In 2003 if you need to get the job as a Web Developer you should be knowing minimum below things

https://www.youtube.com/watch?v=C6MPSLgsGGs

you would need to be able to do some HTML, you'd write some javascript probably for some validation logic or to make some annoying pop-ups appear or something like that you would be doing some styling with CSS you would inevitably be writing some sequel to talk to a relational database and you will be doing some server-side programming with asp.net or asp, PHP, or java

![alt text](http://url/to/img.png)



**Building web applications using plain JavaScript?**

No Way! In the times of rapid app development, we need better ways to quickly develop interactive web applications and that is where JavaScript frameworks come to the rescue.

JavaScript frameworks are the backbone of single page web applications development and provide superpowers to plain HTML and JavaScript.



**These day if you want to get job as a typical Web Developer.**

Now things changed since then what's the landscape and what are we asking now, I have not our rocks start web developers but a web developers or typical blog standard entrance to the industry. Well these days it's not enough just to know JavaScript obviously there's quite a lot of different languages that compile to JavaScript that you’re going to have to be familiar with Typescript notably but also different versions of java script itself, different standards ES2015, ES2016 and son on that they'll have to get used to.

**Some of commonly used languages that compile to JS**

**CoffeeScript**: Unfancy JavaScript

**EmberScript**: Ember.js-infused CoffeeScript.

***Static typing : Some of the projects listed below are also statically typed.***

**TypeScript** :Typed superset of JavaScript by Microsoft.

**ActionScript** :With Apache FlexJS - Based on ECMAScript 4, ActionScript provides typing and can be compiled to JavaScript.

***JavaScript Language Extensions***

**JSX**: A superset to add an XML-like syntax to represent HTML elements in React by Facebook

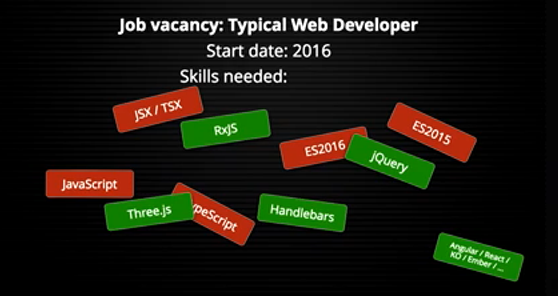
**C#, .NET related languages**

**JSIL:** MSIL (.NET bytecode) to JavaScript

**Bridge.NET**: Open Source C#-to-JavaScript compiler using Microsoft Roslyn. Includes support for jQuery + many other frameworks. Developed and well supported by a professional team. Try online at Deck.NET.



After that you need to start thinking about what frameworks you want to use. You enter in this milky world of picking your favorite client side Model View Library as well as your favorite libraries for all kinds of other tasks like to do with rendering 3d graphics or anything’s else so lots of libraries that you goanna have to learn



**jQuery**: Plain JavaScript along with jQuery has been used to build complex web interfaces but with lot more effort and complexity in code development and maintenance.

**Handlebars:** from various people. It’s also the templating engine that Ember.js uses. Handlebars works by allowing you to define templates using simple script blocks.

**Threejs:** The ever popular Three.js along with the newer Babylon.js offer web developers an abstract foundation for crafting feature rich WebGL creations ranging from animated logos to fully interactive 3D games.Three.js was originally written in ActionScript before being translated to JavaScript. Having been created before the introduction of WebGL, Three.js has the unique convenience of a modular rendering interface allowing it to be used with SVG and HTML5’s canvas element in addition to WebGL.

**JavaScript Frameworks**: give you space to focus on developing interactive elements of the user interface without worrying too much about code structure and code maintenance. JavaScript frameworks work on **MVC design** paradigm and enforce structure to ensure more **scalable**, **reusable**, **maintainable** JavaScript code. It however **is not necessary that all frameworks ride on MVC pattern**, there are many variations to it and one has choice to go with **MV\*, MVVM, MVP** as is best suited to the project needs.

Features

**AngularJS framework gives super powers to HTML** by adding all the necessary features required to build dynamic views (interactive user interface). It gives option to extend HTML **attributes** by the use of Angular **directives**. Extending HTML with AngularJS is very simple, one can use standard AngularJS directive or develop a custom directive and mount it on any div.

**Two way data binding is at the core of Angular.js**. When user interacts with the interface and provides an input, the view and the model (JavaScript objects) are synchronized, the logic in the model is executed and the DOM gets updated.

The reverse is true as well, if model gets updated, view is re-rendered. This essentially takes away all the **pain of writing manual code for DOM manipulation**.

Recently released Framework ReactJS is giving tough competition to AngularJs but Angular is holding the ground tight and growing in demand as ever before. The main reason behind Angular's continuous growth is the improvements and advancements it brings in with every new release.

**2. React.js**

What powers Facebook and Instagram’s user Interface? ***React.js JavaScript Framework*** is behind the user interface of Facebook as well and Instagram. This gives us a quick idea about how powerful is **ReactJS** when it comes to **building large scale applications of extreme dynamic nature**.

ReactJS was first released as open source in 2013 under BSD license. The community is growing rapidly ever since its release and I must say it is the **fastest growing JavaScript framework** as of today. One can find tons of resources, tutorials and React component libraries to get going within no time.

ReactJS is best at rendering complex user interfaces with high performance. The basic fundamental behind React is **the concept of virtual DOM**. ReactJS utilizes a virtual DOM, which can be rendered either at **client side or server side** and communicate back and forth.

When the data manipulation is much more dynamic and complex, client side DOM manipulations become performance intensive.

The React approach to handle this is –

* Render the DOM at server side, the virtual DOM.
* Compare virtual DOM to the browser/actual DOM and figure out differences.
* Update only the selective/changed nodes of browser DOM instead of re-rendering the entire DOM.

Another biggest advantage of react is the re-usability it brings on the table in the form of reactive components. **React component libraries can be created and used across applications** or made available for public use.

**3. Ember.js**

EmberJS is another powerful MVC JavaScript framework. Ember was initially released in 2011 as open source JavaScript framework by Yehuda Katz under MIT license. ***EmberJS competes with the likes of Angular and React*** when it comes to building interactive frontend user interfaces and also has a very active community of developers.

Ember also rides on the principal of two way data binding like AngularJS, i.e. update view when model changes and update model when the view changes, keeping both in sync all of the time.

Ember is managing to be among the top JavaScript Frameworks by continuously strengthening itself with new superpowers. It is coming up with Fastboot.js module that allows **server side rendering of DOM**, the concept similar to what React is already using for **better performance in complex UI rendering**.

Ember targets the best of both AngularJS (two way data binding) and ReactJS (server side rendering). The way Ember community continues to power it with awesome features, I am more than sure that it will continue to ride the growing wave of JavaScript Frameworks.

