

Developing and Distributing Custom Visuals



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/>



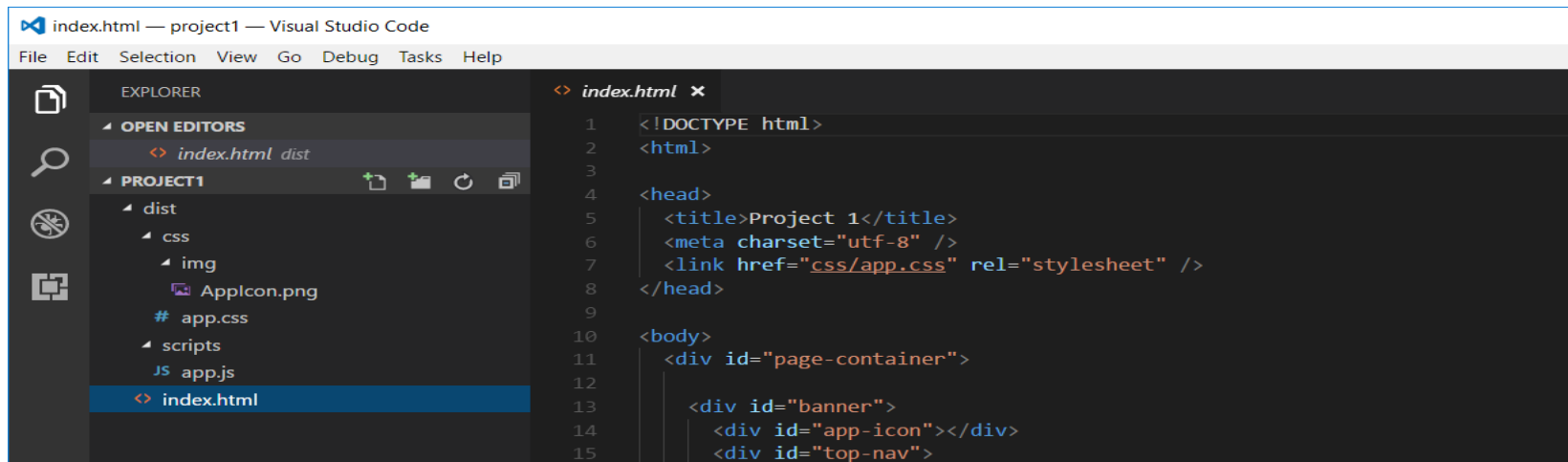
Install Visual Studio Code

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



Developing with Visual Studio Code

- Node.js is agnostic when it comes to developer IDE
 - There are many different IDEs that people use with Node.js
 - This course will be using Visual Studio Code

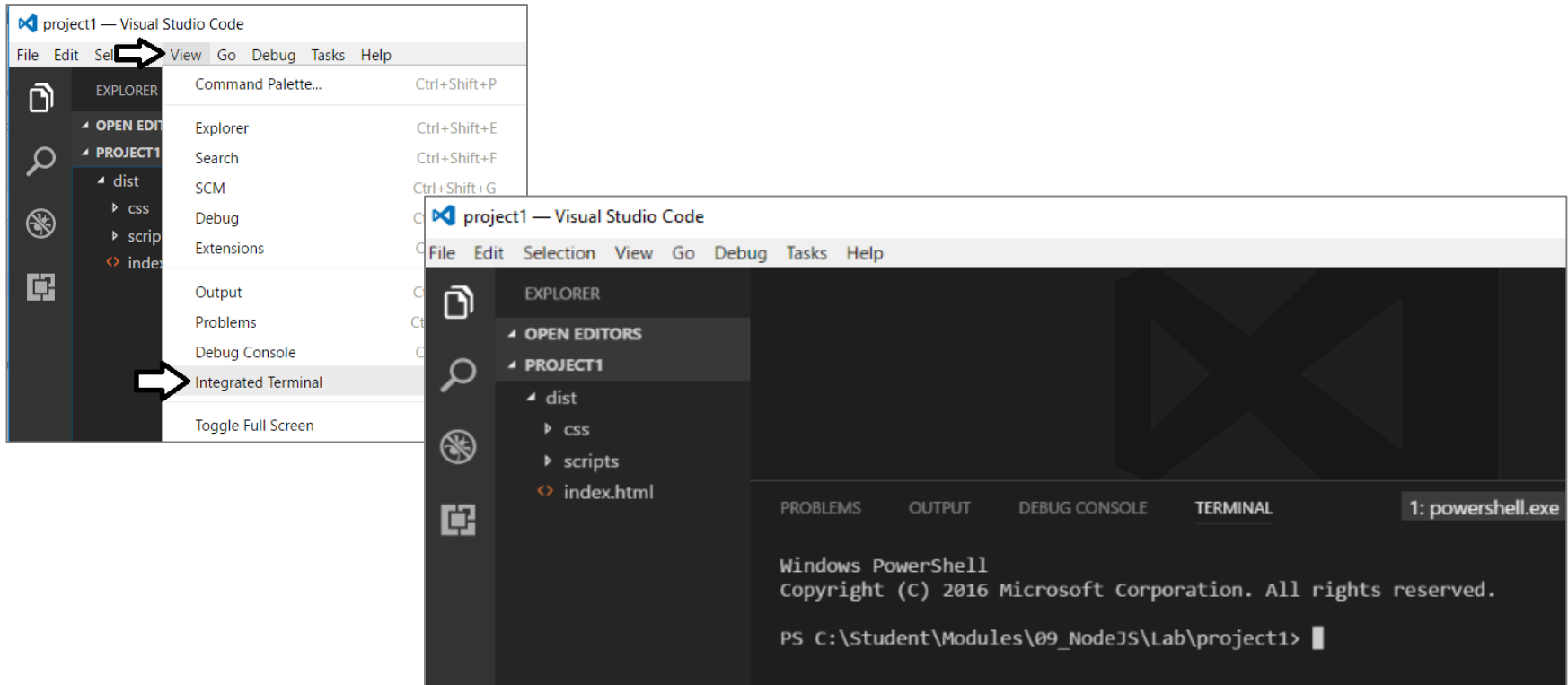


- Visual Studio is not a good fit for Node.js development
 - Visual Studio solution & project files incompatible with Node.js



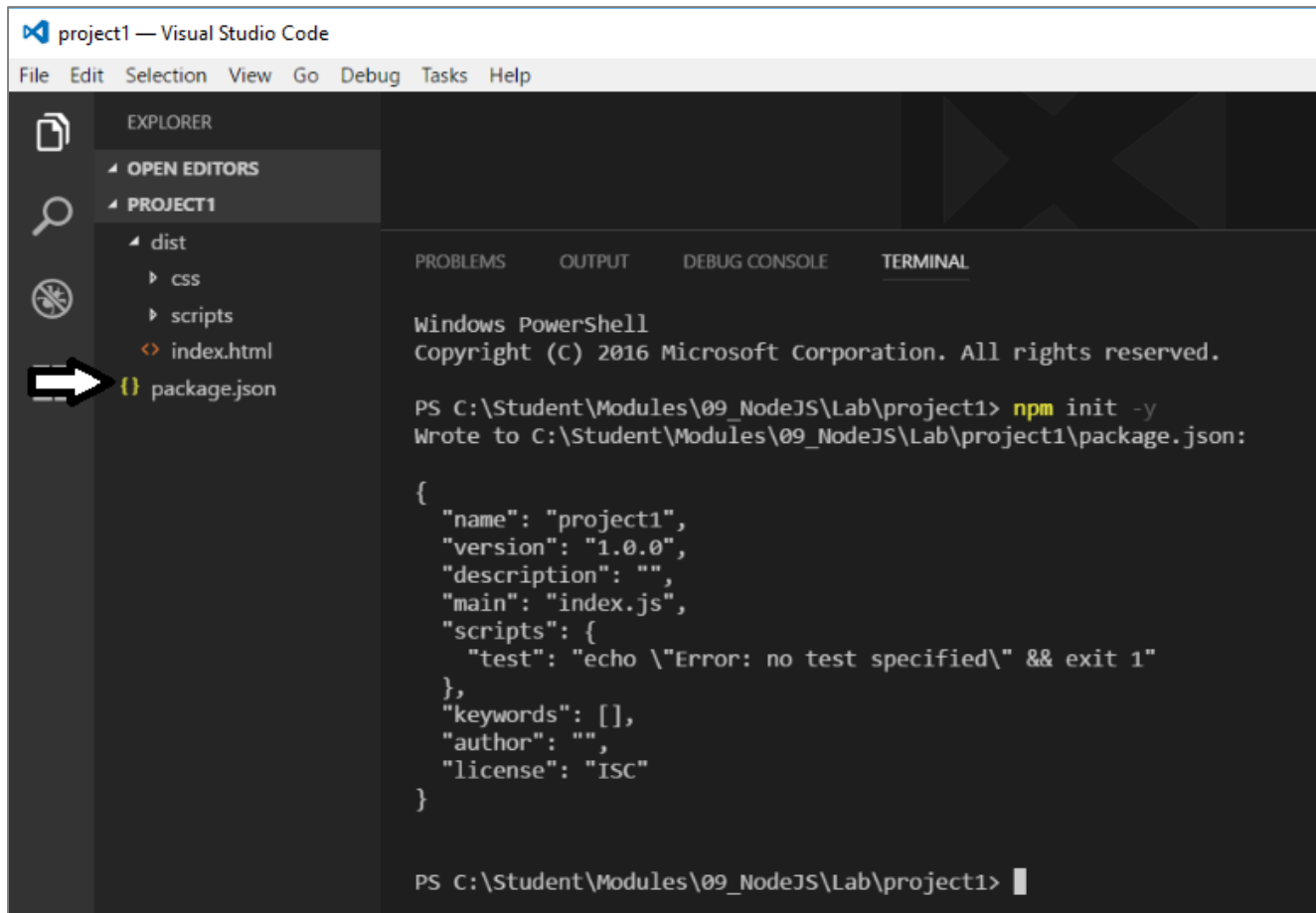
Integrated Terminal

- Use the Integrated Terminal to execute `npm` command



npm init

- Node.js projects initialized with `npm init` command
 - This command created the **package.json** file



