

# AngularJS

Presented by  
VAISHALI TAPASWI

FANDS INFONET Pvt.Ltd.  
[www.fandsindia.com](http://www.fandsindia.com)

# Ground Rules

- Turn off cell phone. If you cannot please keep it on silent mode. You can go out and attend your call.
- If you have any questions or issues please let me know immediately.
- Let us be punctual.

A decorative vertical bar on the left side of the slide, composed of numerous horizontal segments in various shades of blue, black, and yellow, creating a colorful, abstract pattern.

# Agenda



A **Fast AND Steady** Approach

# Fast Recap of Object Oriented JavaScript



# An Object

- An **object** is a collection of properties and methods
- These properties can either be primitive data types, other objects, or functions
- A **constructor function** (or simply, **constructor**) is a function used to create an object
- Role of new keyword
  - ```
function myFunc(){  
    }  
var myObject = new myFunc();  
alert(typeof myObject); // displays "object"
```
  - ```
function myFunc(){  
    return 5;  
    }  
  
var myObject = myFunc();  
alert(typeof myObject); // displays "number"
```

# Properties

- ❑ 

```
function myFunc(){  
  }  
var myObject = new myFunc();  
myObject.StringValue = "This is a String";  
alert(myObject.StringValue); // displays "This is a String"
```
- ❑ 

```
function myFunc(){  
  }  
  
var myObject = new myFunc();  
myObject.StringValue = "This is a String";  
var myObject2 = new myFunc();  
alert(myObject2.StringValue); // displays "undefined"
```

# Initialize Property

```
□ function myFunc(){  
    this.StringValue = "This is a String";  
}
```

```
var myObject = new myFunc();  
var myObject2 = new myFunc();  
alert(myObject2.StringValue); // displays  
"This is a String"
```

# Constructor Arguments

```
function myFunc(StringValue){  
    this.StringValue = StringValue;  
}
```

```
var myObject = new myFunc("This is myObject's  
string");  
var myObject2 = new myFunc("This is a String");  
alert(myObject.StringValue); // displays "This is  
myObject's string"  
alert(myObject2.StringValue); // displays "This is a  
String"
```



# Object Methods

- Constructor

```
function Circle(radius){
  this.radius = radius;
}
```
- Now, let's define some functions

```
function getArea(){
  return
  (this.radius*this.radius*3.14);
}

function getCircumference(){
  var diameter = this.radius*2;
  var circumference =
  diameter*3.14;
  return circumference;
}
```
- Associate names and functions in constructor

```
function Circle(radius){
  this.radius = radius;
  this.getArea = getArea;
  this.getCircumference =
  getCircumference;
}
```
- Call

```
alert(bigCircle.getArea()); //
displays 31400
alert(bigCircle.getCircumfe
nce()); // displays 618
alert(smallCircle.getArea());
// displays 12.56
alert(smallCircle.getCircumfe
rence()); // displays 12.56
```

# Inner Functions

```
❑ function Circle(radius){  
    function getArea(){  
        return (this.radius*this.radius*3.14);  
    }  
    function getCircumference(){  
        var diameter = this.radius*2;  
        var circumference = diameter*3.14;  
        return circumference;  
    }  
    this.radius = radius;  
    this.getArea = getArea;  
    this.getCircumference = getCircumference;  
}
```

# Object Categories

- There are three object categories
  - Native Objects
    - Native objects are those objects supplied by JavaScript. Examples of these are String, Number, Array, Image, Date, Math, etc.
  - Host Objects
    - Host objects are objects that are supplied to JavaScript by the browser environment. Examples of these are window, document, forms, etc
  - User-Defined Objects.
    - user-defined objects are those that are defined by the programmer.
- A fundamental concept in JavaScript is that every element that can hold properties and methods is an object, except for the primitive data types

# Discuss

A decorative horizontal bar consisting of a series of colored segments in black, blue, light blue, yellow, and teal, spanning the width of the slide.

## Rich Internet Applications –AJAX

# What is AngularJS

MVC JavaScript Framework by Google  
for Rich Web Application Development

# Why AngularJS?

“Other frameworks deal with HTML’s shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views”.

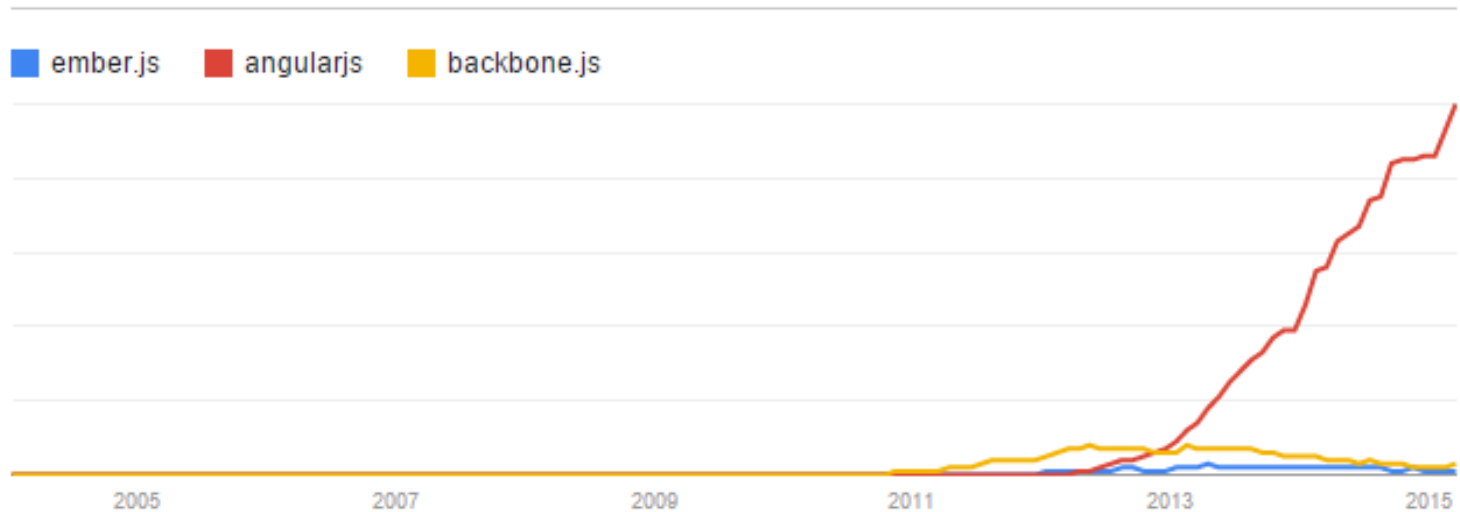
“ HTML? Build UI Declaratively! CSS? Animations! JavaScript? Use it the plain old way!”

# Why AngularJS?

- ❑ Structure, Quality and Organization
- ❑ Lightweight ( < 36KB compressed and minified)
- ❑ Free
- ❑ Separation of concern
- ❑ Modularity
- ❑ Extensibility & Maintainability
- ❑ Reusable Components

# Other Javascript MV\* Frameworks

- BackboneJS
- EmberJS





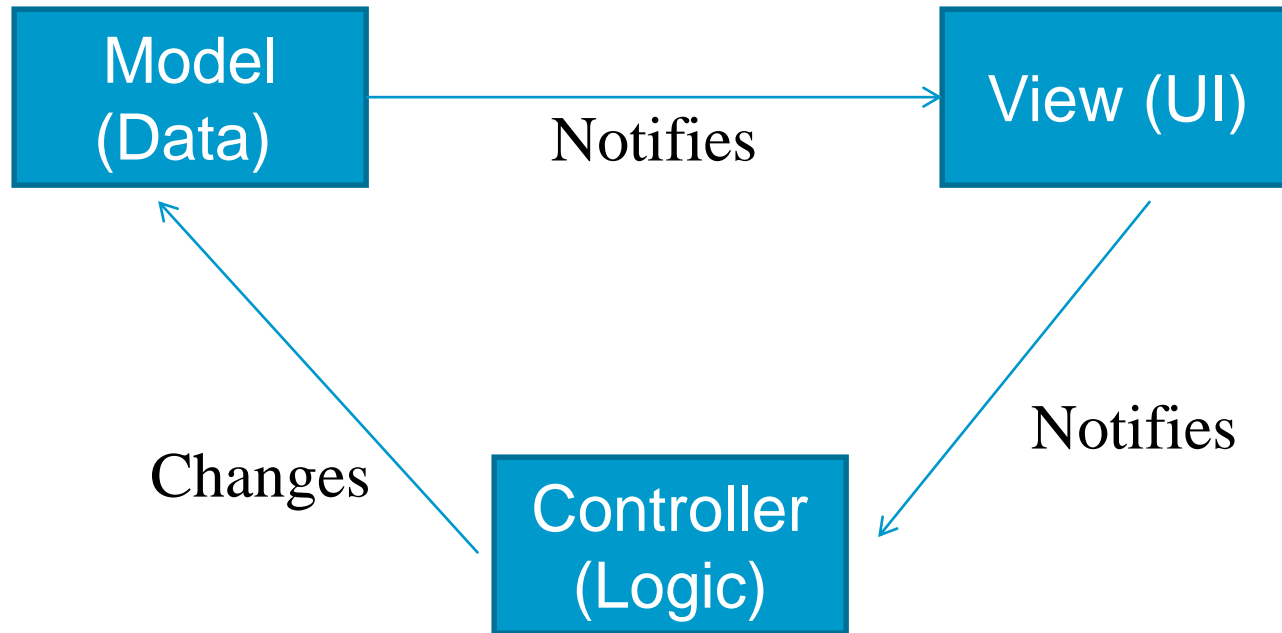
# Features of AngularJS

- ❑ Two-way Data Binding – Model as single source of truth
- ❑ Directives – Extend HTML
- ❑ MVC
- ❑ Dependency Injection
- ❑ Testing
- ❑ Deep Linking (Map URL to route Definition)
- ❑ Server-Side Communication

