JavaScript IntelliSense

Visual Studio 2017 provides a powerful JavaScript editing experience right out of the box. Powered by a TypeScript based language service, Visual Studio delivers richer IntelliSense, support for modern JavaScript features, and improved productivity features such as Go to Definition, refactoring, and more.

> *The JavaScript language service in Visual Studio 2017 uses a new engine for the language service (former code name "Salsa"). Details are included here in this topic, and you might also want to read this blog post. The new editing experience also mostly applies in VS Code. See the VS Code docs for more info.*

For more information about the general IntelliSense functionality of Visual Studio, see Using IntelliSense.

What's New in the JavaScript language service in Visual Studio 2017

- Richer IntelliSense
- Automatic acquisition of type definitions
- Support for ES6 and beyond
- JSX syntax support

Richer IntelliSense

JavaScript IntelliSense in Visual Studio 2017 will now display a lot more information on parameter and member lists. This new information is provided by the TypeScript language service, which uses static analysis behind the scenes to better understand your code. TypeScript uses several sources to build up this information.

- IntelliSense based on type inference
- IntelliSense based on JSDoc
- IntelliSense based on TypeScript Declaration Files

IntelliSense based on type inference

In JavaScript, most of the time there is no explicit type information available. Luckily, it is usually fairly easy to deduce a type given the surrounding code context. This process is called type inference.

For a variable or property, the type is typically the type of the value used to initialize it or the most recent value assignment.

```
var nextItem = 10;
nextItem; // here we know nextItem is a number

nextItem = "box";
nextItem; // now we know nextItem is a string
```

For a function, the return type can be inferred from the return statements.

For function parameters, there is currently no inference, but there are ways to work around this using JSDoc or TypeScript `.d.ts` files (see later sections).

Additionally, there is special inference for the following:

- "ES3-style" classes, specified using a constructor function and assignments to the prototype property.
- CommonJS-style module patterns, specified as property assignments on the `exports` object, or assignments to the `module.exports` property.

```
function Foo(param1) {
    this.prop = param1;
}
Foo.prototype.getIt = function () { return this.prop; };
// Foo will appear as a class, and instances will have a 'prop' property and a
'getIt' method.

exports.Foo = Foo;
// This file will appear as an external module with a 'Foo' export.
// Note that assigning a value to "module.exports" is also supported.
```

IntelliSense based on JSDoc

Where type inference does not provide the desired type information (or to support documentation), type information may be provided explicitly via JSDoc annotations. For example, to give a partially declared object a specific type, you can use the `@type` tag as shown below:

```
/**
 * @type {{a: boolean, b: boolean, c: number}}
 */
var x = {a: true};
x.b = false;
x. // <- "x" is shown as having properties a, b, and c of the types specified
```

As mentioned, function parameters are never inferred. However, using the JSDoc `@param` tag you can add types to function parameters as well.

```
/**
 * @param {string} param1 - The first argument to this function
 */
function Foo(param1) {
    this.prop = param1; // "param1" (and thus "this.prop") are now of type "string".
}
```

