

# Developing Custom Visuals for Power BI



# Agenda

- Installing the Power BI Developer Tools
- Creating Your First Custom Visual
- Defining Data Roles and Data Mappings
- Extending a Visual with Custom Properties
- Migrating to Version 3 of the Power BI Developer Tools



# Installing the Power BI Developer Toolchain

- Install Node.JS
  - Installs Node Package Manager (npm)
- Install Visual Studio Code
  - Lightweight Alternative to Visual Studio for Node.js Development
- Install the Power BI Developer Tools (pbiviz)
  - Install using Node Package Manager (npm)
- Create and install a local self-signed certificate
  - Install using Power BI visuals CLI tool (pbiviz)



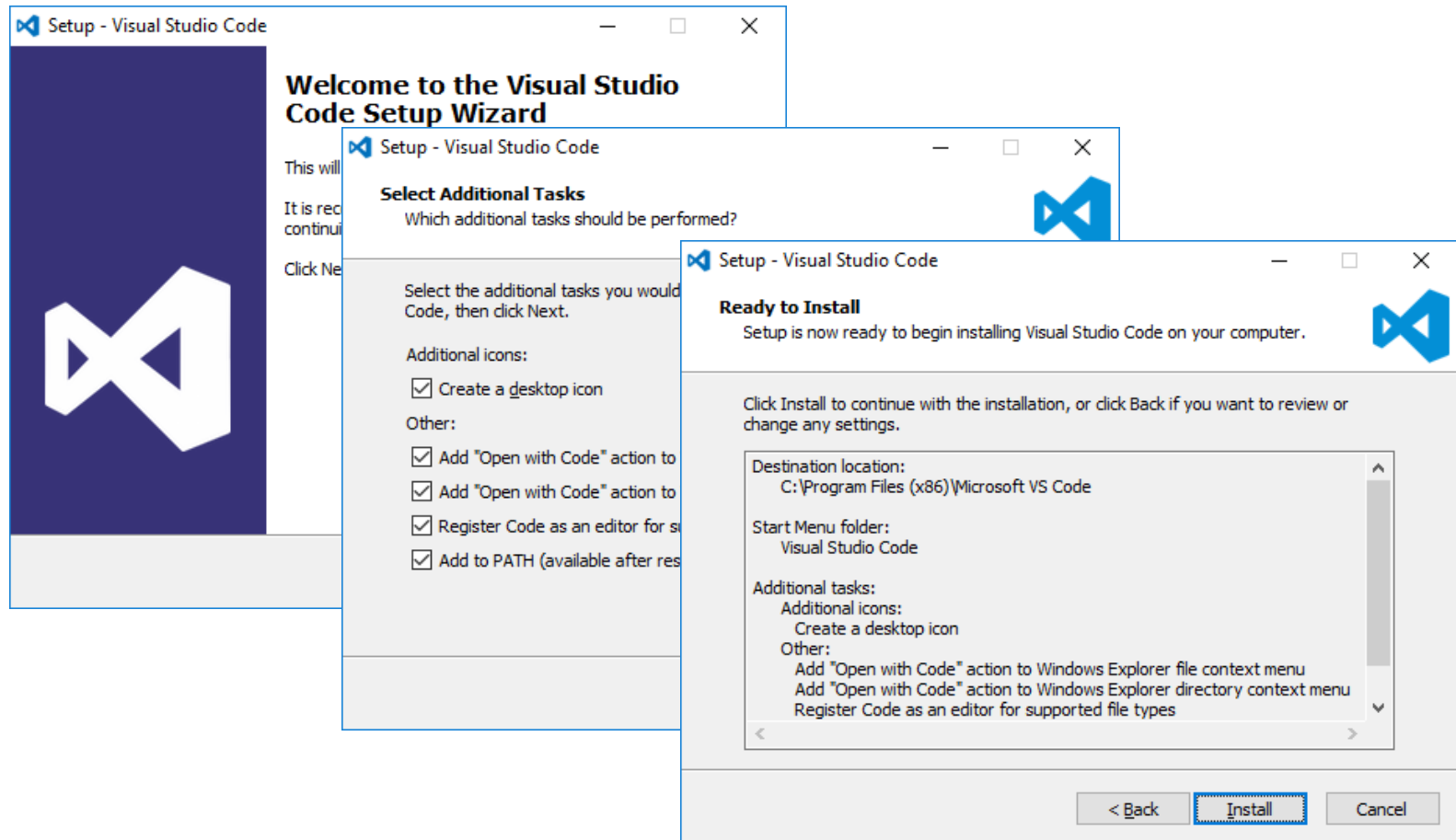
# Installing node.js

- <https://nodejs.org/en/download/>



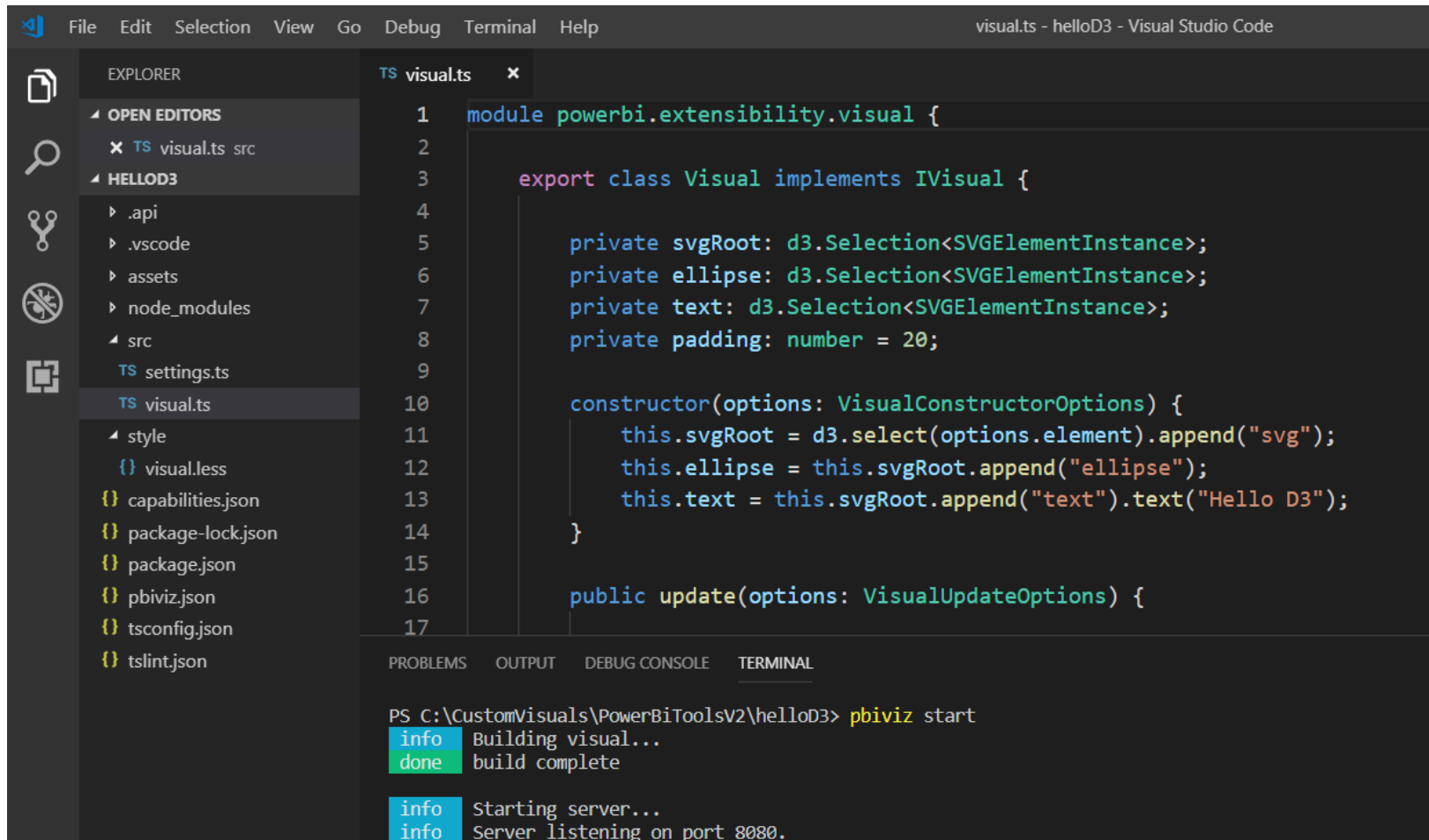
# Install Visual Studio Code

- <http://code.visualstudio.com/>



# Developing with Visual Studio Code

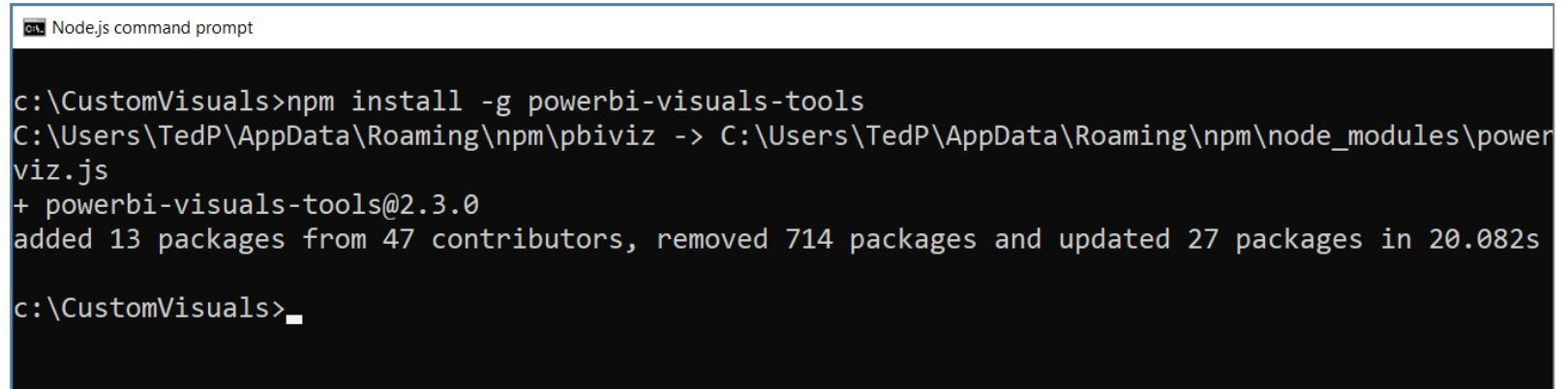
- Provides great development experience with node.js





# Power BI Visual CLI Tool (PBIVIZ)

- What is the Power BI Custom Visual Tool?
  - Command-line utility for cross-platform dev
  - Use it with Visual Studio or Visual Studio Code
  - Requires that you first install node.js
  - Install by running command from node.js command prompt  
**npm install -g powerbi-visuals-tools**



```
Node.js command prompt

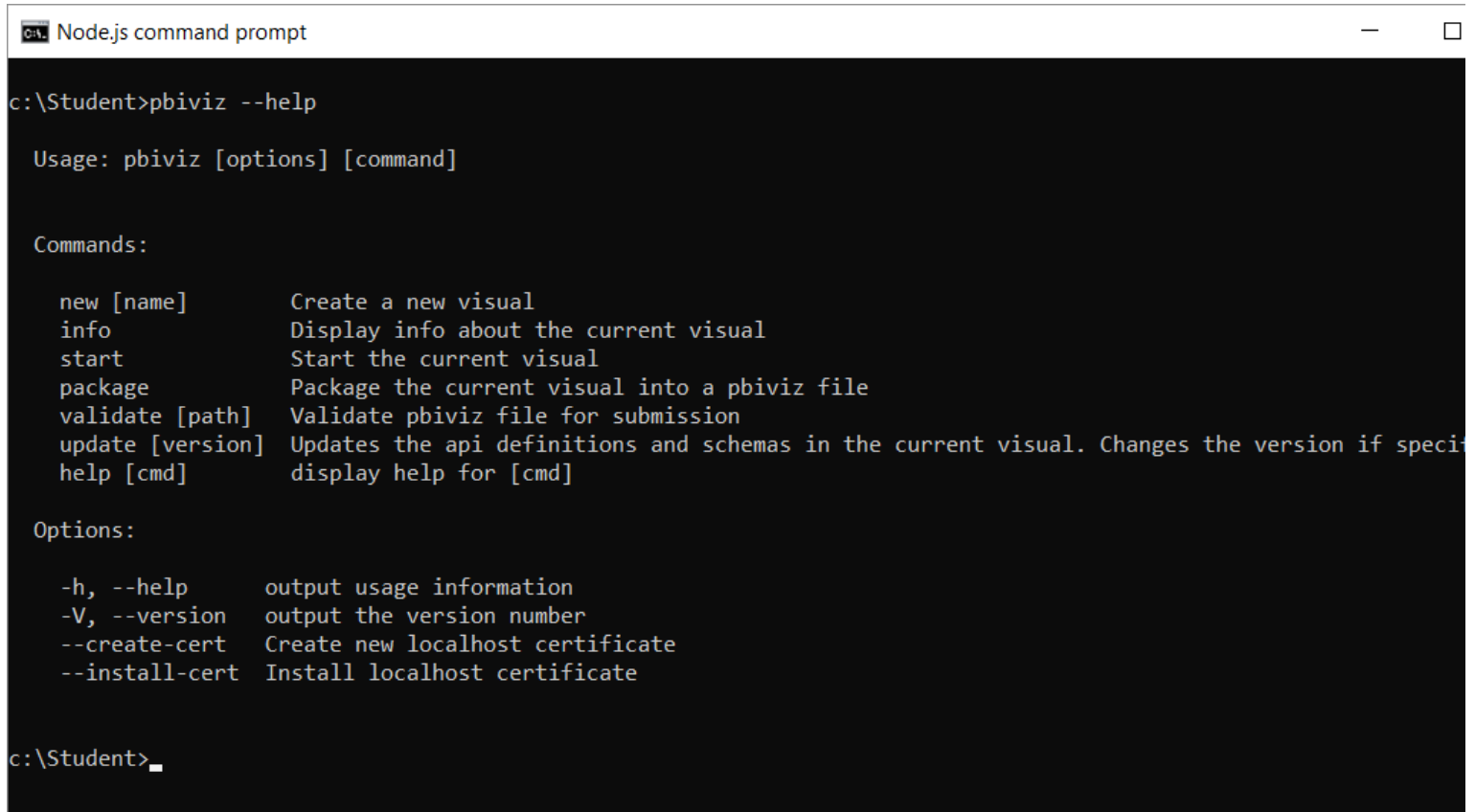
c:\CustomVisuals>npm install -g powerbi-visuals-tools
C:\Users\TedP\AppData\Roaming\npm\pbiviz -> C:\Users\TedP\AppData\Roaming\npm\node_modules\powerbi-visuals-tools
+ powerbi-visuals-tools@2.3.0
added 13 packages from 47 contributors, removed 714 packages and updated 27 packages in 20.082s

