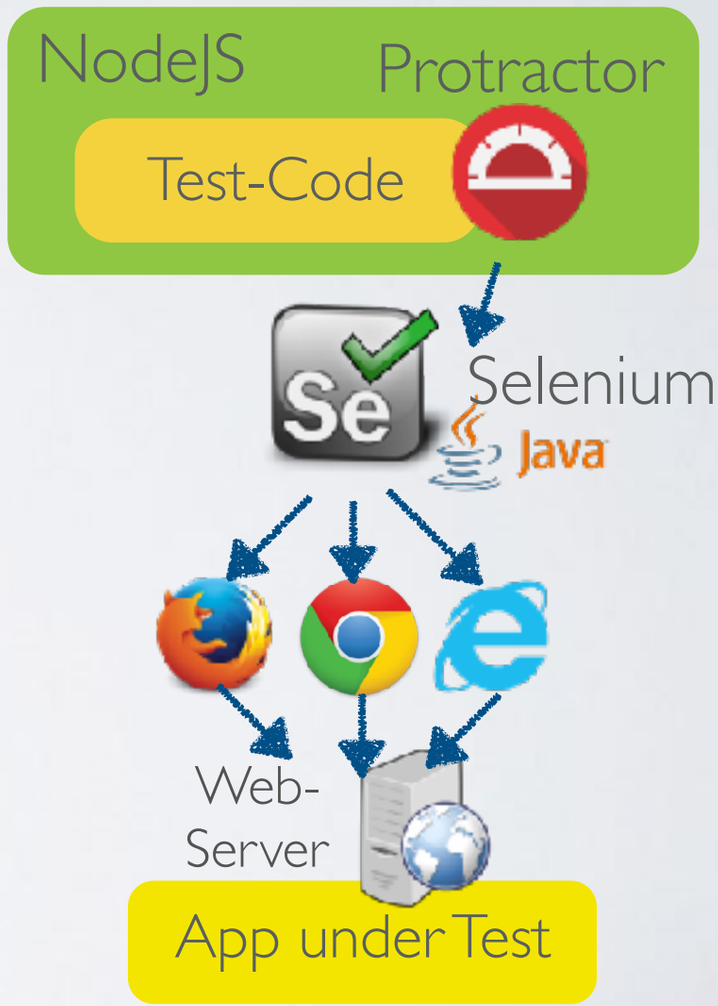
(unit-tests)



Run tests in browser.
Optionaly use PhantomJS as headless
browser (easy installation, speed).

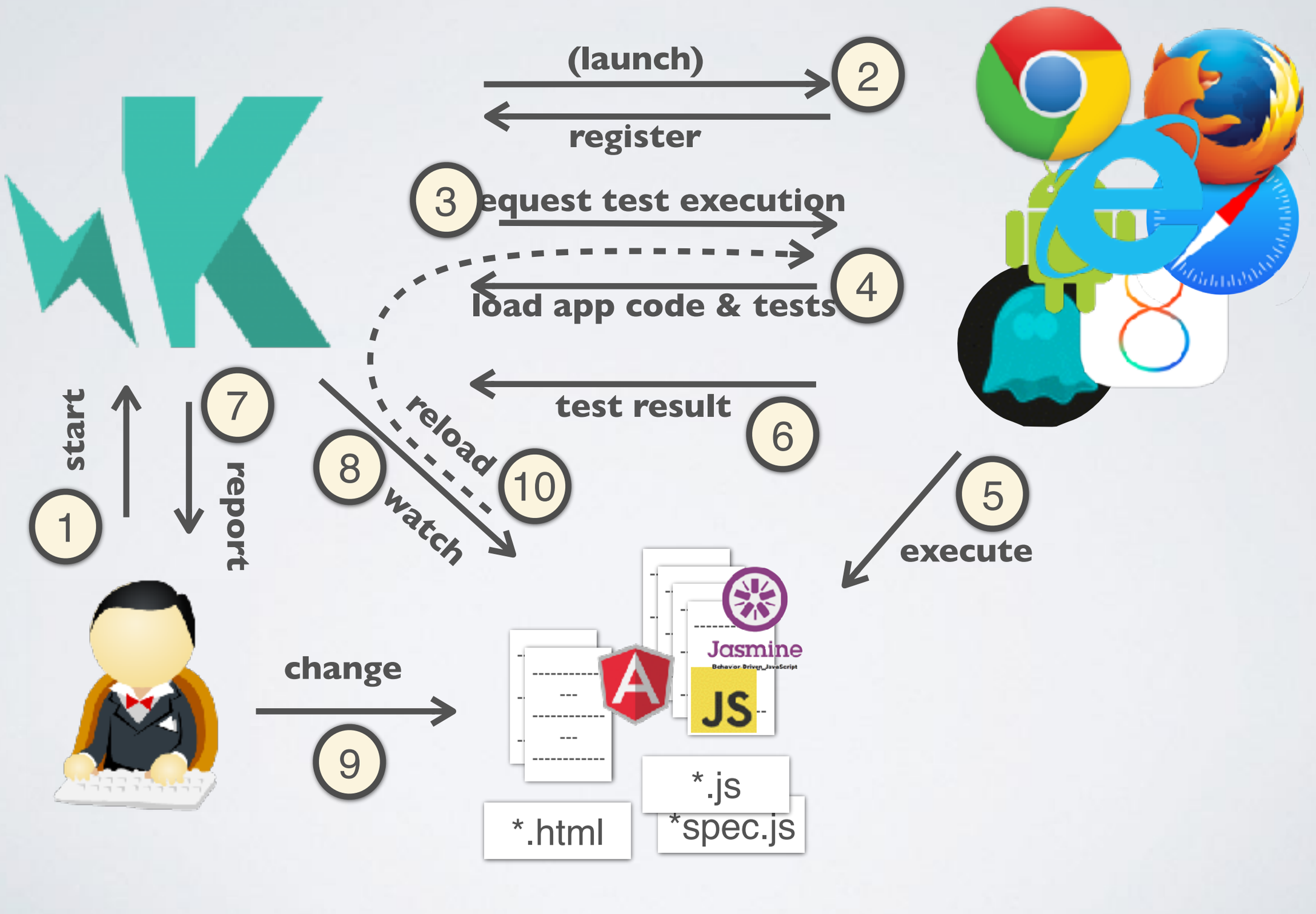
Protractor

(end-to-end tests, e2e)



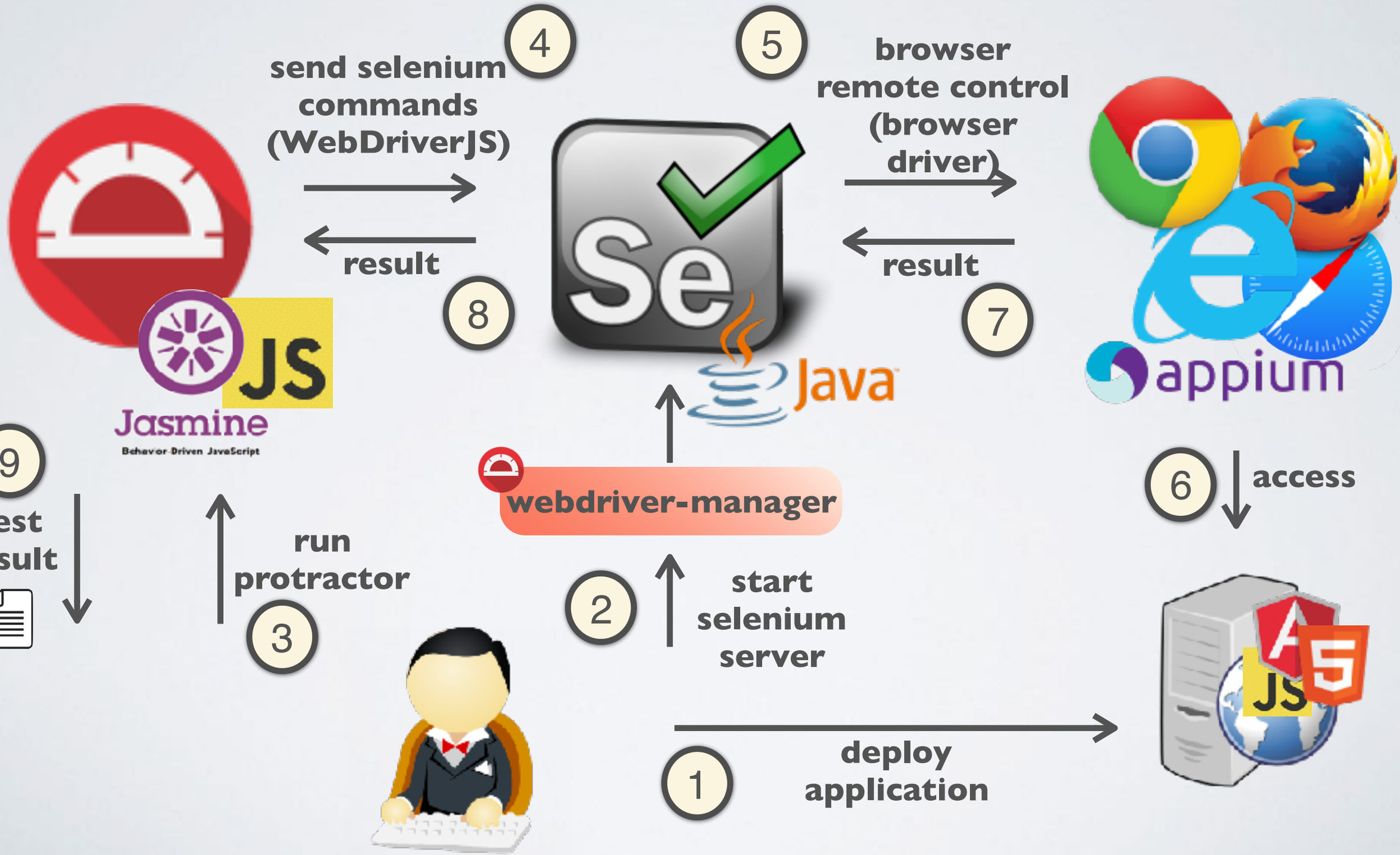
Script a browser to interact
with a deployed app.