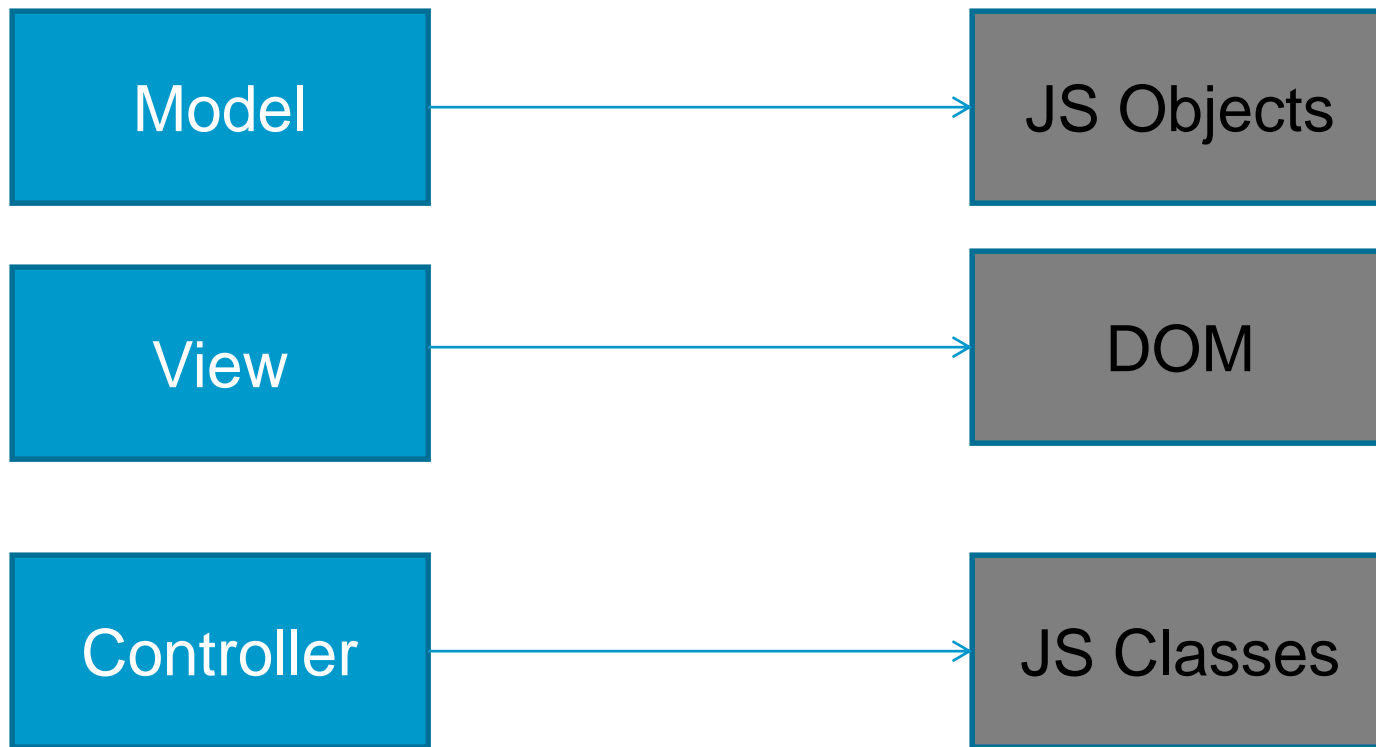
# Data Binding

```
<html ng-app>
<head>
  <script src='angular.js'></script>
</head>
<body>
  <input ng-model='user.name'>
  <div ng-show='user.name'>Hi
    {{user.name}}</div>
</body>
</html>
```

# MVC



# MVC



# Hello AngularJS

```
<p ng:init="greeting = 'Hello  
World!'">{{greeting}}</p>
```

# What Is Angular?

- ❑ AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Angular's data binding and dependency injection eliminate much of the code you would otherwise have to write. And it all happens within the browser, making it an ideal partner with any server technology.

# What Is Angular?

- The impedance mismatch between dynamic applications and static documents is often solved with:
  - **a library** - a collection of functions which are useful when writing web apps. Your code is in charge and it calls into the library when it sees fit. E.g., jQuery.
  - **frameworks** - a particular implementation of a web application, where your code fills in the details. The framework is in charge and it calls into your code when it needs something app specific. E.g., durandal, ember, etc.
- Angular takes another approach. It attempts to minimize mismatch by creating new HTML constructs. Angular teaches the browser new syntax through a construct we call *directives*.

# A complete client-side solution

- Angular is not a single piece in the overall puzzle of building the client-side of a web application. It handles all of the DOM and AJAX glue code you once wrote by hand and puts it in a well-defined structure.
- Angular comes with the following out-of-the-box:
  - Everything you need to build a CRUD app : Data-binding, basic templating directives, form validation, routing, deep-linking, reusable components and dependency injection.
  - Testability story: Unit-testing, end-to-end testing, mocks and test harnesses.
  - Seed application with directory layout and test scripts as a starting point.



# The Zen of Angular

- Angular is built around the belief that declarative code is better than imperative when it comes to building UIs and wiring software components together, while imperative code is excellent for expressing business logic.
- Considerations
  - Decouple DOM manipulation from app logic which dramatically improves the testability of the code.
  - Regard app testing as equal in importance to app writing. Testing difficulty is dramatically affected by the way the code is structured.
  - Decouple the client side of an app from the server side, allowing development work to progress in parallel, and allows for reuse of both sides.
  - Framework should guide developers through the entire journey of building an app: From designing the UI, through writing logic, to testing.
  - It is always good to make common tasks trivial and difficult tasks possible.

# The Zen of Angular

- Angular frees you from the following pains
  - Registering callbacks
  - Manipulating HTML DOM programmatically
  - Marshaling data to and from the UI
  - Writing tons of initialization code just to get started

# Conceptual Overview

- Template - HTML with additional markup
- Directives - extend HTML with custom attributes and elements
- Model - data shown to user in view and with which user interacts
- Scope - context where model is stored so that controllers, directives and expressions can access it
- Expressions - access variables and functions from the scope
- Compiler - parses the template and instantiates directives and expressions
- Filter - formats the value of an expression for display to the user

# Conceptual Overview

- View - what the user sees (the DOM)
- Data Binding - sync data between the model and the view
- Controller - the business logic behind views
- Dependency Injection - Creates and wires objects and functions
- Injector - dependency injection container
- Module - container for the different parts of app including controllers, services, filters, directives which configures the Injector
- Service - reusable business logic independent of views

# Example

```
<div ng-app ng-init="qty=1;cost=2"> <b>Invoice:</b>
<div>
```

```
Quantity: <input type="number" min="0" ng-model="qty">
```

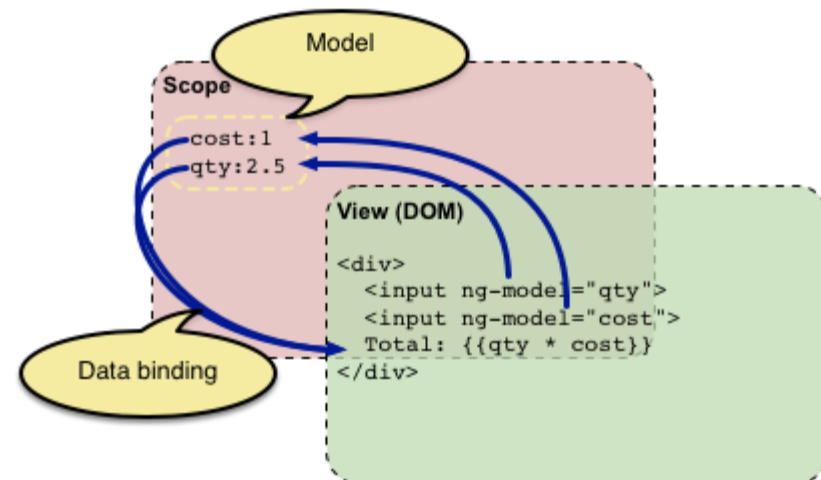
```
Costs:<input type="number" min="0" ng-model="cost">
```

```
<b>Total:</b> {{qty * cost | currency}} </div>
```

- This looks like normal HTML, with some new markup. In Angular, a file like this is called a **template**. When Angular starts your application, it parses and processes this new markup from the template using the compiler. The loaded, transformed and rendered DOM is then called the **view**.

# Example

- The first kind of new markup are the **directives**. They apply special behavior to attributes or elements in the HTML. In the example above we use the *ng-app* attribute, which is linked to a directive that automatically initializes our application. Angular also defines a directive for the *input* element that adds extra behavior to the element. The *ng-model* directive stores/updates the value of the input field into /from a variable/from a variable.



# Example

- The second kind of markup - `{{ expression | filter }}`:
  - When the compiler encounters this markup, it will replace it with the evaluated value of the markup. An **expression** in a template is a JavaScript - like code snippet that allows to read and write variables. Angular provides a scope for the variables accessible to expressions. Values that are stored in variables on the scope are referred to as the model.
- The example above also contains a **filter**.
  - A filter formats the value of an expression for display to the user. In the example above, the filter currency formats a number into an output that looks like money.
- The important thing in the example is that Angular provides live bindings: Whenever the input +values change, the value of the expressions are recalculated and the DOM is updated with their values. The concept behind this is two-way data binding.

# Requirement

- Create a text box to accept name, and display Hello, name. As soon as name changes message should be modified.
  - JS
    - Create text input, handle onchange or keydown kind of events and modify
  - AngularJS
    - Data Binding (Model)M



# JS Vs Angular

```
<script lang="text/javascript" >
function test() {
    document.getElementById("
abc").innerHTML =
    document.getElementById("t
est").value; }
</script> </head> <body>
<input type="text" id ="test"
    onchange="test()"
    onkeypress=".. ><p/>
Hello, <label id="abc" ></label>
</body> </html>
```

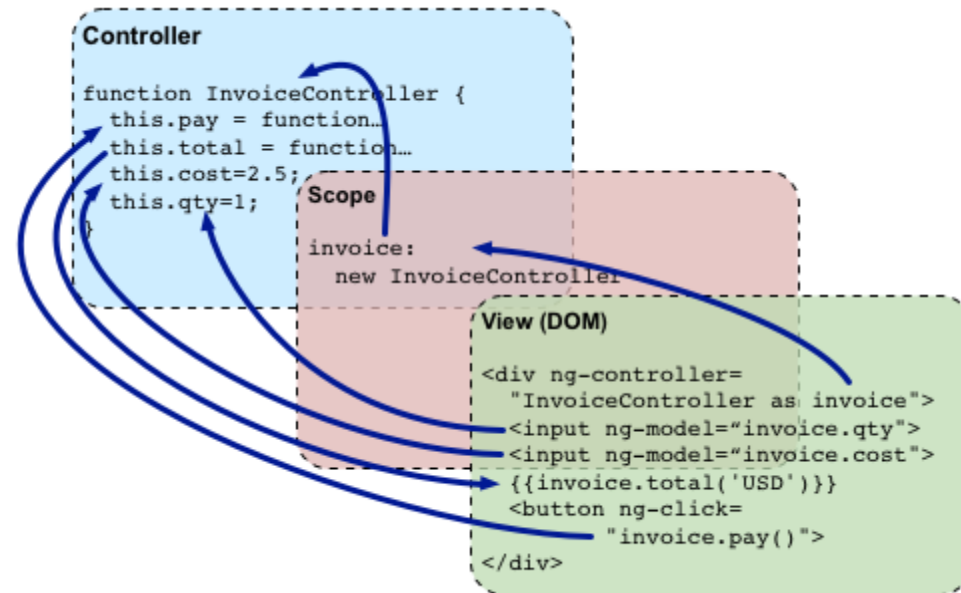
```
<script
    src="angular.js"></script>

</head>
<body>
<input type="text" ng-
    model="timepass" /><p/>
Hello, {{ timepass }}
</body>
</html>
```

# Adding UI logic: Controllers

```
angular.module('invoice1', [])
.controller('InvoiceController', function()
{.....});
```

```
<div ng-app="invoice1"
ng-
controller="InvoiceController as invoice">
```



## View Independent Business Logic: Services

- When the application grows it is a good practice to move view independent logic from the controller into a **Service**, so it can be reused by other parts of the application as well. Later on, we could also change that service without changing controller.