c:\CustomVisuals>_
```



# Getting Started with PBIVIZ

- PBIVIZ.EXE is a command-line utility
  - You execute PBIVIZ commands from the NODE.JS command line



```
Node.js command prompt

c:\Student>pbiviz --help

Usage: pbiviz [options] [command]

Commands:

  new [name]      Create a new visual
  info            Display info about the current visual
  start          Start the current visual
  package        Package the current visual into a pbiviz file
  validate [path] Validate pbiviz file for submission
  update [version] Updates the api definitions and schemas in the current visual. Changes the version if specified
  help [cmd]      display help for [cmd]

Options:

  -h, --help      output usage information
  -V, --version    output the version number
  --create-cert    Create new localhost certificate
  --install-cert   Install localhost certificate

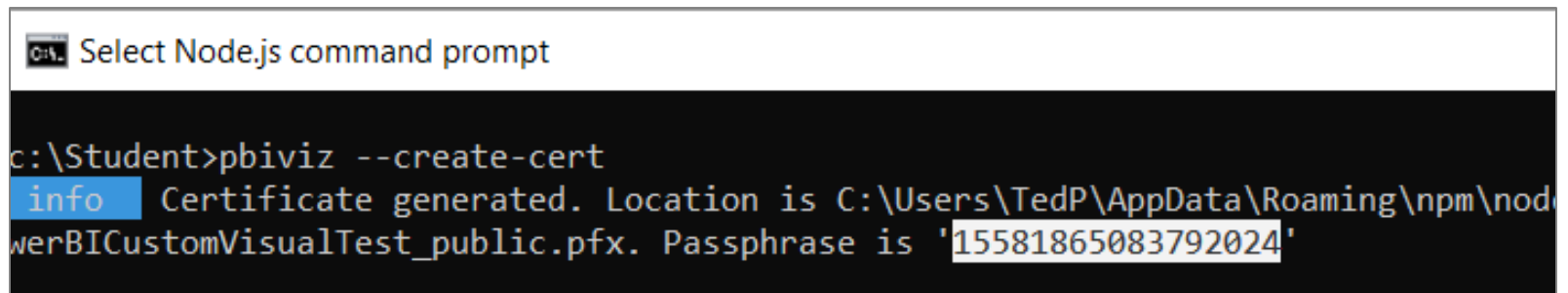
c:\Student>
```





# Creating a Certificate for Local Testing

- PBIVIZ provide local web server for testing & debugging
  - Web server runs locally on developer's workstation in Node.js
  - Makes it possible to test custom visuals in Power BI Service
  - Custom visual resources served up from <https://localhost>
  - Setup requires creating self-signed SSL certificate
  - SSL certificate created using **pbiviz --create-cert** command
  - You must copy a passphrase to properly install the certificate



```
C:\> Select Node.js command prompt

c:\Student>pbiviz --create-cert
info Certificate generated. Location is C:\Users\TedP\AppData\Roaming\npm\nod
werBICustomVisualTest_public.pfx. Passphrase is '15581865083792024'
```