The screenshot shows the Visual Studio Code interface for a project named 'project1'. The Explorer sidebar on the left shows the file structure: 'dist' (containing 'css' and 'scripts'), 'index.html', and 'package.json'. A white arrow points to 'package.json', indicating it is the file created by the command. The TERMINAL pane on the right shows the command prompt output of the 'npm init -y' command, which wrote a default package.json file to the project directory. The output of the command is as follows:

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Student\Modules\09_NodeJS\Lab\project1> npm init -y
Wrote to C:\Student\Modules\09_NodeJS\Lab\project1\package.json:

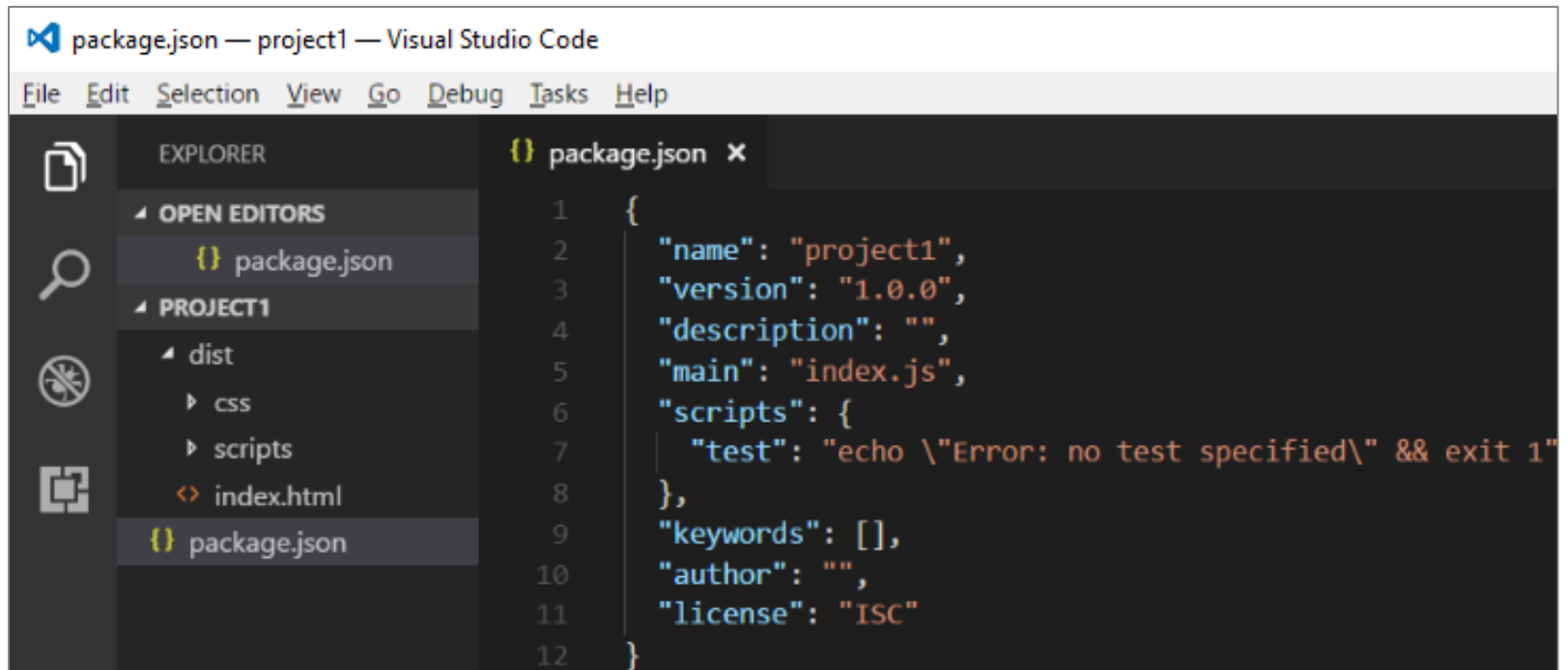
{
  "name": "project1",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

The terminal prompt is now at 'PS C:\Student\Modules\09_NodeJS\Lab\project1> '.



package.json

- **package.json** serves as project manifest file
 - Tracks project name and version number
 - Tracks installed package dependencies



The screenshot shows the Visual Studio Code interface with the `package.json` file open in the Editor. The Explorer on the left shows the project structure with `package.json` selected. The Editor displays the following JSON content:

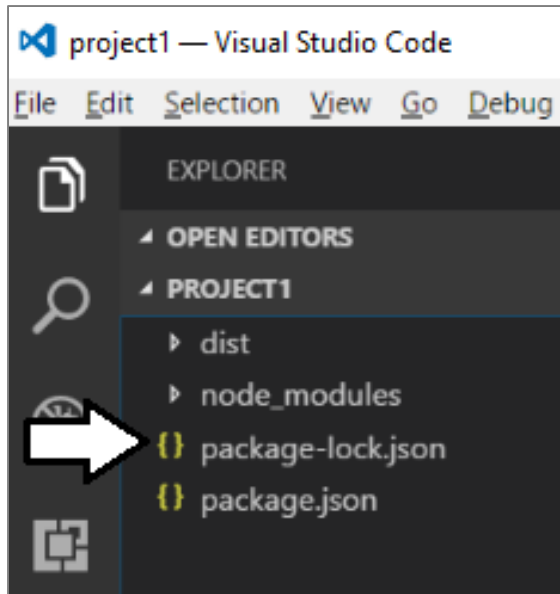

```
1 {  
2   "name": "project1",  
3   "version": "1.0.0",  
4   "description": "",  
5   "main": "index.js",  
6   "scripts": {  
7     "test": "echo \"Error: no test specified\" && exit 1"  
8   },  
9   "keywords": [],  
10  "author": "",  
11  "license": "ISC"  
12 }
```



Installing Packages

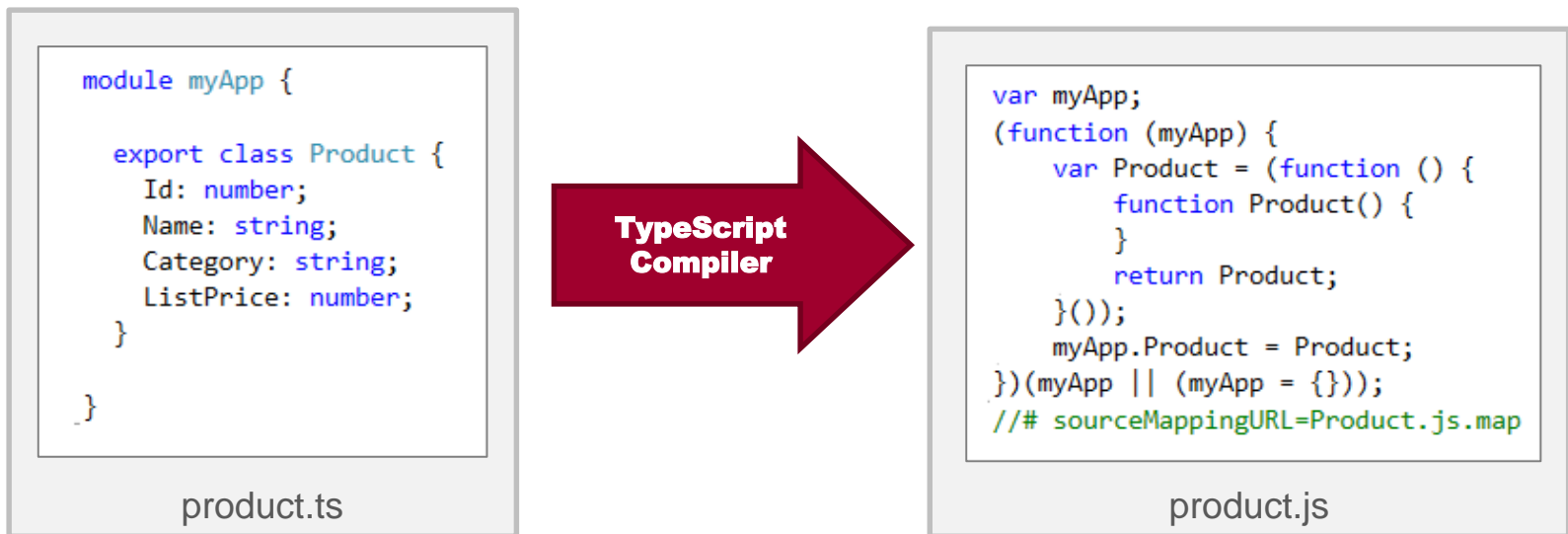
```
npm install browser-sync --save-dev
```

```
"devDependencies": {  
  "browser-sync": "^2.18.12"  
}
```



What is TypeScript?