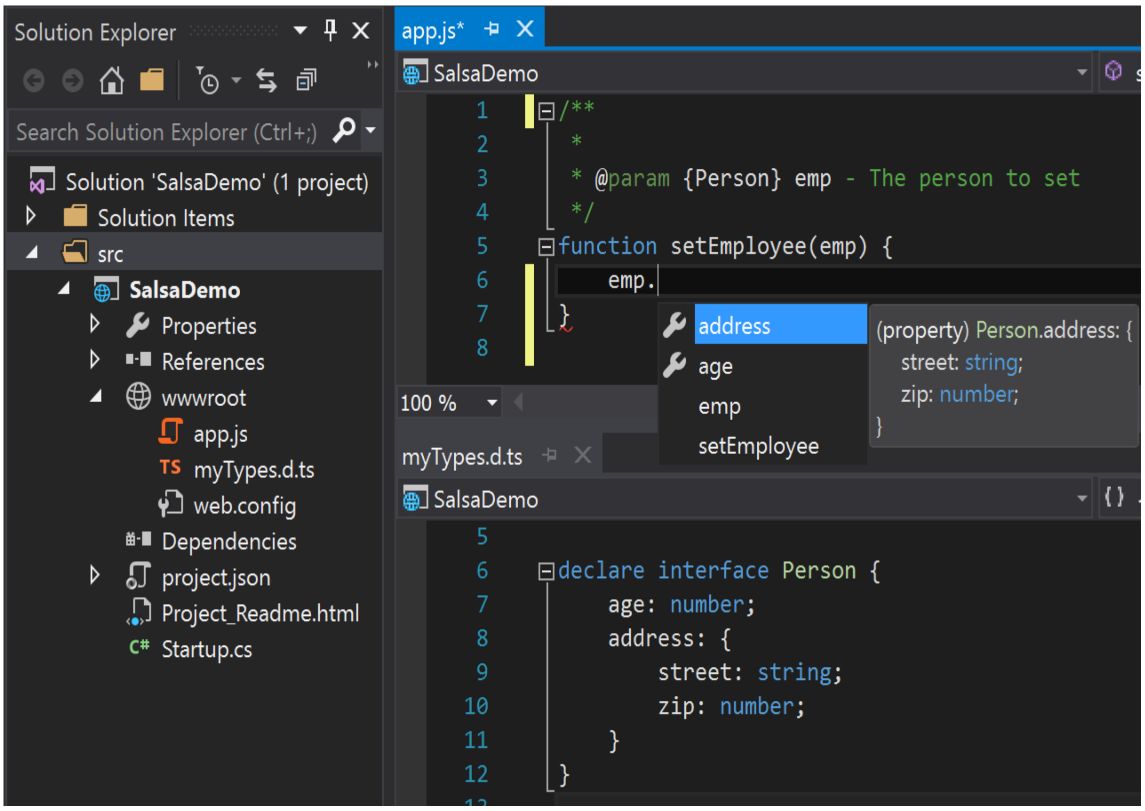
See this doc for the JsDoc annotations currently supported.

IntelliSense based on TypeScript Declaration Files

Because JavaScript and TypeScript are now based on the same language service, they are able to interact in a richer way. For example, JavaScript IntelliSense can be provided for values declared in a `.d.ts` file (more info), and types such as interfaces and classes declared in TypeScript are available for use as types in JsDoc comments.

Below, we show a simple example of a TypeScript definition file providing such type information (via an interface) to a JavaScript file in the same project (using a JsDoc tag).

TypeScript declarations used in JavaScript



```
/**
 *
 * @param {Person} emp - The person to set
 */
function setEmployee(emp) {
    emp.
}
```

```
address
age
emp
setEmployee
```

```
(property) Person.address: {
    street: string;
    zip: number;
}
```

```
declare interface Person {
    age: number;
    address: {
        street: string;
        zip: number;
    }
}
```

Automatic acquisition of type definitions

In the TypeScript world, most popular JavaScript libraries have their APIs described by `.d.ts` files, and the most common repository for such definitions is on DefinitelyTyped.

By default, the Salsa language service will try to detect which JavaScript libraries are in use and automatically download and reference the corresponding `.d.ts` file that describes the library in order to provide richer IntelliSense. The files are downloaded to a cache located under the user folder at `%LOCALAPPDATA%\Microsoft\TypeScript`.

> *This feature is **disabled** by default if using a `tsconfig.json` configuration file, but may be set to enabled as outlined further below).*

Currently auto-detection works for dependencies downloaded from npm (by reading the `package.json` file), Bower (by reading the `bower.json` file), and for loose files in your project that match a list of roughly the top 400 most popular JavaScript libraries. For example, if you have `jquery-1.10.min.js` in your project, the file `jquery.d.ts` will be fetched and loaded in order to provide a better editing experience. This `.d.ts` file will have no impact on your project.

If you do not wish to use auto-acquisition, disable it by adding a configuration file as outlined below. You can still place definition files for use directly within your project manually.

Support for ES6 and beyond

ES6, or ECMAScript 2015, is the next version of JavaScript. It brings new syntax to the language such as classes, arrow functions, `let` / `const`, and more. All of this new syntax is supported in Visual Studio.

One of the key features TypeScript provides is the ability to use ES6 features, and emit code that can execute in JavaScript runtimes that don't yet understand those newer features. This is commonly known as "transpiling". Because JavaScript uses the same language service, it too can take advantage of ES6+ to ES5 transpilation.

Before this can be set up, some understanding of the configuration options is required. TypeScript is configured via a `tsconfig.json` file. In the absence of such a file, some default values are used. For compatibility reasons, these defaults are different in a context where only JavaScript files (and optionally `.d.ts` files) are present. To compile JavaScript files, a `tsconfig.json` file must be added, and some of these defaults must then be set explicitly.