**4. Aurelia.js**

AureliaJS is the creation of Rob Eisenberg and team who come mostly from the world of Angular and Durandal. Aurelia though is an open source product is officially managed by Durandal Inc., a startup company that creates libraries, **tools and frameworks to support next generation of web development**.

Aurelia is just released, in January 2015, and is ready for production use. It extends the capabilities of Durandal and is termed as NextGen version of it by Eisenberg. For the existing developers who work in Durandal or Angular1 and 2, Aurelia comes with a clear migration path.

**AureliaJS** is new but if you are evaluating a JavaScript Framework, it definitely needs a consideration. It is managed by **highly professional community** and carries a great legacy.

A power fact about AureliaJS is that it is highly modularized and comprises of many independent small libraries. One can use entire framework in the project, use few of the required libraries, or extend the selected libraries to create custom framework.

Aurelia is self-contained package and doesn't have any external dependencies except for polyfills.

**5. Meteor.js**

Love developing complete web application in pure JavaScript? MeteorJS is the magical full stack platform for **building end to end mobile and web applications completely in JavaScript** at lightning speed. MeteorJS is the power player and comes equipped with all the features you need for frontend rendering, backend development, business logic and database management.

Meteor is the baby of Meteor Development Group, it was first released in 2012 as an open source JavaScript framework under MIT license.

Ever since its release, the MeteorJS ecosystem has grown huge at rapid pace and the community too is vibrant and helpful. You would find tons of resources, tutorials and custom packages that give super powers to MeteorJS.

The best thing about MeteorJS is that you use only JavaScript for end to end application development, no need to invest time learning anything else. **Meteor.JS is modular** and the packages and libraries can be used on demand.

The server side packages run in the node.js and you do not need anything else but MeteorJS packages to access the database, all in JavaScript, this makes **MeteorJS applications real time web applications**.

From Performance perspective, any changes in the database are reflected back on the UI in the real time and vice versa without the handshake between different languages or without major overhead of server response times.

**6. Backbone.js**

Are you looking for a lightweight but full featured JavaScript Framework? Backbone.JS is the one for you!

The popularity and **power of backbone can be judged from the fact that biggies like Pinterest, Foursquare, Walmart, Disqus and Delicious are using backone.js**. This is just a small subset of backbone users and the actual list is really huge to be covered here.

The good thing about backbone is that it is simple, small size package and easy to learn. You can get started building apps with Backbone JavaScript Framework within no time.

Backbone is very flexible in a way that it comes with just the minimal and you can build anything on top of it, by writing your own code or by using third party JavaScript frameworks. One can even build a **fully functional opinionated framework** with backbone at the core.

Over the past few months or so, I see comparatively slow growth in Backbone usage as compared to the rivals Angular, Ember or the newly launched React.

The main reason for that appears to be the sluggish release cycle of backbone and lack of power features that other frameworks offer out of the box.

It of course is still useful but more as a **side framework rather than the core framework** of your web project.

**7. Polymer.js**

Polymer.js is a different ball game altogether. Polymer was released by Google back in 2013. Polymer uses the concept of web components to extend HTML capabilities.

Web Components is the browser technology released by W3C using which new custom HTML elements can be created. For example <audio> is a standard HTML5 element but with the web components and related technologies you can create your own custom element - like <my-audio>.

**Polymer gives powers and brings structure in building custom HTML elements** using browser based technologies that includes web components.

**8. Knockout.js**

KnockoutJS was initially released by Steve Sanderson in 2010 as open source JavaScript Framework under MIT license. **Knockout works under the MVVM design paradigm** and that makes it a little different from Ember and Angular.

Knockout has had its golden time but as of now it is growing at a much slower pace as compared to the earlier competitors like Angular, Ember or Backbone. The slow growth is simply because it lags way behind in terms of improvements and adding more features.

Developer community is slowly moving to frameworks like React and Angular. Knockout has a great legacy and can definitely make a come back but only if someone else adopts it and starts nurturing it with the **latest and greatest of the world of JavaScript technologies**..

**9. Vue.js**

Vue.Js is the creation of Evan you. It was first released in 2014 as open source framework for public use under MIT license.

VueJs is relatively new and is gaining lot of traction among the community of developers. VueJs works with MVVM design paradigm and has a very simple API. **VueJs demonstrate minimalism to the extreme** and allows you to use selective modules, as required.

Vue is inspired by AngularJS, ReactiveJs, knockoutJS and RivetsJS and updates model and view via two way data binding.

While **Vue is not in the league of Angular or Ember but conceptually has all the potential** to be the JavaScript framework of future and hold respactable market share. It is holding the ground firmly after 10 months of its release and getting regular updates by the creator, Evan You.

**10. Mercury.js**

We are seeing release of new JavaScript frameworks at a pace witnessed never before. The promising new entrants in any field definitely get a place in my top ten lists. Mercury.JS is one such framework that has just been launched and seems to have a bright future.

Mercury is the creation of Raynos and available as open source under MIT license. It is fast growing in popularity and is **getting lot of attention from the community of JavaScript Developers**.

Mercury appears to be inspired by react and runs on the concept of virtual DOM rendering. It is **modern JavaScript framework, fully modular and can be used to the extent you need**.

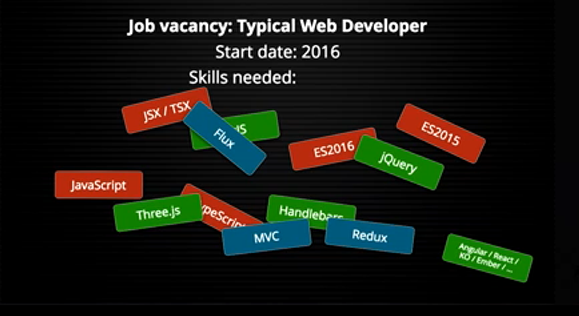
Looking at the underlying concepts, compact size, modular approach, performance and popularity, Mercury makes its place in list of best 10 JavaScript frameworks of modern times.

**Client-side application architecture/ Pattern**

That’s alone is not quite enough to write your application because you have to make some other choices for example what kind of client-side application architecture do you want to use it's not just MVC these days we've now got things like redux and other members of flux family that are bidding for our attention there.

MVC: bidirectional data flow.

Redux: unidirectional data flow.



When you go with all these stuff together you can probably start to write some code but then you need to think about how it's going to get into the browser and get executed there are many **different systems for packaging bundling** and distributing the code and for actually getting it to load inside the browsers for making references between modules and so on you have to understand the pros and cons of the different options there but you don’t just write there bundles by hand obviously you're going to need to use some sort for build tool.

