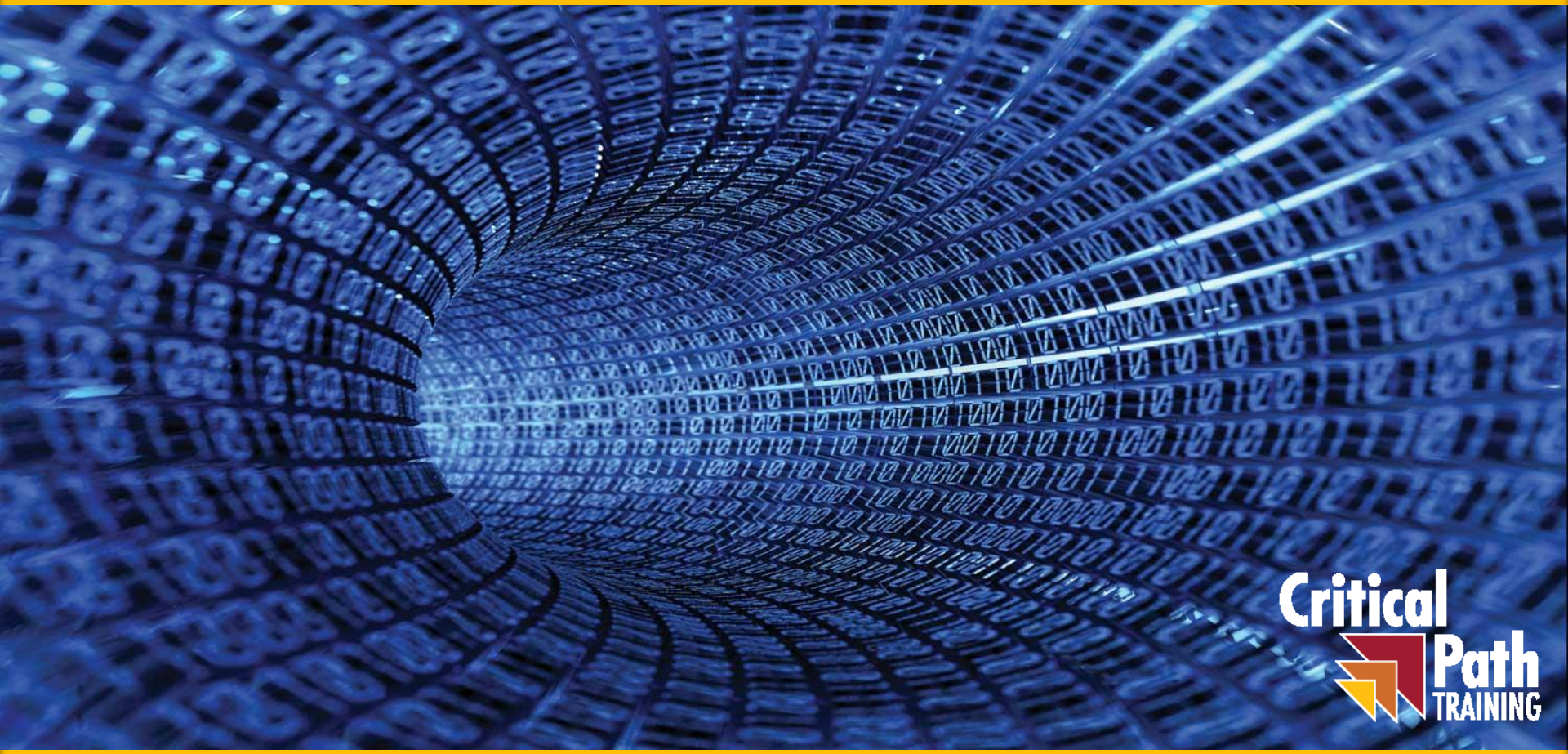
Developing with TypeScript and AngularJS



Agenda

- Moving from JavaScript to TypeScript
- Introduction to AngularJS
- Angular Routes, Views and Controllers
- View Models and Form Validation
- Programming with Asynchronous Services
- Creating Angular Components