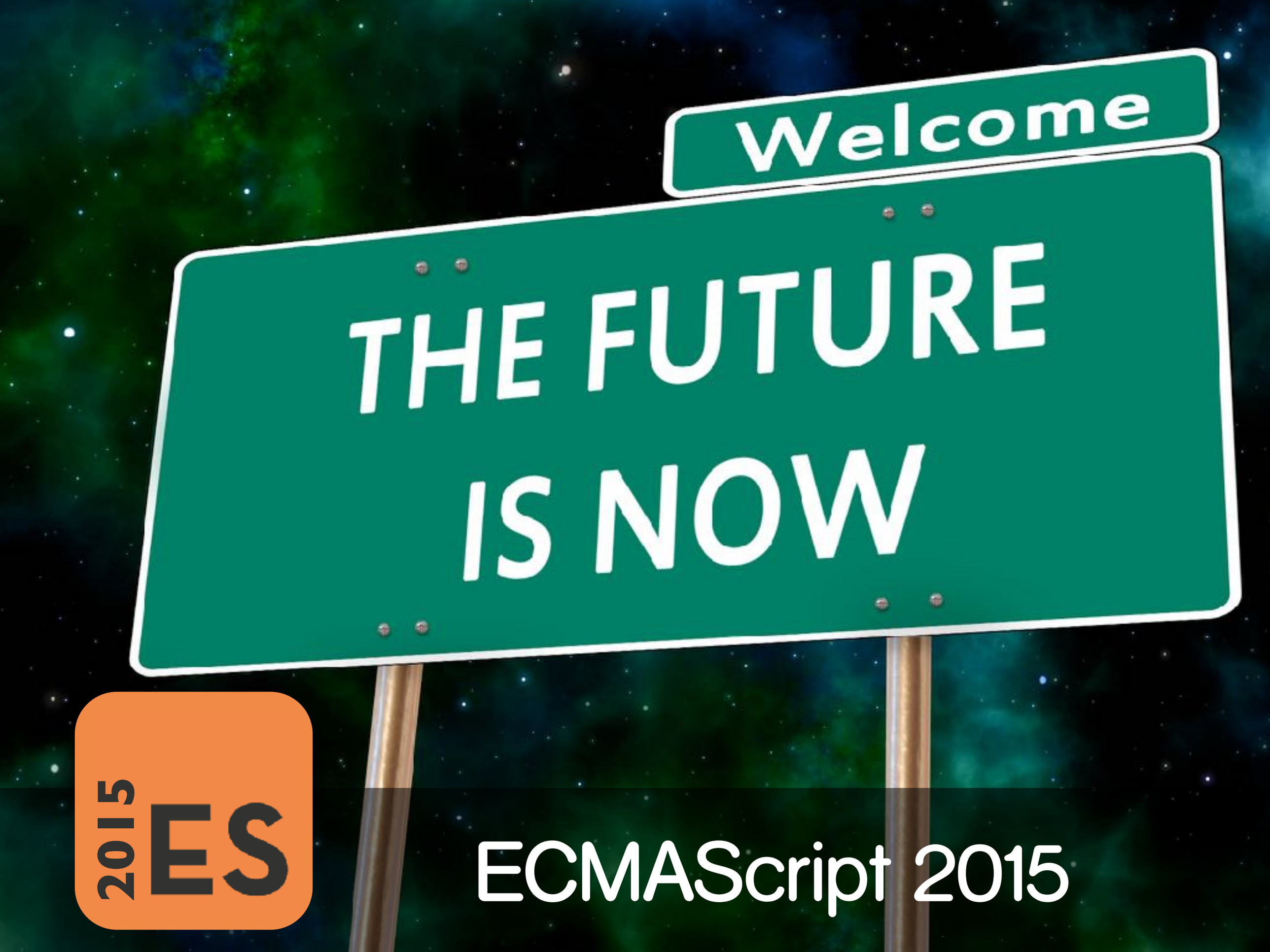
KARMA


(launch)
2
register
3
equest test execution
load app code & tests
4
test result
6
5
execute
reload
10
8
watch
7
report
start
1
change
9
Jasmine
Behavior Driven JavaScript
JS
*.js
*spec.js
*.html

Protractor

end to end testing for AngularJS




Welcome
THE FUTURE
IS NOW
2015
ES
ECMAScript 2015

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
- ECMAScript 2018: June 2018

lead to ActionScript 3

also called ES6

also called ES7

also called ES8

also called ES9

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

ES2015: Runtime & Syntax

New Language Features (Syntax)

New Built-In Functionality (Runtime Objects)

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.

```
function assign(){
  x = 43;
  if (true) {
    var x = 42;
  }
  var x;
  return x;
}
```

`var` creates a property on the global object (window) when declared on the top level scope.

Variables declared with `let` or `const` have block scope and "can't be accessed before their declaration".

Technically variables are still hoisted but not initialized and can't be accessed until their lexical binding is evaluated (aka "temporal dead zone").

```
function assign(){
  let x = 42;
  if (true) {
    let x = 43;
  }
  return x;
}
```

`let` and `const` do not create a property on the global object (window).

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

Hoisting vs. Initialization

`let` bindings are created at the top of the (block) scope containing the declaration, commonly referred to as "hoisting".

Unlike variables declared with `var`, which will start with the value undefined, let variables are *not initialized* until their definition is evaluated.

Accessing the variable before the initialization results in a ReferenceError. The variable is in a "*temporal dead zone*" from the start of the block until the initialization is processed.

```
let foo = () => bar;
let bar = 'bar';
foo();

console.log(foo()); // -> bar
```

hoisting

```
let x = 10;
if (true) {
    console.log(x); // -> ReferenceError
    let x = 11;
}
```

temporal dead zone

Variable Declarations with `let` and `const`

- A variable with a given name can only be declared once with `let` and `const`
- `const` must be assigned on declaration and can't be changed later
- `let` should be preferred for for-loops:

```
for (let i = 0; i < max; i += 1){…};
for (let e of elements){…};
```

(block scope & automatic closure)

- Try to use `const` over `let`

`for-of` Loop

The for-in loop iterates over the indexes of an object.

ES5 JS

```
var names =
          ['John', 'Jane', ‘Doe'];
for (var name in names) {
    console.log(name);
}
// -> 0, 1, 2
```

The for-of loop iterates over the elements of an iterable.

2015 ES

```
let names =
          ['John', 'Jane', ‘Doe'];
for (let name of names) {
    console.log(name);
}
// -> 'John', 'Jane', ‘Doe'
```

(iterables are a “standardized interface” in ES2015, many built-ins are iterables and you can implement your own iterable objects)

https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Iterators_and_Generators

Iterators

In ES2035 the iteration protocol is standardized.
Many built in constructs are iterable.
Custom iterators can be implemented using `Symbol.iterator`.

```
const a = [1,2,3,4,5];

const iter = a[Symbol.iterator]();

console.log(iter.next());
// ->  {value: 1, done: false}
```

- The `for...of` loop works with iterables.
- The spread operator (`...args`) works with iterables.
- *Generators* can be used to create custom iterators.

```
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);
}
```

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

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


```
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: '!!'};
}
```

```
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

Default Parameters

```
function greet(message, name = 'Joe'){
    console.log(message + ', ' + name);
}

greet('Hello'); // -> Hello, Joe
greet('Goodbye', 'Jeff'); // -> Goodbye, Jeff
```

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 values 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);
```

Multiple return values:

```
function useElement(){
  let element = 42;
  const getElement = () => element;
  const setElement = e => element = e;
  return [getElement, setElement];
}
const [getValue, setValue] = useElement();
```

Nested destructuring:

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

Object Literal Enhancements

```
var firstName = 'Jonas';
var lastName = 'Bandi';
var greet = function(){
        console.log('Hello!');}

var o1 = {
    firstName: firstName,
    lastName: lastName,
    greet: greet
};
```

ES5 JS

```
const firstName = 'Jonas';
const lastName = 'Bandi';
const greet = function(){
        console.log('Hello!');}
const propName = 'myprop';

var o1 = {
    firstName,
    lastName,
    greet,
    say(){console.log(‘Hi!’)};
    [propName]: 'gugus'
};
```

2015 ES

Example for Dynamic Keys

```
function isObject(item) {
  return (item && typeof item === 'object' && !Array.isArray(item));
}

function mergeDeep(target, ...sources) {
  if (!sources.length) return target;
  const source = sources.shift();

  if (isObject(target) && isObject(source)) {
    for (const key in source) {
      if (isObject(source[key])) {
        if (!target[key]) Object.assign(target, { [key]: {} });
        mergeDeep(target[key], source[key]);
      } else {
        Object.assign(target, { [key]: source[key] });
      }
    }
  }
  return mergeDeep(target, ...sources);
}
```

http://stackoverflow.com/questions/38345937/object-assign-vs-extend

Classes

Syntactic Sugar for constructor functions and prototypes

```
class Person {
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    fullName() { return this.firstName + ' ' + this.lastName};
}

const pers = new Person('John', 'Doe');
console.log(pers.fullName());
console.log(typeof Person) // -> function
```

```
// De-sugared:
function Person(firstName, lastName){
       this.firstName = firstName;
       this.lastName = lastName;
}
Person.prototype.fullName = function() { return this.firstName
                                        + ' ' + this.lastName};
```

There is no class *type* at runtime: the constructors defines a constructor function and methods get added to it's prototype.

