

# DOM COMPILATION



# Objectives

2

- Deep dive into the \$compile service
- Look at additional related services
  - ▣ \$parse
  - ▣ \$interpolate
  - ▣ \$sce

# Classical Templating

3

- Consumes static string template and combines it with data
- The result is just a string
- Then the developer manually appends it to the DOM
- However,
  - ▣ Any change to the data requires repeating the process
  - ▣ Leads to lost of user input
  - ▣ No easy way to define reusable behaviors

# Angular Templating

4

- Works against a DOM tree, not a string
- The result is a “live” DOM
  - ▣ The DOM is automatically updated by Angular whenever the data model changes
  - ▣ The data model is automatically being modified when the DOM changes
- Has extensive API for integrating with the compilation and linking phases

# HTML Compiler

5

- Allows the developer to teach the browser new HTML syntax
  - ▣ Without server side or pre compilation
- New behavior can be attached to any HTML element or attribute
- New HTML elements can be created
- These extensions become a Domain Specific Language (DSL)
- Angular calls it Directive

# \$compile

6

- Build a template string and wrap it inside a jQuery element
- Compile the element using **\$compile**
  - ▣ May transform the DOM into something else
- Link the template to a Scope instance
  - ▣ Cannot use arbitrary object
  - ▣ Because directives use the Scope API
- Initiate a digest cycle
  - ▣ Watchers are executed
  - ▣ DOM is updated

# \$compile Usage

7

```
var template = angular.element("<span>{{name}}</span>");
console.log("Template: " + template[0].outerHTML);

var templateFn = $compile(template);
console.log("$compile: " + template[0].outerHTML);

var scope = $rootScope.$new();
templateFn(scope);
console.log("link: " + template[0].outerHTML);

scope.name = "App Data";
scope.$digest();
console.log("$digest: " + template[0].outerHTML);
```

```
<span class="ng-binding ng-scope">
  {{name}}
</span>
```

```
<span class="ng-binding ng-scope">
  {{name}}
</span>
```

```
<span class="ng-binding ng-scope">
  App Data
</span>
```

# Clone the Template

8

- In some cases you may want to compile once but link multiple times to different scope instances
- For example, ng-repeat

Original  
template is not  
effected

```
var template = angular.element("<span>{{name}}</span>");
console.log("Template: " + template[0].outerHTML);

var templateFn = $compile(template);
console.log("$compile: " + template[0].outerHTML);

for (var i = 0; i < 10; i++) {
  var scope = $rootScope.$new();
  scope.name = "My Data " + (i + 1);

  var view = templateFn(scope, function (clone) {});
  scope.$digest();

  console.log("view " + (i+1) + ": " + view[0].outerHTML);
}
```

This is a clone



# Compilation Debug Info

9

- Angular injects some debug info into the compiled DOM
  - ▣ **ng-scope**: When an element is attached to a new scope
  - ▣ **ng-binding**: When an element is bound through `{{}}` or `ng-bind`
- The debug info makes the DOM heavier and therefore slower
- The compilation phase is longer
- You may disable it using **`$compileProvider`**

# Disable Compilation Debug Info

10

```
angular.module("MyApp", [])  
  .config(function ($compileProvider) {  
    $compileProvider.debugInfoEnabled(false);  
  });
```

- See <http://jsperf.com/angular-debug-info-inpact> for performance comparison
- Does not show significant improvement
  - ▣ Only ~2% better
- However, the tested DOM is small

# Compile Phase

11

- #1: Traversing
  - ▣ Inspect the DOM element and collect all directives
  - ▣ A single element may match multiple directives
  - ▣ Sorts directives by their priority
- #2: Compiling
  - ▣ Execute each directive's compile function
  - ▣ The compile function may change the DOM and therefore order is important
  - ▣ Collect link functions