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

```
module myApp {

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

}
```

product.ts

TypeScript Compiler

```
var myApp;
(function (myApp) {
    var Product = (function () {
        function Product() {
        }
        return Product;
    }());
    myApp.Product = Product;
})(myApp || (myApp = {}));
//# sourceMappingURL=Product.js.map
```

product.js

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);
```

Assignment with let versus var

- `var` does not recognize nor honor scope
- `let` will recognize and honor scope

```
var x:number = 2016;
let y: number = 2016;

{
  var x:number = 2017;
  let y:number = 2017;
}

let message = "x=" + x + " and " + "y=" + y;
```

x=2017 and y=2016

Parameter Arrays

```
// define function with a parameter array using (...) syntax
function createOrderedList(...names: string[]): string{
  let html: string = "<ol>";
  for (let index: number = 0; index < names.length; index++) {
    html += "<li>" + names[index] + "</li>"
  }
  return html += "</ol>";
};

// create a string array
let stooges: string[] = ["Moe", "Curly", "Larry"];

// call function with a parameter array
let stoogesList: string = createOrderedList(...stooges);
```

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 sytax
let myFunction = () => {
  console.log("Hello world");
};

// use function arrow sytax 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.
```

Category
Name

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;
}
```

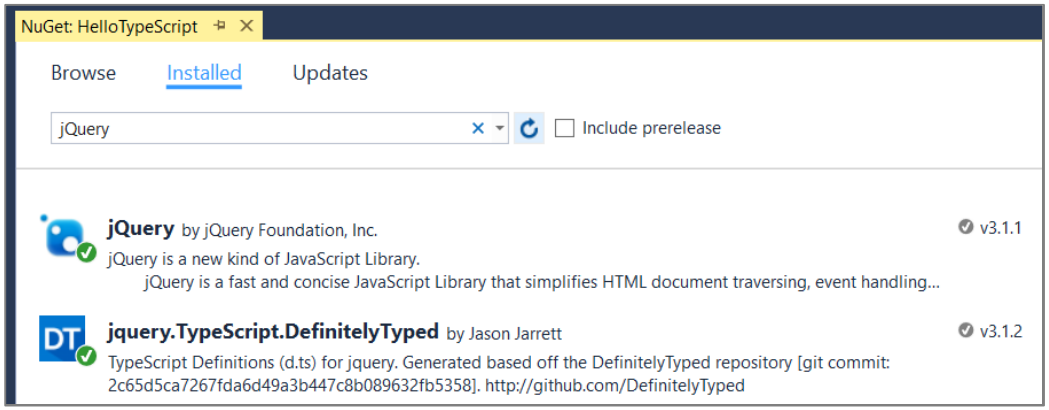
```
export class MyProductDataService implements IProductDataService {

  private products: Product[] =...;

  GetAllProducts(): Product[]...;

  GetProduct(id: number): Product...;

  AddProduct(product: Product): void...;

  DeleteProduct(id: number): void...;

  UpdateProduct(product: Product): void...;
}
```

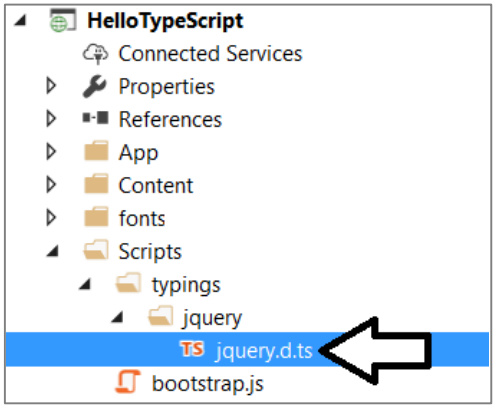
  - Client code can be decoupled from concrete classes

```
// program against variables based on interface type
let productService: IProductDataService = new MyProductDataService();