- A programming language which compiles into plain JavaScript
- A superset of JavaScript that adds a strongly-typed dimension
- It can be compiled into ECMAScript3, ECMAScript3 or ECMAScript 6
- It runs in any browser, in any host and on any OS



Type Annotation

- TypeScript allows you to annotate types
 - Provides basis for strongly-typed programming
 - Type annotations used by compiler for type checking
 - Type annotations are erased at the end of compile time

```
// define strongly-typed function
var myFunction = function (param1: number): string {
  | return "You passed " + param1;
};

// define strongly-typed variables
var myNumber: number = 2017;
var myMessage: string = myFunction(myNumber);
var myContent: JQuery = $("<p>").text(myMessage);
var contentBox: JQuery = $("#content-box");

contentBox.empty().append(myContent);
```



Arrow Function Syntax

- TypeScript supports arrow function syntax
 - Concise syntax to define anonymous functions
 - Can be used to retain this pointer in classes

```
// create anonymous function using function arrow syntax
let myFunction = () => {
  console.log("Hello world");
};

// use function arrow syntax with typed parameters
let myOtherFunction = (param1: number, param2: string) : string => {
  return param1 + " - " + param2;
};

// create function to assign to DOM event
window.onresize = (event: Event) => {
  let window: Window = event.target as Window;
  console.log("Window width: " + window.outerWidth);
  console.log("Window height: " + window.outerHeight);
};
```



Classes

- TypeScript supports defining classes
 - Class defines type for object
 - Export keyword makes class created across files
 - Class can be passed as factory function
 - Default accessibility is public

```
export class Product {  
  Id: number;  
  Name: string;  
  Category: string;  
  ListPrice: number;  
}
```

```
// create new Product instance  
let product1: Product = new Product();  
product1.Id = 1;  
product1.Name = "Batman Action Figure";  
product1.Category = "Action Figure";  
product1.ListPrice = 14.95;
```



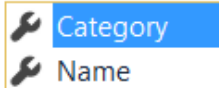
Class Constructors

- Constructor parameters become fields in class

```
export class Product {  
  
  constructor(private Id: number, public Name: string, public Category: string, private ListPrice: number) {  
    // no need to do anything here  
  }  
  
  MyPublicMethod() {  
    // access to private fields  
    let id: number = this.Id  
    let price: number = this.ListPrice  
  }  
  
}
```

- Client-side code calls constructor using new operator

```
// create new Product instance  
let product1: Product = new Product(1, "Batman Action Figure", "Action Figure", 14.95);  
  
// access public properties  
let product1Name: string = product1.Name;  
let product1Category: string = product1.
```



Interfaces

- Interface defines a programming contract
 - Classes can implement interfaces

```
export interface IProductDataService {  
  GetAllProducts(): Product[];  
  GetProduct(id: number): Product;  
  AddProduct(product: Product): void;  
  DeleteProduct(id: number): void;  
  UpdateProduct(product: Product): void;  
}
```

```
export class MyProductDataService implements IProductDataService {  
  
  private products: Product[] = [];  
  
  GetAllProducts(): Product[] {  
    return this.products;  
  }  
  
  GetProduct(id: number): Product {  
    return this.products.find(p => p.id === id);  
  }  
  
  AddProduct(product: Product): void {  
    this.products.push(product);  
  }  
  
  DeleteProduct(id: number): void {  
    this.products = this.products.filter(p => p.id !== id);  
  }  
  
  UpdateProduct(product: Product): void {  
    const index = this.products.findIndex(p => p.id === product.id);  
    if (index !== -1) {  
      this.products[index] = product;  
    }  
  }  
}
```

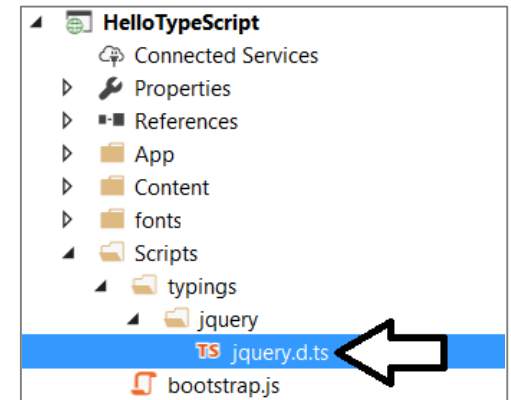
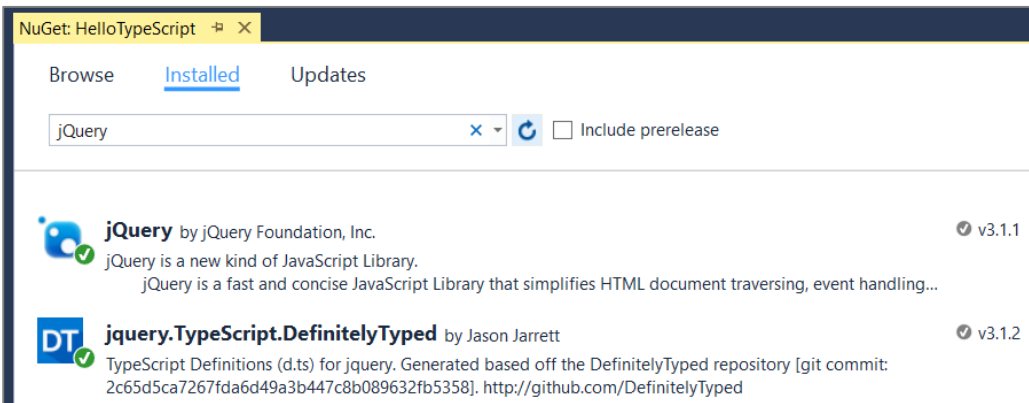
- Client code can be decoupled from concrete classes

```
// program against variables based on interface type  
let productService: IProductDataService = new MyProductDataService();  
  
// client code is decoupled from underlying data access class implementations  
let products: Product[] = productService.GetAllProducts();  
let product1: Product = productService.GetProduct(1);
```



TypeScript Definition Files (d.ts)

- What are TypeScript definition files
 - Typed definitions for 3rd party JavaScript libraries
 - DefinitelyTyped provides great community resource
 - Typed definition files have a **d.ts** extension

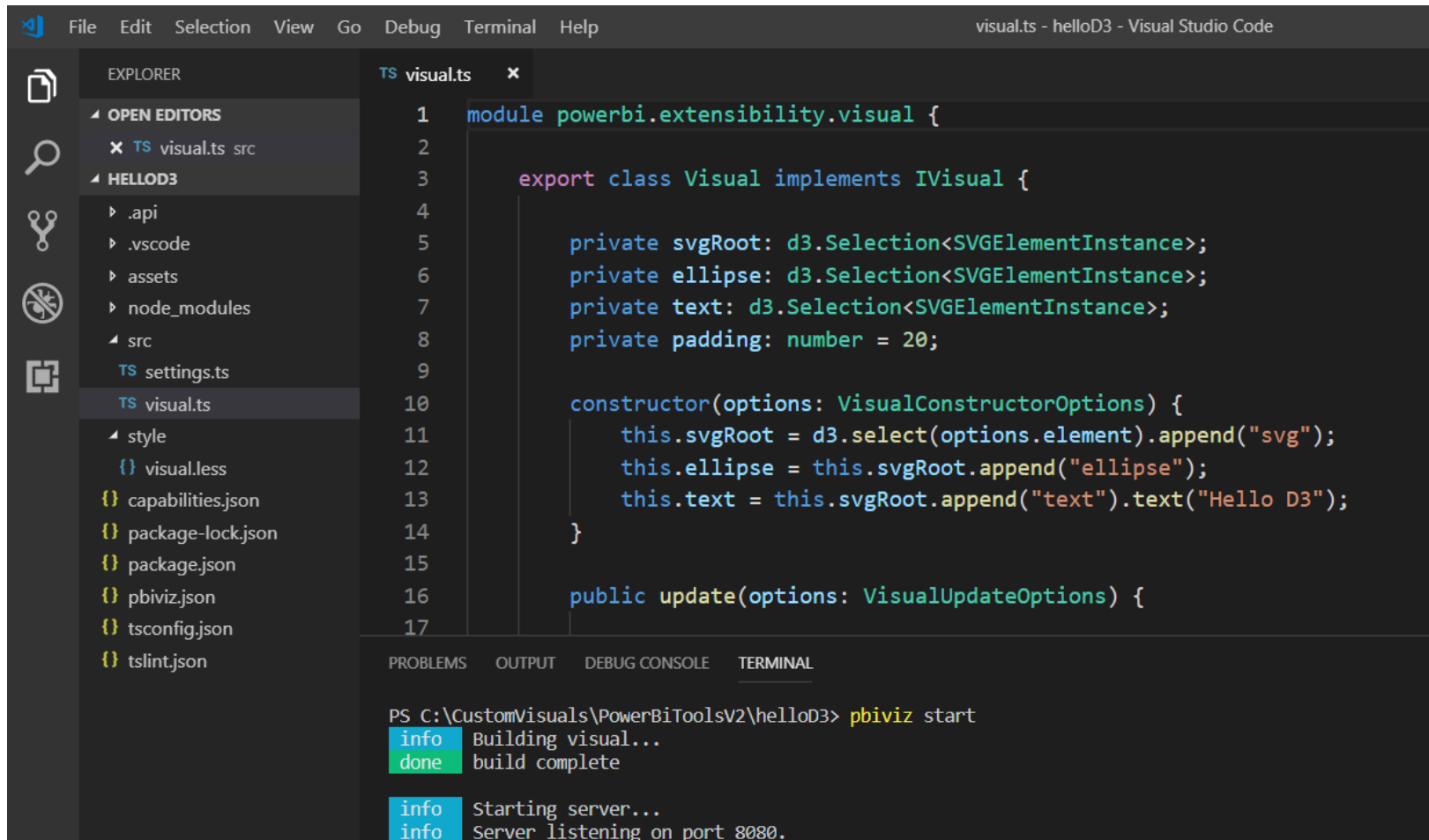


```
// define strongly-typed variables
var myNumber: number = 2017;
var myMessage: string = myFunction(myNumber);
var myContent: JQuery = $("<p>").text(myMessage);
var contentBox: JQuery = $("#content-box");
```