JavaScript programming language has exploded in popularity. It has become the go-to language for developing both rich web applications, as well as hybrid mobile applications. And as JavaScript projects are becoming increasingly complex, developers are experiencing new requirements of the language. One of these is modularity

**ES6 modules**

One of the main goals of ES6 modularity is to make it really simple to install and use any Javascript library from anywhere on the Internet (github, npm, etc.). Only two things are needed:

•a single command to install the library (jspm install jquery)

•One single line of code to import the library and use it (var $ = require('jquery');)

**Libraries export objects in a common module format (ES6 modules).**

export default function $() { ... }

**We import a module into a local scope and use it.**

import $ from 'jquery';



**Why is modularity important in JavaScript?**

Modularity is important in JavaScript by the same reasons it’s important in any other language:

•it encourages the development of small isolated modules with clear interfaces, as opposed to large chunks of monolithic code

•it helps with testability, as modules can be replaced at runtime with mocks that implement the same interface

•it improves code maintainability, as smaller and well isolated modules are easier to understand

**Variety of module formats out there in use:**

CommonJS

AMD

UMD

ES6 Modules

**Tools Module Bundlers for JavaScript Tools for assets come in a variety of shapes and sizes:**

Browserify

jspm

Webpack

Rollup

Brunch / Brocolli

Sprockets

**Build your own with** Gulp / Grunt

**Package manager** that allows us to download a package and include it in our application. Package managers (such as Bower, NPM, JSPM and Duo) help us download front-end dependencies without having to visit a projects’ website /

1. NPM(Tagline: "npm is the package manager for JavaScript"): This one is usually thought of as the node package manager. NPM even stands for node package manager. Obviously you aren't using node in the **browser/frontend**. You can use NPM packages for frontend AND backend. Any NPM package that you find might be intended for node only, browser only, or both. These days it seems like most packages can be used in both places.
2. Bower(requires [node, npm](http://nodejs.org/) : Tagline: A package manager for the web): When Bower was created, NPM existed already, but it was for node, not the browser. Node packages didn't usually include assets (like bundled JS and CSS) in their NPM packages. If you needed the assets, you would just download them. That's junky - and that's why Bower was created. Nowadays NPM packages include frontend assets, so if you are already using NPM for your node backend, you might want to stick with NPM and skip Bower.

• bring your own CJS/AMD loader.

•bring your own build tool.

• bring your own organization conventions.

•ultimately, coupled with either make or grunt, bower looks like a viable option

1. JSPM (Tagline: Frictionless browser package management): This is for users of the SystemJS bundler. / It’s a package manager for the SystemJS universal module loader, built on top of the dynamic ES6 module loader. JSPM doesn't host any of its own packages. It allows you to install packages that are hosted on NPM or github. So if you use JSPM.
2. Duo : Duo lets you specify your require() statements as github paths, (with an optional version) like this:

***var uid = require('matthewmueller/uid');***

***var fmt = require('yields/fmt');***

***var reactive = require('component/reactive@0.14.x');***

The idea is that it saves you time and effort because you don't have to install the package or create a package.json file. Duo finds and installs the package automatically.

I would argue that this is foolish*. If you require() a package in multiple places*, you have to update all your require() statements any time you want to change the version.

Duo may save a little time in the short term, but it will cost you more in the long term.

**Older Package Managers**

•**component: is** more of a concept and a framework for building and distributing front-end components.

•jam

•enter

**Non-Package-Managers**

They aren't package managers but work as additional tools:

•**napa**: If you use NPM as your package manager, napa helps you install github repositories that haven't been configured to be used as packages yet.

•**volo**: "volo is a tool which lets you quickly create projects, add libraries, and automate common tasks"

**Feature of package managers**

**npm & bower** are package managers. They just download the dependencies and don't know how to build projects on their own. What they know is to call webpack/gulp/grunt after fetching all the dependencies.

**bower is like npm**, but builds flattened dependencies trees (unlike npm which do it recursively). Meaning npm fetches the dependencies for each dependency (**may fetch the same a few times**), while **bower** expects you to manually include sub-dependencies. Sometimes bower and npm are used together for front-end and back-end respectively (since each megabyte might matter on front-end).

**In addition, there are various libraries that allow dynamic loading of modules:**

Require.js

System.js

**Working with bower or npm**

1. Install Typescript

npm install -g typescript

***1) At the root directory of your project type***

> bower init or npm init

which will create bower.json or package.json metadata file for your project.

***2) To search for available packages to install, type***

> bower search <keywords> or npm search <keywords>.

***3) To install a package into the local module cache and then copy its latest version into your project, type***

> bower install <package-name>#<package-version> or npm install <package-name>@<package-version>.

***4) The best way to save a package's version info into your project's metadata file.***

> bower install --save <package-name>#<package-version> or npm install --save <package-name>@<package-version>.

***5) Running the tests***

> npm run build && npm run test

***6) Install first node js module***

> npm install

***7) Webpack install globally***

> npm install -g webpack webpack-dev-server

***8) To run the start-webpack-server***

> npm run start-webpack-server

***9) To install intellisense***

> npm install -g typings

***10) To run the server***

> npm run server

***11) To install Browserfy***

> npm install -g –save-dev browserify

**12) To Install Redux**

> npm intall –S redux

Or > npm I –S redux

**13) object spread operator**

npm install --save-dev babel-plugin-transform-object-rest-spread

npm install --save babel-preset-stage-2

then in config.js

query: {

presets:[ 'es2015', 'react', 'stage-2' ]

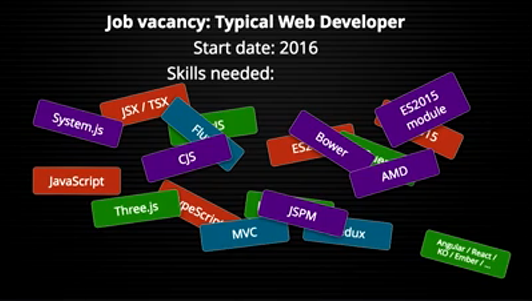
}

//z = { x, ...y };

//z = Object.assign({ x }, y);

//z = Object.assign({}, x, y);





There are many different build tools that you need to trade off and make the right choice among now for the really old school one an could use something like glup or grunt but to be honest they were like they were around in 2014 so most of us have moved on numerous times since then an these days we'll be using things like webpack and rollup and something that someone came up with this morning are however.

**Build Tools**

Most of the common action present in build tools are

* Transcompiling JavaScript: CoffeeScript, Dart, Babel, Traceur etc.
* JavaScript Transforms: wrapping in modules or ng-annotate etc.
* Bundling/Concatenation: combining of scripts and styles into multiple files
* Minification: scripts, styles and html
* Source Maps: for both scripts and styles
* CSS Preprocessor: Less, Sass, Stylus etc.
* Style Transforms: Autoprefixer, PostCSS etc.
* Cache Busting: renaming files with a hash to prevent incorrect caching
* Image Optimization
* Compiling Templates: Mustache or HandlebarsJS, Underscore, EJS, Jade etc.
* Copying Assets: html, fav icon etc.
* Watching for Changes
* Incremental Rebuild
* Clean Build: deleting all files at start or preferably cleaning up files as needed
* Injecting References: generating script and style tags that reference the bundled files
* Build Configurations: separate Dev, Test and Prod configuration, for example to not minify html in dev build
* Serve: running a development web server
* Running Unit Tests
* Running JavaScript and CSS Linters: jshint, csslint etc.
* Dependencies: handle dependencies on npm and Bower packages, Browserfy etc.