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

Classes: Inheritance / Setter & Getter

```
class Employee extends Person{
    constructor(employer, firstName, lastName){
        super(firstName, lastName);
        this._employer = employer;
    }

    get employer() {
        return this._employer;
    }

    set employer(val) {
        this._employer = val.toUpperCase();
    }

     getDescription(){
        return super.getDescription() + this.employer;
     }
}
```

```
let emp = new Employee('Fight Club', 'Tyler', 'Durden');
emp.employer = 'Project Mayhem';
console.log(emp.emplyoyer);
console.log(emp.getDescription());
```

In ES5 you could already define properties (setters/getters) with `Object.defineProperty`, but it was much more clumsy …

Classes are "Syntactic Sugar"

```
const empl = new Employee('Fight Club', 'Tyler', 'Durden');

console.log(typeof empl); // object
console.log(typeof Employe); // function
console.log(empl.constructor);
console.log(empl.__proto__);
console.log(empl.__proto__.__proto__);
```

```
const empl = new Person('Tyler', 'Durden');

setTimeout(empl.fullName, 10); // this is undefined!
```

Differences between classes & traditional constructor functions

All constructors are functions, but not all functions are constructors.

Classes *enforce strict mode*, i.e. constructor, static and prototype methods, getter and setter functions are executed in strict mode.

```
class Person {
    getMe(){
        return this;
    }
    triggerError(){
        leakingVar = 42;
        return 43;
    }
}

const p = new Person();
const getThePerson = p.getMe;
console.log(getThePerson()); // undefined
console.log(p.triggerError()); // ReferenceError
```

- Classes can't be called only instantiated with **`new`**
- classes have a **`super`** binding
- the **`prototype`** property is not writable in classes
- The prototype (**`__proto__`**) of a class is it's super constructor when there is an extends clause

https://stackoverflow.com/questions/46820121/in-javascript-what-are-the-differences-between-a-class-and-a-constructor/46821201#46821201

Generators

A generator is a special type of function that creates an iterator. It can **`yield`** the control flow and maintain it's state.

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

var iter = idMaker();
console.log(iter.next().value); // 0
console.log(iter.next().value); // 1
```

https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Iterators_and_Generators

Promises

Promises are a way to model asynchronous operations.
A promise is an object that represents the completion of an asynchronous operation and the resulting value.

Many different libraries implement promises for ES5.
(i.e. bluebird, Q, AngularJS, jQuery v3, ...)

https://promisesaplus.com/implementations

`Promise` is a built in object in ES2015.

```
const p = new Promise((resolve, reject) => {
   setTimeout(() => resolve('result), 1000);
});
p.then((value) => console.log(value));
p.catch((error) => console.log('Oh, crap!));
```

There are several polyfills for ES2015 promises.
i.e: https://github.com/taylorhakes/promise-polyfill

Set & Map

Efficient data structures for common algorithms.

In ES5 objects could be used as "has maps" (key-value stores).

`Set`, `Map`, `WeakSet`, `WeakMap` are built in objects in ES2015.

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

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

Set, Map:

- values (and keys) can be primitive types and objects.
- are iterable

WeakSet, WeakMap:

- values (and keys) can objects.
- are not iterable
- values are garbage collected if not referenced from somewhere else

Symbol

`symbol` is a new primitive data type in ES2015.

A JavaScript runtime has built in ("well-known") symbols.

The `Symbol()` function creates a new value of type symbol.

Built in symbols expose language behaviors:

```
const a = [1,2,3,4,5];

const iter = a[Symbol.iterator]();

console.log(iter.next());
// ->  {value: 1, done: false}
```

New symbols can be used to create unique values:

```
Symbol('foo') === Symbol('foo'); // false
```

```
const key = Symbol("hidden");

const obj = {
  name: 'Jonas',
  [key]: 'Not visible!'
};

console.log(obj.hidden); // undefined
console.log(obj[key]); // "Not visible!"
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol

Proxies

Proxies can wrap an object and "trap" (intercept) interactions with the wrapped object.

```
var target = {};
var handler = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(target, handler);
console.log(p.world); // Hello, world!
```

```
var target = function () {
  return "Hello, world!"
};
var handler = {
  apply: function (receiver, ...args) {
    return "Proxy: " + receiver();
  }
};

var p = new Proxy(target, handler);
console.log(p('world')); // Proxy: Hello, world!
```

Note: Proxies can not be transpiled nor polyfilled!

Reflect API

API exposing the runtime-level meta-operations on objects.

ES5 exposes **`Object`** methods like:

`defineProperty()`, `getOwnPropertyDescriptors()` ...

`Reflect` is a built in objects in ES2015, that exposes similar methods:

```
const o = {a: 42, b: '43', [Symbol('c')]: 44}
console.log(Reflect.ownKeys(o));; // ['a', 'b', Symbol(c)]

function C(a, b){
  this.c = a + b;
}
var instance = Reflect.construct(C, [20, 22]);
console.log(instance.c);; // 42
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect/Comparing_Reflect_and_Object_methods

EcmaScript Modules

Classes & Packages in Java

Split code into several files

Isolate units of code & expose API

Declare dependencies

development


```
import javax.persistence.EntityManager;
import org.mycomp.domain.MyEntity;
...
EntityManager em = ...
MyEntity myEntity = new MyEntity();
myEntity.setName("Test");
...
em.persist(myEntity);
```

build

compilation

resolve dependencies

runtime

deployment provides dependencies

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(){…};
```

`program.js`

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

ES5 JS

ES2015 introduces modules to declare dependencies explicitly.

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

2015 ES

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

doSomething();
```

https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/

ES2015 Modules Semantics

- 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 defaultExport
       from './library5.js';
```

Modules at Runtime

Module loading is not part of the ECMA specification!
It took a while until browser supported modules natively.

Today all modern browsers support *static* module imports:
https://caniuse.com/#feat=es6-module

```
<script type="module" src="app.js"></script>
```

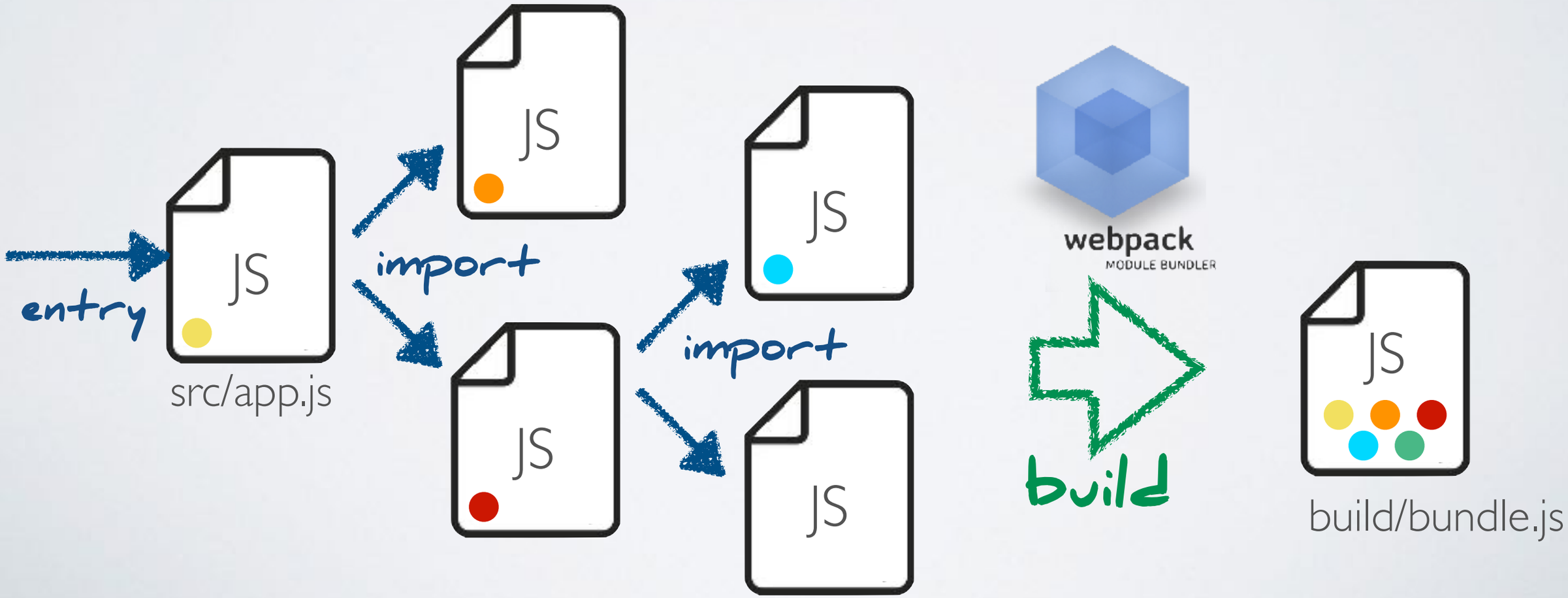
For better performance non-trivial web apps should use a bundler and not native module loading (even with http/2).
https://developers.google.com/web/fundamentals/primers/modules

Ecma script modules are not yet supported in Node.js!

Modules at Build Time

The primary use-case for modules today is build-tooling.
Modern *bundlers* (WebPack, Rollup, Parcel ...) work with modules:

- build a dependency tree based on fine grained ecma script modules (import/export)
- create coarse grained JavaScript bundles which are optimized to be loaded by a browser.