// clioent 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");
```

Agenda

- ✓ Moving from JavaScript to TypeScript
- ➢ Introduction to AngularJS
- Angular Routes, Views and Controllers
- View Models and Form Validation
- Programming with Asynchronous Services
- Creating Angular Components

Introducing AngularJS

- What is AngularJS?
  - A JavaScript framework for building web applications
  - Based on Single-Page Application (SPA) model
  - Implements Model-View-Controller (MVC) Pattern

- What version should you use?
  - AngularJS 1.0
  - AngularJS 1.5
  - Angular 2.0
  - Angular 4.0

AngularJS Features

- Directive
  - A shared unit of declarative functionality
- Module
  - A container for a reusable unit of code
- Controller
  - A JavaScript functions which processes incoming requests
- View
  - An HTML template that serves as a partial view on a page
- View Model
  - JavaScript object containing domain-specific data prepared by controller
  - Object properties declaratively bound to HTML elements in the view
- Service
  - Built-in Angular services include $http, $window and $route
  - Custom services used to write code which is shared across controllers

Key Angular Directives

- `ng-app:` initialize the Angular app
- `ng-controller:` designate controller scope
- `ng-view:` define placeholder for dynamic views
- `ng-bind:` one-way binding of HTML element to model
- `ng-model:` two-way binding of HTML element to model
- `ng-repeat:` `create` for-each loop
- `ng-click:` handle click event.
- `ng-hide:` shows or hides an HTML element
- `ng-href:` creates Angular-compliant anchor tags
- `ng-src:` creates Angular-compliant img tags

Service Components included with AngularJS

- Angular includes many built-in service components
  - This tables lists some of the more commonly used services

Service	Purpose
$http	used to communicate with the remote HTTP servers using XMLHttpRequest object
$location	used to retrieve the URL in the browser address bar
$log	safely writes the message into the browser's console
$q	promise/deferred implementation
$window	reference to the browser's window object
$anchorScroll	scrolls to the related element
$filter	used for formatting data displayed to the user
$route	used for deep-linking URLs to controllers and views
$routeParams	allows you to retrieve the current set of route parameters

Understanding AngularJS Modules

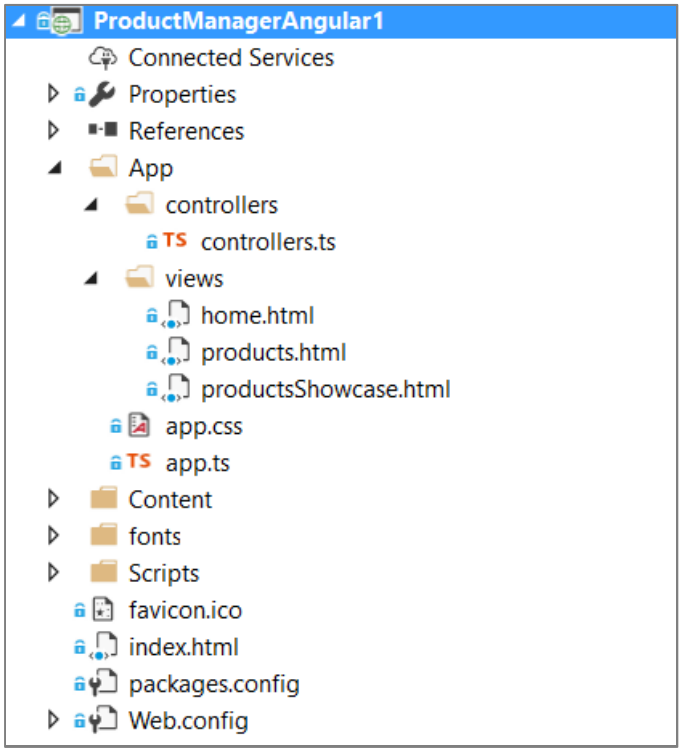
- Module represents a container of code
  - AngularJS provides several built-in Modules
  - Third parties libraries often created using Modules
- Named module can be created for app
  - App module named using **ng-app** Directive
  - App module initialized using **angular.module** function

```html
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initi
  <title>Product Manager NG</title>
  <link href="/Content/bootstrap.css" rel="stylesheet" />
  <!-- required by Angular routing -->
  <base href="/">
</head>