# Installing the SSL Certificate

- Installing certificate enables SSL through <https://localhost>
  - Installing certificate is a one time operation – not once per project
  - SSL certificate installed using **pbiviz --install-cert** command
  - Running **--install-cert** command starts Certificate Import Wizard

```
Node.js command prompt

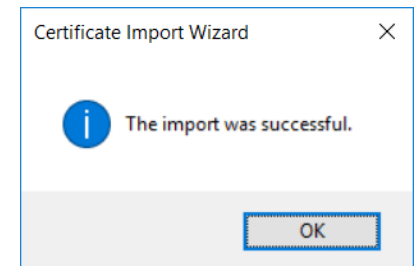
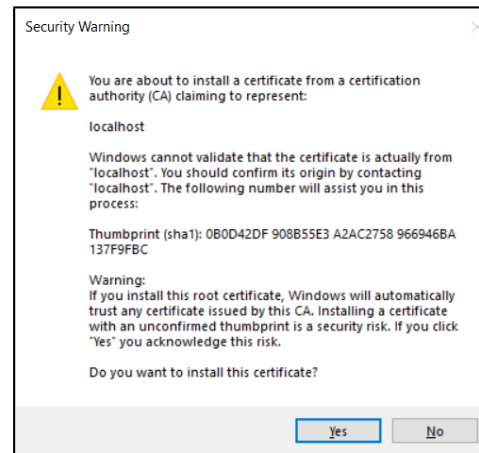
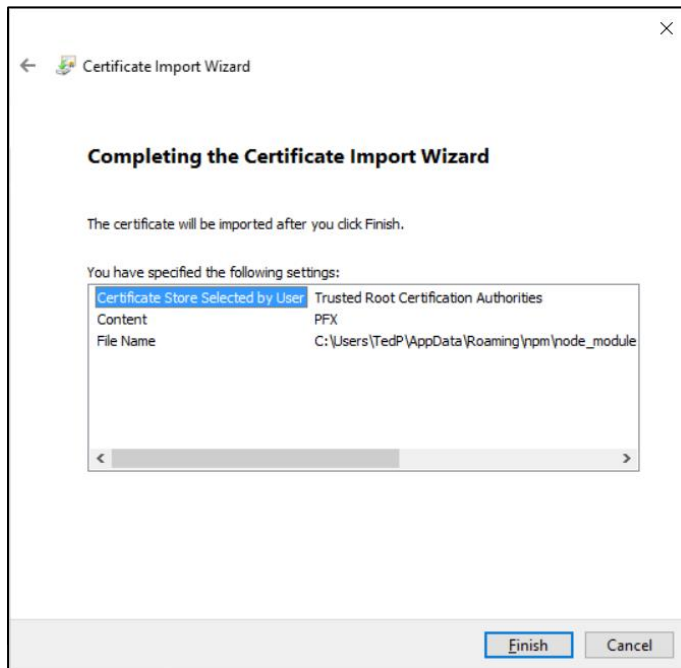
c:\Student>pbiviz --install-cert
info Use '15581865083792024' passphrase to install PFX certificate.

c:\Student>_
```



# The Certificate Import Wizard

- Wizards steps you through process of installing certificate
  - You enter certificate passphrase as part of installation process



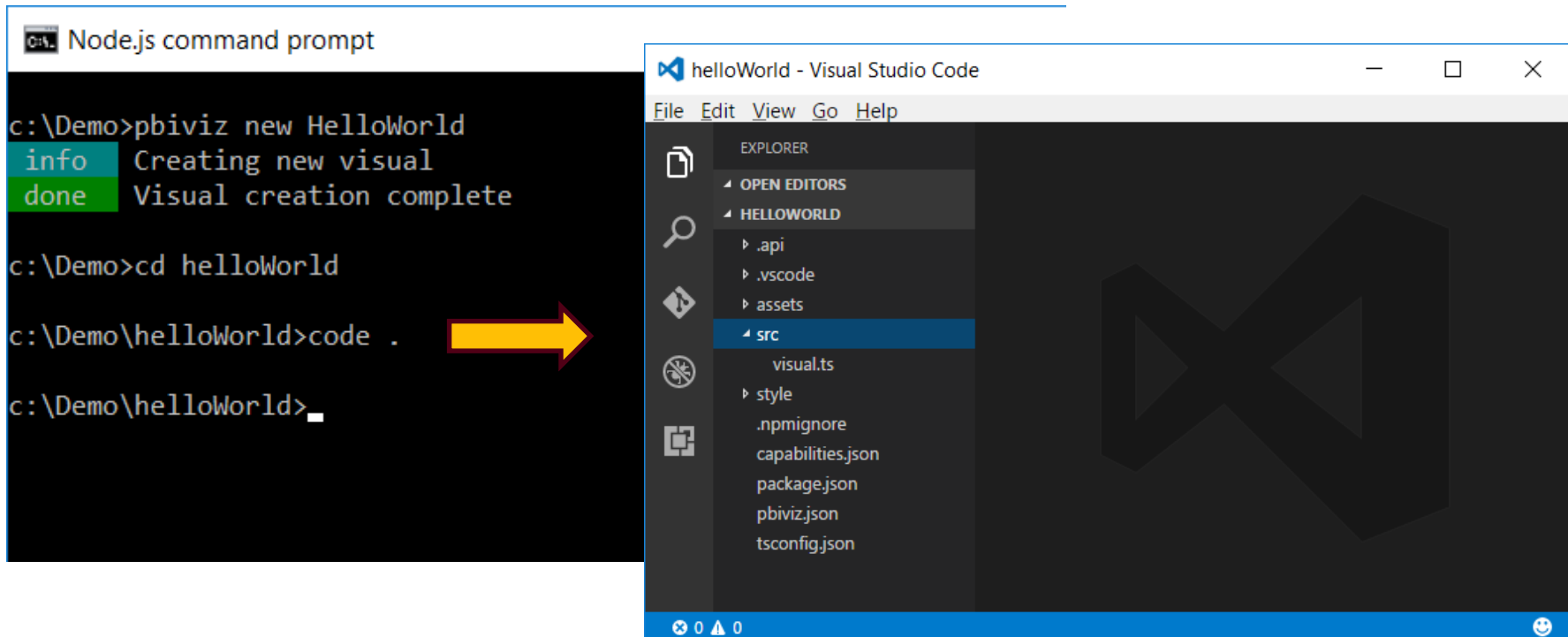
# Agenda

- ✓ Installing the Power BI Developer Tools
- Creating Your First Custom Visual
  - Defining Data Roles and Data Mappings
  - Extending a Visual with Custom Properties
  - Migrating to Version 3 of the Power BI Developer Tools



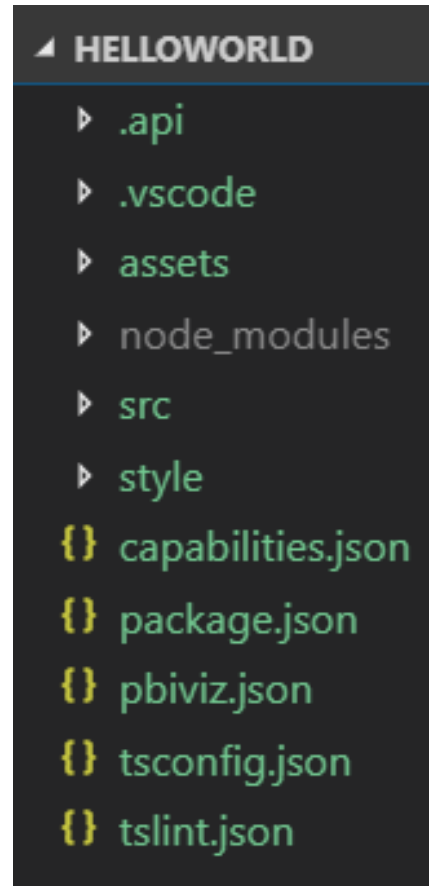
# Creating a New Custom Visual Project

- Creating a new project  
`pbiviz new <ProjectName>`
- Open the Project with Visual Studio Code  
`code .`



# Top-level project files

- `package.json`
  - Used by npm to manage packages
- `pbiviz.json`
  - Main manifest file for your custom visual project
- `capabilities.json`
  - File used to define visual capabilities
- `tsconfig.json` & `tslint.json`
  - Typescript compiler settings



# The pbiviz.json File

- Acts as top-level manifest file for custom visual project
  - Indicated which version of the Custom Visual API is used
  - External JS library files must be referenced in **externalJS** section

```
1 {
2   "visual": {
3     "name": "helloWorld",
4     "displayName": "HelloWorld",
5     "guid": "helloWorldE7F4986C5F864D7589F9F4E14FAAE3EF",
6     "visualClassName": "Visual",
7     "version": "1.0.0",
8     "description": "",
9     "supportUrl": "",
10    "gitHubUrl": ""
11  },
12  "apiVersion": "2.3.0",
13  "author": {
14    "name": "",
15    "email": ""
16  },
17  "assets": {
18    "icon": "assets/icon.png"
19  },
20  "externalJS": [
21    "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.js"
22  ],
23  "style": "style/visual.less",
24  "capabilities": "capabilities.json",
```