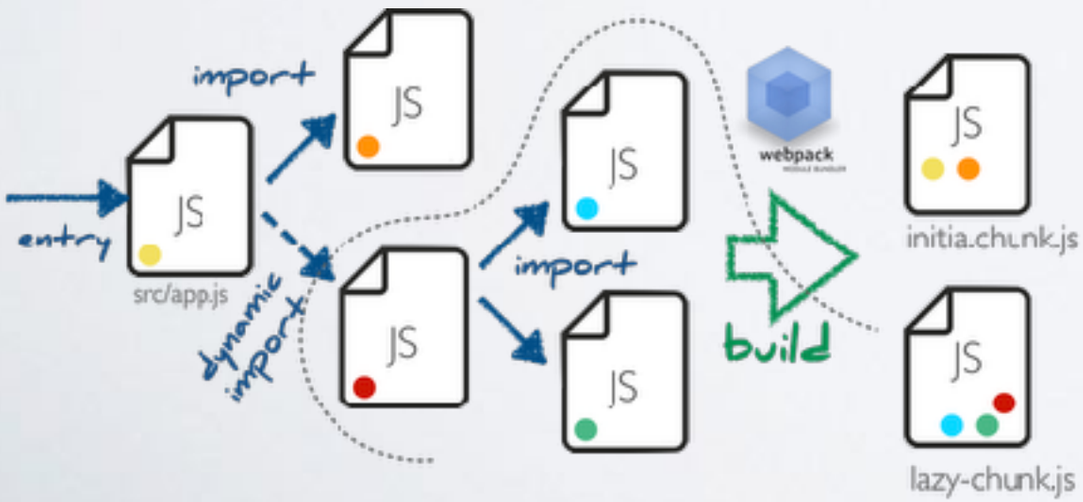
Dynamic Module Loading

Dynamic module loading is a ECMA proposal:
https://github.com/tc39/proposal-dynamic-import

(currently only supported natively in Chrome & Safari: https://caniuse.com/#feat=es6-module-dynamic-import)

```
import('./second.js')
  .then(m  => m.sayHello());
```

Dynamic imports are supported by modern bundlers (webpack, rollup, parcel): Based on `import` a seperate JavaScript bundle is created at build time, that can be lazyly-loaded by the browser when needed. The goal is a small initial bundle. This is also referred to as *code-splitting*, *lazy-loading* or *chunks*.



https://webpack.js.org/guides/code-splitting/
https://parceljs.org/code_splitting.html
https://rollupjs.org/guide/en#experimental-code-splitting

Node.js: CommonJS

CommonJS is the module system of Node.js.
CommonJS does not work natively in the browser.
A future version of Node will support ES2015 modules (esm).

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer(function(req, res){
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, function(){
  console.log('Running at http://'+ hostname + ‘:' + port);
});
```

Node.js: CommonJS

app.js

```
var calculator = require('./lib/calculator');

var input = [1,2,3,4];
var output = calculator.doComplicatedCalculation(input);

console.log('Output: ' + output);
```

import

export

```
function doComplicatedCalculation(input){
    var output = [];
    for (var i = 0; i < input.length; i++){
        var val = input[i];
        output.push(i*i, i*i*i);
    }
    return output;
}
exports.doComplicatedCalculation = doComplicatedCalculation;
```

lib/calculator.js

ES Beyond ECMAScript 2015

ECMAScript Specification Process

https://github.com/tc39/ecma262/

- Stage 0: Strawman Proposals
- Active Proposals
  Stage 1-3 (proposal, draft, candidate)
- Finished Proposals
  (two browser vendors must implement the feature)
- Finished Proposals will be included in the next (yearly) revision of the language specification.

http://www.2ality.com/2015/11/tc39-process.html

ES2016
(released in June 2016)

- Array.prototype.includes:

```
let months = ['June', 'July', 'August'];
months.includes('June');    // => true
months.includes('January'); // => false
```

```
let items = ['1', 5, 8];
items.includes(1); // => false
items.includes(8); // => true
```

- Exponential operator:

```
2 ** 3;    // => 8
5 ** 2;    // => 25
```

```
let num = 2;
num **=4;
num;   // => 16
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes
https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators
https://rainsoft.io/must-know-details-about-es2016-features/

ES 2017

(released in June 2017)

- async functions
- Shared memory and atomics
- Object.values
  Object.entries
  Object.getOwnPropertyDescriptors
- String padding
- Trailing commas in function parameter lists and calls

ES2017: async functions

```
async function main(){
 var result = await doItAsync();
 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

ES 2018

(released in June 2018)

- Asynchronous Iteration
- Object Rest/Spread
- Promise.prototype.finally
- RegExp features
- Template Literal Revision

https://medium.com/front-end-weekly/javascript-whats-new-in-ecmascript-2018-es2018-17ede97f36d5

Asynchronous Iteration

`for-await-of` and async generators

```
const peopleUrls = [
  'https://swapi.co/api/people/1/',
  'https://swapi.co/api/people/2/'
];

async function* createAsyncIterable() {
  for (const url of peopleUrls) {
    yield axios.get(url);
  }
}
```

```
async function run() {
  const asyncIterable = createAsyncIterable()
  const asyncIterator =
            asyncIterable[Symbol.asyncIterator]();

  for await (const response of asyncIterator) {
    console.log(response);
  }
}
run();
```

Available in Chrome, Firefox & Safari. Not yet available in Edge.
Supported in Babel & TypeScript.

ES 2018: Object Rest/Spread Properties

```
const sourceObj1 = {a: 1, b:2, c:3};
const sourceObj2 = {c: 5, d:6, e:7};
const mergedObject = {...sourceObj1, ...sourceObj2, f:9};
console.log(mergedObject);
```

Object rest/spread properties allows elegant copying & merging of objects

Available in Chrome, Firefox & Safari.
Not yet available in Edge.
Supported in Babel & TypeScript.

ES 2019

(planned released in June 2019)

- Array.prototype.{flat,flatMap}
- Object.fromEntries
- String.prototype.{trimStart,trimEnd}
- Symbol.prototype.description
- Optional catch binding
- ...

http://2ality.com/2018/02/ecmascript-2019.html

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 for class fields is currently at stage 3.
Implemented in Chrome > 72. Currently not implemented in other browsers.
Class fields are supported in TypeScript and Babel.

https://github.com/tc39/proposal-class-fields
https://codingitwrong.com/2018/05/26/javascript-private-fields-and-object-oriented-design.html

ESnext: Private Fields

```
class Car {
  #milesDriven = 0
  drive(distance) {
    this.#milesDriven += distance
  }
  getMilesDriven() {
    return this.#milesDriven
  }
}