**Grunt:** almost doesn’t qualify as a build tool. It really is a system for running a sequence of commands/tasks. Each Grunt task is a plug-in that represents a tool one might run from the command line. Each plug-in defines what its configuration in the Gruntfile is named and what the configuration options are. The fixed section names mean one can’t organize or name the configuration in a way that makes sense to them.

**Gulp:** touts its high performance due to both streams and parallel task running. Yet, many users have complained that the parallel task execution is confusing and there are plans to extend the configuration to give more control over this.

**Broccoli:** Support for watch and incremental build is built in so that Broccoli “will figure out by itself which files to watch, and only rebuild those that need rebuilding”. It achieves that through a sophisticated caching system built on top of its core abstraction of file trees. Trees are Broccoli’s answer to Gulp streams and are easily chained. To simplify the system, they don’t run tasks in parallel like Gulp, but claim that it isn’t necessary because performance is still good. Broccoli is still very immature and in beta. It is lacking in the area of documentation. [The read me](https://github.com/broccolijs/broccoli/blob/master/README.md) boldly states windows support is spotty. The fact that there is little to no source map support was a big issue.

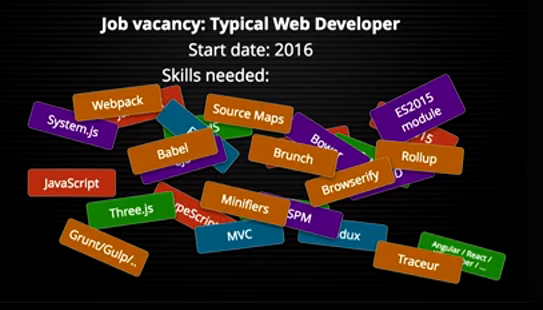
**Brunch** relies heavily on convention over configuration and assumes your build pipeline will include most of the steps. This means that a config that might be [600+ lines in Grunt](https://gist.github.com/paulmillr/eb3ae139aadbbb87ab9b#file-grunt-js) or [150+ lines in Gulp](https://gist.github.com/paulmillr/eb3ae139aadbbb87ab9b#file-gulp-js) could end up being [20 lines in Brunch](https://github.com/paulmillr/brunch-with-chaplin/blob/master/brunch-config.coffee). Brunch is written in CoffeeScript and all the examples are in CoffeeScript, but it isn’t too hard to translate to JavaScript.

[**Webpack**](https://webpack.github.io/) has the highest interest - almost 1.5 times or more higher than [Grunt](http://gruntjs.com/), [Gulp](http://gulpjs.com/), probably also due to its flexibility and ability to handle many file types.

Webpack users overlap with ES6, React, React Native, and Redux, forming one happy family.

Both Grunt and Gulp are more likely to be used with Angular, Backbone, and Jasmine, indicating their age.

Grunt and Gulp users also appear to lean on Bower to manage package version control.



**Transpilers**

CoffeeScript,

TypeScript and

Babel

**ES6 code with in-browser transpilation,**

•Babel for ES6

•TypeScript

•Traceur

**Browserify**

browser-side require() the node way

Browserify doesn't offer any tool you can do this only after downloading plugins, but using both leads to very similar results.

**Webpack(webpack-dev-server)**

Webpack do the bundling your modules to be used in a browser environment (though you can target other environments, like bundling your server-side ES6 code for Node).

It's a task runner with hot reloading of changes which allows you to forget about all JS/CSS watchers.

Webpack offers many tools by default (e.g. code splitting),Packs CommonJs/AMD modules for the browser. Allows to split your codebase into multiple bundles, which can be loaded on demand. Support loaders to preprocess files, i.e. json, jsx, es7, css, less, ... and your custom stuff.

**Grunt (configuring )**

It's JavaScript Task Runner to automate everything that can be automated (i.e. compile CSS/Sass, optimize images, make a bundle and minify/transpile it).

Grunt is based on configuring separate independent tasks, each task opens/handles/closes file.

Tasks Grunt maybe easy to read even for people not so familiar with JS

**Gulp (code)**

It's also JavaScript Task Runner,The streaming build system

Gulp requires less amount of code and is based on Node streams, which allows it to build pipe chains (w/o reopening the same file) and makes it faster.

While tasks in Gulp are easy to read even for people not so familiar with JS

**Soruce Maps**

When debugging in a browser, how to tell where the original code is? Source maps solve this problem by providing a mapping between the original and the transformed source code. In addition to source compiling to JavaScript, this works for styling as well.

Webpack can generate both inline source maps included within bundles or separate source map files. The former are valuable during development due to better performance while the latter are handy for production usage as it keeps the bundle size small.

**Rollup**

focuses particularly on bundling ES6 code. Tree shaking is one of its selling points. You can use Rollup with webpack through rollup-loader.

**Minifiers**

The point of minification is to convert the code into a smaller form. Safe transformations do this without losing any meaning by rewriting code.

**Traceur**

Traceur is a future-EcmaScript to current-EcmaScript compiler. It doesn’t add anything to the language that isn’t already proposed for the next version of EcmaScript.

If you just want to write EcmaScript 6 today, use Traceur

While Tracer does an extremely great job at making ES6 code work on older browsers, it also comes at a cost of readability. The output of Traceur generated code looks nowhere near as readable as BabelJS

**Babel**

Babel is a much safer option, since it follows the actual EcmaScript specifications, without additions

If support for JSX is a requirement for you, then you should go with BabelJS or BabelJS is the only transpiler that supports JSX.

**TypeScript**

TypeScript on the other end adds additional stuff on top of EcmaScript official specs (e.g. Types and Generics to name a few) out of the box, which many could consider a good thing, but definitely out of the standard.

If you want optional strict typing and all the benefits that go along with that, plus some ES6 features, use TypeScript.

That's not to even mention any of the advances that have happened in the world of styling which is a whole discipline in its own right which I know very little about but they've got their whole collection of technologies now in order to write effective and maintainable styles.

**CSS**

The standard CSS @import allows you to split into multiple files. The problem with this is that it creates additional HTTP requests.

**LESS**

LESS is written in javascript so you will need NodeJS to run it.

Less tends to be simpler to learn (it's a strict superset of CSS, so any valid CSS file is a valid Less file, so your CSS knowledge already technically translates to Less knowledge).

Sass and Less work a little different. Instead of creating another HTTP request, they combine the files into one

When you write Less or Sass styles, you'll need to set up some sort of compilation step to translate them to CSS that browsers can understand. Grunt, Gulp, Rails gems,

Less doesn’t have as many awesome frameworks, like Compass.