- Syntax

```
angular.module('finance2', [])  
    .factory('currencyConverter', function() {}  
);
```



A **F**ast **AND** **S**teady Approach

# Bootstrap



# Bootstrap

- Angular initialization process

```
<html xmlns:ng="http://angularjs.org" ng-app>
```

```
  <body> ...
```

```
    <script src="angular.js"></script>
```

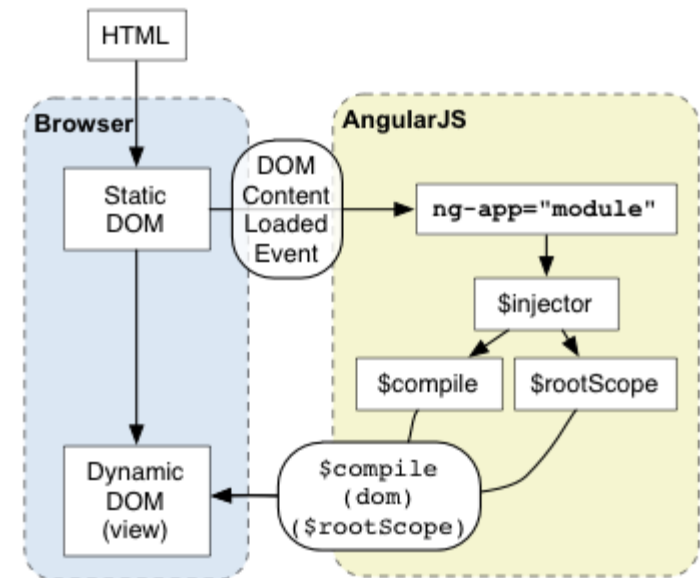
```
  </body>
```

```
</html>
```

- Place the script tag at the bottom of the page. It improves app load time because the HTML loading is not blocked by loading of the angular.js script.
- Place ng-app to the root of your application, typically on the <html> tag if you want angular to auto-bootstrap your application.
- If you choose to use the old style directive syntax ng: then include xml-namespace in html to make IE happy. (This is here for historical reasons, and we no longer recommend use of ng:.)

# Automatic Initialization

- Angular initializes automatically upon DOMContentLoaded event or when the angular.js script is evaluated if at that time document.readyState is set to 'complete'. At this point Angular looks for the ng-app directive which designates your application root. If the ng-app directive is found then Angular will:
  - load the module associated with the directive.
  - create the application injector
  - compile the DOM treating the ng-app directive as the root of the compilation. This allows you to tell it to treat only a portion of the DOM as an Angular application.



# Manual Initialization

**manually bootstrapping your app - not use the ng-app directive**

```
<!doctype html>
<html>
<body>
<div ng-controller="MyController"> Hello {{greetMe}}! </div>
  <script src="http://code.angularjs.org/snapshot/angular.js">
</script>
<script>
angular.module('myApp', []) .controller
('MyController', ['$scope', function ($scope) { $scope.greetMe =
  'World'; }]);
  angular.element(document).ready(function() {
    angular.bootstrap(document, ['myApp']); });
</script>
</body>
</html>
```

# Digest Cycle

- Understand two-way binding
- <http://www.sitepoint.com/understanding-angulars-apply-digest/>
- Understand delayed binding



# Application Structure

- How to maintain directory structures
- Tools to generate simple directory structure
  - Yeoman
- Seed application of Angular

# App Skeleton

## FOLDERS

- ▼ flfeeder-part1
  - ▼ app
    - ▶ bower\_components
    - ▶ css
    - ▶ img
    - ▼ js
      - app.js
      - controllers.js
      - directives.js
      - filters.js
      - services.js
    - ▶ partials
      - index-async.html
      - index.html
      - npm-debug.log

# Data Binding



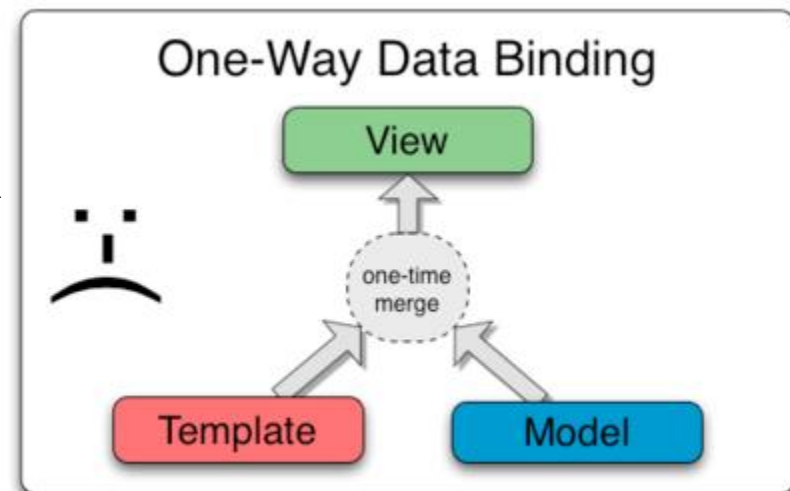
# Data Binding

- Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

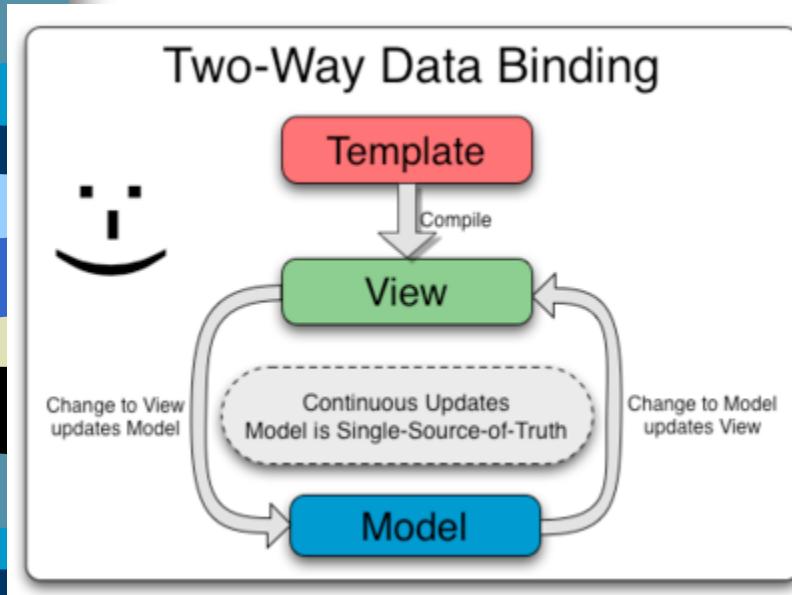
# Data Binding in Classical Template Systems

Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model.

This means that the developer has to write code that constantly syncs the view with the model and the model with the view.



# Data Binding in Angular Templates



Angular templates work differently. First the is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

# Requirement

- Create Controller to initialize array of first names and a function to add new entry in array
  - Functions or variables which need to be exposed outside should be initialized like
    - `$scope.names= ["aa", "bb"];`
    - `$scope.add = function (newname) {...}`

# Requirement

- In a simple table show all elements using ng-repeat tag

```
<table border='1' >  
  <tr ng-repeat="n in names">  
    <td>{{n}}</td></tr>  
</table>
```

- Create a textbox and button to add new names in the array

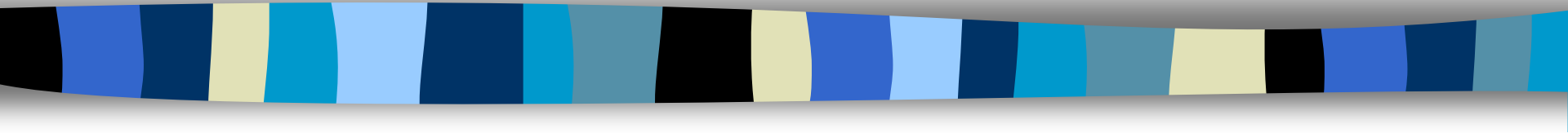
```
<input ng-model="name" /><input type="button" ng-click="add(name)" /><p>
```





A **F**ast **AND** **S**teady Approach

# Templates



# Templates

- Templates are written with HTML that contains Angular-specific elements and attributes. Angular combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.
- Following are the types of Angular elements and attributes you can use:
  - Directive - An attribute or element that augments an existing DOM element or represents a reusable DOM component.
  - Markup - The double curly brace notation `{{ }}` to bind expressions to elements is built-in Angular markup.
  - Filter - Formats data for display.
  - Form controls - Validates user input.

# Templates

- In a simple app, the template consists of HTML, CSS, and Angular directives contained in just one HTML file (usually index.html).
- In a more complex app, you can display multiple views within one main page using "partials" – segments of template located in separate HTML files. You can use the ngView directive to load partials based on configuration passed to the \$route service.

# What are Directives?

- At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** ([\\$compile](#)) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

# Matching Directives

- Angular normalizes an element's tag and attribute name to determine which elements match which directives. We typically refer to directives by their case-sensitive camelCase normalized name (e.g. ngModel). However, since HTML is case-insensitive, we refer to directives in the DOM by lower-case forms, typically using dash-delimited attributes on DOM elements (e.g. ng-model).
- The normalization process is as follows:
  - Strip x- and data- from the front of the element/attributes.
  - Convert the :, -, or \_-delimited name to camelCase.