# Compilation – High Level View

12

```
function compileNodes(nodeList, ...) {  
  var linkFns = [], directives, nodeLinkFn, childLinkFn, linkFnFound;  
  
  for (var i = 0; i < nodeList.length; i++) {  
    directives = collectDirectives(nodeList[i], ...);  
  
    nodeLinkFn = (directives.length) ? applyDirectivesToNode(directives, ...) : null;  
  
    childLinkFn = !nodeList[i].childNodes ? null : compileNodes(childNodes, ...);  
  
    if (nodeLinkFn || childLinkFn) {  
      linkFns.push(i, nodeLinkFn, childLinkFn);  
      linkFnFound = true;  
    }  
  }  
  
  return linkFnFound ? compositeLinkFn : null;  
}
```

Collect all matching directives for current node. Directives are sorted by priority

Execute compile function for each directive. Keep resultant link function

Do the same for all child nodes

# Collecting Directives

13

- Each node is compared against a list of all registered directives
- Need to compare
  - ▣ Element name
  - ▣ Attribute list
  - ▣ Element class
  - ▣ Comment
  - ▣ Element text - For interpolation expression
  - ▣ Attribute's value - For interpolation expression

# Collecting Directives

14

addDirective adds  
directive only if  
name and type are  
valid

Directives are  
sorted by priority

```
function collectDirectives(node, directives, ...) {  
  switch (node.nodeType) {  
    case NODE_TYPE_ELEMENT:  
      addDirective(directives, 'E', ...);  
  
      foreach (attr in attrs) {  
        addAttrInterpolateDirective(node, directives, value, ...);  
        addDirective(directives, 'A', ...);  
      }  
  
      if (isString(node.className) && node.className !== '') {  
        addDirective(directives, 'C', ...)  
      }  
  
      break;  
  
    case NODE_TYPE_TEXT:  
      addTextInterpolateDirective(directives, node.nodeValue);  
      break;  
  
    case NODE_TYPE_COMMENT:  
      match = COMMENT_DIRECTIVE_REGEXP.exec(node.nodeValue);  
      addDirective(directives, 'M', ...);  
      break;  
  }  
  
  directives.sort(byPriority);  
  
  return directives;  
}
```

# addDirective

15

- ❑ Iterates all registered directives looking for a matching directive (by name)
- ❑ Instantiates the directive
- ❑ Verifies directive's restrict option

Directive is instantiated before checking the restrict option ☹

```
function addDirective(tDirectives, name, location, maxPriority) {  
  var match = null;  
  if (hasDirectives.hasOwnProperty(name)) {  
    for (var directive, directives = $injector.get(name + Suffix), i = 0, ii = directives.length; i < ii; i++) {  
      try {  
        directive = directives[i];  
        if ((maxPriority === undefined || maxPriority > directive.priority) &&  
            directive.restrict.indexOf(location) != -1) {  
          tDirectives.push(directive);  
          match = directive;  
        }  
      } catch (e) { $exceptionHandler(e); }  
    }  
  }  
  return match;  
}
```

Collection of all registered directives

There may be multiple directives with the same name

# Registering a Directive

16

```
function directive(name, directiveFactory) {
  if (!hasDirectives.hasOwnProperty(name)) {
    hasDirectives[name] = [];

    $provide.factory(name + 'Directive', ['$injector', '$exceptionHandler',
      function ($injector, $exceptionHandler) {
        var directives = [];
        forEach(hasDirectives[name], function (directiveFactory, index) {
          var directive = $injector.invoke(directiveFactory);
          if (isFunction(directive)) {
            directive = { compile: valueFn(directive) };
          } else if (!directive.compile && directive.link) {
            directive.compile = valueFn(directive.link);
          }
          directive.priority = directive.priority || 0;
          directive.index = index;
          directive.name = directive.name || name;
          directive.require = directive.require || (directive.controller && directive.name);
          directive.restrict = directive.restrict || 'EA';
          if (isObject(directive.scope)) {
            directive.$$isolateBindings = parseIsolateBindings(directive.scope, directive.name);
          }
          directives.push(directive);
        });
        return directives;
      }
    ]);
    hasDirectives[name].push(directiveFactory);
    return this;
  }
};
```