<body ng-app="myApp">
```

```
module myApp {

  var app = angular.module("myApp", ['ngRoute']);

  app.config( ($locationProvider: ng.ILocationProvider,
               $routeProvider: ng.route.IRouteProvider) => {
    // code to initialize app
  });

}
```

App Project Structure

- All application code maintained in **App** folder
  - App start page implemented using **start.html**
  - App initialization code maintained in **app.js**
  - Child folders added for controllers, services and views



Agenda

- ✓ Moving from JavaScript to TypeScript
- ✓ Introduction to AngularJS
- ➢ Angular Routes, Views and Controllers
- ▪ View Models and Form Validation
- ▪ Programming with Asynchronous Services
- ▪ Creating Angular Components

Routes, View Template and Controllers

- What are Routes?
  - Route represents endpoint in the app's route map
  - Route configured with View template and Controller
- What is a View Template?
  - HTML fragment in .html file which acts as partial view
  - HTML in view template often created using Directives
- What is a controller?
  - JavaScript function which provides view logic
  - Controller creates and passes model to View Template

Defining Routes

- Steps to defining route map for an app
  - Define routes using the injected `$routeProvider` object
  - You must supply `templateUrl`, `controller` and `controllerAs`
  - Setting `HTML5Mode` eliminates need for `#` in routing URLs

```
app.config( ($locationProvider: ng.ILocationProvider,
            $routeProvider: ng.route.IRouteProvider) => {

  $locationProvider.html5Mode(true);

  $routeProvider
    .when("/", {
      templateUrl: 'App/views/home.html',
      controller: "homeController",
      controllerAs: "vm"
    })
    .when("/products", {
      templateUrl: 'App/views/products.html',
      controller: "productsController",
      controllerAs: "vm"
    })
    .when("/products/showcase", {
      templateUrl: 'App/views/productsShowcase.html',
      controller: "productShowcaseController",
      controllerAs: "vm"
    })
    .otherwise({ redirectTo: "/" });

});
```

Web.config File for an Angular Web App

- HTML5 Mode requires server-side URL rewriting support
  - Can be configured using Web.config file

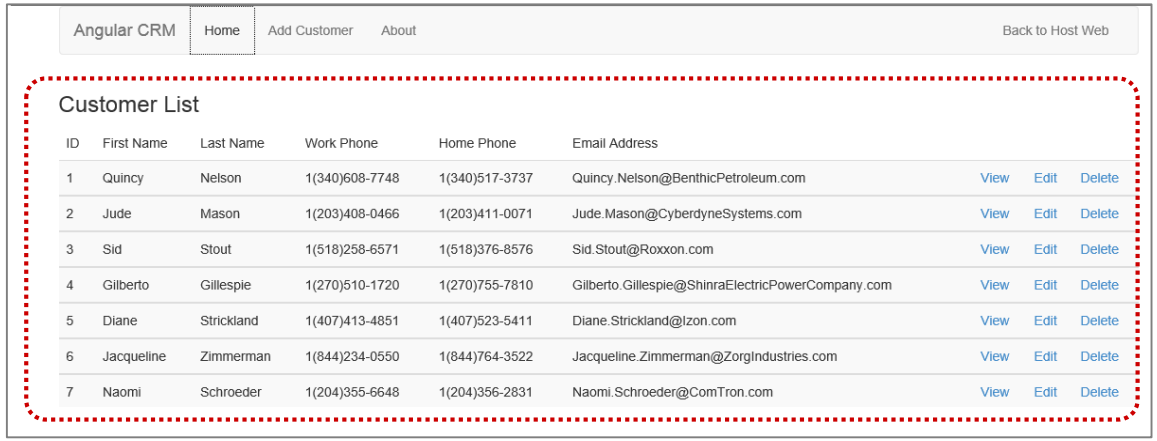
```
<configuration>

  <system.webServer>
    <rewrite>
      <rules>
        <rule name="AngularJS Routes" stopProcessing="true">
          <match url=".*" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.html$" negate="true" />
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.js$" negate="true" />
            <add input="{REQUEST_FILENAME}" pattern="(.*?)\.css$" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
            <add input="{REQUEST_URI}" pattern="^/(api)" negate="true" />
          </conditions>
          <action type="Rewrite" url="/" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>