Developing with Visual Studio Code

- Provides great development experience with node.js



The screenshot displays the Visual Studio Code editor interface. The top menu bar includes File, Edit, Selection, View, Go, Debug, Terminal, and Help. The Explorer sidebar on the left shows the project structure for 'helloD3', including files like .api, .vscode, assets, node_modules, src, settings.ts, visual.ts, style, visual.less, capabilities.json, package-lock.json, package.json, pbiviz.json, tsconfig.json, and tslint.json. The main editor area shows the 'visual.ts' file with the following TypeScript code:

```
1 module powerbi.extensibility.visual {
2
3   export class Visual implements IVisual {
4
5     private svgRoot: d3.Selection<SVGElementInstance>;
6     private ellipse: d3.Selection<SVGElementInstance>;
7     private text: d3.Selection<SVGElementInstance>;
8     private padding: number = 20;
9
10    constructor(options: VisualConstructorOptions) {
11      this.svgRoot = d3.select(options.element).append("svg");
12      this.ellipse = this.svgRoot.append("ellipse");
13      this.text = this.svgRoot.append("text").text("Hello D3");
14    }
15
16    public update(options: VisualUpdateOptions) {
17
```

At the bottom, the Terminal window shows the following output:

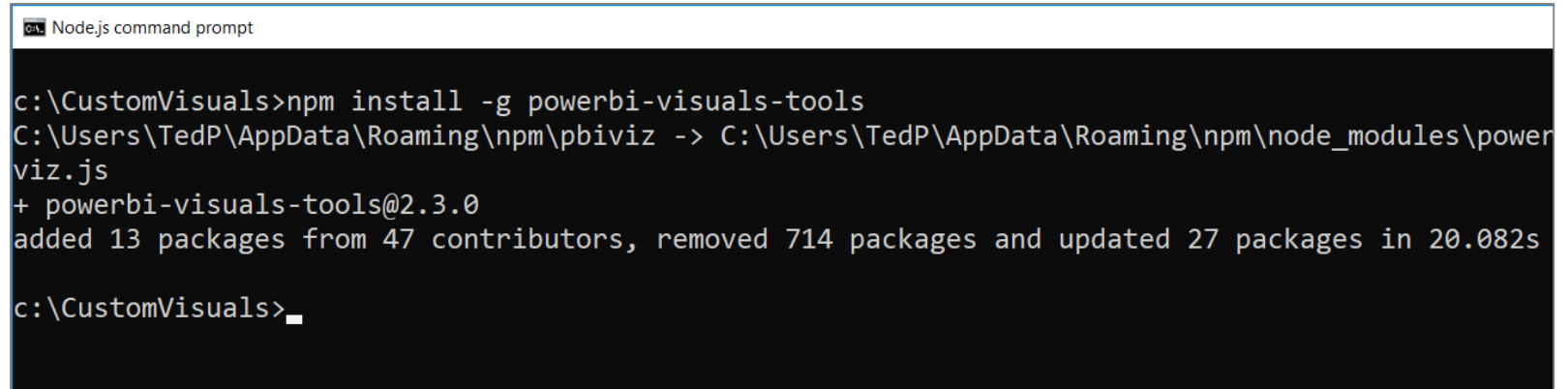
```
PS C:\CustomVisuals\PowerBiToolsV2\helloD3> pbiviz start
info Building visual...
done build complete

info Starting server...
info Server listening on port 8080.
```



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

```

Select Node.js command prompt

c:\Student>pbiviz --help

Usage: pbiviz [options] [command]

Options:

  -V, --version          output the version number
  --install-cert         Creates and installs localhost certificate
  -h, --help             output usage information

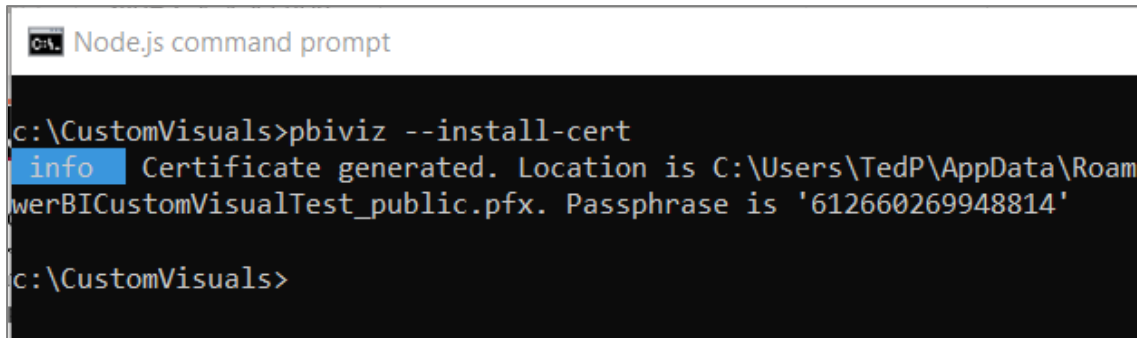
Commands:

  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
  update [version]       Updates the api definitions and schemas in the current visual.
  help [cmd]             display help for [cmd]
```



Creating a Certificate for Local Testing

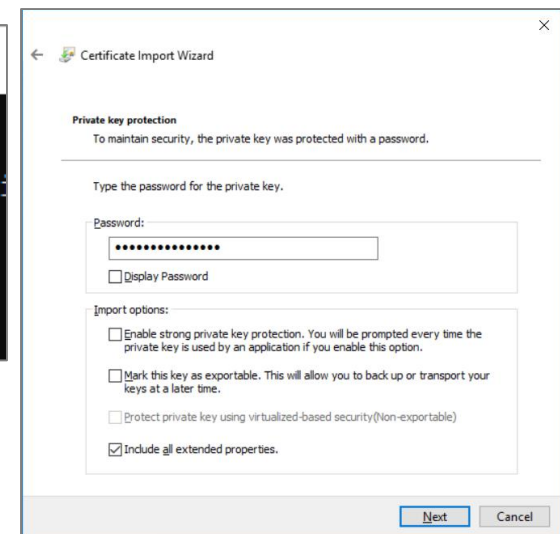
- PBIVIZ provide local web server for testing & debugging
 - Web server runs locally on developer's workstation in Node.js
 - Makes it possible to test custom visuals in Power BI Service
 - Custom visual resources served up from <https://localhost>
 - Setup requires creating self-signed SSL certificate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



```
Node.js command prompt

c:\CustomVisuals>pbiviz --install-cert
info Certificate generated. Location is C:\Users\TedP\AppData\Roaming\PowerBICustomVisualTest_public.pfx. Passphrase is '612660269948814'

c:\CustomVisuals>
```



← Certificate Import Wizard

Private key protection
To maintain security, the private key was protected with a password.

Type the password for the private key.

Password:

☐ Display Password

Import options:

☐ Enable strong private key protection. You will be prompted every time the private key is used by an application if you enable this option.

☐ Mark this key as exportable. This will allow you to back up or transport your keys at a later time.

☐ Protect private key using virtualized-based security (non-exportable)

☒ Include all extended properties.

Next Cancel

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

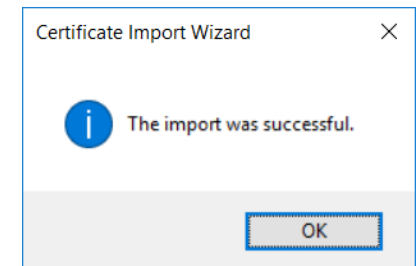
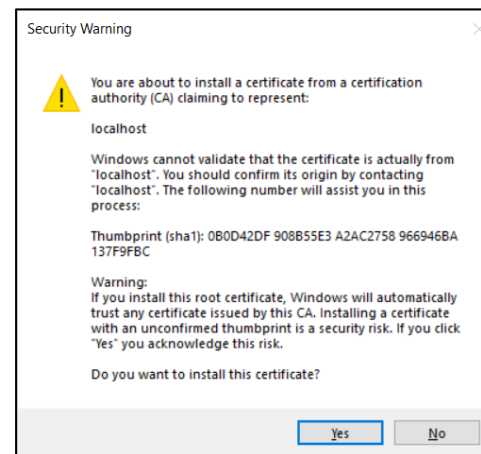
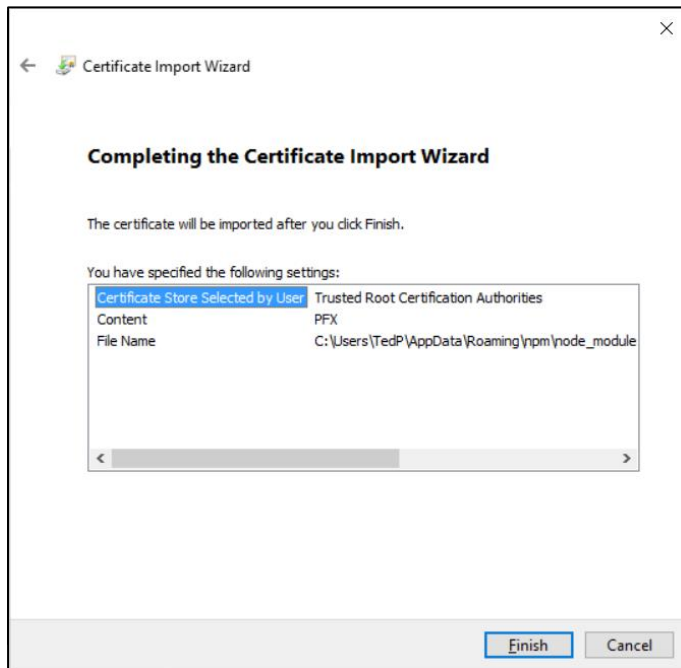
c:\Student>pbiviz --install-cert
info Use '15581865083792024' passphrase to install PFX certificate.