hasDirectives is an array of all registered directives

Multiple directives can have the same name

Register a factory which is responsible for instantiating all directives

Non standard factory since an array is returned

A directive is an object with a compile function which returns a link function



# Applying Directives

17

- Apply directive settings that may modify the DOM
- Execute the directive's compile function

```
function applyDirectivesToNode(directives, compileNode, ...) {  
    ...  
    for (var i = 0, ii = directives.length; i < ii; i++) {  
        directive = directives[i];  
  
        if (directiveValue = directive.scope) {...}  
  
        directiveName = directive.name;  
  
        if (!directive.templateUrl && directive.controller) {...}  
  
        if (directiveValue = directive.transclude) {...}  
  
        if (directive.template) {...}  
  
        if (directive.templateUrl) {...}  
  
        else if (directive.compile) {  
            linkFn = directive.compile($compileNode, templateAttrs, childTranscludeFn);  
        }  
  
        if (directive.terminal) {...}  
    }  
    return nodeLinkFn;  
}
```

# Compilation Order

18

- Each directive may change the DOM
  - ▣ One directive sees the modification of the others
  - ▣ Therefore order is important
- Default directive's priority is zero
  - ▣ Higher priority means “compile earlier”
- If priority is the same then sort by
  - ▣ name
  - ▣ index

```
function byPriority(a, b) {  
    var diff = b.priority - a.priority;  
    if (diff !== 0) return diff;  
    if (a.name !== b.name) return (a.name < b.name) ? -1 : 1;  
    return a.index - b.index;  
}
```

# Modify DOM During Compilation

19

- What if one directive completely removes itself from the DOM ?
- All other lower priority directives manipulate an already removed element
- What about the sibling directive ?
  - ▣ Angular does not compile it ☹
  - ▣ Is this a bug ?
- In general, compile should only “play” with the element content

# Removing an Element

20

- Assuming dir1 & dir2 are directives
- dir1 removes itself
- dir2 is not compiled

```
angular.module("MyApp", [])  
  .directive("dir1", function () {  
    return {  
      compile: function (element) {  
        element.remove();  
      }  
    };  
  })  
  .directive("dir2", function () {  
    return {  
      compile: function (element) {  
        element.append("<h1>Directive2</h1>");  
      }  
    };  
  });
```

```
<body>  
  <dir1></dir1><dir2></dir2>  
</body>
```

# Linking Phase

21

- Select the appropriate scope instance
  - ▣ May create a new one
- Invoke pre link on current directive
  - ▣ PRE DOM modification (not safe)
  - ▣ Register PRE watchers
- Recursively link child directives
- Invoke post link on current directive
  - ▣ Post DOM modification
  - ▣ Register POST watchers

# Linking Phase

22

```
function nodeLinkFn(childLinkFn, scope, ...) {
  if (newIsolateScopeDirective) {
    isolateScope = scope.$new(true);
  }

  // PRELINKING
  for (i = 0, ii = preLinkFns.length; i < ii; i++) {
    linkFn = preLinkFns[i];
    invokeLinkFn(linkFn, ...);
  }

  // RECURSIVE
  childLinkFn && childLinkFn(scopeToChild, linkNode.childNodes, ...);

  // POSTLINKING
  for (i = postLinkFns.length - 1; i >= 0; i--) {
    linkFn = postLinkFns[i];
    invokeLinkFn(linkFn, ...);
  }
}
```

# Linking Phase - Disclaimer

23

- The linking phase is more complex than introduced at previous slides
- Need to take into account
  - ▣ Directive's template
    - Might need to load it from server
    - Which means delayed linking
  - ▣ Directive's controller
- Will be discussed at next module

# Interpolation

24

- During DOM compilation Angular analyzes each node's text and each attribute's value
- Seeking for interpolation expression

```
function addTextInterpolateDirective(directives, text) {  
  var interpolateFn = $interpolate(text, true);  
  if (interpolateFn) {  
    directives.push({  
      priority: 0,  
      compile: function textInterpolateCompileFn(templateNode) {  
        return function textInterpolateLinkFn(scope, node) {  
          scope.$watch(interpolateFn, function interpolateFnWatchAction(value) {  
            node[0].nodeValue = value;  
          });  
        };  
      }  
    });  
  }  
}
```