</configuration>
```

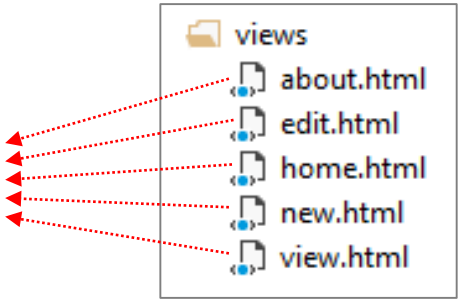
Dynamically Loading View Templates

- View placeholder element defined using ng-view attribute

```
<body ng-app="AngularCRM" >

  <div class="container">
    <div class="navbar navbar-default">...</div>
  </div>

  <div class="container">
    <div id="content-box" ng-view></div>
  </div>

</body>
```

- View templates are loaded into view placeholder element





Understanding AngularJS Controllers

- Controllers are implemented using JavaScript functions
  - Controller registered using **controller** function on app module
  - Controller initializes object to serve as view model
  - View model accessible to view template with ControllerAs variable

```
class ProductsController {
  static $inject: Array<string> = ['$location', 'ProductDataService'];
  products: Product[];
  productCategories: string[];
  // add constructor
  constructor(private $location: ng.ILocationService,
    private ProductDataService: IProductDataService) {
    this.products = ProductDataService.GetAllProducts();
    this.productCategories = ProductDataService.GetProductCategories();
  }
  // add delete user action
  deleteProduct(id: number) {
    this.ProductDataService.DeleteProduct(id);
    this.$location.path("/products");
  }
}

app.controller('productsController', ProductsController);
```

Understanding AngularJS View Templates

- View Templates are implemented using HTML

products.html

```
<p>
  <a class="btn btn-primary" href="/products/add">Create Product</a>
</p>

<table id="products-table" class="table table-striped table-hover table-responsive ">
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th class="hidden-sm hidden-xs">Category</th>
      <th class="hidden-sm hidden-xs">List Price</th>
      <th> </th>
      <th> </th>
      <th> </th>
    </tr>
  </thead>
  <tbody ng-repeat="product in vm.products">
    <tr>
      <td>{{product.Id}}</td>
      <td>{{product.Name}}</td>
      <td class="hidden-sm hidden-xs">{{product.Category}}</td>
      <td class="hidden-sm hidden-xs">{{product.ListPrice | currency }}</td>
      <td><a href="/products/view/{{product.Id}}" class="navbar-link">View</a></td>
      <td><a href="/products/edit/{{product.Id}}" class="navbar-link">Edit</a></td>
      <td><a href="" ng-click="vm.deleteProduct(product.Id)" class="navbar-link">Delete</a></td>
    </tr>
  </tbody>
</table>
```

Defining the IProductDataService Interface

- Interface can help with design
  - It decouples controller code from data access code

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

Custom Services in AngularJS

- What type of code should be written in a service?
  - Any code shared across controllers or which calls across network
- Create service by calling **service** method on App Module object

```
export class InMemoryProductDataService implements IProductDataService...