c:\Student>_
```



The Certificate Import Wizard

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



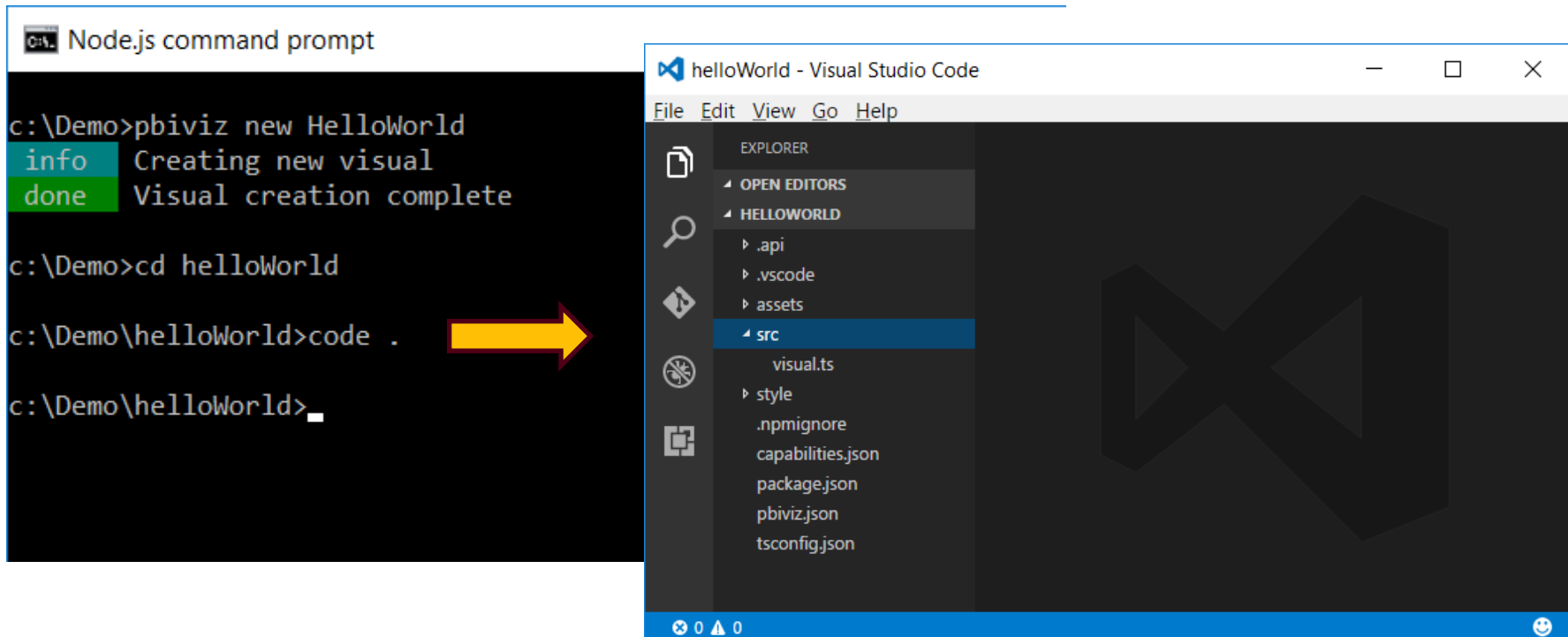
Agenda

- ✓ 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



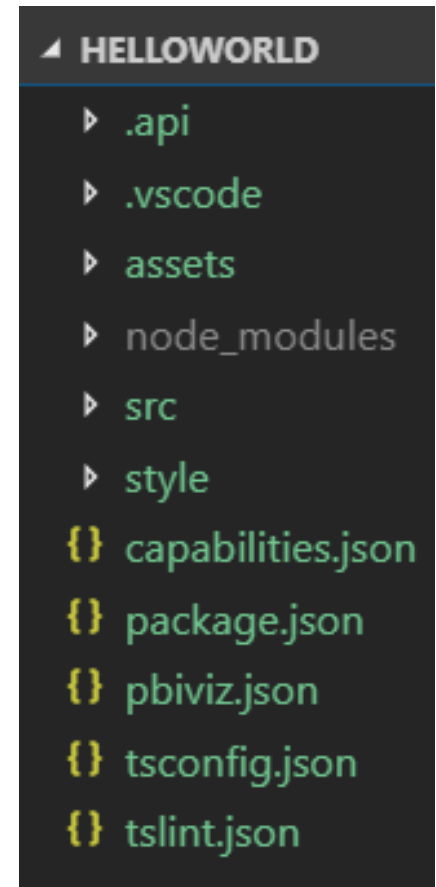
Creating a New Custom Visual Project

- 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



Package.json

```
{ package.json ×
1  {
2    "name": "visual",
3    "scripts": {
4      "pbiviz": "pbiviz",
5      "start": "pbiviz start",
6      "package": "pbiviz package",
7      "lint": "tslint -r \"node_modules/tslint-microsoft-contrib\" \"+(src|test)/**/*.ts\""
8    },
9    "dependencies": {
10     "@babel/runtime": "^7.4.5",
11     "@babel/runtime-corejs2": "^7.4.5",
12     "@types/d3": "5.5.0",
13     "d3": "5.5.0",
14     "powerbi-visuals-utils-dataviewutils": "^2.2.0",
15     "powerbi-visuals-api": "~2.6.0",
16     "core-js": "3.1.3"
17   },
18   "devDependencies": {
19     "ts-loader": "5.2.2",
20     "typescript": "3.0.1"
21   }
22 }
```



The pbiviz.json File

- Acts as top-level manifest file for custom visual project
 - Indicates which version of the Custom Visual API is used

```
pbiviz.json
1  {
2    "visual": {
3      "name": "helloWorld",
4      "displayName": "helloWorld",
5      "guid": "helloWorld59EA4B1D32E442159C244949D73D7994",
6      "visualClassName": "Visual",
7      "version": "1.0.0",
8      "description": "",
9      "supportUrl": "",
10     "githubUrl": ""
11   },
12   "apiVersion": "2.6.0",
13   "author": {
14     "name": "",
15     "email": ""
16   },
17   "assets": {
18     "icon": "assets/icon.png"
19   },
20   "externalJS": null,
21   "style": "style/visual.less",
22   "capabilities": "capabilities.json",
23   "dependencies": null,
24   "stringResources": []
25 }
```



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": false,
4          "emitDecoratorMetadata": true,
5          "experimentalDecorators": true,
6          "target": "es6",
7          "sourceMap": true,
8          "outDir": "./.tmp/build/",
9          "moduleResolution": "node",
10         "declaration": true,
11         "lib": [
12             "es2015",
13             "dom"
14         ]
15     },
16     "files": [
17         "./src/visual.ts"
18     ]
19 }
```



Installing D3 when using PBIVIZ Version 2

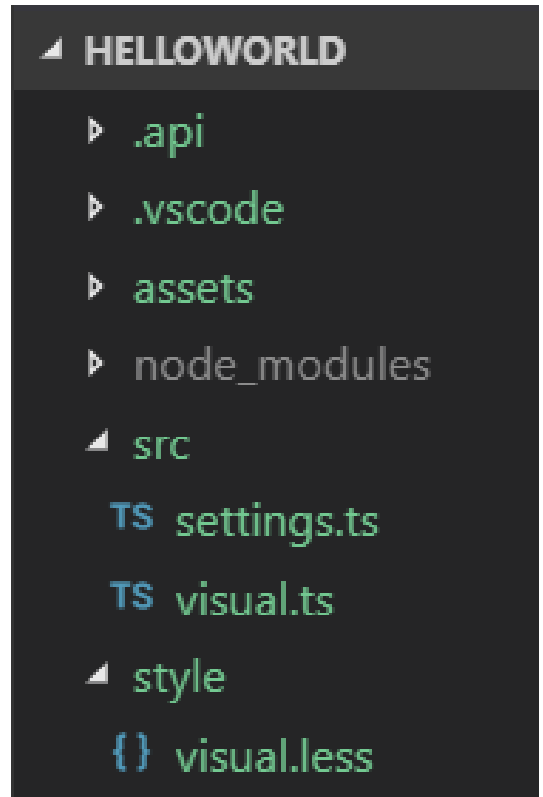
- Install package for D3 library version 3.x
`npm install d3@3 --save-dev`
- Install package for type definition files version 3
`npm install @types/d3@3 --save-dev`
- Update **externalJS** section of **pbiviz.json**

```
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 }
```