Less uses @. to declare variable.

In Less, loops only allowed for numeric values.

Unless you are working with something that relies on Less (like a certain frontend framework).

**SASS**

To run Sass, you will need to have Ruby installed.

Sass tends to be more powerful and extensible.

Sass and Less work a little different. Instead of creating another HTTP request, they combine the files into one

When you write Less or Sass styles, you'll need to set up some sort of compilation step to translate them to CSS that browsers can understand. Grunt, Gulp, Rails gems.

•Sass has a better language syntax

•Sass has more features

•Sass has Compass (and other frameworks to choose from)

Basically, Sass is next level.

In Sass you can iterate through any kind of data.

Sass uses $ to declare variable.

**CSS Reset (reset.css)**

**Normalize/Reboot(Normalize.css)**

Instead of resetting all values, it simply unifies the display of basic styles in browsers

**Bootstrap**

It provides a framework for creating user interfaces that includes a grid layout system, buttons and widgets, base styling, typographic styling, interactive/responsive menu bars, and much more.



So that's a good portion of your application but as well as that these days we also now are concerned about the different platforms that our code is need to run on it's not just desktop browsers anymore obviously it's not even just mobile browsers which themselves have got a whole set of different capabilities and UI interaction paradigm now we also need to think about if we're building a really modern application maybe our client side code doesn't just run on the client maybe it's a universal or isomorphic app where our Javascript runs on the server just as well as it runs in the browser and transparently switches between the two so there's.

**Search Engine Optimization (SEO)**

•Google SEO

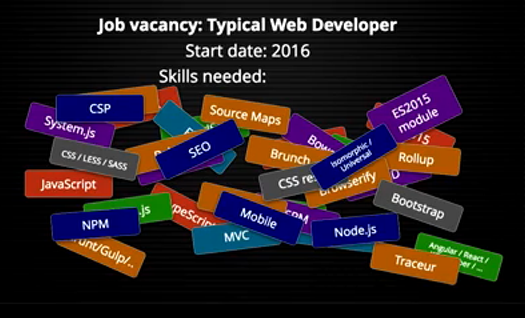
•Bing SEO

•Yahoo SEO

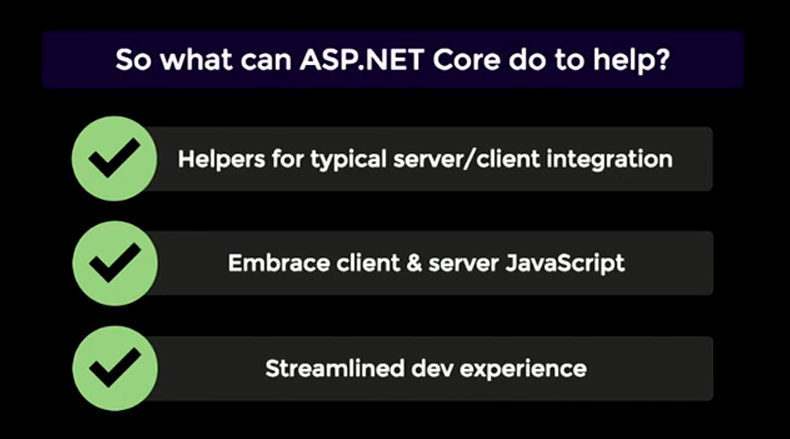
An application that can only run in the client-side cannot serve HTML to crawlers, so it will have poor SEO by default. Web crawlers function by making a request to a web server and interpreting the result

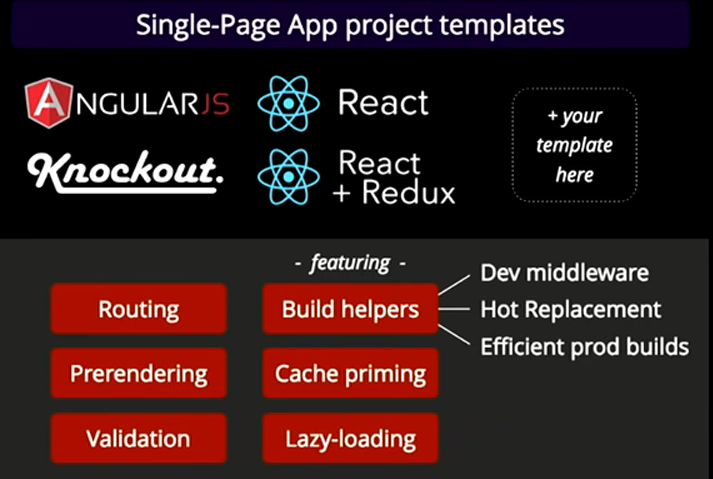
**Mobile Computer Science Principles (CSP)**

•Build socially useful mobile apps



So many different concepts that are all coming together to make a current day web application so how does this fit into the bigger picture then because when we think about .





demo $ npm install -g you generator-aspnetcore-spa

$ mkdir MyNewThing

$ cd MyNewThing

MyNewThing $ yo aspnetcore-spa

MyNewThing $ dotnet watch run

**Build helpers**

**Dev middleware** (dynamically recompiling your application when changes made in the files but still you need to refresh in the browser)

In Startup.cs go to Configure method

add after app.UseDevloperExceptionPage();

if (env.IsDevelopment())

app.UseWebpackDevMiddleware();

**Hot Module Replacement**

app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions

{

HotModuleReplacement = true

});

**Efficient prod builds**

stop the server and configure the webpack

MyNewThing $ webpack -- config webpack.config.vendor.js

Compile the project with this setting

MyNewThing $ webpack

Run the application.

MyNewThing $ dotnet watch run

To use the tree shaking feature build with production config

=====

Every time you refresh the browser page take some time to load.

**Pre-rendering feature**

Home> Index.cshtml

Add below tag.

<app asp-prerender-module="ClientApp/boot-server"

asp-prerender-webpack-config="webpack.config.js">Loading...</app>

<script src="~/dist/vendor.js" asp-append-version="true"> </script>

@section scripts {

<script src="~/dist/main.js" asp-append-version="true"> </script>

}

===

**Caching the control**

put the tag in

<cache vary-by="@Context.Request.Path">

<app asp-prerender-module="ClientApp/boot-server"

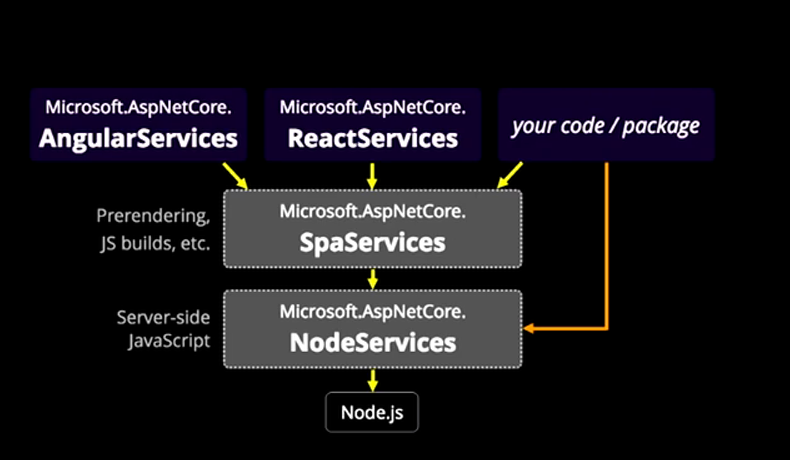
asp-prerender-webpack-config="webpack.config.js">Loading...</app>