# Matching Directives

❑ `<div ng-controller="Controller"> Hello  
<input ng-model='name'> <hr/>  
 <span ng-bind="name"></span> <br/>  
 <span ng:bind="name"></span> <br/>  
 <span ng_bind="name"></span> <br/>  
 <span data-ng-bind="name" /> <br/>  
 <span x-ng-bind="name"></span> <br/>  
</div>`

# Matching Directives

Best Practice: Prefer using directives via tag name and attributes over comment and class names

- All of the Angular-provided directives match attribute name, tag name, comments, or class name. The following demonstrates the various ways a directive (myDir in this case) can be referenced from within a template:

```
<my-dir></my-dir>
```

```
<span my-dir="exp"></span>
```

```
<!-- directive: my-dir exp -->
```

```
<span class="my-dir: exp;"></span>
```

# Bindings

- Text and attribute bindings

```
<a ng-href="img/{{username}}.jpg">Hello  
  {{username}}!</a>
```

- ngAttr attribute bindings

```
<svg> <circle ng-attr-cx="{{cx}}"></circle> </svg>
```

- Where simple `<circle cx="{{cx}}"></circle>` is not processed because of eager processing



# Angular Expressions



# Expressions

- Angular expressions are JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`.
- Valid Expression examples
  - `1+2`
  - `a+b`
  - `user.name`
  - `items[index]`

# Angular vs. JavaScript Expressions

- Context: JavaScript expressions are evaluated against the global window. In Angular, expressions are evaluated against a scope object.
- Forgiving: In JavaScript, trying to evaluate undefined properties generates ReferenceError or TypeError. In Angular, expression evaluation is forgiving to undefined and null.
- No Control Flow Statements: You cannot use the following in an Angular :expression conditionals, loops, or exceptions.
- No Function Declarations: You cannot declare functions in an Angular expression, even inside ng-init directive.
- No RegExp Creation With Literal Notation: You cannot create regular expressions in an Angular expression.

# One-time binding

- Why this feature?
- An expression that starts with `::` is considered a one-time expression. One-time expressions will stop recalculating once they are stable, which happens after the first digest if the expression result is a non-undefined value `{{::name}}`

# Requirement

- ❑ Create a friend list and initialize in controller. Create a UI to add new friends and display current friends table. It should display original list of friends and modified list of friends.
- ❑ Modify same code to add a small button, where click on that button should display original list.



A **F**ast **AND** **S**teady Approach

# Filters



# Filters

- A filter formats the value of an expression for display to the user. They can be used in view templates, controllers or services and it is easy to define your own filter.
- Using filters in view templates
  - {{ expression | filter }}
  - {{ expression | filter1 | filter2 | ... }}
  - {{ expression | filter:argument1:argument2:... }}

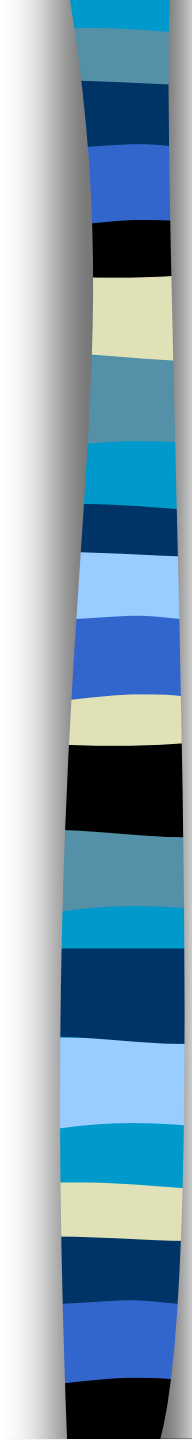
# Using filters in controllers, services, and directives

- ❑ Filters can be used in controllers, services, and directives. Inject a dependency with the name `<filterName>Filter` to your controller/service/directive
- ❑ using the dependency `numberFilter` will inject the number filter. The injected argument is a function that takes the value to format as first argument and filter parameters starting with the second argument.



# Filters in Controller

```
app.controller("arrctrl", function ($scope, $filter){  
    $scope.names= ["aa", "bb"];  
    $scope.add = function (newname)  
    {  
        $scope.names.push($filter("uppercase")(newname));  
    }  
});
```



Name	Description
filter	Selects a subset of items from array and returns it as a new array.
currency	Formats a number as a currency (ie \$1,234.56). When no currency symbol is provided, default symbol for current locale is used.
number	Formats a number as text.
date	Formats date to a string based on the requested format.
json	Allows you to convert a JavaScript object into JSON string.
lowercase	Converts string to lowercase.
uppercase	Converts string to uppercase.
limitTo	Creates a new array or string containing only a specified number of elements. The elements are taken from either the beginning or the end of the source array, string or number, as specified by the value and sign (positive or negative) of limit. If a number is used as input, it is converted to a string.
orderBy	Orders array by the expression predicate. It is ordered alphabetically for strings and numerically for numbers. Note: if you notice numbers are not being sorted correctly, make sure they are saved as numbers and not strings.

# Requirement

- ❑ Sorted friend list should be shown in upper case

# Requirement

- ❑ Create a friend class with fields like firstname, lastname, city.
- ❑ Provide insert / list operations on the same
- ❑ List option should give selection criteria based on firstname, lastname or city.



A **F**ast **AND** **S**teady Approach

# Modules



# What is a Module?

- You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.
  - Why?
    - Most applications have a main method that instantiates and wires together the different parts of the application.
    - Angular apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped.
- There are several advantages to this approach:
- The declarative process is easier to understand.
  - You can package code as reusable modules.
  - The modules can be loaded in any order (or even in parallel) because modules delay execution.
  - Unit tests only have to load relevant modules, which keeps them fast.
  - End-to-end tests can use modules to override configuration.

# Recommended Setup

- Break your application to multiple modules like this:
  - A module for each feature
  - A module for each reusable component (especially directives and filters)
  - And an application level module which depends on the above modules and contains any initialization code.

# Applications

- Applications
- Model
  - How to create a model
  - Explicit models
  - Implicit models



# Controllers



# ngController

- The ngController directive attaches a controller class to the view. This is a key aspect of how angular supports the principles behind the Model-View-Controller design pattern.
- MVC components in angular:
  - Model — Models are the properties of a scope; scopes are attached to the DOM where scope properties are accessed through bindings.
  - View — The template (HTML with data bindings) that is rendered into the View.
  - Controller — The ngController directive specifies a Controller class; the class contains business logic behind the application to decorate the scope with functions and values
- Note that you can also attach controllers to the DOM by declaring it in a route definition via the \$route service. A common mistake is to declare the controller again using ng-controller in the template itself. This will cause the controller to be attached and executed twice.

# Understanding Controllers

- ❑ Controller is a JavaScript constructor function that is used to augment the Angular Scope.
- ❑ When a Controller is attached to the DOM via the ng-controller directive, Angular will instantiate a new Controller object, using the specified Controller's constructor function. A new child scope will be available as an injectable parameter to the Controller's constructor function as \$scope.

# Do and Don't for Controller

- Use controllers to:
  - Set up the initial state of the \$scope object.
  - Add behavior to the \$scope object.
- Do not use controllers to:
  - Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. Angular has databinding for most cases and directives to encapsulate manual DOM manipulation.
  - Format input — Use angular form controls instead.
  - Filter output — Use angular filters instead.
  - Share code or state across controllers — Use angular services instead.
  - Manage the life-cycle of other components (for example, to create service instances).

# Scope Inheritance Example

```
var myApp = angular.module('scopeInheritance', []);  
myApp.controller('MainController', ['$scope', function($scope) {  
    $scope.timeOfDay = 'morning';  
    $scope.name = 'Nikki';  
}]);  
myApp.controller('ChildController', ['$scope', function($scope) {  
    $scope.name = 'Mattie';  
}]);  
myApp.controller('GrandChildController', ['$scope', function($scope) {  
    $scope.timeOfDay = 'evening';  
    $scope.name = 'Gingerbread Baby';  
}]);
```

# Scope Inheritance Example

```
<div class="spicy">
  <div ng-controller="MainController">
    <p>Good {{timeOfDay}}, {{name}}!</p>
    <div ng-controller="ChildController">
      <p>Good {{timeOfDay}}, {{name}}!</p>
      <div ng-controller="GrandChildController">
        <p>Good {{timeOfDay}}, {{name}}!</p>
      </div>
    </div>
  </div>
</div>
```

Good morning, Nikki!

Good morning, Mattie!

Good evening, Gingerbread Baby!

```
div.spicy div {
  padding: 10px;
  border: solid 2px blue;
}
```

# Dependency Injection



# Dependency Injection

- Is a software design pattern that deals with how components get hold of their dependencies.
- The Angular injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.



# Using Dependency Injection

- ❑ DI is pervasive throughout Angular. You can use it when defining components or when providing run and config blocks.
- ❑ Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function. These components can be injected with "service" and "value" components as dependencies.
- ❑ Controllers are defined by a constructor function, which can be injected with any of "service" & "value" components as dependencies, but can also be provided with special dependencies.
- ❑ The run method accepts a function, which can be injected with "service", "value" and "constant" components as dependencies. Note that you cannot inject "providers" into run blocks.
- ❑ The config method accepts a function, which can be injected with "provider" and "constant" components as dependencies. Note that you cannot inject "service" or "value" components into configuration.

# Factory Methods

- The way you define a directive, service, or filter is with a factory function. The factory methods are registered with modules.
- The recommended way of declaring factories is:  

```
angular.module('myModule', [])  
.factory('serviceId', ['depService', function(depService) {  
  // ...}])  
.directive('directiveName', ['depService', function(depService)  
  { // ...}])  
.filter('filterName', ['depService', function(depService) {  
  // ...}]);
```

# Module Methods

- We can specify functions to run at configuration and run time for a module by calling the config and run methods. These functions are injectable with dependencies just like the factory functions above.

```
angular.module('myModule', [])  
.config(['depProvider', function(depProvider) {  
  // ... }])  
.run(['depService', function(depService) {  
  // ... }]);
```

# Controllers

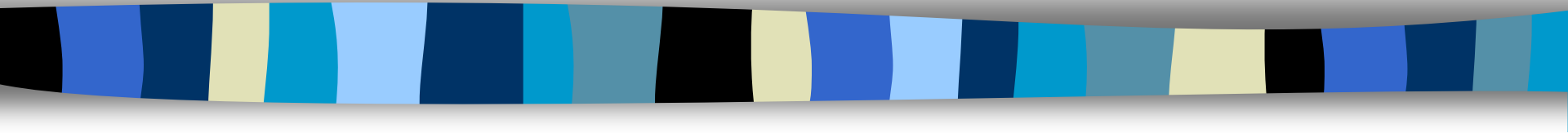
- Controllers are "classes" or "constructor functions" that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way of declaring Controllers is using the array notation:

```
someModule.controller('MyController', ['$scope', 'dep1', 'dep2',  
  function($scope, dep1, dep2) {  
    ...  
    $scope.aMethod = function() { ...}  
    ...  
  }]);
```