Update DOM  
whenever the  
expression changes

Is null if text does  
not contain any  
interpolation



# \$interpolate

25

- Compiles a string into an interpolation function
- Later, can be linked to a scope and thus generating the final string

"Hello, Ori"

```
angular.module("MyApp", [])  
  .run(function ($interpolate) {  
    var template = "Hello, {{name}}";  
  
    var interpolateFn = $interpolate(template);  
  
    var str = interpolateFn({ name: "Ori" });  
    console.log(str);  
  });
```

# \$interpolateProvider

26

- ❑ Curly braces can be replaced
- ❑ You can switch to other opening and ending characters

```
angular.module("MyApp", [])  
  .config(function ($interpolateProvider) {  
    $interpolateProvider.startSymbol("<%");  
    $interpolateProvider.endSymbol("%>");  
  });
```

```
<div ng-controller="HomeCtrl">  
  <%message%>  
</div>
```

# \$interpolate – How does it work ?

27

## □ Small wrapper around **\$parse**

```
function $interpolate(text) {
  var expressions = [], parseFns = [];

  while (index < textLength) {
    if (((startIndex = text.indexOf(startSymbol, index)) !== -1) &&
        ((endIndex = text.indexOf(endSymbol, startIndex + startSymbolLength)) !== -1)) {
      exp = text.substring(startIndex + startSymbolLength, endIndex);
      expressions.push(exp);
      parseFns.push($parse(exp, parseStringifyInterceptor));
    }
  }

  return function interpolationFn(context) {
    var values = new Array(ii);
    for (var i=0, ii=expressions.length; i < ii; i++) {
      values[i] = parseFns[i](context);
    }
    return compute(values);
  }
}
```

# \$interpolate - XSS

28

- By default **\$interpolate** is willing to inject any string into the interpolation expression
- Even plain JavaScript

The JavaScript code  
is executed

```
var template = "{{name}}";  
  
var interpolateFn = $interpolate(template);  
  
var context = {  
  name: "<script>alert('XSS');</script>",  
};  
  
var str = interpolateFn(context);  
  
angular.element(".message").append(str);
```

# \$interpolate – XSS Protection

29

- You may ask **\$interpolate** to guard against XSS
- This is done by setting the value “html” for the 3<sup>rd</sup> parameter

```
var template = "{{name}}";  
  
var interpolateFn = $interpolate(template, undefined, "html");  
  
var context = {  
  name: "<script>alert('XSS');</script>",  
};  
  
var str = interpolateFn(context);  
  
angular.element(".message").append(str);
```

- **interpolateFn** throws an exception: “Attempting to use unsafe value in a safe context”

# \$interpolate – Too Strict

30

- Angular does not let you interpolate even a simple HTML like `<h1>`
- Use the relaxed module **angular-sanitize**

Need to download  
this module

```
angular.module("MyApp", ["ngSanitize"])  
  .run(function ($interpolate, $rootElement) {  
    var template = "{{name}}";  
    var interpolateFn = $interpolate(template, undefined, "html");  
    var context = {  
      name: "<h1>Hello</h1>"  
    };  
    var str = interpolateFn(context);  
    angular.element(".message").append(str);  
  });
```

Returns empty  
string if content is  
suspicious

# angular-sanitize – How does it work ?

31

- **angular-sanitize** registers a service named **\$sanitize**
- **\$interpolate** asks **\$sce** whether a value can be trusted
- **\$sce** looks for **\$sanitize** service
  - ▣ If not found throws “Attempting ...”
  - ▣ Else, delegates request to **\$sanitize**
- **\$sanitize** returns empty string for any suspicious content
  - ▣ Does not throw an error

# \$sce

32

- ❑ Strict Contextual Escaping
- ❑ Assist in writing code that is secured by default
- ❑ Usually, no need to interact with it directly
- ❑ Directives are using it to preserve secured code
  - ❑ `$sce.getTrusted` - Convert a value to a trusted one or throws an exception
  - ❑ `$sce.parseAs` – Like `$parse` but with security checking
- ❑ Can be disabled using `$sceProvider.enabled(false)`