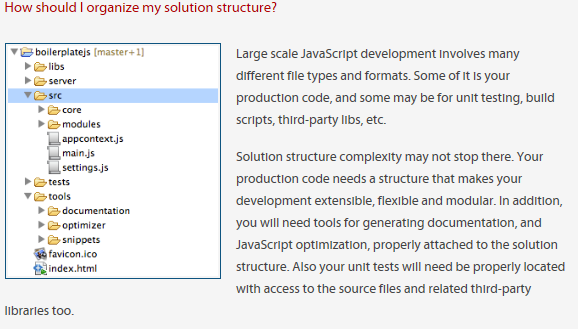
</cache>

**Running React template**

ReactSpa $ export ASPNETCORE\_ENVIRONMENT=Development

ReactSpa $ dotnet watch run





1**. Package Manager**

Package managers simplify installing and updating project dependencies, which are libraries such as: jQuery, Bootstrap,Angularjs,react

**2. Module Loader/Bundling**

Most projects of any scale will have their code split between a number of files. And while you could just include each file with an individual <script> tag. However, <script> establishes a new http connection, and for small files – which is a goal of modularity – the time to set up the connection can take significantly longer than transferring the data. While the scripts are downloading, no content can be changed on the page.

•The problem of download time can largely be solved by concatenating a group of simple modules into a single file, and minifying it.

•The performance comes at the expense of the flexibility though. If your modules have inter-dependency, this lack of flexibility may be a showstopper.

Then we heard about RequireJS, Browserify, Webpack and SystemJS

•RequireJS: is a JavaScript file and module loader. It is optimized for in-browser use, but it can be used in other JavaScript environments, like Node.

•Browserify: set out to allow use of CommonJS formatted modules in the browser. Consequently, Browserify isn’t as much a module loader as a module bundler: Browserify is entirely a build-time tool, producing a bundle of code which can then be loaded client-side.

•Webpack: It bundles all of your static assets, including JavaScript, images, CSS and more, into a single file. It also enables you to process the files through different types of loaders. You could write your JavaScript with CommonJS or AMD modules syntax. It attacks the build problem in a fundamentally more integrated and opinionated manner. In Browserify you use Gulp/Grunt and a long list of transforms and plugins to get the job done. Webpack offers enough power out of the box that you typically don’t need Grunt or Gulp at all.

•SystemJS: is a module loader that can import modules at run time in any of the popular formats used today (CommonJS, UMD, AMD, ES6). It is built on top of the ES6 module loader polyfill and is smart enough to detect the format being used and handle it appropriately. SystemJS can also transpile ES6 code (with Babel or Traceur) or other languages such as TypeScript and CoffeeScript using plugins.

**3. Task runner**

Task runners and build tools are primarily command-line tools. Why we need to use them: In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting which previously cost us a lot of times to do with command line or even manually.

•Grunt: You can create automation for your development environment to pre-process codes or create build scripts with a config file and it seems very difficult to handle a complex task. Popular in last few years.

•Gulp: Automation just like Grunt but instead of configurations, you can write JavaScript with streams like it's a node application. Prefer these days.

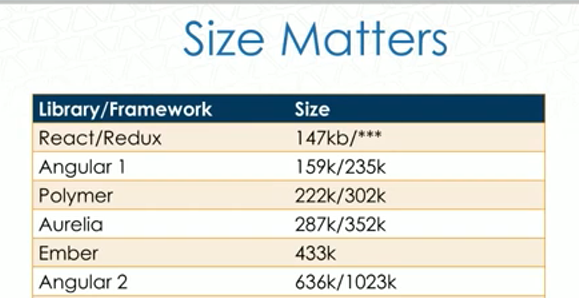
**4. Scaffolding tools**

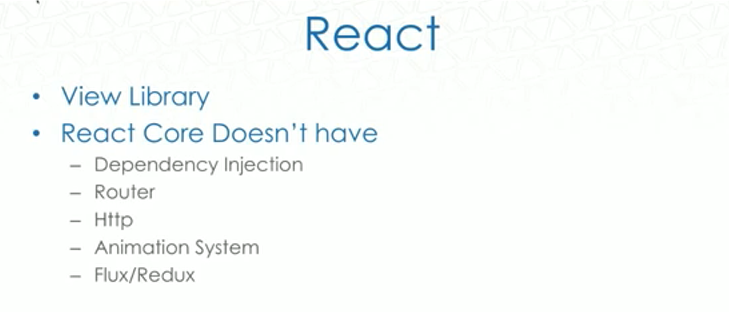
•Slush and Yeoman: You can create starter projects with them. For example, you are planning to build a prototype with HTML and SCSS, then instead of manually create some folder like scss, css, img, fonts. You can just install yeoman and run a simple script. Then everything here for you.

**What JS frameworks do you use?**

Angular 1 is very popular it was amazing release, it was one of good framework during that time having very less size and we can do lists of cool things in that, like directives and other. But it made one mistake using an observer pattern. The way it works in angular is that during every digest cycle the framework checks if any of the values in the app have changed and that's really dirty checking now imagine you have several hundred pages in your spa in your performance is bound to get slow especially when you have thousands of elements that you know are being check for and that was big issue with angular 1.

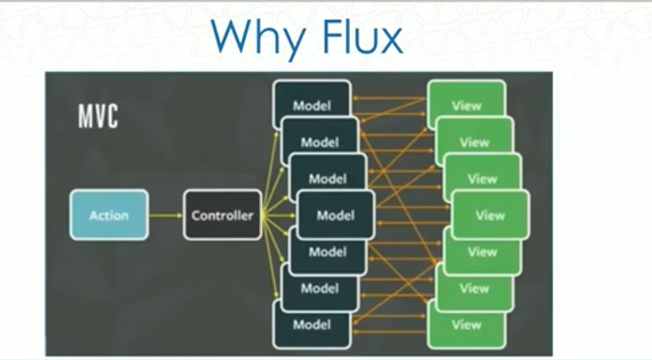
Angular 2 is really good something like what reacted it which was one-way data flow and the data was immutable so good news is that angular 2 embraces that now and it only update once the data changes so the change detection which is a new thing in angular 2 is now a quick check of an object's the reference and not really the values and change detection is what it should have been in the big you know in the first place because you know when you're creating bigger apps on a framework like this you need them to perform really well and that's