angular.module('myApp').service('ProductDataService', InMemoryProductDataService);
```

- How do you use the service from a controller?

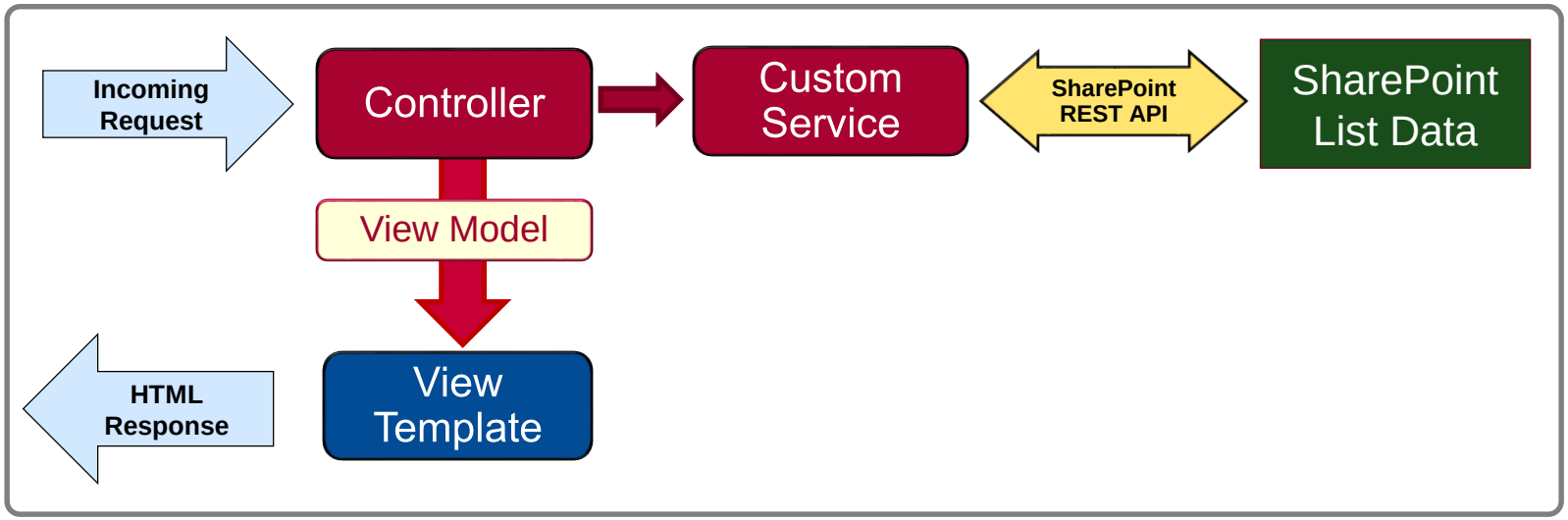
```
class ProductsController {
  static $inject: Array<string> = ['$location', 'ProductDataService'];
  products: Product[];
  productCategories: string[];
  // add constructor
  constructor(private $location: ng.ILocationService, private ProductDataService: IProductDataService) {
    this.products = ProductDataService.GetAllProducts();
    this.productCategories = ProductDataService.GetProductCategories();
  }
  // add delete user action
  deleteProduct(id: number)...
}
```

Best Practices with Services and Controllers

- Controllers should never reference the DOM
  - DOM manipulation done using custom Directives
- Controllers should define view behavior
  - What happens when user clicks Delete button?
  - What happens when user clicks Save button?
- Controllers should not contain any data access code
  - Code to call across network should be written in service(s)

Controller Processing Flow

1. Incoming request routed to Controller using app's route map
2. Controller calls data access function provided by custom service
3. Custom service calls across network to fetch SharePoint list data
4. Custom service returns SharePoint list data to Controller
5. Controller uses SharePoint list data to create model
6. Controller passes model to View Template using $scope
7. View Template binds to model data using Directives
8. View Templates renders HTML which is returned to client



Key Filters

- Format
  - currency
  - date
  - number
- Displaying data sets
  - orderBy
  - limitTo
- String manipulation
  - uppercase
  - lowercase

Customer Filters

- Custom filter created using `filter` function

```
class AppFilters {
  static $inject: Array<string> = ['$filter'];
  public static listPriceFilter($filter) {
    return (price: number) => {
      return "$" + $filter('number')(price, 2) + " USD";
    }
  }
}