The required settings for the tsconfig file are outlined below:

- `allowJs` : This value must be set to `true` for JavaScript files to be recognized. By default this is `false` , as TypeScript compiles to JavaScript, and this is necessary to avoid the compiler including files it just compiled.
- `outDir` : This should be set to a location not included in the project, in order that the emitted JavaScript files are not detected and then included in the project (see `exclude` below).
- `module` : If using modules, this setting tells the compiler which module format the emitted code should use (e.g. `commonjs` for Node or bundlers such as Browserify).
- `exclude` : This setting states which folders not to include in the project. The output location, as well as non-project folders such as `node_modules` or `temp` , should be added to this setting.
- `enableAutoDiscovery` : This setting enables the automatic detection and download of definition files as outlined above.
- `compileOnSave` : This setting tells the compiler if it should recompile any time a source file is saved in Visual Studio.

In order to convert JavaScript files to CommonJS modules in an `./out` folder, settings similar to the below might be included in a `tsconfig.json` file.

```
{
  "compilerOptions": {
    "module": "commonjs",
    "allowJs": true,
    "outDir": "out"
  },
  "exclude": [
    "node_modules",
    "wwwroot",
    "out"
  ],
  "compileOnSave": true,
  "typingOptions": {
    "enableAutoDiscovery": true
  }
}
```

With the above settings in place, if a source file (`./app.js`) existed which contains several ECMAScript 2015 language features as shown below:

```
import {Subscription} from 'rxjs/Subscription';
```

```
class Foo {
    sayHi(name) {
        return `Hi ${name}, welcome to Salsa!`;
    }
}

export let sqr = x => x * x;
export default Subscription;
```

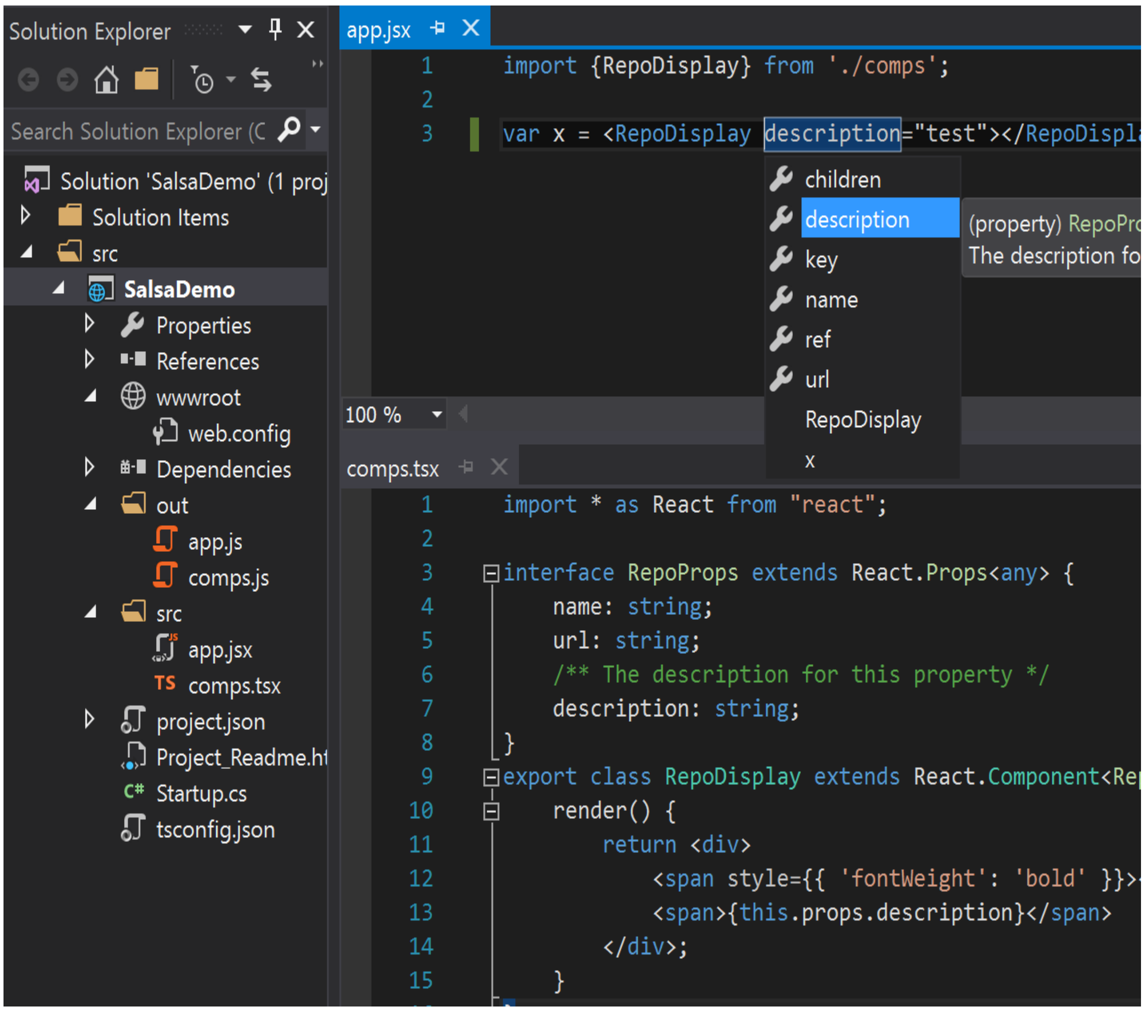
Then a file would be emitted to `./out/app.js` targeting ECMAScript 5 (the default) that looks something like the below:

```
"use strict";
var Subscription_1 = require('rxjs/Subscription');
var Foo = (function () {
    function Foo() {
    }
    Foo.prototype.sayHi = function (name) {
        return "Hi " + name + ", welcome to Salsa!";
    };
    return Foo;
}());
exports.sqr = function (x) { return x * x; };
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = Subscription_1.Subscription;
//# sourceMappingURL=app.js.map
```