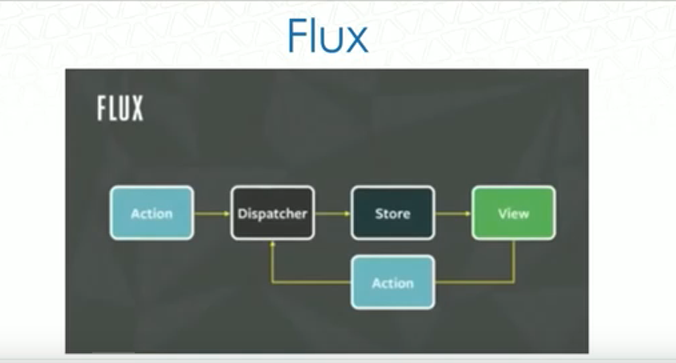




Redux is actually implementation for Flux, Redux is nothing but more than a predictable state container for JavaScript apps, so how does it really works. Suppose you want your apps to behave consistently run in different environments be it in native client or server and easy to test, Redux helps you with that so Redux gives you a great developer experience. So one of the big things about it is hot reloading instant changes on the browser without losing the current client side state and then you also have trying to time-travel debugging that allows you to travel forward and backward though state changes in your code it also allows you to play with interactions of the user and the best part where it actually is better than Flux is "Now you have a centralized graph" for all the application state so you are avoiding the complexity of interacting between multiple stores which is what you have to do with Flux and of course flux is a better pattern for getting one way observer pattern. So Redux is actually an improvement over Flux, the best part is that you also have reduced boilerplate over flux the top level of the container patterns are subscribed to the Redux store in a better way, And redux is less than 3k. Redux is works great with React or any other view Library.



This picture is taken from the Facebook where they explained why they need a Flux. The NBC app where you have a model that refresh the view and then inside the view something else needed data from another model and you see the pic you have a situation that you have so many references that you don’t know you have circular reference going on here and In the diagram you see bidirectional arrows and that's where the complexity is creating the issues and what they had was a chat bug where the user would go and see what there's an unread message and the moments clicks on it the counter says there's 100 messages if you click on it there was no unread message so they started having issues like these they had testability issues and a lot of that kind of stuff so they simplify this model. So in order to simplify this model they come up with flux



**What really is Flux?**

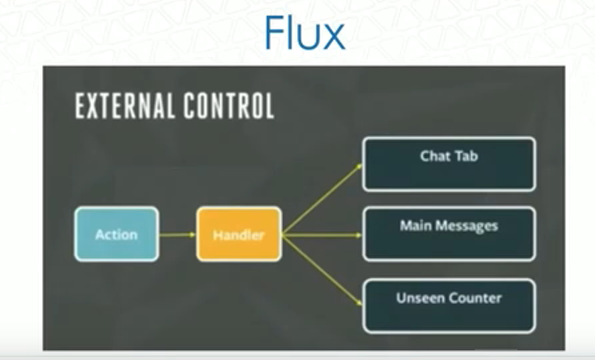
Well you have an action which has a dispatcher and that dispatcher talks to the store which really goes to the view and its really nothing more than it's called a controller view which is listening for change events and then if you have another action the dispatcher is going to wait till the first action gets all the way to the view through the store and the dispatcher will send over the next action.

**What’s an action?**

Action is really nothing more than helper method which is collected into a library that creates an action from method parameters and assigns it to a type and provides it to the dispatcher, in the dispatcher every action that the dispatcher really takes is sent to all the stores be a callback so this is more like a listener pattern and the stores are registered with the dispatcher.

**What’s the store?**

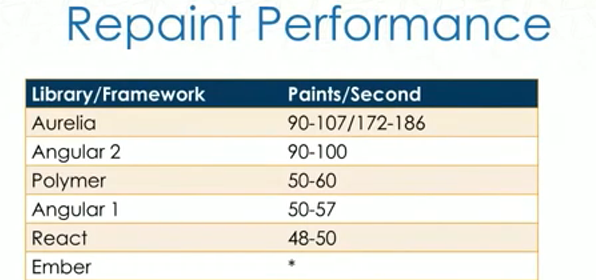
So often the store really updates the view in response to an action it actually creates or triggers a change event so these special views which are called controller view are listening for change events and they retrieve new data from the stores and provide the new data to the entire tree of their child views so you will have something like below

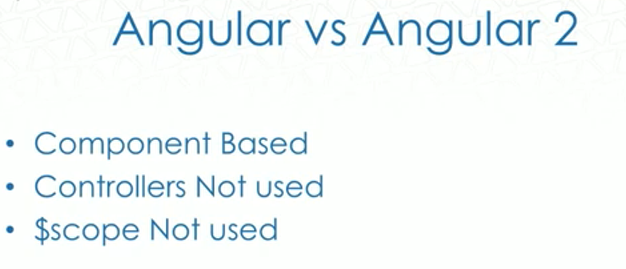


You can have the handler, The chat Tab is listening to it, the main messages are listening to it and unseen counter is listening to it so once everything gets updated you see you only have unidirectional updates which is a better improvement over earlier.

Benefits of this is you have better performance you know at the same time as lots more simplified architecture and testability.

React also work with angular or other libraries, it’s more like view library. With React 2 component model it’s a great start but it’s not a very standard web component model because every other framework is going towards more like standard web component model whereas React has its own web component model which is great but at the same time they are not going in the same direction just with community.





<https://github.com/AjinkyaKher/Podcast_React_Redux_Angular2_RxJs>

**Redux - Basics**

* . Single source of truth
* . Read only state
* . State transformed using pure functions.

**What is Redux:?**

Its a state container for your javascript applications.

**Why do you need a state container?**

To manage the state of your application.

**What is the state of your application?**

When you start writing application from scratch things are simple, the of number of variables that you want to keep a track of is less and also probably they are static but as the complexity of your application starts growing you will see that you will make a lot of rest service calls, lots of third-party calls there's a chain of callbacks that's going on also the number of variables in your application keep on increasing now it become complex app, now you don’t have concrete way of managing your state then if something breaks in your application logic or rather if you see your UI and see that a particular piece of the UI is broken to figure out what piece of functionality actually it’s the hard thing, And answering to the question e.g. If state of my application changes from A to B let's say what are the UI pieces that are affected due to it. What kind of side effects or other effects of that should I expect in my application such questions seem pretty difficult to be answered?

If you go down the Redux route it's like tailor-made for answering such questions and you should start thinking of building your application in this way so when I say the Redux. The way you start building an application is by thinking very first and foremost about the state of your application now in the case of a to-d-app.

Let’s start thinking of the state of a to-do application as an array of to do's right. So let’s go through these three commandments, if you want to know about Redux absolutely you should know about these things

**Three Principles**

***Redux can be described in three fundamental principles:***

*1) Single source of truth :* The state of your whole application is stored in an object tree within a single store.