angular.module("myApp").filter("listPrice", AppFilters.listPriceFilter);
```

```
<tr>
  <td>{{product.Id}}</td>
  <td>{{product.Name}}</td>
  <td class="hidden-sm hidden-xs">{{product.Category}}</td>
  <td class="hidden-sm hidden-xs">{{product.ListPrice | listPrice }}</td>
  <td><a href="/products/view/{{product.Id}}" class="navbar-link">View</a></td>
  <td><a href="/products/edit/{{product.Id}}" class="navbar-link">Edit</a></td>
  <td><a href="" ng-click="vm.deleteProduct(product.Id)" class="navbar-link">Delete</a></td>
</tr>
```

ID	Name	Category	List Price
1	Batman Action Figure	Action Figures	$14.95 USD
2	Captain America Action Figure	Action Figures	$12.95 USD
3	Easel with Supply Trays	Arts and Crafts	$49.95 USD

Agenda

- ✓ Moving from JavaScript to TypeScript
- ✓ Introduction to AngularJS
- ✓ Angular Routes, Views and Controllers
- ➢ View Models and Form Validation
- ▪ Programming with Asynchronous Services
- ▪ Creating Angular Components

AngularJS Form Validation

productsAdd.html

```
<h3>Create Product</h3>

<form name="thisForm" novalidate ng-submit="thisForm.$valid && vm.addProduct()">
  <div class="form-horizontal">

      <div class="form-group">...</div>
```

```
<div class="form-group">
  <label class="control-label col-md-2" for="ListPrice">List Price:</label>
  <div class="col-md-10">

    <input id="ListPrice" name="ListPrice" type="number" required min="1" max="10000" step="any"
           ng-model="vm.product.ListPrice" class="form-control" >

    <div class="text-danger" ng-show="thisForm.ListPrice.$invalid && (thisForm.ListPrice.$dirty || thisForm.$submitted)">
      <span ng-show="thisForm.ListPrice.$error.required">List price is a required field</span>
      <span ng-show="thisForm.ListPrice.$error.number">List price must be a number</span>
      <span ng-show="thisForm.ListPrice.$error.min || thisForm.ListPrice.$error.max">
        List price must be between $1 and $10,000
      </span>
    </div>

  </div>

</div>
```

Agenda

- ✓ Moving from JavaScript to TypeScript
- ✓ Introduction to AngularJS
- ✓ Angular Routes, Views and Controllers
- ✓ View Models and Form Validation
- ➢ Programming with Asynchronous Services
- ▪ Creating Angular Components

Asynchronous Interfaces

```
export interface IProductDataServiceAsync {
  GetAllProductsAsync(): ng.IPromise<Product[]>;
  GetProductAsync(id: number): ng.IPromise<Product>;
  AddProductAsync(product: Product): ng.IPromise<void>;
  DeleteProductAsync(id: number): ng.IPromise<void>;
  UpdateProductAsync(product: Product): ng.IPromise<void>;
  GetProductCategoriesAsync(): ng.IPromise<string[]>;
  GetProductsByCategoryAsync(category: string): ng.IPromise<Product[]>;
}
```

```
export class AsyncInMemoryProductDataService implements IProductDataServiceAsync...

angular.module('myApp').service('ProductDataService', AsyncInMemoryProductDataService);
```

Calling Asynchronous Methods

- Calling asynchronous method requires callback
  - Callback method pass using `then` method

```
class ProductsController {
  static $inject: Array<string> = ['$location', 'ProductDataService'];

  products: Product[];
  productCategories: string[];

  constructor(private $location: ng.ILocationService,
    private ProductDataService: IProductDataServiceAsync) {
    ProductDataService.GetAllProductsAsync()
      .then((result: Product[]) => {
        this.products = result;
      });
    ProductDataService.GetProductCategoriesAsync()
      .then((result: string[]) => {
        this.productCategories = result;
      });
  }