# The tsconfig.json File

- Used to add references to other TypeScript files
  - Controls which TypeScript files are passed to TypeScript compiler
  - No need to reference \*.d.ts files in the **node\_modules/@types** folder


```
{ } tsconfig.json •
1  {
2    "compilerOptions": {
3      "allowJs": true,
4      "emitDecoratorMetadata": true,
5      "experimentalDecorators": true,
6      "target": "ES5",
7      "sourceMap": true,
8      "out": "./.tmp/build/visual.js"
9    },
10   "files": [
11     ".api/v1.11.0/PowerBI-visuals.d.ts",
12     "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.d.ts",
13     "node_modules/powerbi-visuals-utils-typeutils/lib/index.d.ts",
14     "node_modules/powerbi-visuals-utils-formattingutils/lib/index.d.ts",
15     "src/settings.ts",
16     "src/visual.ts"
17   ]
18 }
```



# Installing D3 when using PBIVIZ Version 2

- Install package for D3 library version 3.x  
`npm install d3@3 --save-dev`
- Install package for type definition files version 3  
`npm install @types/d3@3 --save-dev`
- Update **externalJS** section of **pbiviz.json**

```
17   "assets": {  
18     "icon": "assets/icon.png"  
19   },  
20   "externalJS": [  
21     "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.js",  
22     "node_modules/d3/d3.js"  
23   ],  
24   "style": "style/visual.less",  
25   "capabilities": "capabilities.json",  
26   "dependencies": "dependencies.json",  
27   "stringResources": []  
28 }
```



# Visual Source Files

- `visual.ts`
  - visual class definition
- `settings.ts`
  - helper class to manage visual properties
- `visual.less`
  - CSS used to style custom visual

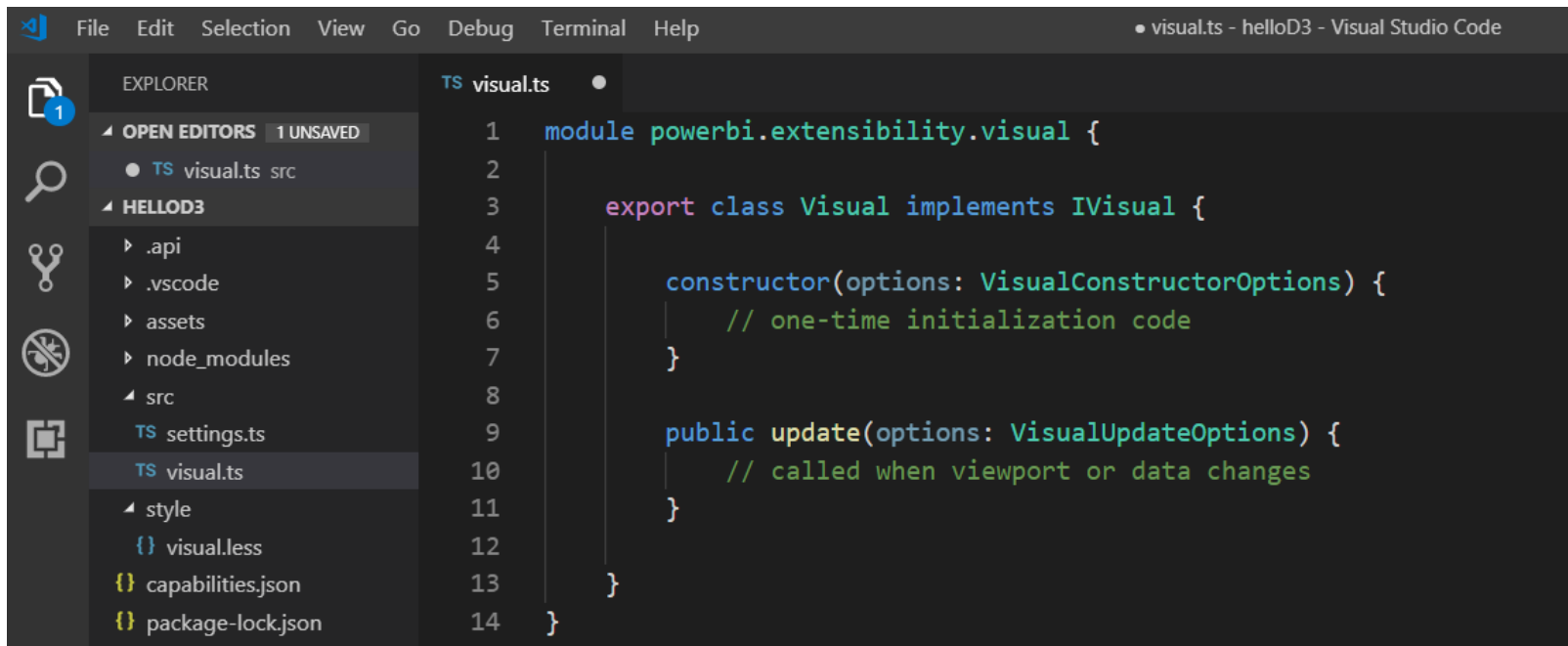
## ▲ HELLOWORLD

- ▶ `.api`
- ▶ `.vscode`
- ▶ `assets`
- ▶ `node_modules`
- ▲ `src`
  - TS `settings.ts`
  - TS `visual.ts`
- ▲ `style`
  - { } `visual.less`



# Authoring a Custom Visual Class

- Custom visual is a class that implements **IVisual**
  - Class must be defined in **powerbi.extensibility.visual** namespace
  - Minimum visual class must provide **update** method
  - Parameterized **constructor** used to create visual elements

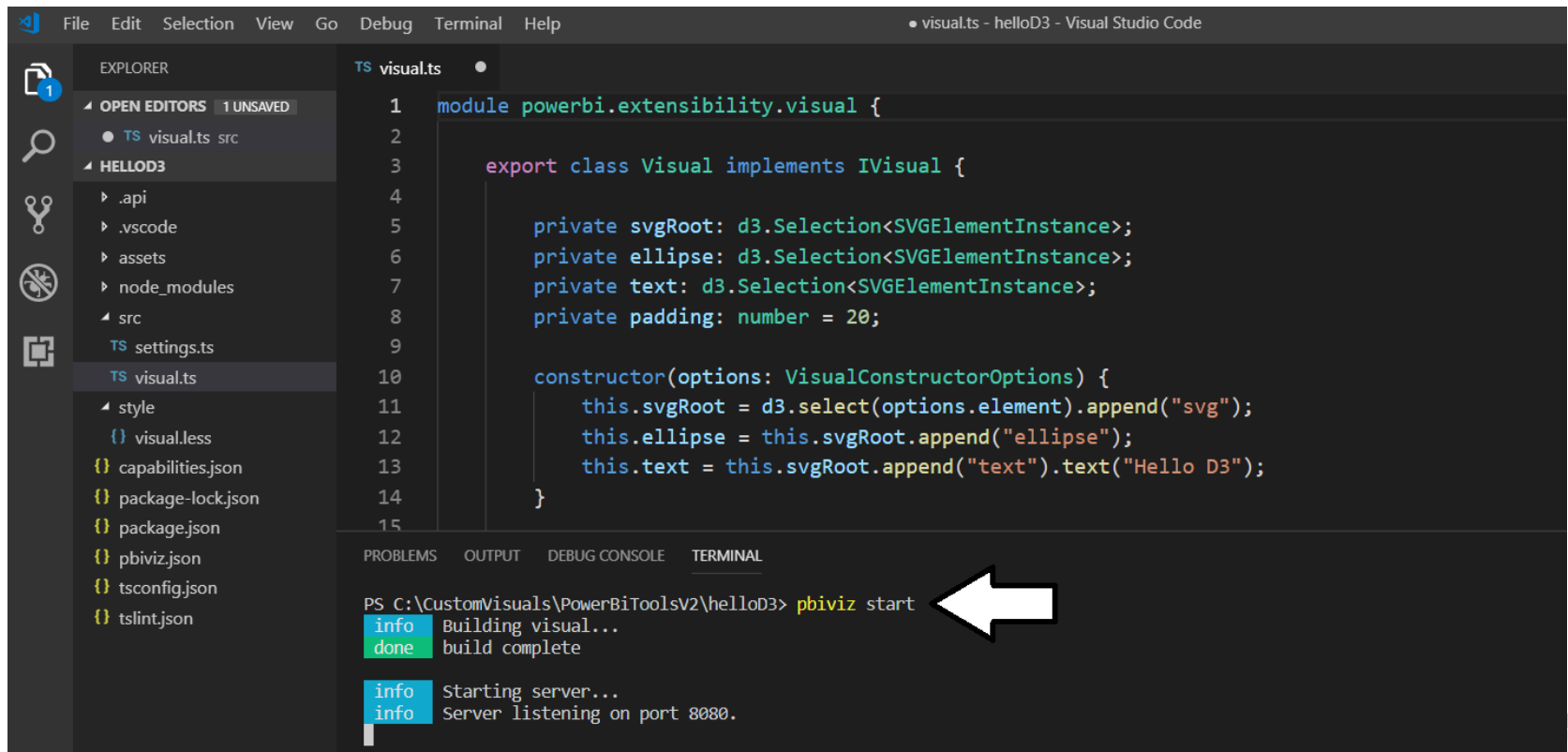


```
1 module powerbi.extensibility.visual {
2
3   export class Visual implements IVisual {
4
5     constructor(options: VisualConstructorOptions) {
6       // one-time initialization code
7     }
8
9     public update(options: VisualUpdateOptions) {
10      // called when viewport or data changes
11    }
12
13  }
14 }
```