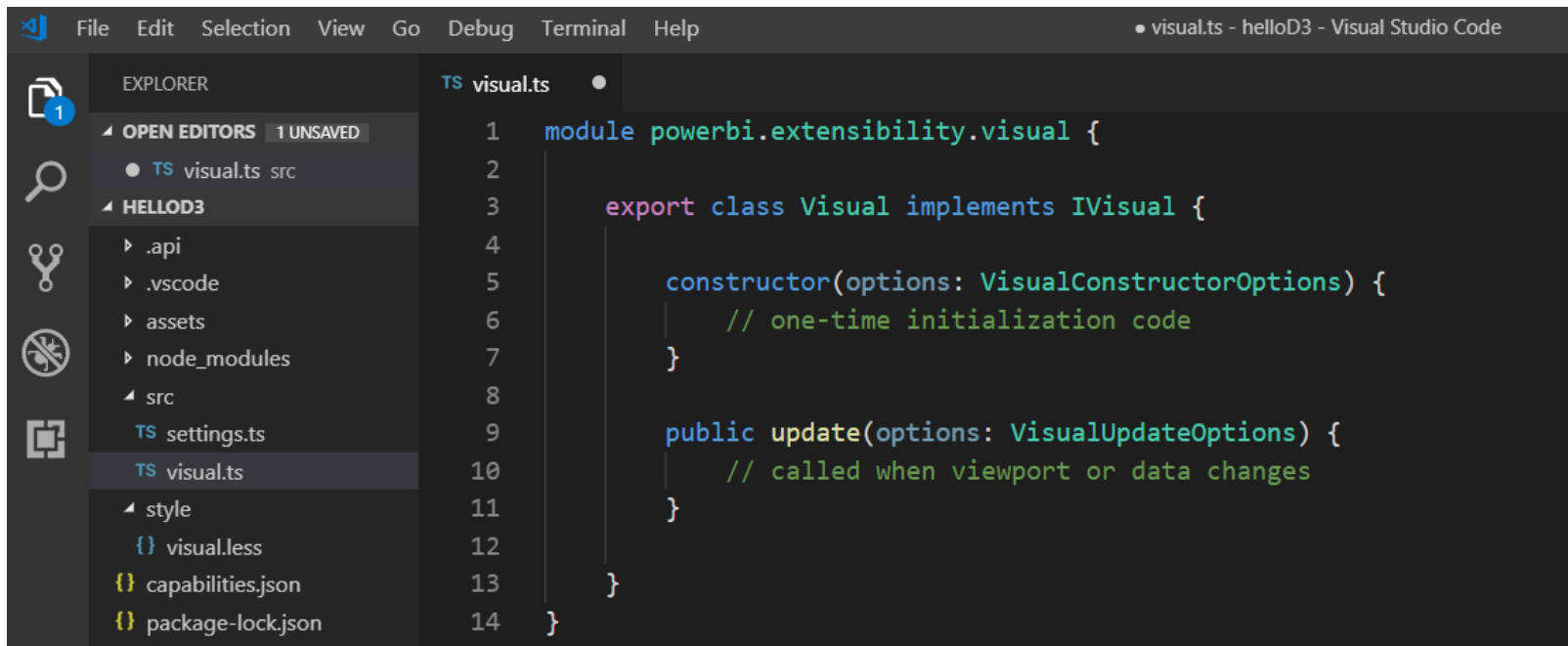
Visual Source Files

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



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

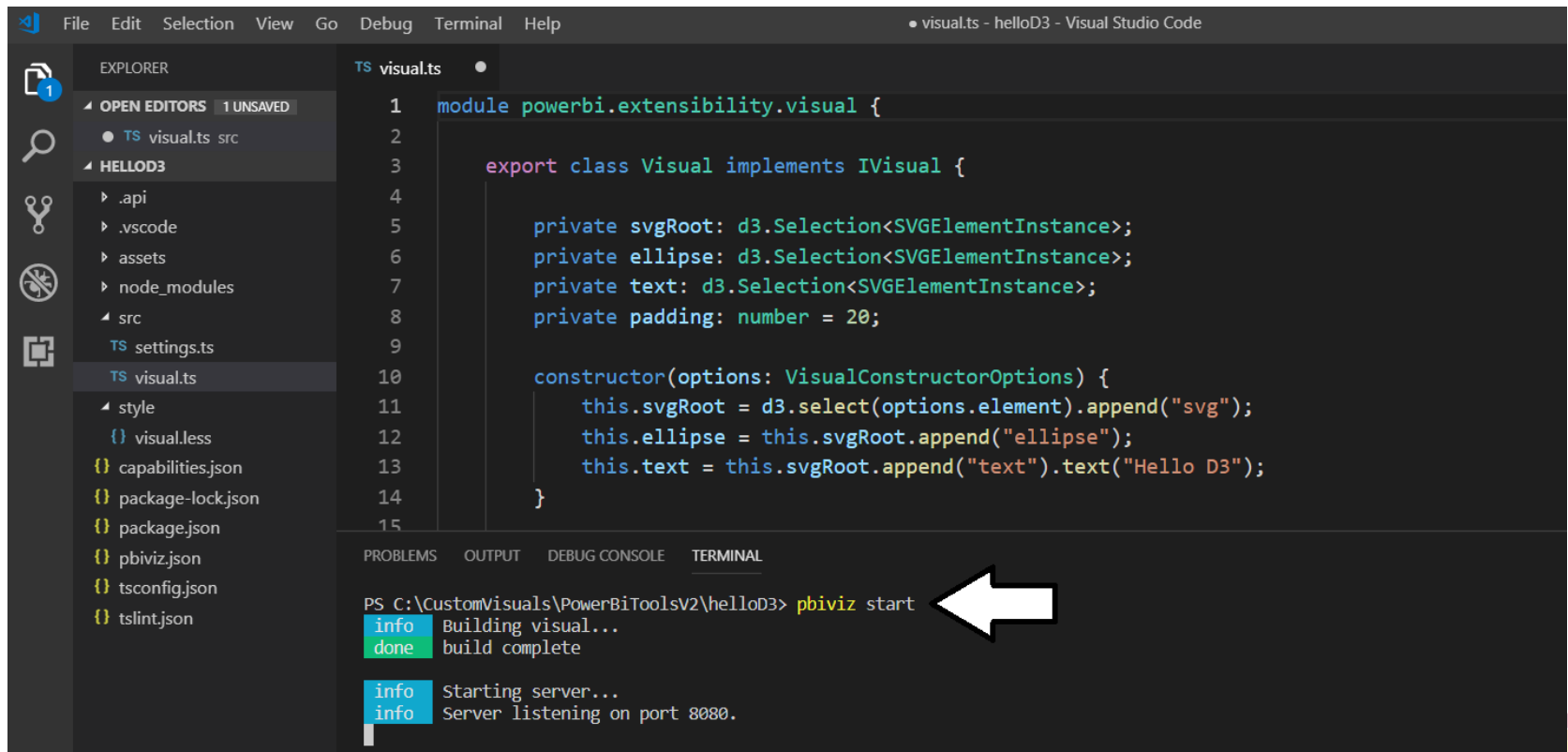
A screenshot of the Visual Studio Code editor interface. The Explorer sidebar on the left shows a project named 'HELLOD3' with a 'src' folder containing 'visual.ts' and 'settings.ts'. The main editor area displays the 'visual.ts' file with the following TypeScript code:

```
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 }
```



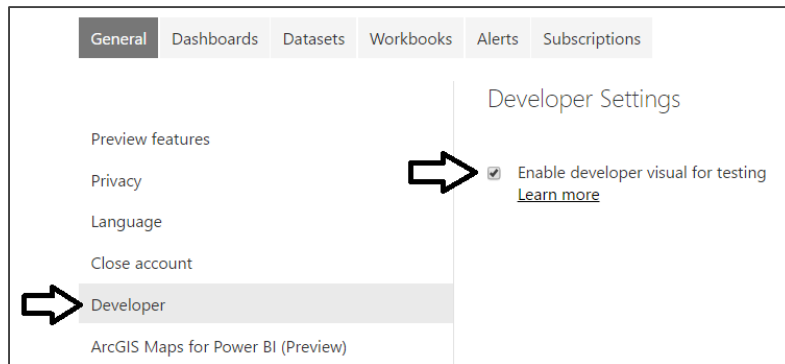
Running a Custom Visual Project

- 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

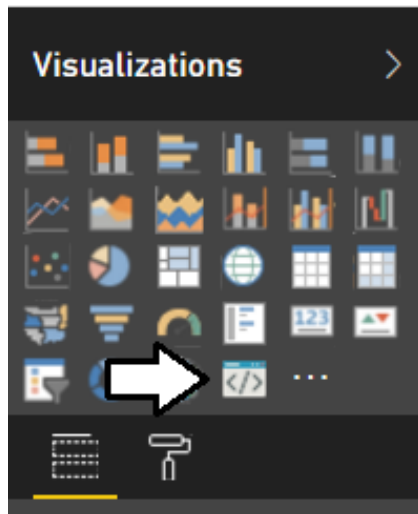


The Developer Visual

- 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



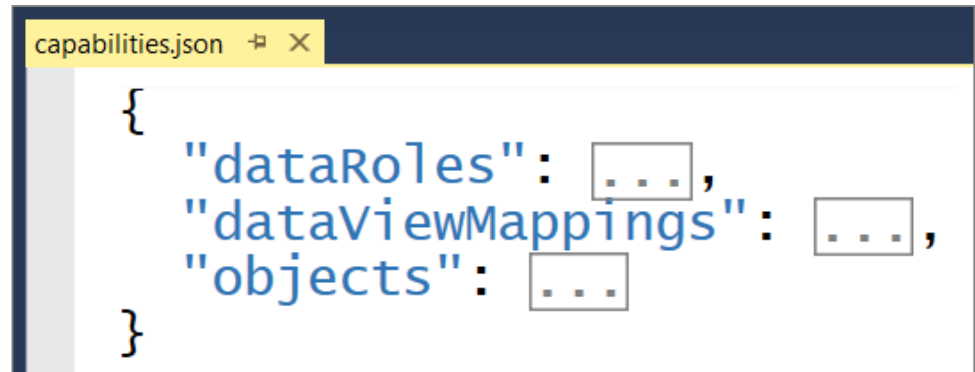
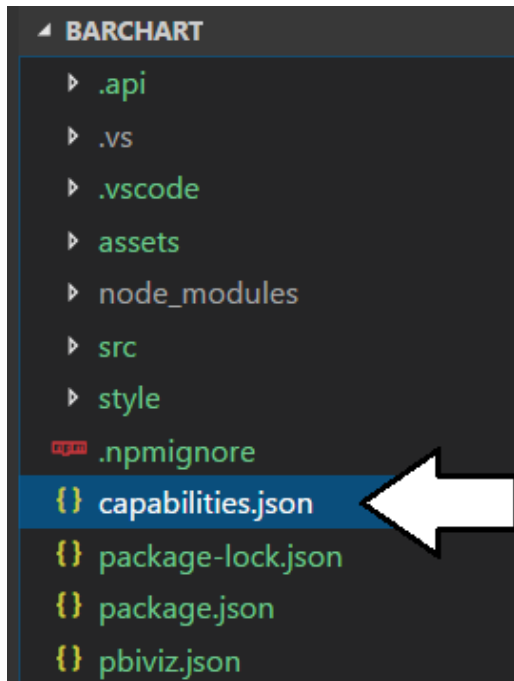
Agenda