  deleteProduct(id: number) {
    this.ProductDataService.DeleteProductAsync(id)
      .then(() => {
        this.$location.path("/products");
      });
  }
}
```

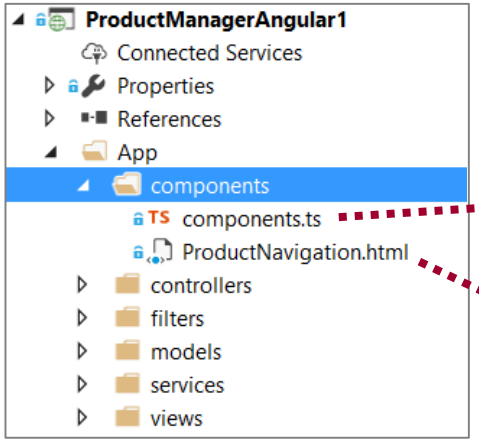
```
ProductDataService.GetAllProductsAsync()
  .then((result: Product[]) => {
    this.products = result;
  });
```

Agenda

- ✓ Moving from JavaScript to TypeScript
- ✓ Introduction to AngularJS
- ✓ Angular Routes, Views and Controllers
- ✓ View Models and Form Validation
- ✓ Programming with Asynchronous Services
- ➢ Creating Angular Components

AngularJS Components

- AngularJS 1.5 introduces components
  - Designed to replace custom AngularJS directives
  - Applications can be designed in terms of components
  - Represents component-oriented shift into Angular 2.0




```
let app = angular.module("myApp");

class ProductNavigationController...

class ProductNavigation implements ng.IComponentOptions...

app.component("productNavigation", new ProductNavigation());
```

ProductNavigation.html

```
<nav id="left-nav" class="navbar" role="navigation">
  <div id="left-nav-title">Filter by Category</div>
  <ul class="nav navbar-nav">
    <li class="nav navbar-link"><a href="/products/showcase/">All Categories</a></li>
    <li class="nav navbar-link" ng-repeat="category in $ctrl.productCategories">
      <a href="/products/showcase/?category={{category}}">{{category}}</a>
    </li>
  </ul>
</nav>
```

AngularJS Component Classes

- Angular component requires two classes
  - Component controller class

```
class ProductNavigationController {
  static $inject: string[] = ['ProductDataService'];
  public productCategories: string[];

  constructor(private ProductDataService: IProductDataServiceAsync) {
    // initialize view model inside $onInit not in constructor
  };

  public $onInit() {
    this.ProductDataService.GetProductCategoriesAsync()
      .then((result: string[]) => {
        this.productCategories = result;
      });
  }
}
```

  - Component options class

```
class ProductNavigation implements ng.IComponentOptions {

  public bindings: { [binding: string]: string };
  public controller: any;
  public templateUrl: any;

  constructor() {
    this.bindings = {};
    this.controller = ProductNavigationController;
    this.templateUrl = '/App/components/productNavigation.html';
  }

}
```

AngularJS Component View Template

- Component options provides path to view template

```
class ProductNavigation implements ng.IComponentOptions {

  public bindings: { [binding: string]: string };
  public controller: any;
  public templateUrl: any;

  constructor() {
    this.bindings = {};
    this.controller = ProductNavigationController;
    this.templateUrl = '/App/components/productNavigation.html';
  }

}
```

- View templates uses `$ctrl` variable to access view model

ProductNavigation.html

```
<nav id="left-nav" class="navbar" role="navigation">
  <div id="left-nav-title">Filter by Category</div>
  <ul class="nav navbar-nav">
    <li class="nav navbar-link"><a href="/products/showcase/">All Categories</a></li>
    <li class="nav navbar-link" ng-repeat="category in $ctrl.productCategories">
      <a href="/products/showcase/?category={{category}}">{{category}}</a>
    </li>
  </ul>
</nav>
```

Instantiating an AngularJS Component

```
let app = angular.module("myApp");

class ProductNavigationController...

class ProductNavigation implements ng.IComponentOptions...

app.component("productNavigation", new ProductNavigation());
```

productsShowcase.html

```
<div class="row">
  <div class="container">
    <div class="row row-offcanvas row-offcanvas-left">

      <!-- sidebar nav -->
      <div class="col-xs-6 col-sm-2 sidebar-offcanvas" id="sidebar" role="navigation">
        <product-navigation />
      </div>

      <!-- main area -->
      <div class="col-xs-12 col-sm-10">...</div>

    </div>
  </div>
</div>
```

Summary

- ✓ Moving from JavaScript to TypeScript
- ✓ Introduction to AngularJS
- ✓ Angular Routes, Views and Controllers
- ✓ View Models and Form Validation
- ✓ Programming with Asynchronous Services
- ✓ Creating Angular Components