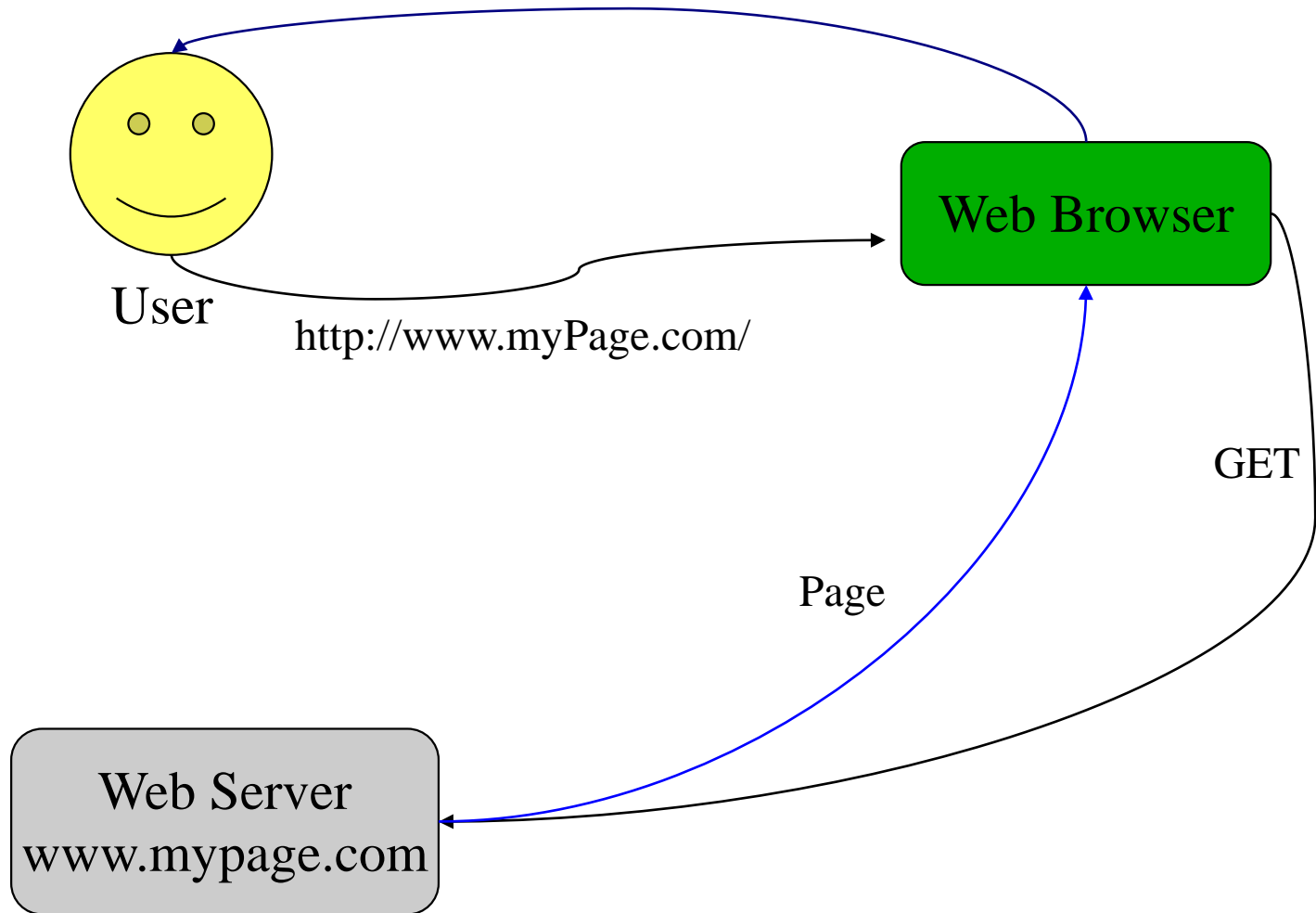


A **F**ast **AND** **S**teady Approach

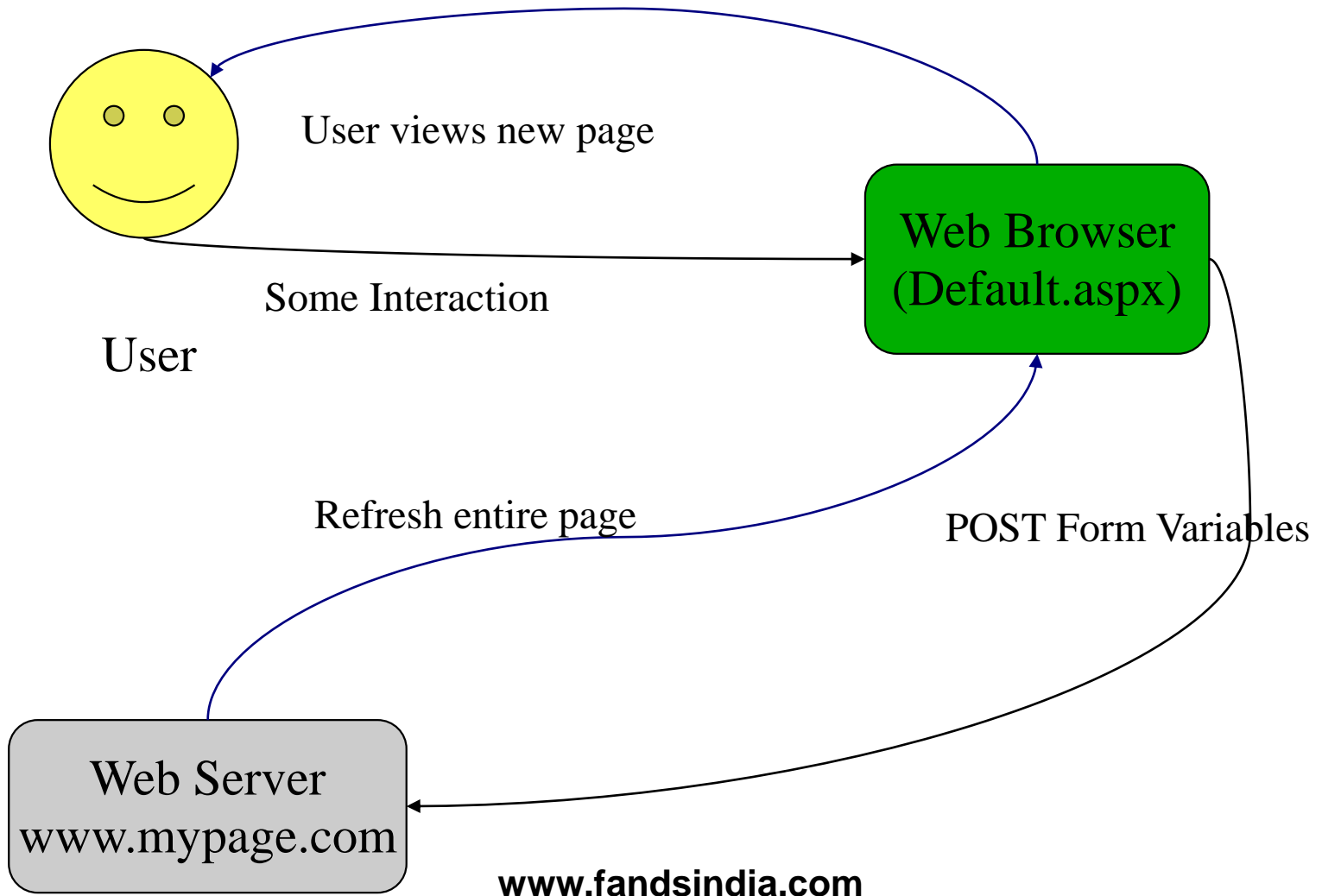
# AJAX



# Non-Ajax

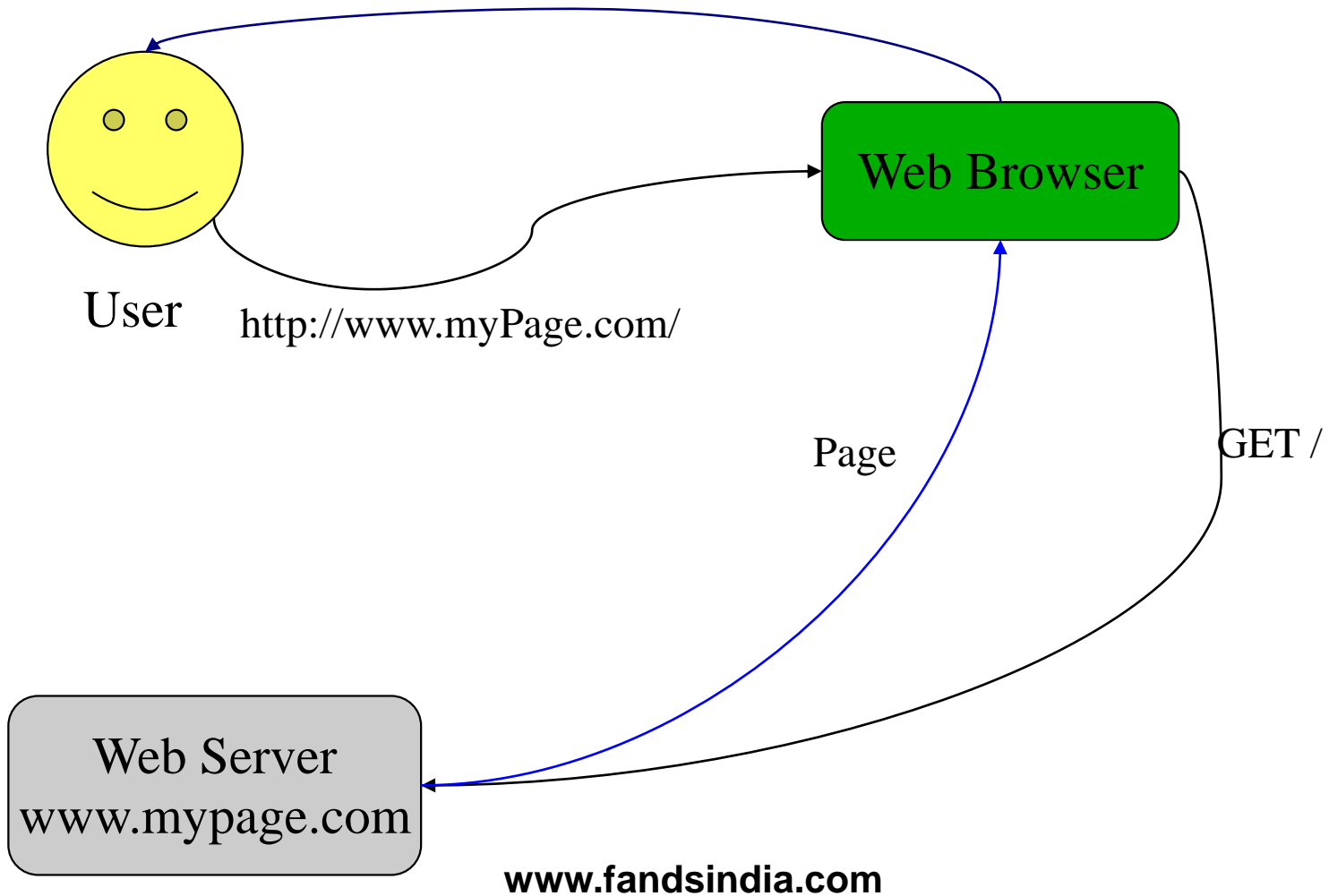


# Non-Ajax



# Ajax Enabled

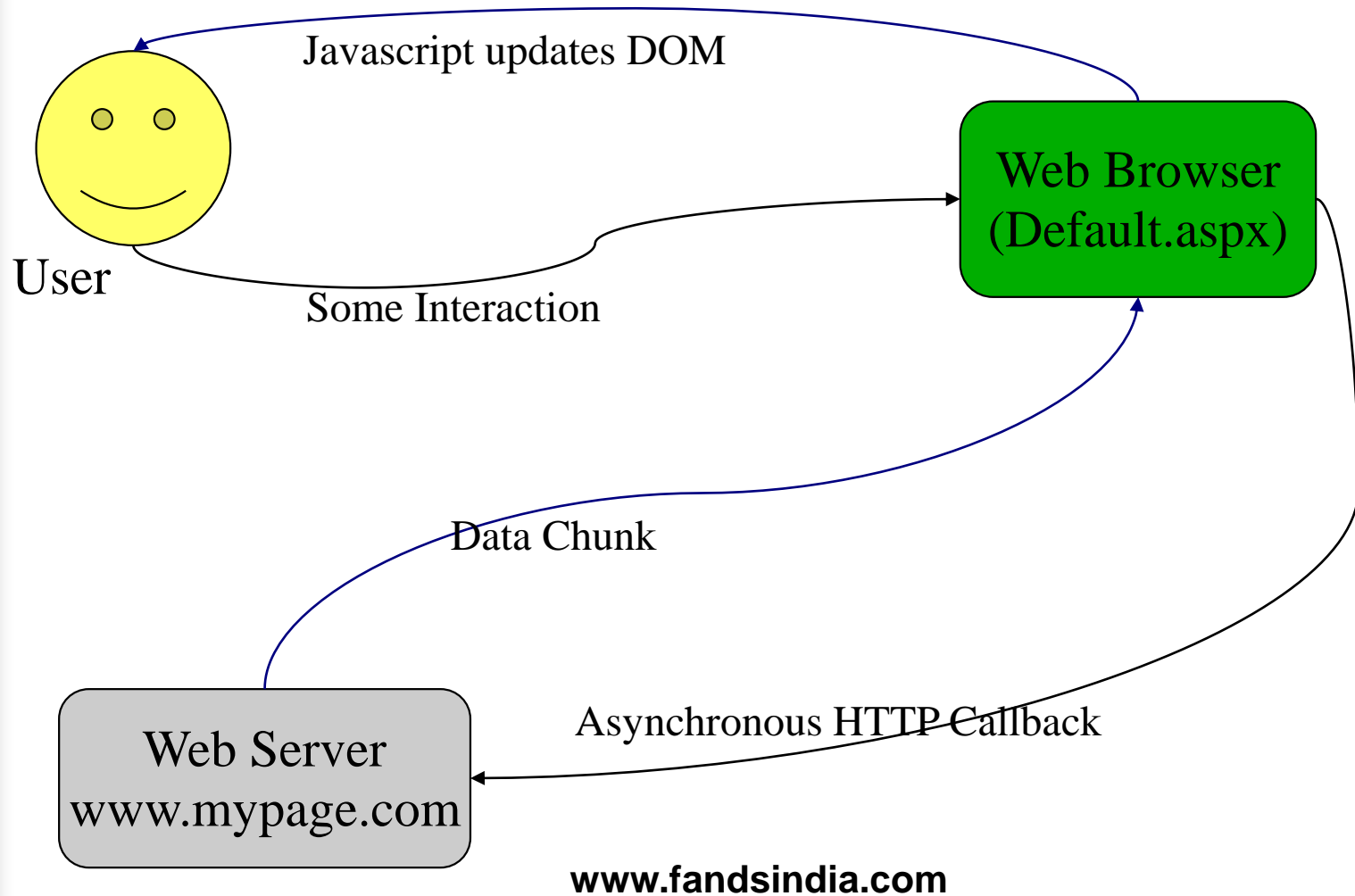
1<sup>st</sup> part same as before





# Ajax Enabled

## Here is the different





A **F**ast **AND** **S**teady Approach

# XMLHttpRequest



# The XMLHttpRequest object

- How to Create Object
  - xmlhttp = new XMLHttpRequest(); ????
  - JavaScript and Browser Dependency
- Basic methods and Properties
  - Open
  - readyState
  - responseXML
  - .responseText
  - send

# How to create XMLHttpRequest

- Firefox, Opera 8.0+, Safari
  - `xmlHttp=new XMLHttpRequest();`
- Internet Explorer 6.0+
  - `xmlHttp=new ActiveXObject("Msxml2.XMLHTTP");`
- ie 5.5+
  - `xmlHttp=new`  
`ActiveXObject("Microsoft.XMLHTTP");`
- Browser doesn't support Ajax

# XMLHttpRequest

```
var xmlHttp;
try {
    xmlHttp = new XMLHttpRequest();
    alert("your browser - Firefox, Opera 8.0+, Safari ...");
} catch (e) {
    try {
        xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
        alert("your browser - Internet Explorer 6.0 +++++...");
    } catch (e) {
        try {
            xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
            alert("your browser - Internet Explorer 5.5...");
        } catch (e) {
            alert("Your browser does not support AJAX!");
        }
    }
}
```

# Process Request

```
xmlHttp.open("GET","aa.txt",true);
xmlHttp.onreadystatechange=function()
{
    if (xmlHttp.readyState ==4)
    {
        if (xmlHttp.status ==200)
        {
            document.getElementById("spid").innerHTML =xmlHttp.responseText;
        }
        else
        {
            alert("Some Problem");
        }
    }
}
xmlHttp.send(null);
}
```

# \$http

- Simple Form for get/post etc

```
$http.get('/someUrl')  
  . success(function(data, status, headers, config) { })  
  . error(function(data, status, headers, config) {});
```

- To pass data to get  
url?info

- To pass data to post

```
$http.post('/someUrl', {msg:'hello word!'})
```

OR

```
var req = { method: 'POST', url: 'http://example.com', headers: {  
'Content-Type': undefined }, data: { test: 'test' } };
```

```
$http(req)
```

# List of shortcut methods

- ❑ `$http.get`
- ❑ `$http.head`
- ❑ `$http.post`
- ❑ `$http.put`
- ❑ `$http.delete`
- ❑ `$http.jsonp`
- ❑ `$http.patch`



# \$q

- The AngularJS \$q service is said to be inspired by Chris Kowal's Q library ([github.com/kris/kowal/q](https://github.com/kris/kowal/q)).
- The library's goal is to allow users to monitor asynchronous progress by providing a "promise" as a return from a call.

# Promise Use

```
var promise = callthatrunsInbackground();  
promise.then(  
  function(answer) {  
    // do something  
  },  
  function(error) {  
    // report something  
  },  
  function(progress) {  
    // report progress  
  });
```

# Use in Angular

- A number of Angular services return promises: `$http`, `$interval`, `$timeout`, for example. All promise returns are single objects; you're expected to research the service itself to find out what it returns. For example, for `$http.get`, the promise returns an object with four keys: `data`, `status`, `headers`, and `config`. Those are the same as the four parameters fed to the success callback if you use those semantics:

# Promise with http

// this

```
$http.get('/api/v1/movies/avengers')  
  .success(function(data, status, headers, config) {  
    $scope.movieContent = data;  