- ✓ 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

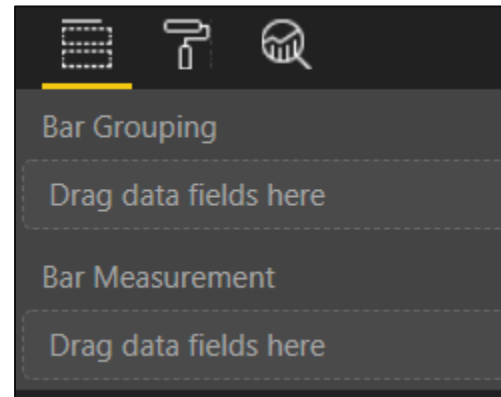
- Visual capabilities defined inside **capabilities.json**
 - **dataRoles** defines the field wells displayed on Fields pane
 - **dataViewMappings** defines the type of DataView used by visual
 - **objects** defines custom properties for visual



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

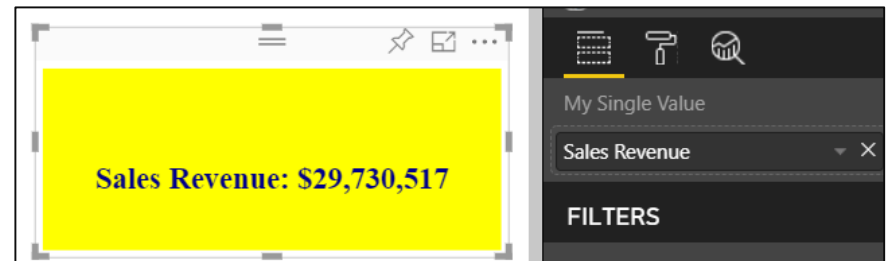
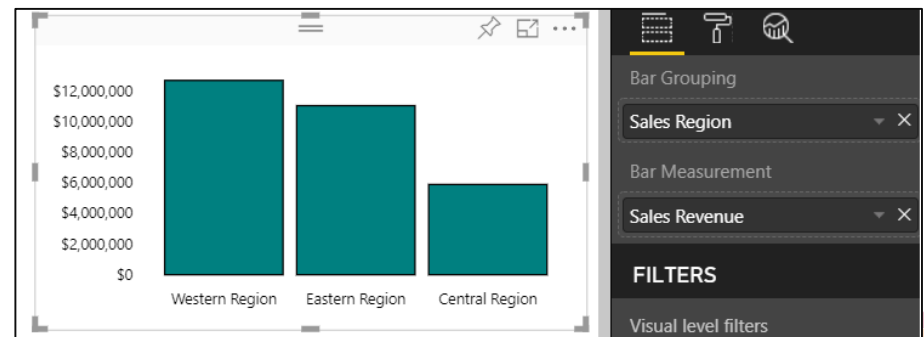


Table Mapping

The visual displays a table with three columns: Sales Region, Sales Revenue, and Units Sold. The right-hand pane shows the configuration: 'Values' is the title, 'Sales Region', 'Sales Revenue', and 'Units Sold' are the selected fields, and the 'FILTERS' pane is empty.

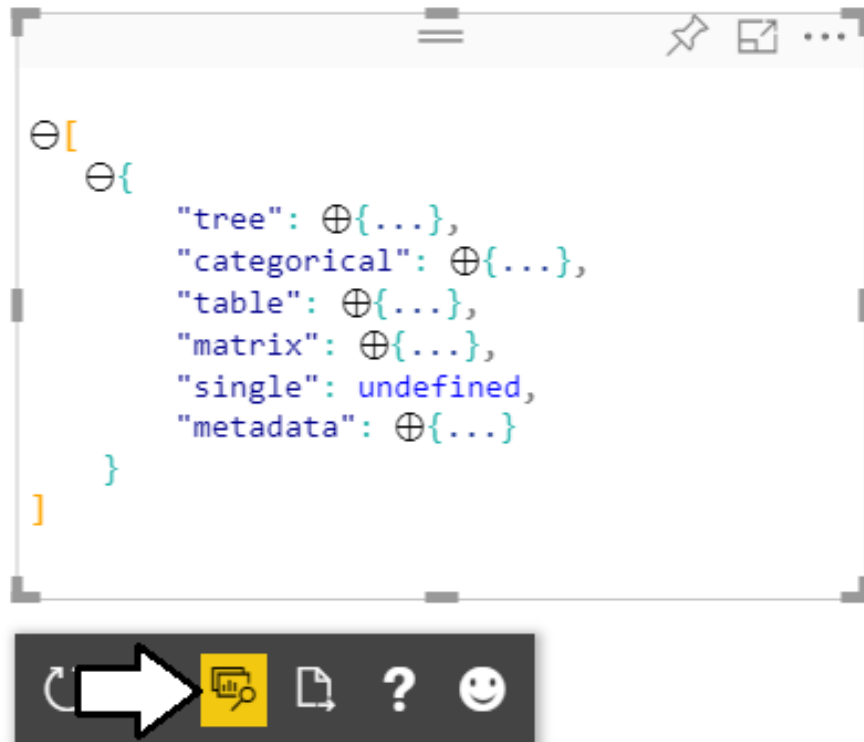
Sales Region	Sales Revenue	Units Sold
Western Region	\$12,733,888	1,598,125
Central Region	\$5,915,449	994,680
Eastern Region	\$11,081,180	1,959,240

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

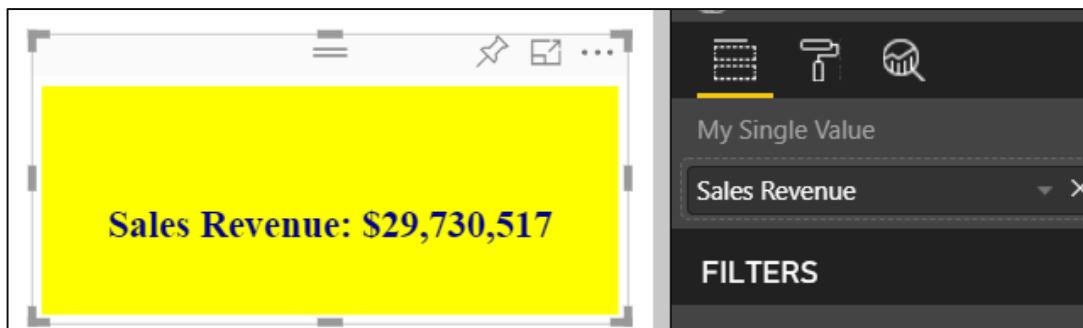
```
export interface BarchartDataPoint {  
    Category: string;  
    Value: number;  
}  
  
export interface BarchartViewModel {  
    IsValid: boolean;  
    DataPoints?: BarchartDataPoint[];  
    Format?: string;  
    SortBySize?: boolean;  
    XAxisFontSize?: number;  
    YAxisFontSize?: number;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

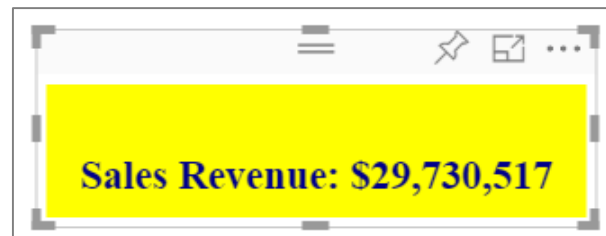
- Used to format values using Power BI formatting strings
 - Requires installing powerbi-visuals-utils-formattingutils package

```
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",
```



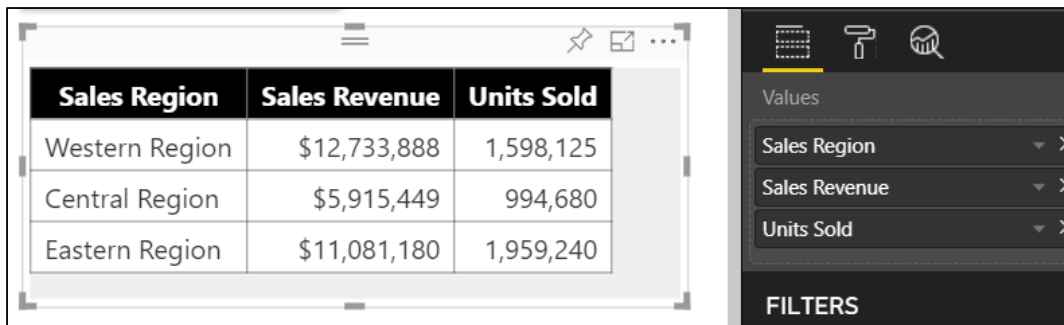
```
"column": {
  "roles": [...],
  "type": [...],
  "format": "#,0",
  "displayName": "Units Sold",
  "queryName": "Sales.Units Sold",
```



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
```