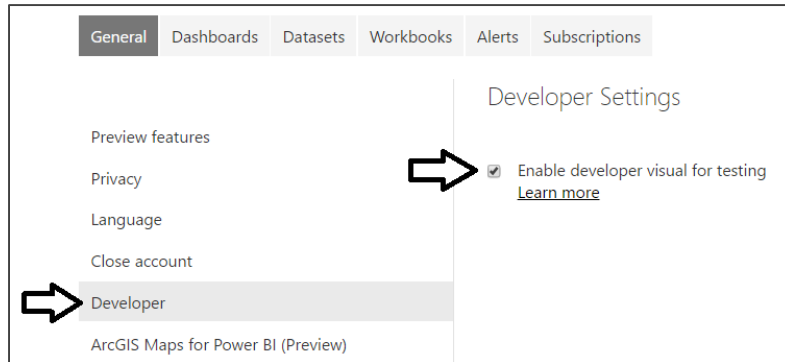
# Running a Custom Visual Project

- Visual projects run & tested using **pbviz start** command
  - Run **pbviz start** from Visual Studio Code from Integrated console
  - Command starts local debugging session in node.js

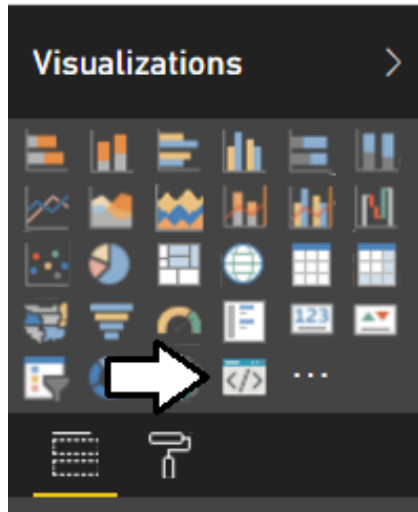


# The Developer Visual

- Must be enabled on Developer Settings page



- Provides new visual for testing and debugging custom visuals



# Working with the Developer Visual

- Developer visual loads custom visual from node.js
  - Makes it possible to test custom visual inside Power BI Service
  - Developer visual provides toolbar with development utilities





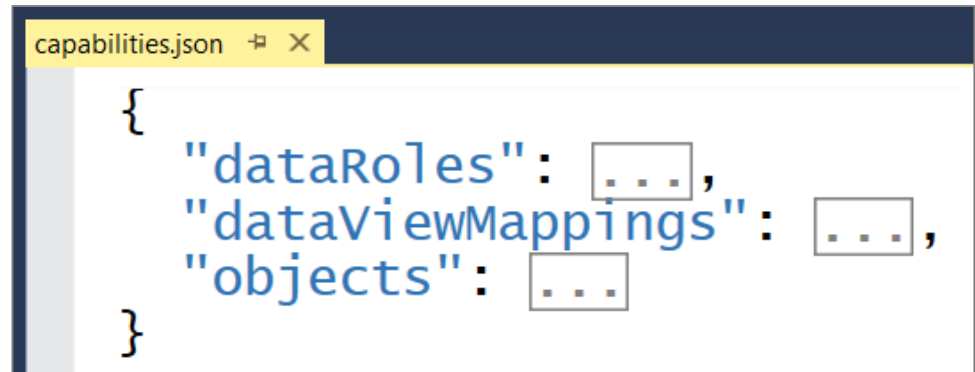
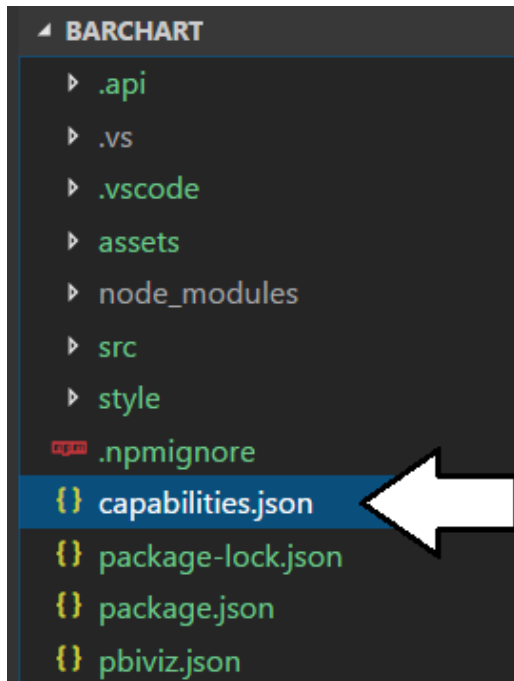
# Agenda

- ✓ Installing the Power BI Developer Tools
- ✓ Creating Your First Custom Visual
- Defining Data Roles and Data Mappings
  - Extending a Visual with Custom Properties
  - Migrating to Version 3 of the Power BI Developer Tools



# Visual Capabilities

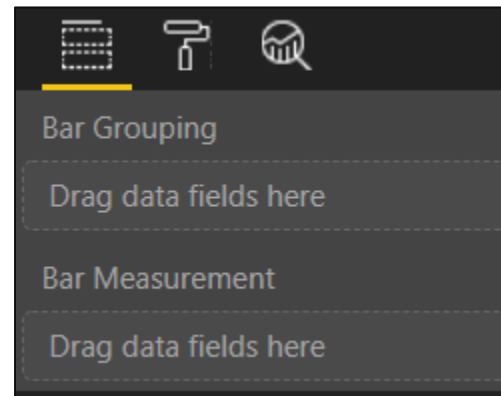
- Visual capabilities defined inside **capabilities.json**
  - **dataRoles** defines the field wells displayed on Fields pane
  - **dataViewMappings** defines the type of DataView used by visual
  - **objects** defines custom properties for visual



# Data Roles

- DataRoles define how fields are associated with visual
  - Each dataRole is display as field well in the Field pane
  - dataRoles can be defined with conditions and data mappings

```
"dataRoles": [  
  {  
    "displayName": "Bar Grouping",  
    "name": "myCategory",  
    "kind": "Grouping"  
  },  
  {  
    "displayName": "Bar Measurement",  
    "name": "myMeasure",  
    "kind": "Measure"  
  }  
]
```



# Data Mapping Modes

- Power BI visual API provides several mapping modes

- Single
- Table
- Categorical
- Matrix
- Tree

Single Mapping

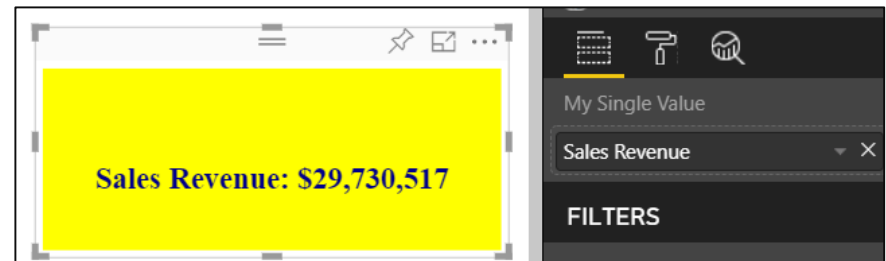
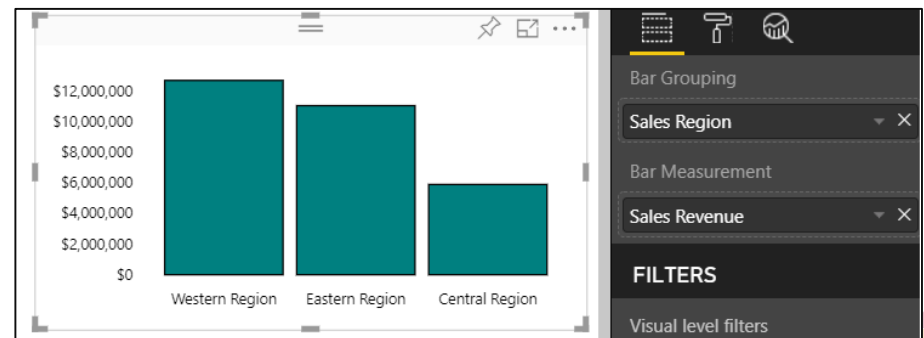


Table Mapping

A Power BI visual in Table Mapping mode. The visual is a table with three columns: "Sales Region", "Sales Revenue", and "Units Sold". The table contains three rows of data. The right-hand pane shows the configuration: "Values" is selected, and "Sales Region", "Sales Revenue", and "Units Sold" are the chosen fields. The "FILTERS" pane is empty.