const tesla = new Car()
tesla.drive(10)
tesla.getMilesDriven() // 10
tesla.#milesDriven // Invalid
```

The specification for class fields is currently at stage 3.
Implemented in Chrome > 74. Currently not implemented in other browsers.
Private class fields are supported in Babel but not yet in TypeScript.

https://github.com/tc39/proposal-class-fields
https://jamie.build/javascripts-new-private-class-fields.html
https://codingitwrong.com/2018/05/26/javascript-private-fields-and-object-oriented-design.html
https://github.com/Microsoft/TypeScript/issues/9950

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 class fields is currently at stage 2.
Browsers do not yet supports decorators.
Decorators are supported in TypeScript and Babel (v7.1.0).

https://github.com/tc39/proposal-decorators

ES2015 browser support:

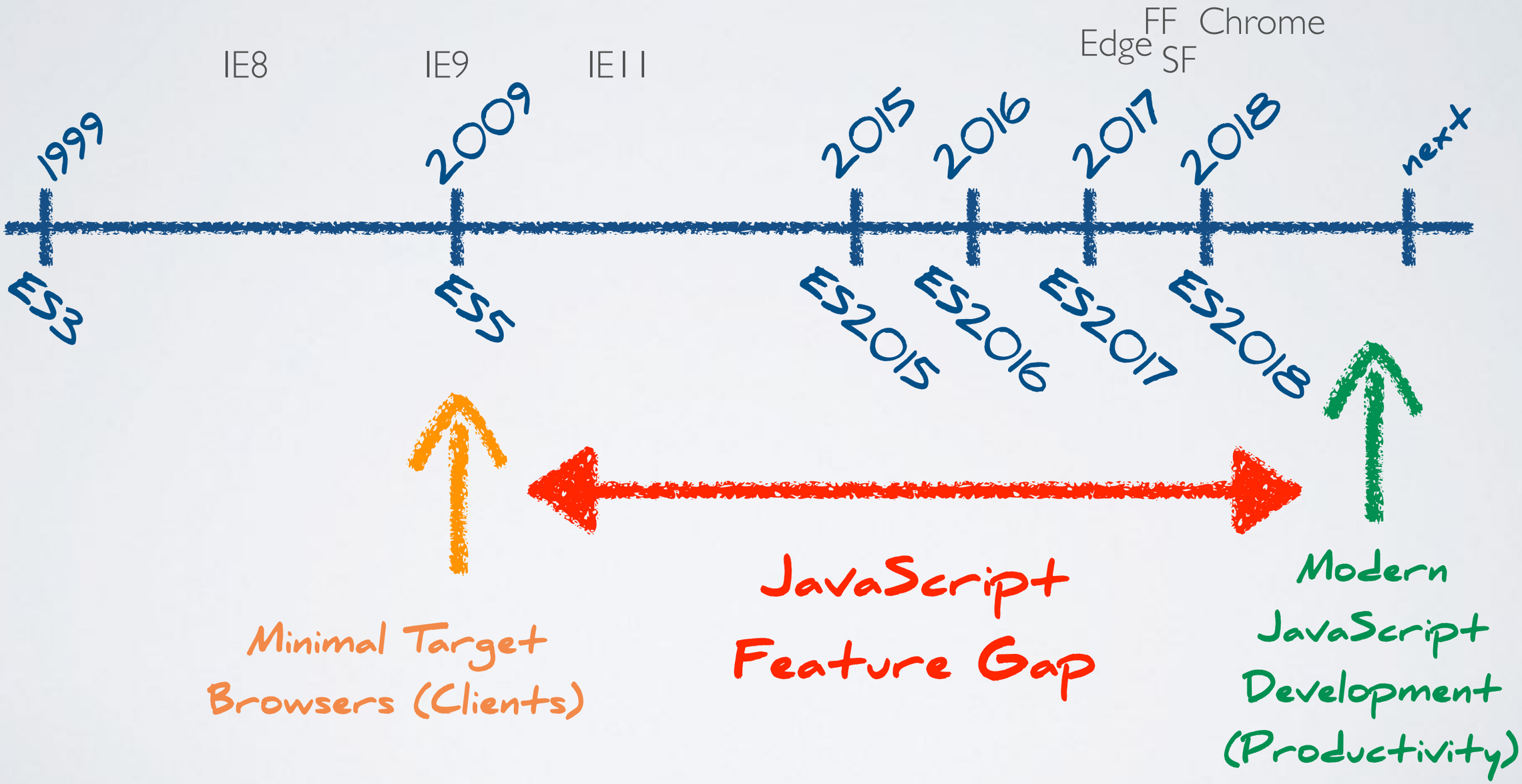
Desktop browsers

IE 11	Edge 17	Edge 18	FF 60 ESR	FF 66	CH 72, OP 59	CH 73, OP 60	SF 12.1
11%	96%	96%	98%	98%	98%	98%	99%

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

The Feature Gap

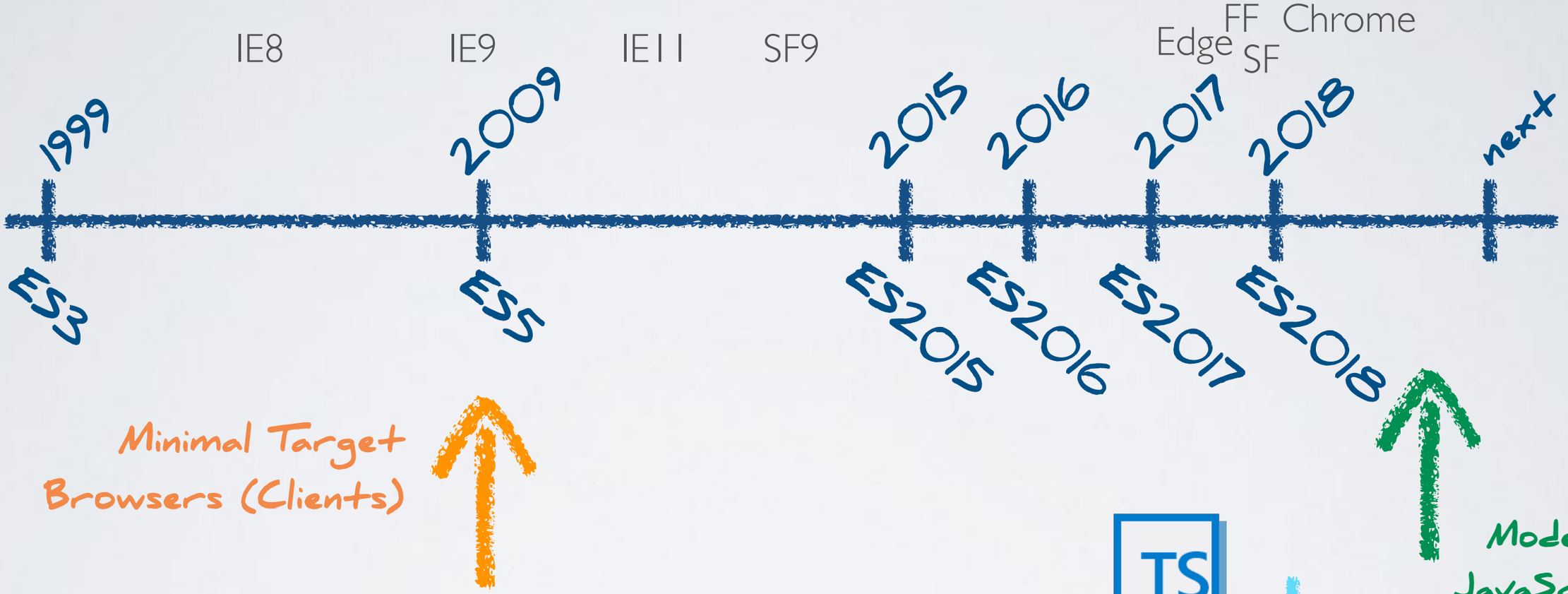
JavaScript has evolved rapidly in the past few years.



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

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

ES2015 Today: Transpiler & Polyfills

New Language Features (Syntax)

transpiling to ES5 for older browsers

TypeScript

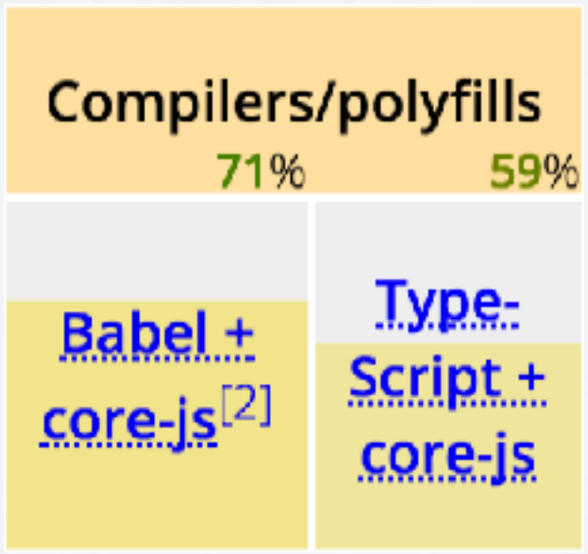
BABEL

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

TypeScript

JavaScript that scales!

TypeScript is a language for application-scale JavaScript development.

http://www.typescriptlang.org/

JavaScript was never engineered for large applications. It was intended for 100-1000 lines of codes.

Today we build apps with millions of lines of code. Very large JavaScript code bases tend to become “read-only”.

– Anders Hejlsberg, Build 2016

https://channel9.msdn.com/Events/Build/2016/B881

The goal of TypeScript

+

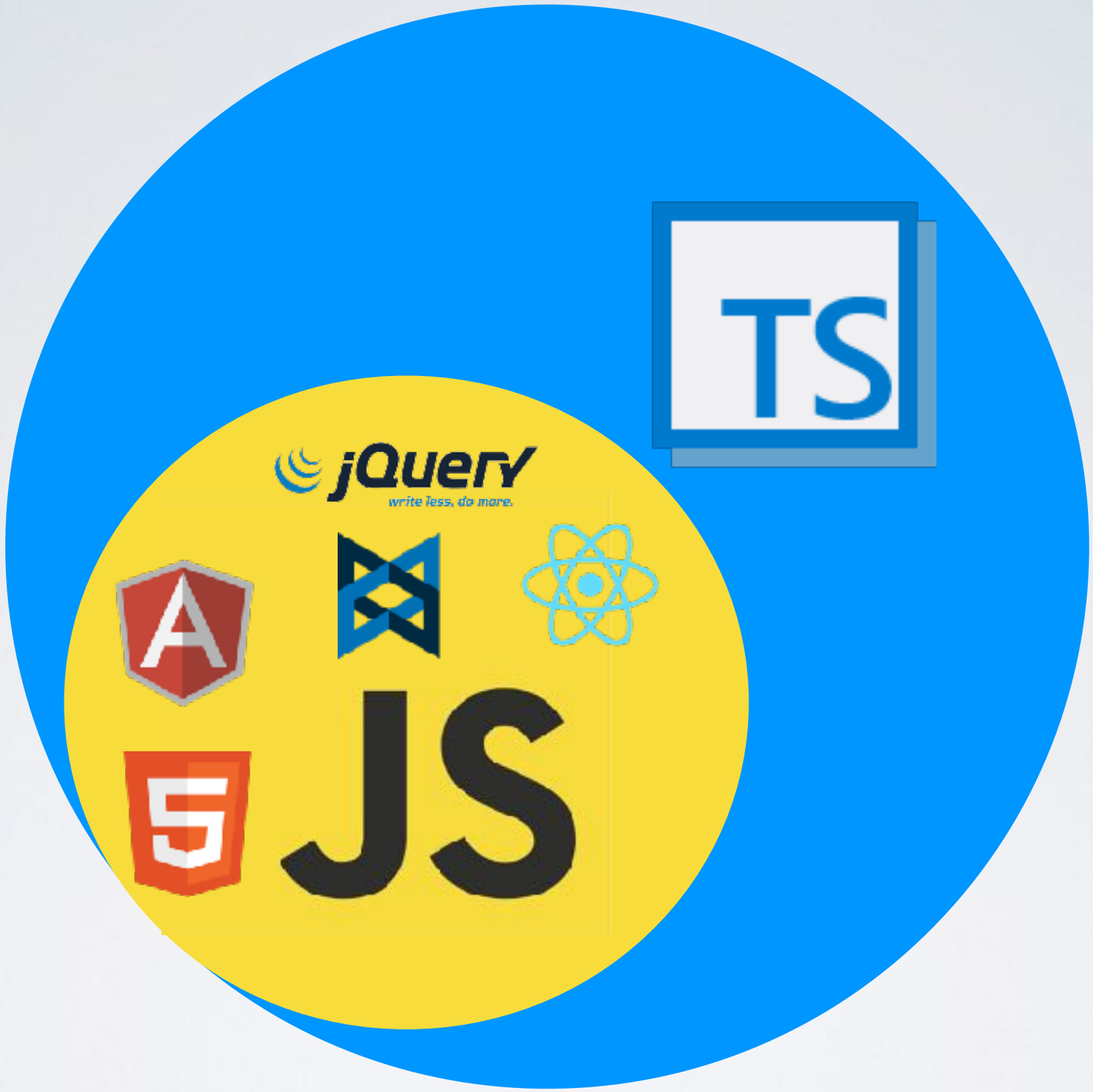
on top of ES5

new (future)
language features

modules, classes, ...

static types

enables tools



TypeScript is a superset of JavaScript

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

It starts with JavaScript.

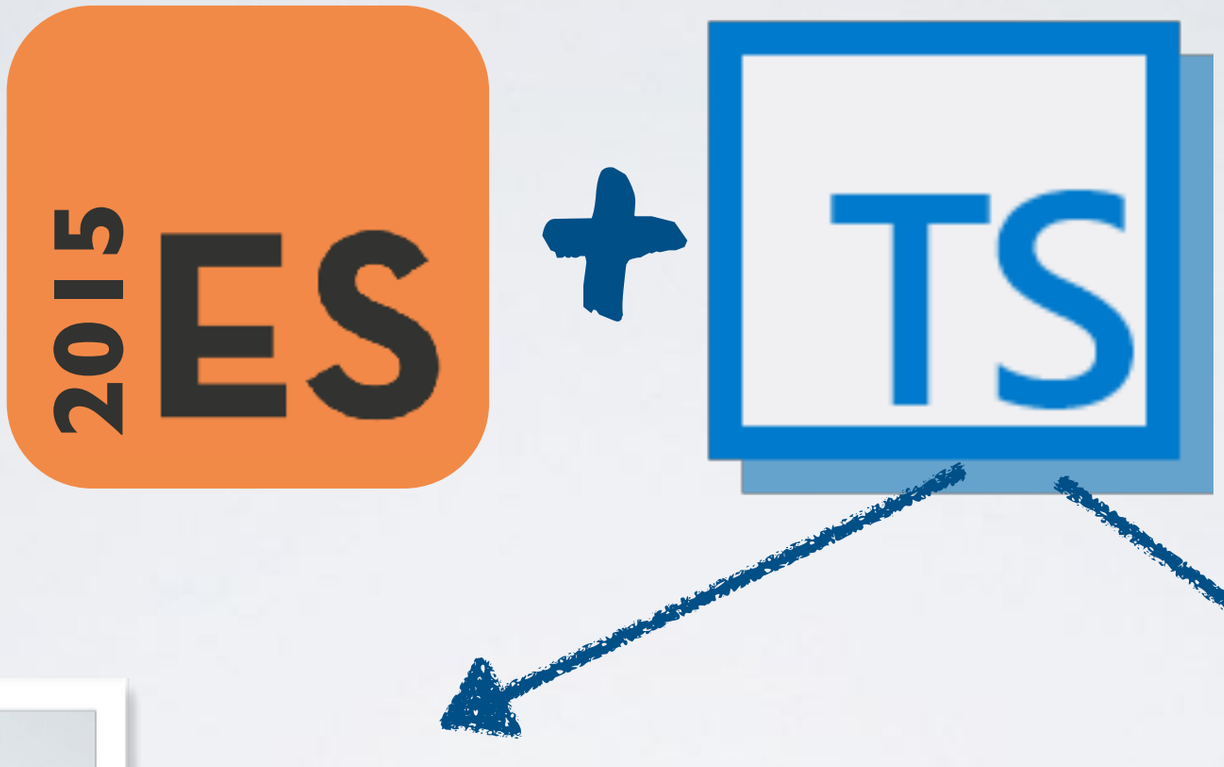
It ends with JavaScript.

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

TypeScript Today

- … is a Transpiler that provides ES2015 language features for older Browsers
- … is a Transpiler that provides some future ESnext language features (i.e. decorators)
- … is a compiler that adds an optional static type system on top of JavaScript.

TypeScript Today



new (future) language features

modules, classes, ...

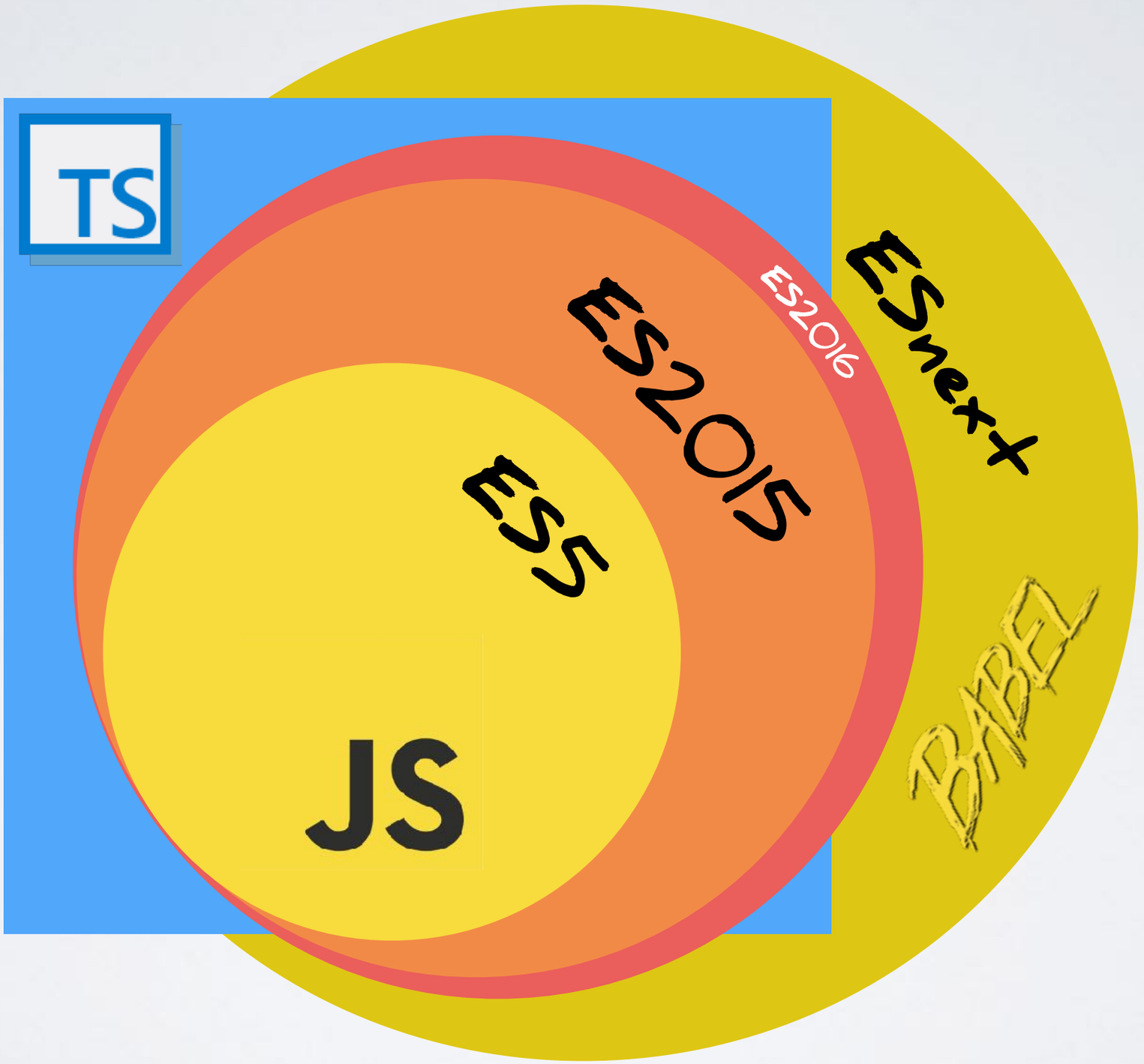
static types

enables tools

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

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

Superset ?



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

tsconfig setup

The TypeScript compiler is configured in `tsconfig.json`