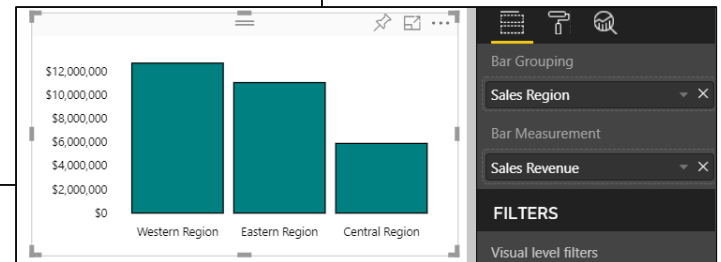
```
public update(options: VisualUpdateOptions) {  
    var dataView: DataView = options.dataViews[0];  
    var table: DataViewTable = dataView.table;  
    var columns: DataViewMetadataColumn[] = table.columns;  
    var rows: DataViewTableRow[] = table.rows;
```



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

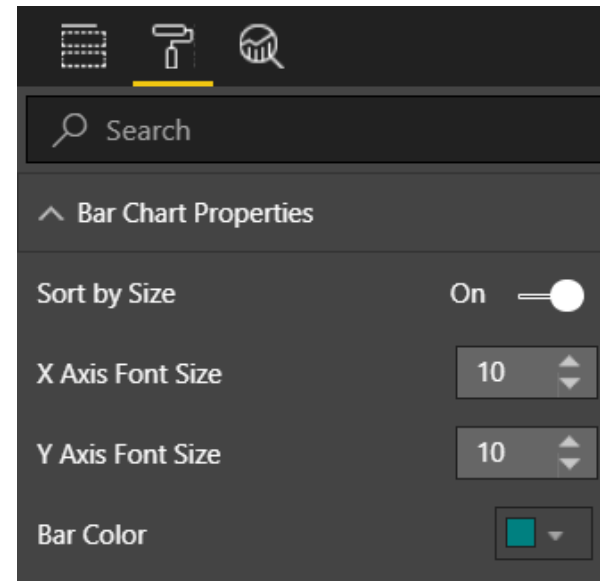
- ✓ Installing the Power BI Developer Tools
- ✓ Creating Your First Custom Visual
- ✓ Defining Data Roles and Data Mappings
- Extending a Visual with Custom Properties
- Migrating to Version 3 of the Power BI Developer Tools



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" } } }  
      }  
    }  
  }  
}
```

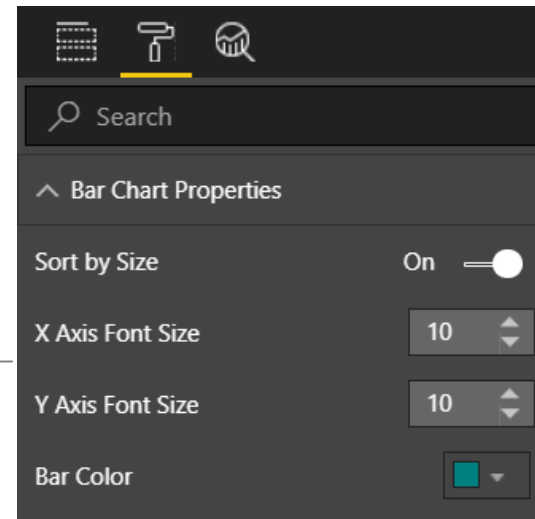
```
export class VisualSettings extends DataModel {  
  public barchartProperties: BarchartProperties;  
}  
  
export class BarchartProperties {  
  sortBySize: boolean = true;  
  xAxisFontSize: number = 10;  
  yAxisFontSize: number = 10;  
  barColor: Fill = { "solid": { "color": "#FFFFFF" } }  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



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;
```



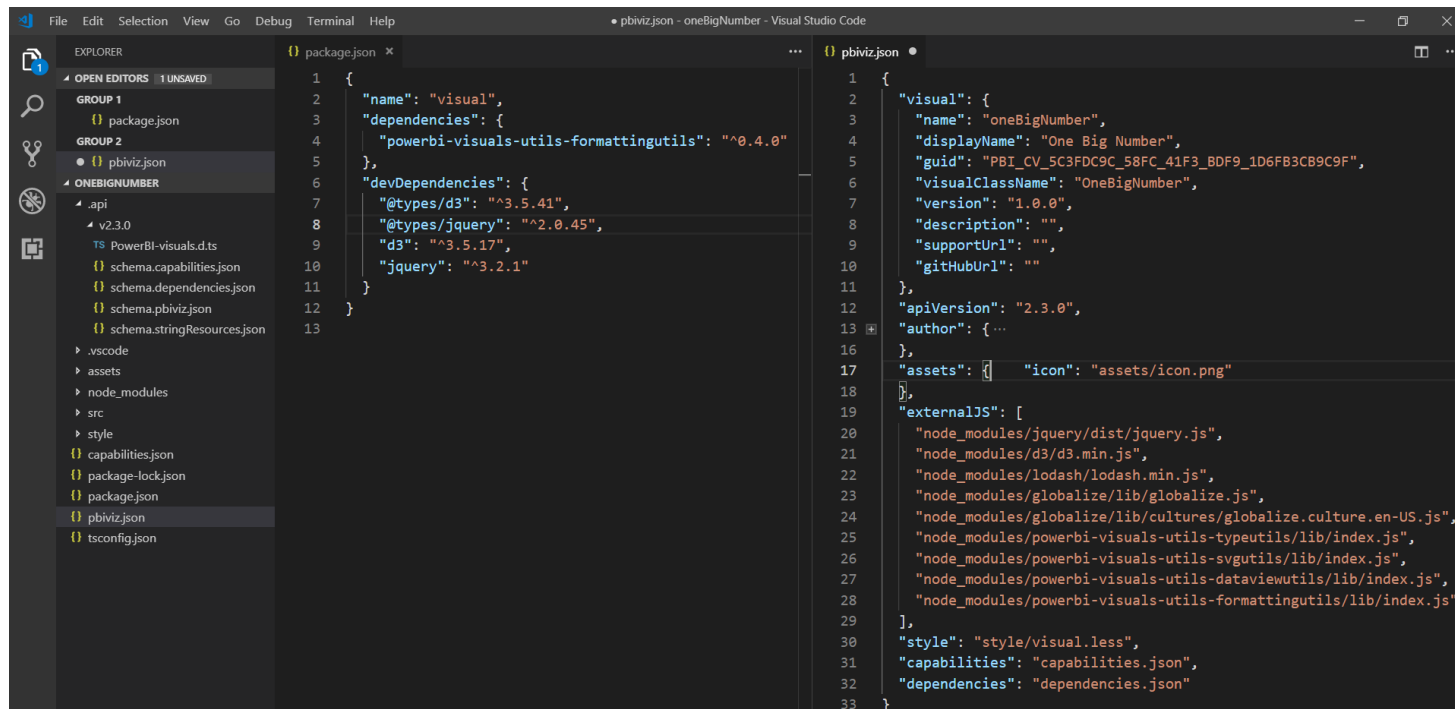
Agenda

- ✓ 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 the following files and folders:

- EXPLORER
 - OPEN EDITORS 1 UNSAVED
 - GROUP 1
 - package.json
 - GROUP 2
 - pbiviz.json
 - ONEBIGNUMBER
 - .api
 - v2.3.0
 - PowerBI-visuals.d.ts
 - schema.capabilities.json
 - schema.dependencies.json
 - schema.pbiviz.json
 - schema.stringResources.json
 - .vscode
 - assets
 - node_modules
 - src
 - style
 - capabilities.json
 - package-lock.json
 - package.json
 - pbiviz.json
 - tsconfig.json

The pbiviz.json file content is as follows:

```
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 content is as follows:

```
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": "OneBigNumber",
15    "url": "https://github.com/OneBigNumber/OneBigNumber"
16  },
17  "assets": [
18    { "icon": "assets/icon.png" }
19  ],
20  "externalJS": [
21    "node_modules/jquery/dist/jquery.js",
22    "node_modules/d3/d3.min.js",
23    "node_modules/lodash/lodash.min.js",
24    "node_modules/globalize/lib/globalize.js",
25    "node_modules/globalize/lib/cultures/globalize.culture.en-US.js",
26    "node_modules/powerbi-visuals-utils-typeutils/lib/index.js",
27    "node_modules/powerbi-visuals-utils-svgutils/lib/index.js",
28    "node_modules/powerbi-visuals-utils-dataviewutils/lib/index.js",
29    "node_modules/powerbi-visuals-utils-formattingutils/lib/index.js"
30  ],
31  "style": "style/visual.less",
32  "capabilities": "capabilities.json",
33  "dependencies": "dependencies.json"
34 }
```



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. MacDonald"),
7     new Quote("Behind every great man is a great woman", "Robert A. Heinlein"),
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 4
 - Install packages for webpack and webpack-cli
 - `npm install webpack webpack-cli --save-dev`





DEMO

Working with the V3 Tools

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