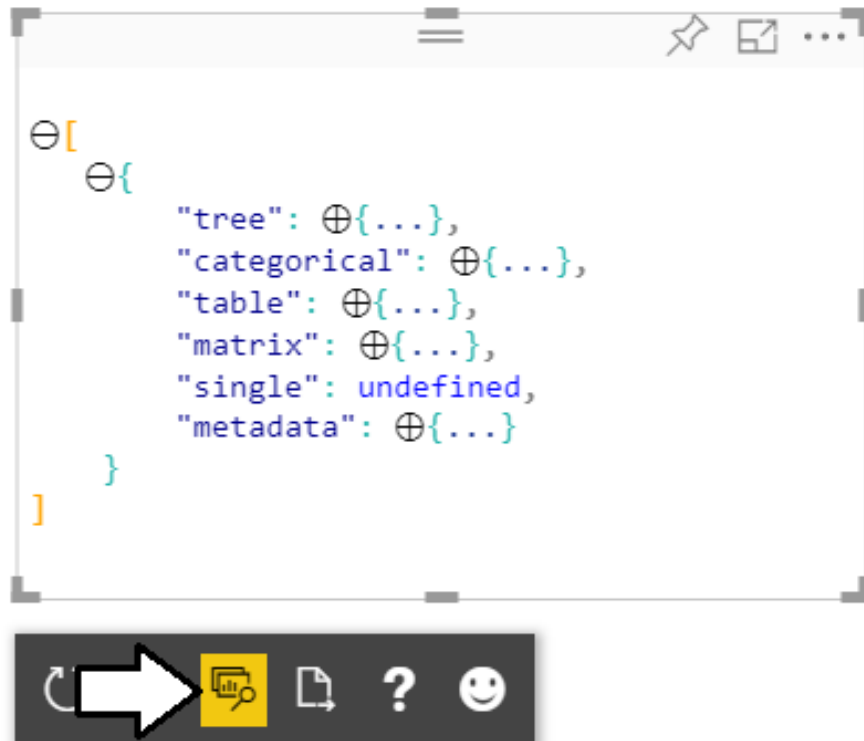
Sales Region	Sales Revenue	Units Sold
Western Region	\$12,733,888	1,598,125
Central Region	\$5,915,449	994,680
Eastern Region	\$11,081,180	1,959,240

Categorical Mapping



# Developer Visual DataView

- Developer visual supports DataView mode
  - Allows you to see and explore data mapping
  - Allows you to see metadata for custom properties



# Designing with View Model

- Best practice involves creating view model for each visual
  - View model defines data required for rendering
  - createViewModel method gets data to generate view model
  - update method calls createViewModel to get view model

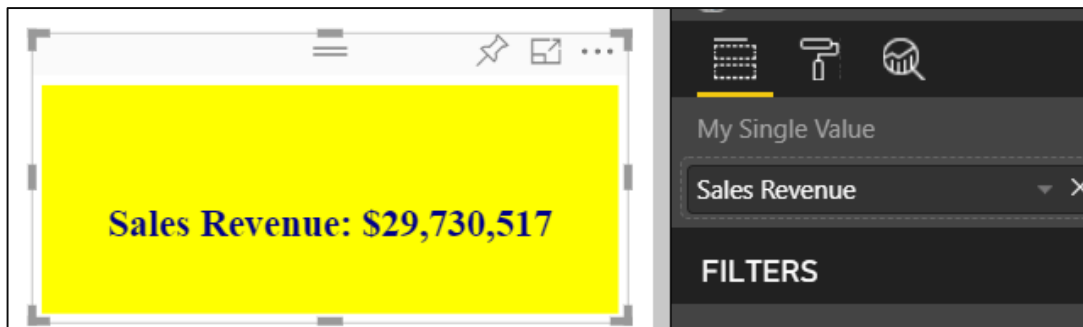
```
export interface BarchartDataPoint {  
  Category: string;  
  Value: number;  
}  
  
export interface BarchartViewModel {  
  IsValid: boolean;  
  DataPoints?: BarchartDataPoint[];  
  Format?: string;  
  SortBySize?: boolean;  
  XAxisFontSize?: number;  
  YAxisFontSize?: number;  
  BarColor?: string;  
}
```



# Single Mapping Example: oneBigNumber

- dataRole can use dataViewMapping mode of single
  - For visuals like Card which only display single value
  - Condition can define that a dataRole requires exactly one measure

```
"dataRoles": [  
  {  
    "displayName": "My single value",  
    "name": "myvalue",  
    "kind": "Measure"  
  }  
],  
"dataViewMappings": [  
  {  
    "conditions": [ { "myvalue": { "min": 1, "max": 1 } } ],  
    "single": { "role": "myvalue" }  
  }  
]
```





# Programming in Single Mapping Mode

- Single mapping easy to access through visuals API
  - DataView object provides single.value property
  - value property defined as PrimitiveValue { bool | number | string }
  - PrimitiveValue must be explicitly cast
  - Other measure properties available through column metadata

```
"tree": ⊕{...},
"categorical": ⊕{...},
"table": ⊕{...},
"matrix": ⊕{...},
"single": ⊖{
  "column": ⊕{...},
  "value": 29730517.14
},
"metadata": ⊖{
  "columns": ⊖[
    ⊖{
      "roles": ⊕{...},
      "type": ⊕{...},
      "format": "\\$#,0;(\\$#,0);\\$#,0",
      "displayName": "Sales Revenue",
      "queryName": "Sales.Sales Revenue",
      "expr": ⊕{...},
      "index": 0,
      "isMeasure": true
    }
  ]
}
```

```
public update(options: VisualUpdateOptions) {
  // get DataView object
  this.dataView = options.dataViews[0];

  // get single value
  var value: number = <number>this.dataView.single.value;

  // get metadata to discover field name and format string
  var column: DataViewMetadataColumn = this.dataView.metadata.columns[0];
  var valueName: string = column.displayName
  var valueFormat: string = column.format;
```



# Using the Power BI Formatting Utilities

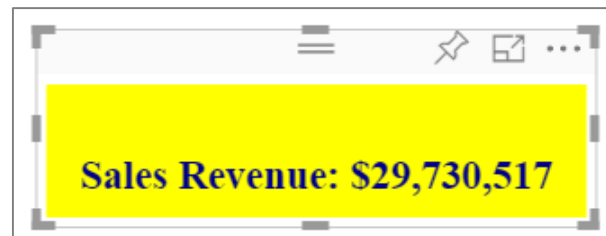
- Used to format values using Power BI formatting strings
  - Requires installing powerbi-visuals-utils-formattingutils package

```
var value: number = <number>this.dataView.single.value;
var column: DataViewMetadataColumn = this.dataView.metadata.columns[0];
var valueName: string = column.displayName
var valueFormat: string = column.format;

var valueFormatterFactory = powerbi.extensibility.utils.formatting.valueFormatter;
var valueFormatter = valueFormatterFactory.create({
    format: valueFormat,
    formatSingleValues: true
});

var valueString: string = valueFormatter.format(value);
```

```
"column": {
  "roles": [...],
  "type": [...],
  "format": "\\$#,0;(\\$#,0);\\$#,0",
  "displayName": "Sales Revenue",
  "queryName": "Sales.Sales Revenue",
```



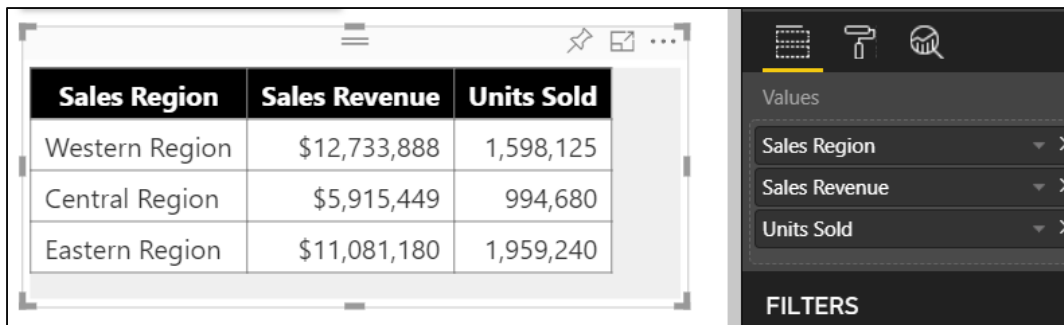
```
"column": {
  "roles": [...],
  "type": [...],
  "format": "#,0",
  "displayName": "Units Sold",
  "queryName": "Sales.Units Sold",
```



# Table Mapping Example: Snazzy Table

- dataRole can use dataViewMapping mode of table
  - For visuals which display rows & columns for ordered set of fields
  - condition can define number of fields that can be added

```
"dataRoles": [  
  {  
    "displayName": "values",  
    "name": "values",  
    "kind": "GroupingOrMeasure"  
  }  
],  
"dataViewMappings": [  
  {  
    "conditions": [ { "values": { "min": 1, "max": 5 } } ],  
    "table": { "rows": { "for": { "in": "values" } } }  
  }  
]
```



The screenshot displays a Power BI report interface. On the left, a table visual is shown with three columns: Sales Region, Sales Revenue, and Units Sold. The table contains three rows of data. On the right, the filter pane is visible, showing the 'Values' section with three filters applied: Sales Region, Sales Revenue, and Units Sold. The filter pane also includes a 'FILTERS' section at the bottom.