```
{
 "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "baseUrl": "src",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2016",
      "dom"
    ]
  }
}
```

(standard angular-cli configuration)

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

Type System Features

- Zero Cost: Static types only during development time, no effect at runtime
- Type inference
- Gradual typing
- Configurable type checking

Using Basic Types

Types are an optional, opt-in feature in TypeScript.

Basic Types:

```
let b:boolean = true;
let n:number = 42;
let s:string = 'Hello!';
let a1:[number] = [1,2,3];
let a2:Array<string> = ['Hello','World', '!'];
enum ArrowDirection { Up, Down, Left, Right}
let e: ArrowDirection = ArrowDirection.Up;
```

Basic type checking:

```
let x:string = 'Hello World';
x = 42; // => compiler error
```

Type inference:

```
let x = 'Hello World'; // the type of x is 'number'
x = 42; // => compiler error
```

The compiler will infer types if possible. It's considered a 'good style' to omit type annotations if htey are not needed.

Advanced Types

Types are an optional, opt-in feature in TypeScript.

Union Types:

```
let pet: IBird | IFish  = {fly(){}, layEggs(){}};
pet = new Fish();
```

Intersection Types:

```
let flyingFish: IFish & IBird = {swim(){}, fly(){}, layEggs(){}};
flyingFish.swim();
```

Type Guards:

```
(pet as IFish).swim();
(<IFish>pet).swim();

if (pet instanceof Fish){
    pet.swim();
}
```

User defined type-guards:

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (<Fish>pet).swim !== undefined;
}
```

```
if (isFish(pet)) {
    pet.swim();
} else {
    pet.fly();
}
```

any Type

Types are optional. TypeScript allows you to to gradually opt-in and opt-out of type-checking during compilation.

```
let v: any = 42;
v = 'Hello!'; // anything can be assigned to v

let n:number = 43;
n = v; // v can be assigned to anything!!!
```

Variables have the implicit type **any** if TypeScript does not know it's type.

Typing can be enforced by setting **`noImplicitAny: true`** in **`tsconfig.json`**

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

Function Types

A function has a type defined by its signature

```
let operation: string = 'add';

let mathFunc: (x: number, y:number) => number;

function add(x: number, y:number) : number {
  return x + y;
}

const subtract = (x: number, y:number):number => x - y;

if (operation === 'add')
  mathFunc = add;
else
  mathFunc = subtract;

console.log(mathFunc(4,2))
```

Note: functions with fewer parameters are assignable to functions that take more parameters.

https://github.com/Microsoft/TypeScript/wiki/FAQ#why-are-functions-with-fewer-parameters-assignable-to-functions-that-take-more-parameters

Classes & Interfaces

With classes & interfaces we can create custom types in TypeScript

```
interface IPerson {
    name: string,
}

class Person implements IPerson {
    name: string = 'Tyler';
    private age: number = 42;

    getInfo() {
        console.log(this.name); // inspect `this`
        return this.name + this.age;
    }
}

const p1: IPerson = new Person();
const p2: Person = new Person();
```

Interfaces are a pure build-time construct. They are not present at runtime!

Classes

Differences to ES2015 classes:

- property declarations
- parameter properties
  (properties declared in constructor signature)
- optional type annotations
- access modifiers (private, public, protected)

The Structural Type System

The structure of an objects defines the compatiblity with a type, the object does not have to explicitly declaread as a type (aka. "Duck Typing")

```
interface IPerson {
    name: string,
    age: number
}

const p3: IPerson = {
    name: 'Tyler',
    age: 42
};
```

TypeScript Decorators & Emit Metadata

`tsconfig.json`

```
{
 "compilerOptions": {
   ...
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
   ...
  }
}
```

The TypeScript compiler generates code that provides metadata (type information) at runtime.

```
Reflect.getMetadata('design:paramtypes', Cat);
Reflect.getMetadata('design:paramtypes', Cat.prototype, 'meow')
```

This feature is used by the dependency injection mechanism of Angular.