This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle. Some functionality which has been traditionally difficult to implement.

*2) State is read-only :*The only way to change the state is to emit an action, an object describing what happened.

This ensures that neither the views nor the network callbacks will ever write directly to the state. Instead, they express an intent to transform the state. Because all changes are centralized and happen one by one in a strict order, there are no subtle race conditions to watch out for. As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes.

***3)Changes are made with pure functions :***To specify how the state tree is transformed by actions, you write pure reducers.

Reducers are just pure functions that take the previous state and an action, and return the next state. Remember to return new state objects, instead of mutating the previous state. You can start with a single reducer, and as your app grows, split it off into smaller reducers that manage specific parts of the state tree. Because reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.

==========================================

1) Single source of truth: so there is a store in the Redux which manages the state of your application and whenever you think of you think of Redux everything is JSON so the state of your application is also a JSON and it maintain the state of your application and its read-only meaning you. Means you can’t make changes to this state if something causes a change in the state then new copy of the state is available to you the existing state is never changed which means that at one point of time there would be multiple copies of state for your application which toggling between the states is easy. Actions like redo undo are going to be easy.

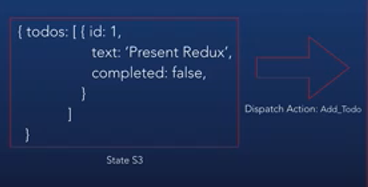
2) Read only state: If state is read only then how do I change the state? You can change the state using "Reducers", Reducers are pure functions it means that when you pass input to this function it produces the same output no matter what. So things like generating a random number or getting a current date time can’t be done this function should produce the same output for the same given input even thousand years from now and for you need to make sure you don’t have any unwanted side effects in your functions.

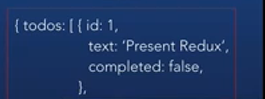
What is it that actually causes a change in the state? It's called an action and once again everything in the Redux is JSON object, even your action going to be the JSON object. When this action is dispatched Its(store) understands that I need to transform the state and how does the store transform the state using the reducers which you register with the store at the beginning of the application.

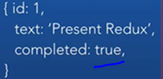
Toggling the state



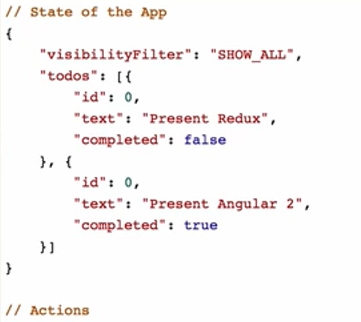
Adding the state.

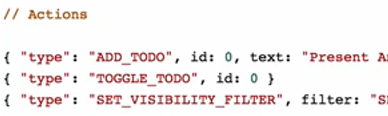


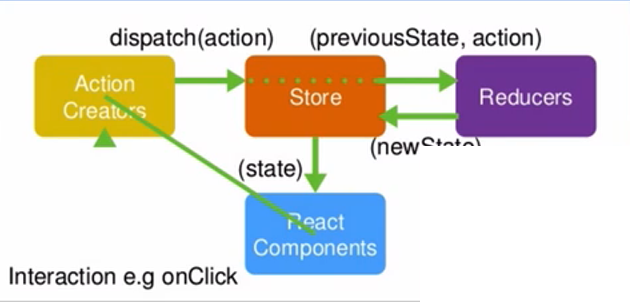




Example







There is a store, react components mean UI pieces of our application and when user play with it, e.g. let’s say when user click the button in response to this action is an action is dispatched now for this you can see in diagram there is an action creator which basically means functions that return or rather that dispatch action, which store receives, now let’s say the type of this action was an action "E" now the store looks up the reducers that we have registered with it at the beginning of the application and find out for action type "E" what is it that I supposed to do is get that piece of functionality and says I know the current state of the application let me apply this functionality, compile to new state and return that new state that's what it exactly does it says previous state in this diagram is the new current state and apply the action and it returns a new state. It means that the UI components or some UI components are subscribed, they are actively listening to changes in the state this store notifies these components of a change in the state and they do their job accordingly now they get the new state and from the new state they can do according to what they supposed to do like updating control based on state. In this case you have seen data flows in one direction as you can see it never flows from the UI to the storage.

Always flows from the store to the UI, and from the UI to the store it’s notify any update through action dispatchers not directly. In short only data flow from one direction which from the Store to the UI. This is all about Redux.

**In React we have two types of components**

1) **Presentational Components.**

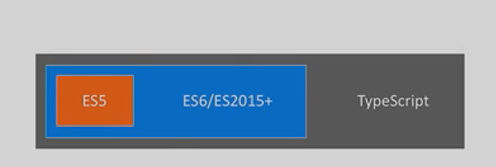
* . How things look (markup, styles)
* . It don’t know about Redux, its dumb components
* . In terms of functionality it read data from props
* . On data changes invoke callbacks from props to smart components

**2) Container components.**

* It’s know about Redux, It’s the one that is actively listening to the changes of state and its going to react.
* How things work (data fetching, state updates)
* Its smart components it dispatches action. So for every presentational components there is container components

Angular 2: when we are coding angular 2 we have couple of choices. We can use ES5(2014), ES6(2015) , ES7(2016) is coming out or TypeScripts.

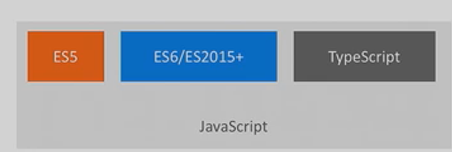
To understand little bit about ECMAScript that’s what ES stands for. Along the say, Javascript was taking years to come out with new versions.



If you are looking at number of ES is one year off. Every year new version of javascript (ES) and browser is adopting it. But how can we use new stuff of ES and keep up with the changes, that’s where TypeScipt comes in, TypeScript makes it easy so it’s always looking ahead to what angular is doing. ES5 is older browser compatibility like if you are using IE8 probably a great option for you. With ES5/6 there is no compilation. With ES we won’t get types and some of these decorators and other futures. With the TypeScript you get all the stuff and any Javascript is valid TypeScript.

CoffeeScript and other languages(Backbone.js, embered.js) which also provide the abstraction to javascript.

But with TypeScript we have option where any javascript is valid with TypeScripts which is not true with other. So TypeScirpt is super set of all old version of javascript.



There was a great question “Have you ever adopted/used something in the TypeScript that was then taken out of ECMAScript. No, this was never happened to us and the reason for that is there’s very specific process for the TypeScript team to go through to adopt new features. Like there is a category for feature and if the feature is reaches to that category like e.g( category 3) where still it’s under discussion this feature goes to TypeScript first. Which means we are still getting Advance but stable code in the TypeScript.

<https://www.youtube.com/watch?v=zoCLEwFUpqg>

<https://angular.io/docs/ts/latest/quickstart.html>