Node Package Manager (NPM)

NPM is used to install javascript tools for server side, or dev tool usage. NPM hosts packages for client and server (typically called browserify compatible) but they must be installed via JSPM commands. The contents and commands related to a node package are all stored within the package.json file in its root.

NPM Commands

help	npm --help
install package from https://www.npmjs.com/	npm install packagename --save
install internal package (windows cli)	npm install friendlyName=%wgit%repository.git
update installed version	npm update [friendlyname e.g. packagename]
run a command from package.config under the scripts section	npm run [commandname]

NPM Tasks

Delete a package	delete the package entry from package.config, and run "npm prune" to remove stale libraries
Install global package	tools to be used standalone from the command-line can be installed with -g or --global, which installs them next to your npm executable instead of in the current project

Javascript Package Manager (JSPM)

JSPM is a package manager that installs packages in a format understood by SystemJs. It was designed to allow packages to be pulled from any configurable endpoint (github/npm/private repositories), in a way that allows for versioning and dependencies across systems.

JSPM Commands

help	jspm --help
install package from http://kasperlewau.github.io/registry/#/	jspm install packagename
install package from windsor	jspm install friendlyName=windsor:repo-name.git
install package from npm https://www.npmjs.com/	jspm install npm:packageName
install package from github	jspm install github:user/reponame
update installed version	jspm update [friendlyname e.g. packagename]
remove package	jspm uninstall packageName

JSPM Tasks

install file loader	For loading non-js files using custom loaders, set install with friendlyname as the file extension ex: jspm install html=text (to load html as a string using the text loader). Used in code by placing a trailing ! after the extension ex: import 'test.html!';

SystemJs

SystemJs is a runtime package loader. Its configuration is found in config.js (by default), or the file specified as the jspm config file in package.json.It loads javascript packages via javascript commands to make their services available. It is comparable to existing loaders such as commonjs, requirejs, and webpack.

Features of SystemJS include:

optional transpilation of es6 syntax via plugins

package version management

non-js content loaders

production build capability into standalone js file, or multiple module formats

SystemJs Config

defaultJSExtensions	adds js after files in import statements. This is deprecated, it must be true currently, but will be false in the future, so always explicitly specify file extensions.
transpiler	plugin used to convert es6 into javascript
paths	created by jspm to lookup packages in file structure, do not modify
meta	overrides for working with incompatible packages (ex: allows making JQuery a global for plugins that don't know how to load it as a module)
map	maps package names to files/folders generally managed by JSPM and should not be modified

Bullet to Demystify the Package Tree Mental Model

Ideally everything is a package, including your SPA (its entry point is the act of loading index.html)

Packages always have a single entry point, and optionally, children/descendants.

The topmost package will not support es6 until browsers do, because of this, the topmost package must use the SystemJs global to load its children

Packages only need references to children that they work with directly, all other descendants are taken care of automatically.

A production package (SPA) is just the entry point to the top level package (all descendant code is present, but technically obscured/private).

Using Javascript Packages

For thinking about javascipt packages in .Net terms:

Library:

package.json defines a main file which is the entry point to the package. This file (generally index.js) should contain exports for all public classes within the package. This makes a package equivalent to a .Net dll, where all of the public classes are listed in index.js.

Class / Service:

Export from a single js file.

Sharing code:

Import statements are like "using" statements but they aren't optional because javascript has no concept of automatic sharing within namespaces.

index.js example:

Using Javascript Packages (cont)

```
import Class1 from './lib /cl ass 1.js';
import Service1 from './lib /se rvi ce1.js';
export {Class1, Service1};
```

class1.js example:

```
import Service2 from './ser vic e2.js';
export default function thething{ Servic e2.l og('oh yeah');}
```

Top Level Package:

When loading a package into a browser, because ES6 syntax is not yet supported, systemjs has a global with an import method which returns a promise with the resulting module. **In production this is not required because Systemjs will compile the package into a single js file which can be loaded using a script tag.**

Example:

```
var lodash = null;
System.im por t(' lod ash ').t hen(
functi on( res ult){ lodash = result;}
);
```

More complex info on modules and syntax here: http://www.2ality.com/2014/09/es6-modules-final.html

Promises

Promises are a simple way of performing async code in javascript. They are included natively in most browsers, and easily polyfilled in non-compliant browsers. Complex promise libraries should not be used as polyfills for the same reason it is dangerous to extend other native JS browser objects.

Promises (Extremely Basic) Usage

Create a promise from a value	`var promise = Promis e.r eso lve (value)`
Do something when a promise resolves or rejects	`promis e.t hen (fu nction fulfil led (va lue){}, function reject ed( err) {})`
Do something when a promise rejects	`promis e.c atc h(f unction reject ed( err){})`
Resolve an array of promises (can also contain raw values)	`Promis e.a ll( [va lue 1,p rom ise 1]).th en( fun ction fulfil led (ar g) { /*a rg[ 0]= value1 arg[1] =result of promis e1*/})`

Basic Rules inside a "then":

1. If a promise is returned, the returned promise will be resolved and used for the next "then".
2. If a value is returned, it will be passed to the next then.
3. If an error is thrown, or promise rejected, the chain will skip to the next "then" or "catch" that handles the error.
4. Promise.all rejects if any promises reject.

Better documentation here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Semver

Semver is a versioning scheme used across most package management systems. It is used within a package to specify when patches, minor features, and breaking changes occur. It is used by package managers to specify which versions of a package are considered compatible, and what the allowed automatic-upgrade path should be.

Package managers and semver

Package managers install the latest version of a dependency, that meets all of the requirements of other dependencies.
Example:
Package relies on lodash@1 (any release of lodash major version 1)
The latest release of lodash is 1.6.8.
Package contains 1.6.8 as installed dependency.
Package adds a dependency "foo" that requires lodash@1.5.2 (this is hypothetical, 3rd party sources generally lock on major version only)
Package now contains lodash@1.5.2 which meets requirements of both.
There are also generally devices for locking versions and solving conflicts built into every package manager, but use cases are for extreme legacy support and generally semver results in smooth upgrade paths making them unnecessary.

Semver (2.0) Syntax Inside A Package

Version format *1	[v][Major].[Minor].[Patch]
When you change something and tests break or you don't believe in testing but you still pretend to be collaborative	npm version major
When you add something and no tests break	npm version minor
When you fix a bug	npm version patch

*1 The leading v on versions is ignored, but some providers aren't smart about how that gets handled so it should always be included on git tags to remain standard in our code.

Semver has many more features/options for complex flows that we probably shouldn't waste our time on.

Semver (2.0) Syntax Consuming A Package

npm package format	`" dep end enc ies ": { " pac kag eNa me": " ver sio n" }`
jspm package format	`" dep end enc ies ": { " pac kag eNa me": " whe ret ofi ndp ack age @ve rsi on" }`
npm installed from git format *1	`" dep end enc ies ": { " pac kag eNa me": " git +re pou rlw ith rea don lyc red ent -ial s#v ers ion (with leading v)" }`
~ prefix	Allows patch-level changes if a minor version is specified on the comparator. Allows minor-level changes if not.
^ prefix	Allows changes that do not modify the left-most non-zero digit
Partial Version	Includes all versions that match the present parts. ex: 1 = 1.1.0 or 1.5.4 but not 2.anything.anything

*1 npm packages installed directly from git use git tags directly and don't pay attention to semver, so it is best to just force move a major version tag until switching to the next major version. Npm git provider should only be used for internal dev tools so versioning is not overly important.