Angular Components

Angular Components

Components are the main building-block of Angular.

Template + Class + Metadata

= Component

A Simple Component

```
import {Component} from 'angular2/core';

@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

New components can be generated with the Angular CLI:

```
ng generate component hello
```

Directives & Components

A *component* is a special kind of directive.
A *directive* is a declarative instruction, that is embedded into a template and has a special meaning for the framework.

Databinding

DOM
(Template)

Interpolation

{{propName}}

Property Binding

[property]="propName"

Event Binding

(event)="methodName()"

Two Way Binding

[(ngModel)]="propName"

(ngModel is provided by the FormsModule)

component

https://angular.io/guide/template-syntax#binding-syntax

Event Handling

template

```
Greetings: {{name}}
<div [innerText]="name"></div>
<input (input)="onNameChange($event)"/>
```

$event is a keyword in the angular templating syntax

event binding

angular expression, called when event is emitted

DOM event

```
@Component(...)
export class GreeterComponent {
  name = 'Tyler Durden';
  onNameChange(e: Event){
    this.name = (e.target as HTMLInputElement).value;
  }
}
```

component

https://angular.io/guide/user-input

Event Handling

template

```
Greetings: {{name}}
<div [innerText]="name"></div>
<input #nameInput (input)="onNameChange(nameInput.value)"/>
```

expression can use the template reference variable

declaring a template reference variable that points to the dom element

angular expression, called when event is emitted

string

```
@Component(...)
export class GreeterComponent {
  name = 'Tyler Durden';
  onNameChange(name: string){
    this.name = name;
  }
}
```

component

https://angular.io/guide/user-input

Write Your First Component

```
import {Component} from '@angular/core';

@Component({
    selector: 'greeter',
    template: `
<input [(ngModel)]="name"/>
Greetings: {{name}}
`
})
export class GreeterComponent {
  name;
  constructor() {
      this.name = 'John';
    }
}
```

1-way databinding

2-way databinding

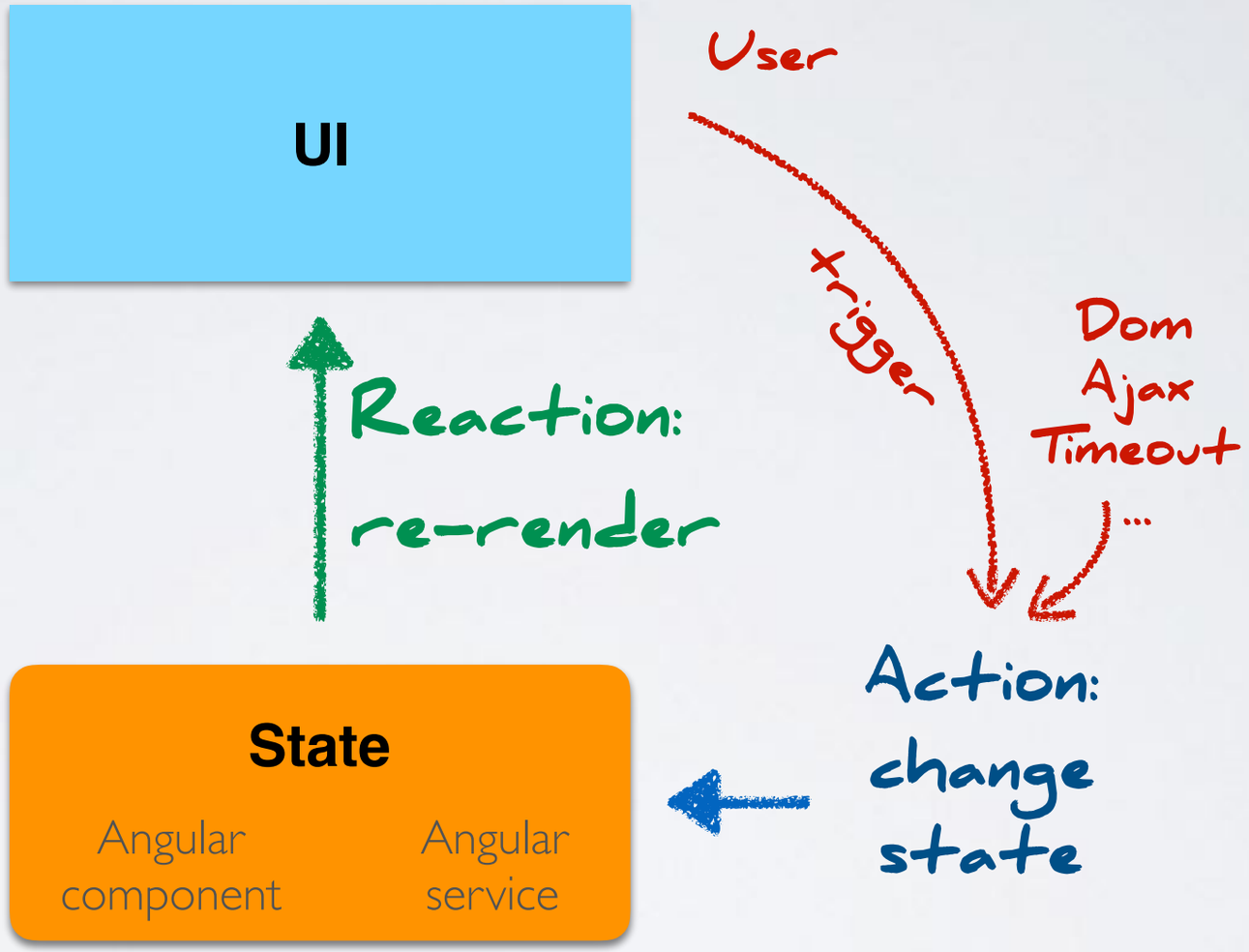
Note: **ngModel** is provided by the **FormsModule**

-> **FormsModule** must be imported in the Angular module (src/app/app.module.ts)

EXERCISES

Exercise 2 - Create your first component

State is Managed in JavaScript



Reactivity: Angular reacts on state changes and updates the UI.

Interpolation vs. Property Binding

Interpolation always provides a string:

```
<h3>
  {{title}}
  <img src="{{heroImageUrl}}" style="height:30px">
</h3>
```

Background: Html attributes can't be changed at runtime, DOM properties can be changed.

Angular converts interpolated "attributes" into a property bindings:

```
<input value="{{title + title2}}">
```

```
<input [value]="title + title2">
```

```
<div innerHTML="{{'<h1>Test2</h1>'}}"></div>
```

Note: There is no 'innerHTML' attribute in html just a innerHTML property in the DOM.

Property binding must be used to provide non-string values:

```
<button [disabled]="isDisabled">Try Me</button>
```

Special Element Bindings

CSS class binding (DOM: classList)

```
<p [class]="myClasses"></p>
```

```
<p [class.is-active]="isActive"></p>
```

an Angular expression, that should evaluate to a string

an Angular expression, that should evaluate to a boolean

`ngClass` directive:

```
<div [ngClass]="['bold-text', 'green']">array of classes</div>
<div [ngClass]="'italic-text blue'">string of classes</div>
<div [ngClass]="{'small-text': true, 'red': true}”>
   object of classes
</div>
```

`ngClass` is a directives and should be preferred when setting several classes.

https://angular.io/guide/template-syntax#class-binding
https://angular.io/guide/template-syntax#ngClass

Special Element Bindings

CSS style binding (DOM: style property)

```
<p [style.display]="!isActive ? 'none' : null"></p>
```

```
<div [ngStyle]="{'color': color, 'font-size': size>}"
style using ngStyle
</div>
```

`ngStyle` is a directives and should be preferred when setting several inline styles.

The style binding allows you to specify a unit extension:

```
<div [style.width.px]="pxWidth"></div>
<div [style.font-size.%]="percentageSize">...</div>
<div [style.height.vh]="vwWidth"></div>
```

Attribute binding (DOM: setAttribute)

```
<p [attr.role]="role"></p>
```

https://angular.io/guide/template-syntax#style-binding

Special Event Bindings

Angular provides "pseudo-events":