  });
```

// is the same as

```
var promise = $http.get('/api/v1/movies/avengers');
```

```
promise.then(  
  function(payload) {  
    $scope.movieContent = payload.data;  
  });
```

# Promises and Services

```
angular.module('atTheMoviesApp', [])
.controller('GetMoviesCtrl', function($log, $scope, movieService) {
  $scope.getMovieListing = function(movie) {
    var promise = movieService.getMovie('avengers');
    promise.then( function(payload) {
      $scope.listingData = payload.data;    },
      function(errorPayload) {
        $log.error('failure loading movie', errorPayload);
      });
  });
}.factory('movieService', function($http) {
  return {
    getMovie: function(id) {
      return $http.get('/api/v1/movies/' + id);
    }
  }
});
```

# Transformer & Interceptors

## □ Transformers

- transformRequest and transformResponse
- these properties can be a single function that returns the transformed value (function(data, headersGetter, status)) or an array of such transformation functions, which allows you to push or unshift a new transformation function into the transformation chain.

# Interceptors

- For purposes of global error handling, authentication, or any kind of synchronous or asynchronous pre-processing of request or postprocessing of responses, it is desirable to be able to intercept requests before they are handed to the server and responses before they are handed over to the application code that initiated these requests. The interceptors leverage the promise APIs to fulfill this need for both synchronous and asynchronous pre-processing.
- The interceptors are service factories that are registered with the `$httpProvider` by adding them to the `$httpProvider.interceptors` array. The factory is called and injected with dependencies (if specified) and returns the interceptor.

# Types of Interceptors

- request: interceptors get called with a http config object. The function is free to modify the config object or create a new one. The function needs to return the config object directly, or a promise containing the config or a new config object.
- requestError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.
- response: interceptors get called with http response object. The function is free to modify the response object or create a new one. The function needs to return the response object directly, or as a promise containing the response or a new response object.
- responseError: interceptor gets called when a previous interceptor threw an error or resolved with a rejection.



# Interceptor as service

```
angular.module("services.myhttpint",[])  
  .factory('myhttpint',['$q',function($q) {  
    var myint =  
    { request : function(config) {  
      console.log("request function of interceptor invoked ...");  
      return config || $q.when(config);  
    } }  
    return myint;  
  }]);
```

```
angular.module("modname",['services.myhttpint']).config(function(  
  $httpProvider) {  
    $httpProvider.interceptors.push("myhttpint");  
  } );
```

# Routes and Views



# Views

- Single Page Application
- Multiple Views
- Design Patterns
  - View Helpers
  - View Resolvers
  - Templates

# Maintainability Issues

- As we add more and more logic to an app, it grows and soon become difficult to manage. Dividing it in Views and using Routing to load different part of app helps in logically dividing the app and making it more manageable.
- Routing helps you in dividing your application in logical views and bind different views to Controllers.