JSX syntax support

JavaScript in Visual Studio 2017 has rich support for the JSX syntax. JSX is a syntax set that allows HTML tags within JavaScript files.

The illustration below shows a React component defined in the `comps.tsx` TypeScript file, and then this component being used from the `app.jsx` file, complete with IntelliSense for completions and documentation within the JSX expressions. You don't need TypeScript here, this specific example just happens to contain some TypeScript code as well.



```
import {RepoDisplay} from './comps';

var x = <RepoDisplay description="test"></RepoDispl
```

```
import * as React from "react";

interface RepoProps extends React.Props<any> {
    name: string;
    url: string;
    /** The description for this property */
    description: string;
}
export class RepoDisplay extends React.Component<Re
    render() {
        return <div>
            <span style={{ 'fontWeight': 'bold' }}>
            <span>{this.props.description}</span>
        </div>;
    }
```

> *To convert the JSX syntax to React calls, the setting `"jsx": "react"` must be added to the `compilerOptions` in the `tsconfig.json` file outlined above.*

The JavaScript file created at `./out/app.js' upon build would contain the code:

```
"use strict";
var comps_1 = require('./comps');
var x = React.createElement(comps_1.RepoDisplay, {description: "test"});
```

Notable Changes from VS 2015

As Salsa is a completely new language service, there are a few behaviors that will be different or absent from the previous experience. The most notable of these changes are the replacement of VSDoc with JSDoc, the removal of custom `.intellisense.js` extensions, and limited IntelliSense for specific code patterns.

No more `///<references/>` or `_references.js`

Previously it was fairly complicated to understand at any given moment which files were in your IntelliSense scope. Sometimes it was desirable to have all your files in scope, other times it wasn't, but this lead to complex configurations involving manual reference management. Going forward you no longer need to think about reference management and so you don't need triple slash references comments or `_references.js` files.

VSDoc

XML documentation comments, sometimes referred to as VSDocs, could previously be used to decorate your source code with additional data that would be used to buff up IntelliSense results. VSDoc is no longer supported in favor of JSDoc which is easier to write and the accepted standard for JavaScript.

`.intellisense.js` extensions

Previously, you could author IntelliSense extensions which would allow you to add custom completion results for 3rd party libraries. These extensions were fairly difficult to write and installing and referencing them was cumbersome, so going forward the new language service won't support these files. As an easier alternative, you can write a TypeScript definition file to provide the same IntelliSense benefits as the old `.intellisense.js` extensions. You can learn more about declaration (`.d.ts`) file authoring here.

Unsupported patterns

Because the new language service is powered by static analysis rather than an execution engine (read this issue for information of the differences), there are a few JavaScript patterns that no longer can be detected. The most common pattern is the "expando" pattern. Currently the language service cannot provide IntelliSense on objects that have properties tacked on after declaration. For example:

```
var obj = {};
obj.a = 10;
obj.b = "hello world";
obj. // IntelliSense won't show properties a or b
```

You can get around this by declaring the properties during object creation:

```
var obj = {
    "a": 10,
    "b": "hello world"
}
obj. // IntelliSense shows properties a and b
```

You can also add a JSDoc comment as shown above:

```
/**
 * @type {{a: number, b: string}}
 */
var obj = {};
obj.a = 10;
obj.b = "hello world";
obj. // IntelliSense shows properties a and b
```