```
<input type="text" (keyup.enter)="handleEnter($event)">
```

```
<input type="text" (keyup.shift.enter)="handleShiftEnter($event)">
```

Unfortunately the supported pseudo events are not documented …?

https://angular.io/guide/user-input#key-event-filtering-with-keyenter

Testing Components

Tests are run via Karma: **`npm run test`**

For unit tests you want to keep the "code under test" small. Angular provides a **`TestBed`** to assemble a module which can only contain the minimal amount of code.

Drawback: The **`TestBed`** must be configured for each test.

When working with external templates, the assembling of the **`TestBed`** must be asynchronous.

The **`TestBed`** can create component fixtures which provide a reference to the component object and the associated DOM element.

https://angular.io/guide/testing

Client Side Routing

Client Side Routing in a SPA

The traditional web is built on the concept of linked documents (URL, bookmarking, back-button …)

In a SPA the document is just the shell for an application. When you navigate away from the document, the application is "stopped".

As a consequence a SPA should "emulate" the traditional user-experience on the web:

- navigate via urls, links & back-button
- bookmarks and deep links

Client-Side Routing in a SPA

http://thedomain:80/app#/first

Browser

Server

app bootstrapping

html-shell is loaded

"navigate" to "first" screen

load data & render screen with JS

http://thedomain:80/app#/**first**

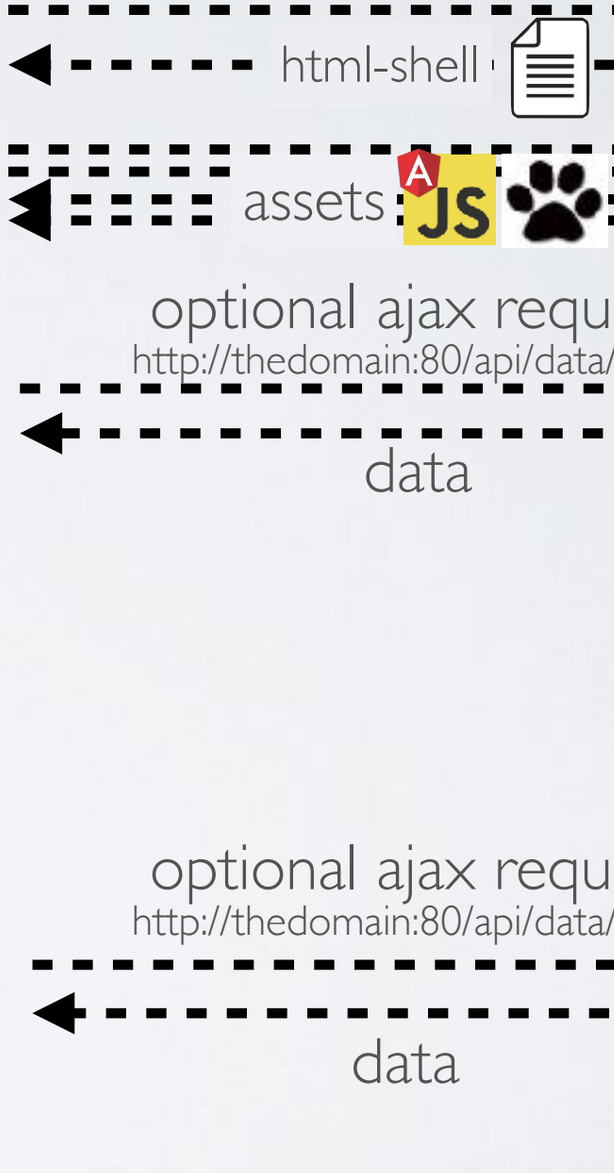
navigation

"navigate" to "second" screen

load data & render screen with JS

http://thedomain:80/app#/**second**

initial request
http://thedomain:80/app



optional ajax request
http://thedomain:80/api/data/first

data

optional ajax request
http://thedomain:80/api/data/second

data

Client-Side Routing

Single Page Applications can contain several "pages"/views.

Routing maps URLs to views. aka: “Deep-Linking”

Parts of the URL are evaluated on the client.

Traditional approach: "Hash-Routing" (#)

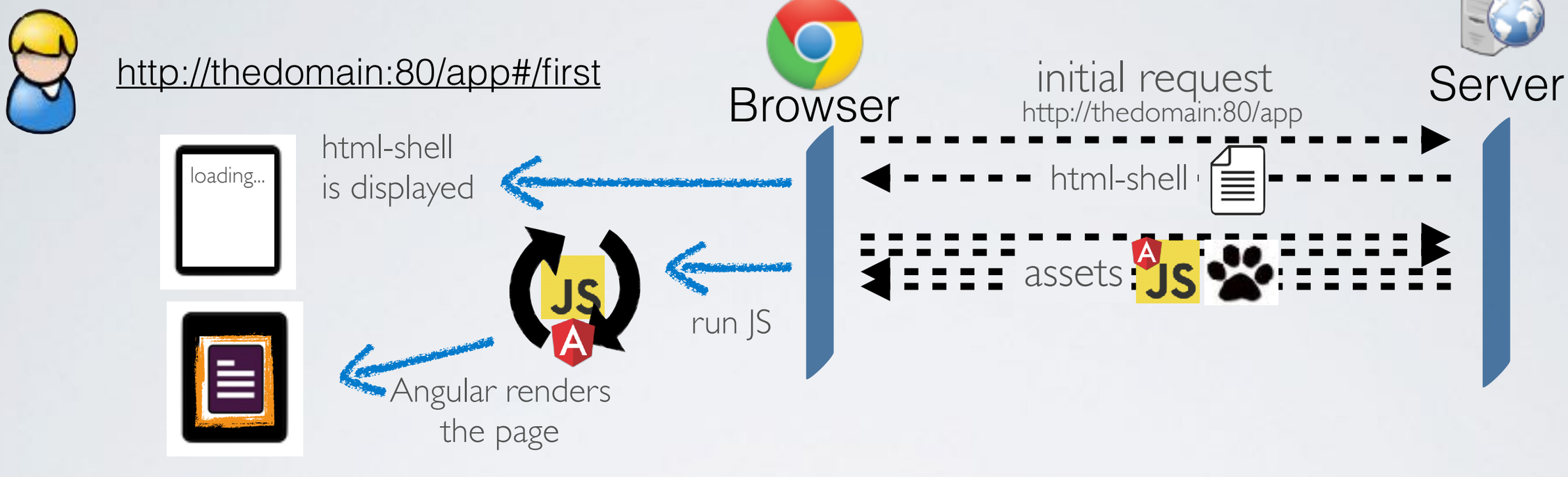
- client-route (part after the **#**) is not sent to the server with the initial http-request
- navigation: browser does not send a request to the server since only the part after the # is changing.

Modern approach: "HTML5 Routing" / "Push-State Routing"

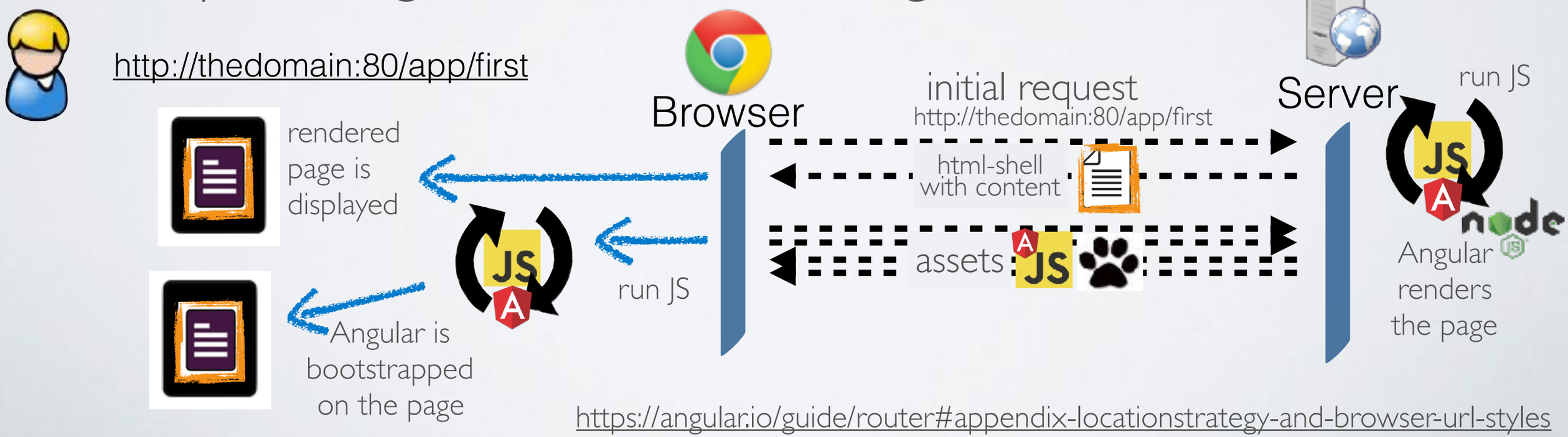
- client-route is sent to the server with the initial http-request
- navigation: url is changed on the client, but the browser does not send a request to the server
- Pitfalls:
  - Server must respond with the "default" shell for unknnown urls (url-rewrite, fallback-url, https://angular.io/guide/deployment#server-configuration)
  - Only supportet in browsers > IE 9 (https://caniuse.com/#feat=history)

Hash-Routing vs. HTML5 Routing

Traditional SPA:



Potentially enabling Server Side Rendering:



https://angular.io/guide/router#appendix-locationstrategy-and-browser-url-styles

Server-Side Rendering of a SPA

The "initial load" of a SPA is always slower than a server rendered application (several roundtrips, JavaScript parse & execute).

Perceived performance can be increased, if the first request (shell) already contains the html representing the initial screen. Current frameworks can rendered on the server to deliver the first screen and then "attach" on the client to provide the functionality.

Implication:
- Node.js on the server!
- Time to first paint increases, time to interaction does not increase!

In many projects server-side rendering is not worth the complexity. For performance optimization there are lower hanging fruits.

Routing in Angular

Angular provides the **`@angular/router`** module, which helps to implement complicated routing scenarios.

With the Angular router module you can map paths to components.

https://angular.io/guide/router

Generating an app with routing with the Angular CLI

```
ng new fantastic-ng --routing --defaults
```

In `src/app/app-routing.module.ts` routes can be configured:

```
const routes: Routes = [
  { path: '', pathMatch: 'full', redirectTo: 'first'  },
  { path: 'first', component: FirstComponent  },
  { path: 'second', component: SecondComponent  }
];
```

The router-module provides the `<router-outlet>` directive which is used in `app.component.html`

Routing

Angular provides the **`@angular/router`** package for client-side routing.

Routing is configured statically when loading a module.
It is *good practice* to configure routing in a separate module, which is imported by the app module.

```
const routes: Routes = [
  { path: '', component: OverviewComponent },
  { path: 'done', component: DoneTodosComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

```
@NgModule({
  declarations: [ ... ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Angular uses the "HTML5 routing" per default. Use

`RouterModule.forRoot(routes, {useHash: true})`

to get Hash-Routing

http://localhost:8080/app

<head>
</head>

<body>

App Component

...

<router-outlet>

...

</body>

SPA Shell

/app/path1

compontent 1

/app/path2

component 2

Router Directives

The router provides directives for placing a "routed" component into a template.

Directives: `router-outlet`, `routerLink`, `routerLinkActive`

```
<nav>
  <a routerLink="/databinding" routerLinkActive="active">Databinding</a>
  <a routerLink="/pipes" routerLinkActive="active">Pipes</a>
</nav>

<router-outlet></router-outlet>
```

You can also pass a *parameter array* to the routerLink directive: `routerLink="[/details, 5]"`

EXERCISES

Exercise 3 – Create an app with routing