# Routing

- The magic of Routing is taken care by a service provider that Angular provides out of the box called `$routeProvider`.
- When we use AngularJS's dependency injection and inject a service object in our Controller, Angular uses `$injector` to find corresponding service injector. Once it get a hold on service injector, it uses `$get` method of it to get an instance of service object. Sometime the service provider needs certain info in order to instantiate service object.
- Application routes in Angular are declared via the `$routeProvider`, which is the provider of the `$route` service. This service makes it easy to wire together controllers, view templates, and the current URL location in the browser. Using this feature we can implement deep linking, which lets us utilize the browser's history (back and forward navigation) and bookmarks.

# Introduction to \$routeProvider

```
var myapplication = angular.module('myapp', ['ngRoute']);
myapplication .config( function($routeProvider) {
$routeProvider.
    when('/addFriend', {templateUrl: 'templates/add-friend.html',
        controller: 'AddFriendController' }).
    when('/showFriend', { templateUrl: 'templates/show-friends.html',
        controller: 'ShowFriendsController' }).
    otherwise({
        redirectTo: '/showFriend'  });
});
myapplication.controller('AddFriendController', function($scope) {});
myapplication.controller('ShowFriendsController', function($scope) {});
```

# Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>AngularJS Routing example</title>
  </head>
  <body ng-app="myapp">
    <ul class="nav">
      <li><a href="#addFriend"> Add Friend </a></li>
      <li><a href="#showFriend"> Show Friends </a></li>
    </ul>
    <div ng-view=""></div>
  <script src="lib/angular.js"></script><script src="lib/angular-
    route.js"></script><script src="js/myapp.js"></script>
</html>
```

# Templates

- Templates/add-friend.html
  - <h2>Add New Friend</h2>
  - {{ message }}
- Templates/show-friends.html
  - <h2>Show Friend</h2>
  - {{ message }}



# Deferred Vs Promise, Chaining

- <http://fdietz.github.io/recipes-with-angular-js/consuming-external-services/deferred-and-promise.html>
- [https://docs.angularjs.org/api/ng/service/\\$q](https://docs.angularjs.org/api/ng/service/$q)

# More on Promises

- <http://chariotsolutions.com/blog/post/angularjs-corner-using-promises-q-handle-asynchronous-calls/>



A **F**ast **AND** **S**teady Approach

# Form and Form Controls



# Form and Form Controls

- ❑ Controls (input, select, textarea) are ways for a user to enter data. A Form is a collection of controls for the purpose of grouping related controls together.
- ❑ Form and controls provide validation services, so that the user can be notified of invalid input before submitting a form.
- ❑ Keep in mind that while client-side validation plays an important role in providing good user experience, it can easily be circumvented and thus can not be trusted. Server-side validation is still necessary for a secure application.

# Simple form

```
<div ng-controller="ExampleController">
  <form novalidate class="simple-form">
    Name: <input type="text" ng-model="user.name" /><br />
    E-mail: <input type="email" ng-model="user.email" /><br />
    Gender: <input type="radio" ng-model="user.gender"
    value="male" />male
    <input type="radio" ng-model="user.gender" value="female"
    />female<br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
<pre>form = {{user | json}}</pre>
<pre>master = {{master | json}}</pre> </div>
```

# HTML Input Types

- ❑ `<input type="text">`
- ❑ `<input type="password">`
- ❑ `<input type="submit">`
- ❑ `<input type="radio">`
- ❑ `<input type="checkbox">`
- ❑ `<input type="button">`

Input types, not supported by old web browsers, will behave as input type text.

HTML5 added new input types:

color  
Date, datetime  
datetime-local  
email  
month  
number  
range  
search  
tel  
time  
url  
week

# Using CSS classes

- To allow styling of form as well as controls, ngModel adds these CSS classes:
  - ng-valid: the model is valid
  - ng-invalid: the model is invalid
  - ng-valid-[key]: for each valid key added by \$setValidity
  - ng-invalid-[key]: for each invalid key added by \$setValidity
  - ng-pristine: the control hasn't been interacted with yet
  - ng-dirty: the control has been interacted with
  - ng-touched: the control has been blurred
  - ng-untouched: the control hasn't been blurred
  - ng-pending: any \$asyncValidators are unfulfilled

# Binding to form and control state

- A form is an instance of `FormController`. The form instance can optionally be published into the scope using the `name` attribute.
- An input control that has the `ngModel` directive holds an instance of `NgModelController`. Such a control instance can be published as a property of the form instance using the `name` attribute on the input control. The `name` attribute specifies the name of the property on the form instance.



# Binding to form and control state

- This implies that the internal state of both the form and the control is available for binding in the view using the standard binding primitives.
- This allows us to extend the above example with these features:
  - Custom error message displayed after the user interacted with a control (i.e. when \$touched is set)
  - Custom error message displayed upon submitting the form (\$submitted is set), even if the user didn't interact with a control

# Binding to form and control state

```
E-mail: <input type="email" ng-model="user.email"
      name="uEmail" required="" />
<br />
<div ng-show="form.$submitted ||
      form.uEmail.$touched">
  <span ng-show="form.uEmail.$error.required">
    Tell us your email.</span>
  <span ng-show="form.uEmail.$error.email">
    This is not a valid email.</span>
</div>
```

# Custom model update triggers

- By default, any change to the content will trigger a model update and form validation.
- Override this behavior using
  - `ng-model-options="{ updateOn: 'blur' }"` will update and validate only after the control loses focus.
  - You can set several events using a space delimited list.
    - `ng-model-options="{ updateOn: 'mousedown blur' }"`
- If you want to keep the default behavior and just add new events that may trigger the model update and validation, add "default" as one of the specified events.
  - `ng-model-options="{ updateOn: 'default blur' }"`

## Non-immediate (debounced) model updates

- ❑ Can delay model update / validation by using debounce key with the ngModelOptions directive. This delay will also apply to parsers, validators & model flags - \$dirty or \$pristine
  - ng-model-options="{ debounce: 500 }" will wait for half a second since the last content change before triggering the model update and form validation.
  - ng-model-options="{ updateOn: 'default blur', debounce: { default: 500, blur: 0 } }" will force immediate updates on some specific circumstances (like blur events).

# Requirement

- Create a ui to accept Customer data from user.
  - Customerid numeric,
  - Name String,
  - Company String,
  - Supportemail as email
  - City with drop down list,
  - Product support as checkbox for product1 to product5.
  - Delivery model supported as radio button with three options Urgent, Routine, Pre-planned.

# Validations

```
<form ng-app=".." ng-controller=".." name="myForm" novalidate>
<p>Email:<br>
<input type="email" name="email" ng-model="email" required>
<span style="color:red" ng-show="myForm.email.$dirty &&
    myForm.email.$invalid">
<span ng-show="myForm.email.$error.required">Email is
    required.</span>
<span ng-show="myForm.email.$error.email">Invalid email
    address.</span>
</span>
```

# Validations

Property	Description
\$dirty	The user has interacted with the field.
\$valid	The field content is valid.
\$invalid	The field content is invalid.
\$pristine	User has not interacted with the field yet.

# Requirement

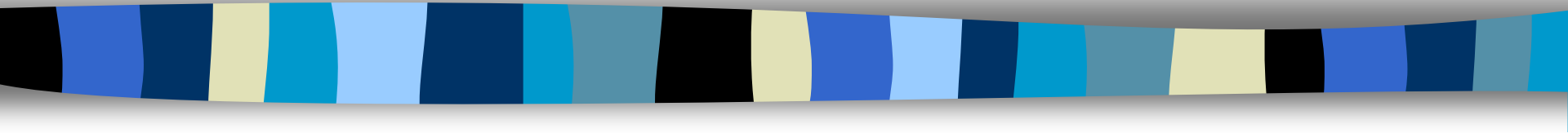
- Modify current form to include validations for
  - Customerid numeric from 10 - 20
  - Name String – min number of characters 5 and max 10
  - City with drop down list – required
  - Add a field to accept date





A **F**ast **AND** **S**teady Approach

# Testing



# Testing a Controller

- Because Angular separates logic from the view layer, it keeps controllers easy to test. Let's take a look at how we might test the controller below, which provides `$scope.grade`, which sets a property on the scope based on the length of the password.

```
angular.module('app', [])  
  .controller('PasswordController',  
    function  
      PasswordController($scope)  
    { $scope.password = "";  
      $scope.grade = function()  
      { var size =  
        $scope.password.length; if  
        (size > 8) { $scope.strength =  
        'strong'; } else if (size > 3) {  
        $scope.strength = 'medium'; }  
        else { $scope.strength =  
        'weak'; } }; });
```

# Testing a Controller

- Because controllers are not available on the global scope, we need to use `angular.mock.inject` to inject our controller first. The first step is to use the `module` function, which is provided by `angular-mocks`. This loads in the module it's given, so it is available in your tests. We pass this into `beforeEach`, which is a function Jasmine provides that lets us run code before each test. Then we can use `inject` to access `$controller`, the service that is responsible for instantiating controllers.

# Create Tests

```
describe('PasswordController', function() {  
  beforeEach(module('app'));  
  var $controller;  
  beforeEach(inject(function(_$controller_)  
  { // The injector unwraps the underscores (_) from around the  
    parameter names when matching  
    $controller = _$controller_; }));  
  describe('$scope.grade', function()  
  { it('sets the strength to "strong" if the password length is >8 chars',  
    function() { var $scope = {}; var controller =  
      $controller('PasswordController', { $scope: $scope });  
      $scope.password = 'longerthaneightchars'; $scope.grade();  
      expect($scope.strength).toEqual('strong'); }); }); });
```

# Testing Filters

```
myModule.filter('length', function()
{ return function(text)
  { return (" + (text || ")).length; }
});
describe('length filter', function()
{ it('returns 0 when given null', function()
  { var length = $filter('length');
    expect(length(null)).toEqual(0); });
  it('returns the correct value when given a string of chars',
    function()
    { var length = $filter('length');
      expect(length('abc')).toEqual(3); }); });
```

# End to End (E2E) Testing

- As applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions. Unit tests are the first line of defense for catching bugs, but sometimes issues come up with integration between components which can't be captured in a unit test. End-to-end tests are made to find these problems.
- Note: In the past, end-to-end testing could be done with a deprecated tool called Angular Scenario Runner. That tool is now in maintenance mode.

# E2E Testing

- Work with Protractor, an end to end test runner which simulates user interactions that will help you verify the health of your Angular application.
- Protractor is a Node.js program, and runs end-to-end tests that are also written in JavaScript and run with node. Protractor uses WebDriver to control browsers and simulate user actions.
- Protractor uses Jasmine for its test syntax.

# Testing - Practicals

- Additional tools for testing Angular applications
  - Jasmine
  - Karma
  - Angular-mocks



# Jasmine

- ❑ Jasmine is a behavior driven development framework for JavaScript that has become the most popular choice for testing Angular applications. Jasmine provides functions to help with structuring your tests and also making assertions. As your tests grow, keeping them well structured and documented is vital, and Jasmine helps achieve this.

# Jasmine

- In Jasmine we use the describe function to group our tests together:

```
describe("sorting the list of users", function() { //  
    individual tests go here });
```

- And then each individual test is defined within a call to the it function:

```
describe('sorting the list of users', function() {  
    it('sorts in descending order by default', function() {  
        // your test assertion goes here  
    });  
});
```

# Jasmine

- Finally, Jasmine provides matchers which let you make assertions:

```
describe('sorting the list of users', function() {  
  it('sorts in descending order by default', function() {  
    var users = ['jack', 'igor', 'jeff'];  
    var sorted = sortUsers(users);  
    expect(sorted).toEqual(['jeff', 'jack', 'igor']);  
  });  
});
```

# Karma

- ❑ Karma is a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests. You can configure Karma to run against a number of browsers, which is useful for being confident that your application works on all browsers you need to support. Karma is executed on the command line and will display the results of your tests on the command line once they have run in the browser.
- ❑ Karma is a NodeJS application, and should be installed through npm. Full installation instructions are available on the Karma website.

# Karma

- “karma init sample.conf.js” command will create sample.conf.js after asking preferences
  - which test framework to use (Jasmine in our case),
  - which files to monitor and use (this includes at least)
    - angular.js
    - angular-mocks.js (contains the definition for **module** and **inject**)
    - our own code to test
    - the files containing the tests or specifications
  - what browsers to use to run the test against (we’ll stick with Chrome)
  - The port on which the server is listening
  - whether to run the test automatically or manually when any of the monitored files changes

# Sample File

```
module.exports = function(config) {  
  config.set({  
    basePath: 'test',  
    frameworks: ['jasmine'],  
    files: [ 'src/hw*.js',          'spec/*Spec.js'    ],  
    exclude: [    ],  
    preprocessors: {    },  
    reporters: ['progress'],  
    port: 9876,  
    colors: true,  
    config.LOG_DEBUG  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome', 'Firefox', 'IE'],  
    singleRun: true  
  });  
};
```

# Run Karma

- karma start sample.conf.js
  - Observe results
  - Do not close browsers



A **F**ast **AND** **S**teady Approach

# Services





# Services

- ❑ Angular services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.
- ❑ Angular services are:
  - Lazily instantiated – Angular only instantiates a service when an application component depends on it.
  - Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.
- ❑ Angular offers several useful services (like \$http), but for most applications you'll also want to create your own.

# Using a Service

- To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. Angular's dependency injection subsystem takes care of the rest.

# Creating and Using a Service

```
Angular.module('myServiceModule', [])  
.factory('notify', ['$window', function(win) {  
  var msgs = [];  
  return function(msg) {  
    msgs.push(msg);  
    if (msgs.length == 3) {  
      win.alert(msgs.join("\n"));      msgs = [];  
    }  };  
}]);
```

```
.controller('MyController', ['$scope', 'notify', function ($scope, notify) {  
  $scope.callNotify = function(msg) {  
    notify(msg); };  
}])
```

# Creating Services

- ❑ Application developers are free to define their own services by registering the service's name and service factory function, with angular module.
- ❑ The service factory function generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service

# Requirement

- Create a service for currency conversion.
  - Return current share rate by fetching data from server.
  - For performance improvisation, we need to do server trip for every third request or after every 10 seconds

# Registering a Service with \$provide

- You can also register services via the \$provide service inside of a module's config function:

```
angular.module('myModule', []).config(['$provide',  
  function($provide) {  
    $provide.factory('serviceId', function() {  
      var shinyNewServiceInstance;  
      // factory function body that constructs shinyNewServiceInstance  
      return shinyNewServiceInstance;  
    });  
  }]);
```

# Services

- Registering a Service with \$provide
  - You can also register services via the \$provide service inside of a module's config function:
- Managing Service Dependencies
  - Refer to Viral Patel Tutorial

# Requirement

- Modify call using \$provide
- Trying calling from two different controllers for confirmation of singleton and loading



# \$location

- The \$location service parses the URL in the browser address bar (based on the window.location) and makes the URL available to your application. Changes to the URL in the address bar are reflected into \$location service and changes to \$location are reflected into the browser address bar.

# Customizations

- Custom Validation
- Custom Directive

# Custom Validation

- ❑ Angular provides basic implementation for most common HTML5 input types: (text, number, url, email, date, radio, checkbox), as well as some directives for validation (required, pattern, minlength, maxlength, min, max).
- ❑ With a custom directive, adding your own validation functions to the \$validators object on the ngModelController is possible.

# Create Custom Validator

```
var app = angular.module('form-example1', []);
var INTEGER_REGEXP = /^-?\d+$/;
app.directive('integer', function()
{ return
{ require: 'ngModel', link: function(scope, elm, attrs, ctrl)
{ ctrl.$validators.integer = function(modelValue, viewValue)
{
    if (ctrl.$isEmpty(modelValue)) { return true; }
    if (INTEGER_REGEXP.test(viewValue)) { return true; }
    return false;
}; } }); });
```

# Use Custom Validator

```
<div> Size (integer 0 - 10):  
<input type="number" ng-model="size" name="size"  
    min="0" max="10" integer />  
{{size}}<br />  
<span ng-show="form.size.$error.integer">  
    The value is not a valid integer!</span>  
<span ng-show="form.size.$error.min ||  
    form.size.$error.max">  
    The value must be in range 0 to 10!</span>  
</div>
```

# Modifying built-in validators

```
var app = angular.module('form-example-modify-validators', []);
app.directive('overwriteEmail', function() {
  var EMAIL_REGEXP = /^[a-z0-9!#$%&'*/+=?^_`{|}~.-
    ]+@example\.com$/i;
  return {
    require: 'ngModel',
    restrict: "",
    link: function(scope, elm, attrs, ctrl) {
      if (ctrl && ctrl.$validators.email) {
        // this will overwrite the default Angular email validator
        ctrl.$validators.email = function(modelValue) {
          return ctrl.$isEmpty(modelValue) ||
            EMAIL_REGEXP.test(modelValue);
        };
      }
    }
  };
});
```

# Requirement

- Supportemail as email – valid email address should have .com at the end

# Custom Directive

```
module.directive('myCustomer', function()  
{  
  return {  
    template: 'Name: {{customer.name}}  
Address: {{customer.address}}'  
  };  
}); or  
templateUrl: 'my-customer.html'
```



A decorative vertical bar is positioned on the left side of the slide. It is composed of numerous horizontal segments of varying widths and colors, including shades of blue, teal, yellow, and black, creating a colorful, abstract pattern.

# Error handling in Angular

# Error Logging

- Uncaught exceptions in AngularJS are all funneled through the `$exceptionHandler` service.
- When unmodified, `$exceptionHandler` sends all uncaught exceptions to the `$log.error` service. The `$log.error` service passes the error through to the client's console.

# Basic Configuration

```
angular.module('exceptionOverride',  
[]).factory('$exceptionHandler',  
function() {  
    return function(exception, cause) {  
        exception.message += 'Angular  
Exception: "' + cause + '"';  
        throw exception;  
    };  
});
```

A decorative vertical bar on the left side of the slide, composed of numerous horizontal segments in various shades of blue, teal, yellow, and black, creating a colorful, abstract pattern.

`$watch()` , `$digest()` and `$apply()`

# Watch

- When you create a data binding from somewhere in your view to a variable on the \$scope object, AngularJS creates a "watch" internally. A watch means that AngularJS watches changes in the variable on the \$scope object. The framework is "watching" the variable.

# Digest

- At key points in your application AngularJS calls the `$scope.$digest()` function.
- Iterates through all watches and checks if any of the watched variables have changed. If a watched variable has changed, a corresponding listener function is called. The listener function does whatever work it needs to do, e.g. display change
- `$digest()` function is what triggers the data binding to update.

# Apply

- The `$scope.$apply()` function is used to execute some code, and then call `$scope.$digest()` after that, so all watches are checked and the corresponding watch listener functions are called. The `$apply()` function is useful when integrating AngularJS with other code.

# Deep Dive in two way binding

- How two-binding works
- Watch
  - Expressions
  - Object Collection
  - Group of Expressions
- When you write an expression (`{{info}}`), behind the scenes Angular sets up a watcher on the scope model, which in turn updates the view whenever the model changes.



# Create Watch

```
$scope.$watch('info', function(newValue, oldValue) {  
    //update the DOM with newValue  
});
```

```
$scope.names =  
['shailendra', 'deepak',  
'mohit', 'kapil'];  
$scope.dataCount = 4;  
  
$scope.$watchCollection('names', function (newVal,  
oldVal) {  
    $scope.dataCount =  
newVal.length;  
});
```

```
$scope.teamScore = 0;  
$scope.time = 0;  
$scope.$watchGroup(['teamScore', 'time'],  
function(newVal, oldVal) {  
    if(newVal[0] > 20){  
        $scope.matchStatus = 'win';  
    }  
    else if (newVal[1] > 60){  
        $scope.matchStatus = 'times up';  
    }  
});
```

# Digest

- It's the \$digest cycle where the watchers are fired. When a watcher is fired, AngularJS evaluates the scope model, and if it has changed then the corresponding listener function is called. So, our next question is when and how this \$digest cycle starts.
- The \$digest cycle starts as a result of a call to `$scope.$digest()`
- \$digest is automatically triggered

# \$scope.\$apply

- Angular doesn't directly call \$digest(). Instead, it calls \$scope.\$apply(), which in turn calls \$rootScope.\$digest(). As a result of this, a digest cycle starts at the \$rootScope, and subsequently visits all the child scopes calling the watchers along the way

# Digest & Apply

- `$scope.$apply()` automatically calls `$rootScope.$digest()`. The `$apply()` function comes in two flavors. The first one takes a function as an argument, evaluates it, and triggers a `$digest` cycle. The second version does not take any arguments and just starts a `$digest` cycle when called. We will see why the former one is the preferred approach shortly.

# Best Practices and Patterns

- JS module pattern

# QUESTION / ANSWERS



# THANKING YOU !