Sales Region	Sales Revenue	Units Sold
Western Region	\$12,733,888	1,598,125
Central Region	\$5,915,449	994,680
Eastern Region	\$11,081,180	1,959,240



# Programming in Table Mapping Mode

- Table mapping data accessible through visuals API
  - DataView object provides table property
  - table property provides columns property and rows property

```
"table": ⊕{  
  "columns": ⊕[  
    ⊕{  
      "roles": ⊕{...},  
      "type": ⊕{...},  
      "format": undefined,  
      "displayName": "Sales Region",  
      "queryName": "Customers.Sales Region",  
      "expr": ⊕{...},  
      "index": 0,  
      "identityExprs": ⊕[ ... ]  
    },  
    ⊕{...},  
    ⊕{...}  
  ],  
  "identity": ⊕[ ... ],  
  "identityFields": ⊕[ ... ],  
  "rows": ⊕[  
    ⊕[  
      "Western Region",  
      12733888.2,  
      1598125
```

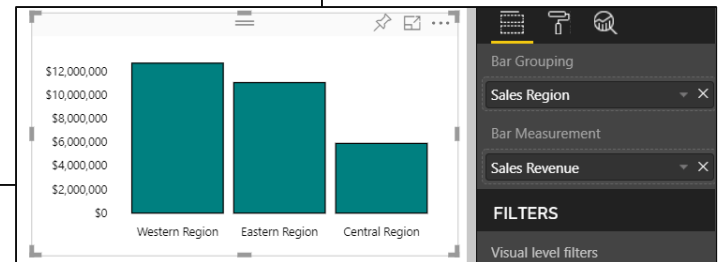
```
public update(options: VisualUpdateOptions) {  
    var dataView: DataView = options.dataViews[0];  
    var table: DataViewTable = dataView.table;  
    var columns: DataViewMetadataColumn[] = table.columns;  
    var rows: DataViewTableRow[] = table.rows;
```



# Categorical Mapping Example: Barchart

- dataRole can use dataViewMapping mode of categorical
  - This is the most common type of data mapping
  - For visuals which divide data into groups for analysis
  - Groups defined as columns and values defined as measures

```
"dataRoles": [  
  { "displayName": "Bar Grouping", "name": "myCategory", "kind": "Grouping" },  
  { "displayName": "Bar Measurement", "name": "myMeasure", "kind": "Measure" }  
],  
"dataViewMappings": [  
  {  
    "conditions": [ { "myCategory": { "max": 1 }, "myMeasure": { "max": 1 } } ],  
    "categorical": {  
      "categories": {  
        "for": { "in": "myCategory" },  
        "dataReductionAlgorithm": { "top": {} }  
      },  
      "values": {  
        "select": [ { "bind": { "to": "myMeasure" } } ]  
      }  
    }  
  }  
]
```



# Agenda

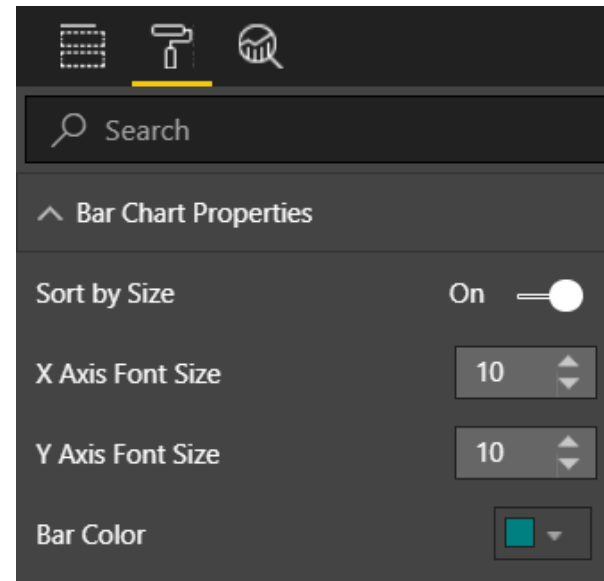
- ✓ Installing the Power BI Developer Tools
- ✓ Creating Your First Custom Visual
- ✓ Defining Data Roles and Data Mappings
- Extending a Visual with Custom Properties
- Migrating to Version 3 of the Power BI Developer Tools



# Extending Visuals with Custom Properties

- Custom properties defined using **objects**
  - You can define one or more objects in **capabilities.json**
  - Each object defined with name, display name and properties
  - object properties automatically persistent inside visual metadata
  - properties can be seen and modified by user in Format pane
  - Custom properties require extra code to initialize Format pane

```
"objects": {  
  "barchartProperties": {  
    "displayName": "Bar Chart Properties",  
    "properties": {  
      "sortBySize": {  
        "displayName": "Sort by Size",  
        "type": { "bool": true }  
      },  
      "xAxisFontSize": {  
        "displayName": "X Axis Font Size",  
        "type": { "integer": true }  
      },  
      "yAxisFontSize": {  
        "displayName": "Y Axis Font Size",  
        "type": { "integer": true }  
      },  
      "barColor": {  
        "displayName": "Bar Color",  
        "type": { "fill": { "solid": { "color": true } } }  
      }  
    }  
  }  
}
```





# DataViewObjectParser and VisualSettings

- Power BI visual utilities provide DataViewObjectParser
  - Abstracts away tricky code to initialize and read property values

```
TS settings.ts •
module powerbi.extensibility.visual {

  import DataViewObjectsParser = powerbi.extensibility.utils.dataview.DataViewObjectsParser;

  export class VisualSettings extends DataViewObjectsParser {
    public barchartProperties: BarchartProperties = new BarchartProperties();
  }

  export class BarchartProperties {
    sortBySize: boolean = true;
    xAxisFontSize: number = 10;
    yAxisFontSize: number = 10;
    barColor: Fill = { "solid": { "color": "teal" } };
  }
}
```



# Mapping Object Properties to VisualSettings

- VisualSettings class must map to named objectnamed
  - VisualSetting class contains named field that maps to object name
  - Named field based on custom class with mapped properties
  - Object & property names must match what's in capabilities.json

```
"objects": {  
  "barchartProperties": {  
    "displayName": "Bar Chart Properties",  
    "properties": {  
      "sortBySize": {  
        "displayName": "Sort by Size",  
        "type": { "bool": true }  
      },  
      "xAxisFontSize": {  
        "displayName": "X Axis Font Size",  
        "type": { "integer": true }  
      },  
      "yAxisFontSize": {  
        "displayName": "Y Axis Font Size",  
        "type": { "integer": true }  
      },  
      "barColor": {  
        "displayName": "Bar Color",  
        "type": { "fill": { "solid": { "color": "#FFFFFF" } } }  
      }  
    }  
  }  
}
```

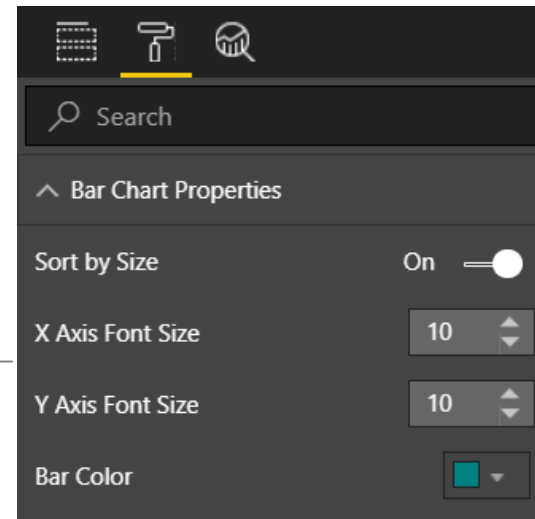
```
export class VisualSettings extends DataModel {  
  public barchartProperties: BarchartProperties;  
}  
  
export class BarchartProperties {  
  sortBySize: boolean = true;  
  xAxisFontSize: number = 10;  
  yAxisFontSize: number = 10;  
  barColor: Fill = { "solid": { "color": "#FFFFFF" } }  
}
```



# Initializing Objects in the Format Pane

- Visual must initialize properties in Format pane
  - Visual must implement enumerateObjectInstances
  - VisualSettings makes this relatively easy
  - Extra code required to make property appear as spinner

```
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration {  
    // register object properties  
    var visualObjects: VisualObjectInstanceEnumerationObject =  
        <VisualObjectInstanceEnumerationObject>VisualSettings  
            .enumerateObjectInstances(this.settings, options);  
  
    // configure spinners for integers properties  
    visualObjects.instances[0].validValues = {  
        xAxisFontSize: { numberRange: { min: 10, max: 36 } },  
        yAxisFontSize: { numberRange: { min: 10, max: 36 } },  
    };  
  
    // return visual object collection  
    return visualObjects;  
}
```



# Retrieving Property Values

- Property values persisted into visual metadata
  - Properties not persisted while they still retain default values

```
"tree": ⊕{...},
"categorical": ⊕{...},
"table": ⊕{...},
"matrix": ⊕{...},
"single": undefined,
"metadata": ⊖{
  "columns": ⊕[ ... ],
  "objects": ⊖{
    "barchartProperties": ⊖{
      "sortBySize": false,
      "xAxisFontSize": 14
    }
  }
}
```

- Property values retrieved using VisualSettings object

```
public update(options: VisualUpdateOptions) {
  if (options.dataViews[0]) {
    // create VisualSettings object
    this.settings = VisualSettings.parse(options.dataViews[0]) as VisualSettings;

    // retrieve property values
    var sortBySize: boolean = this.settings.barchartProperties.sortBySize
    var xAxisFontSize: number = this.settings.barchartProperties.xAxisFontSize;
```



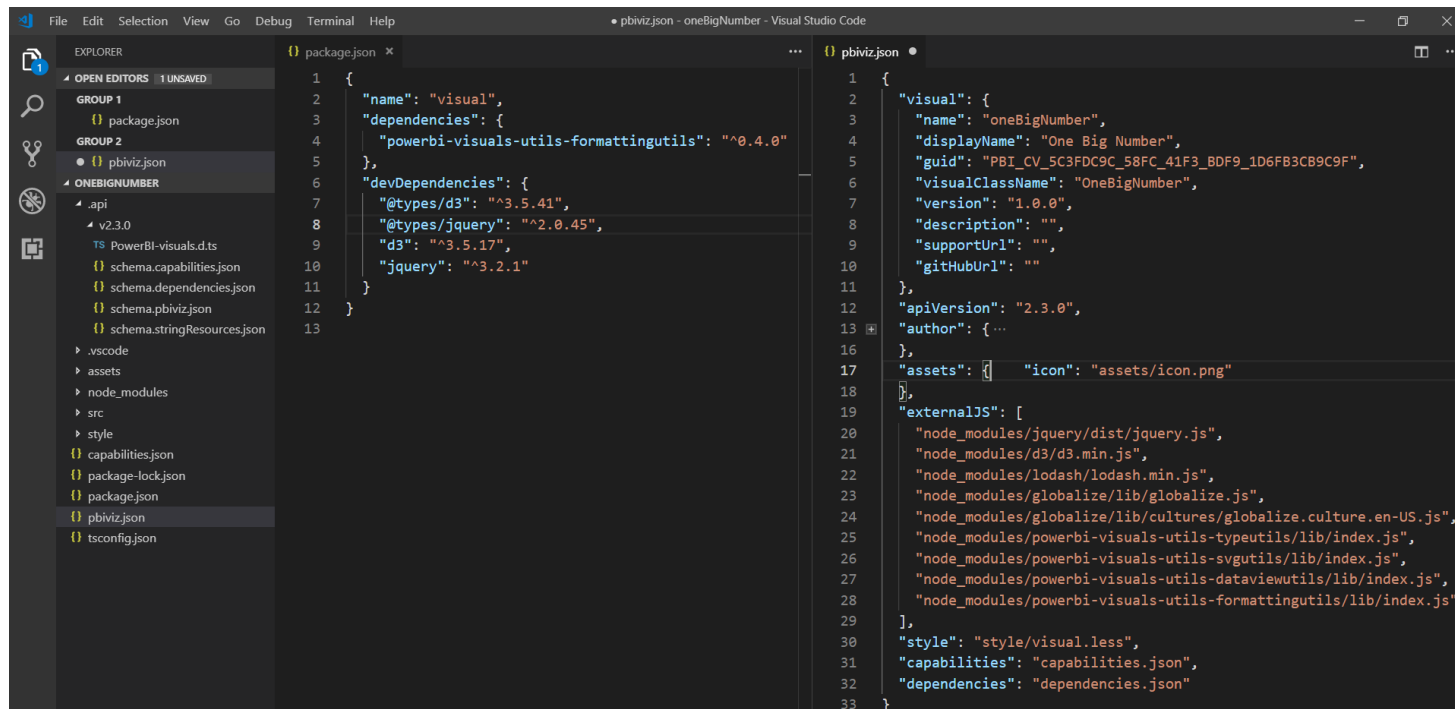
# Agenda

- ✓ Installing the Power BI Developer Tools
- ✓ Creating Your First Custom Visual
- ✓ Defining Data Roles and Data Mappings
- ✓ Extending a Visual with Custom Properties
- Migrating to Version 3 of the Power BI Developer Tools



# Tools v2

- Power BI API added to your project
  - Power BI API files added to project
  - You install packages for utilities
  - Add externalJS entries for JavaScript libraries required on page



The screenshot shows the Visual Studio Code interface with the Explorer sidebar on the left and the pbiviz.json file open in the editor. The Explorer sidebar shows the project structure with folders like .api, v2.3.0, and .vscode. The pbiviz.json file contains the following content:

```
1 {
2   "name": "visual",
3   "dependencies": {
4     "powerbi-visuals-utils-formattingutils": "^0.4.0"
5   },
6   "devDependencies": {
7     "@types/d3": "^3.5.41",
8     "@types/jquery": "^2.0.45",
9     "d3": "^3.5.17",
10    "jquery": "^3.2.1"
11  }
12 }
13 }
```

The pbiviz.json file also contains the following content:

```
1 {
2   "visual": {
3     "name": "oneBigNumber",
4     "displayName": "One Big Number",
5     "guid": "PBI_CV_5C3FDC9C_58FC_41F3_BDF9_1D6FB3CB9C9F",
6     "visualClassName": "OneBigNumber",
7     "version": "1.0.0",
8     "description": "",
9     "supportUrl": "",
10    "githubUrl": ""
11  },
12  "apiVersion": "2.3.0",
13  "author": {
14    "name": "One Big Number",
15    "email": "onebignumber@outlook.com",
16    "url": "https://github.com/onebignumber/onebignumber"
17  },
18  "assets": [
19    { "icon": "assets/icon.png" }
20  ],
21  "externalJS": [
22    "node_modules/jquery/dist/jquery.js",
23    "node_modules/d3/d3.min.js",
24    "node_modules/lodash/lodash.min.js",
25    "node_modules/globalize/lib/globalize.js",
26    "node_modules/globalize/lib/cultures/globalize.culture.en-US.js",
27    "node_modules/powerbi-visuals-utils-typeutils/lib/index.js",
28    "node_modules/powerbi-visuals-utils-svgutils/lib/index.js",
29    "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.js",
30    "node_modules/powerbi-visuals-utils-formattingutils/lib/index.js"
31  ],
32  "style": "style/visual.less",
33  "capabilities": "capabilities.json",
34  "dependencies": "dependencies.json"
35 }
```



# EcmaScript2015 Modules and D3 version 5

- ECMAScript 2015 add modules to JavaScript
  - TypeScript builds on the concept
  - Each file defines its own module
- Modules execute in their own scope not at global scope
  - Code in module not visible to other modules by default
  - Classes and function must be exported to use across modules
  - Modules must import types from other modules
  - relationships between modules defined using imports and exports



# Dynamic Module Loading

- Webpack controls dynamic module loading
  - Your project just references app.ts
  - Compiler dynamically determines other files to include

```
TS app.ts x
import { Quote } from './quote';
import { QuoteManager } from './quote-manager';

$( () => {

  var displayNewQuote = (): void => {
    var quote: Quote = QuoteManager.getQuote();
    $("#quote").text(quote.value);
    $("#author").text(quote.author);
  }
});
```

```
TS quote.ts •
1 export class Quote {
2   value: string;
3   author: string;
4   constructor(value: string, author: string){
5     this.value = value;
6     this.author = author;
7   }
8 }
```

```
TS quote-manager.ts x
1 import { Quote } from './quote';
2
3 export class QuoteManager {
4
5   private static quotes: Quote[] = [
6     new Quote("Always borrow money from a pal", "John D. Rockefeller"),
7     new Quote("Behind every great man is a woman", "William Pittman"),
8     new Quote("In Hollywood a marriage is a business", "Marilyn Monroe")
9   ];
10 }
```





# WebPack

- WebPack serves as a bundling utility
  - Bundles many js/ts files into a single file
  - Can handle dynamic module loading
  - Provides a dev server for testing and debugging
- When using Webpack version 4 or later
  - Install packages for webpack and webpack-cli
    - `npm install webpack webpack-cli --save-dev`



# Webpack Loaders

- Loaders do two things
  - Identify which file or files should be transformed
  - Transform files and add them to dependency graph
- Example loaders
  - awesome-typescript-loader
  - style-loader
  - css-loader
  - url-loader



# Summary

- ✓ Installing the Power BI Developer Tools
- ✓ Creating Your First Custom Visual
- ✓ Defining Data Roles and Data Mappings
- ✓ Extending a Visual with Custom Properties
- ✓ Migrating to Version 3 of the Power BI Developer Tools