# \$sce Usage

33

- Each expression parsed by `$parse` is intercepted by `$interpolate` which enforces SCE checking

```
function $interpolate(text, mustHaveExpression, trustedContext) {  
  ...  
  while (index < textLength) {  
    ...  
    parseFns.push($parse(exp, parseStringifyInterceptor));  
  }  
  
  var getValue = function (value) {  
    return trustedContext ?  
      $sce.getTrusted(trustedContext, value) :  
      $sce.valueOf(value);  
  };  
  
  function parseStringifyInterceptor(value) {  
    ...  
    return value = getValue(value);  
  }  
}
```

# \$parse

34

- ❑ Converts **Angular expression** into a function
- ❑ The returned function can be linked to a context + locals

Result is "Hello  
\$parse"

```
angular.module("MyApp", [])  
  .run(function ($parse) {  
    var expr = "ctrl.message";  
  
    var compiledExpr = $parse(expr);  
  
    var context = { ctrl: { message: "Hello $parse" } };  
    var str = compiledExpr(context);  
  
    console.log(str);  
  });
```

# Angular Expression

35

- Is not a JavaScript expression !!!
  - ▣ And therefore cannot be eval'ed
- Angular implements its own parser and lexer ...
- Differences
  - ▣ Context
  - ▣ Relaxed
  - ▣ No control flow, function, RegExp, comma
  - ▣ Filters

# \$parse implementation

36

```
function $parse(exp, interceptorFn, expensiveChecks) {  
  var parsedExpression, oneTime, cacheKey;  
  switch (typeof exp) {  
    case 'string':  
      var lexer = new Lexer(...);  
      var parser = new Parser(lexer, $filter, ...);  
      parsedExpression = parser.parse(exp);  
      return addInterceptor(parsedExpression, interceptorFn);  
  
    case 'function':  
      return addInterceptor(exp, interceptorFn);  
  
    default:  
      return addInterceptor(noop, interceptorFn);  
  }  
};
```

- Interceptor allows transformation of the evaluated value before it is written into the resultant string
  - ▣ Used internally by Angular (see \$interpolate use case)

# \$parse Metadata

37

- \$parse returns information about the type of the expression

- ▣ literal
- ▣ constant
- ▣ **assign**

```
var expr = "{ 'e-mail': email }";  
  
var compiledExpr = $parse(expr);  
  
console.log("assign: " + !!compiledExpr.assign);  
console.log("literal: " + !!compiledExpr.literal);  
console.log("constant: " + !!compiledExpr.constant);
```

- Angular offers some optimization when literal or constant are true

assign: False  
literal: True  
constant: False

# \$parse.assign

38

- **assign** references a function (or null) which contains the assignment logic
- When executed on an object → Object is modified
- Very useful when implementing directives

Same as executing:  
`context.ctrl.contact =  
 {name: "Ori"};`

```
var expr = "ctrl.contact";  
var getter = $parse(expr);  
var setter = getter.assign;  
  
var context = {};  
if (setter) {  
  setter(context, { name: "Ori" }); }  
  
console.log(context);
```

# Filters

39

- ❑ \$parse allows for embedding filters inside the expression
- ❑ Filter is a stateless function which transforms a value into a string
- ❑ Angular is smart enough to call the filter only if the value has changed (even for custom filter)

Angular assumes that date is stateless and therefore need to re-execute it only if 'when' changes

```
<div ng-controller="HomeCtrl">
  <div>{{when | date}}</div>
  <button ng-click="run()">$apply</button>
  <button ng-click="change()">Change</button>
</div>
```

# Stateful Filter

40

- A filter that holds an internal state which effects the filter's output
- Angular is unaware of the internal state and therefore invokes the filter every digest cycle
  - ▣ Disable previous discussed optimization
- By default Angular assumes that a filter is stateless and can be optimized
- Use **\$stateful** to overwrite default



# Stateful Filter

41

- Filter method must be short
  - ▣ Else, digest cycle performance is reduced

```
angular.module("MyApp", [])  
  .filter("x", function () {  
    var fn = function (value) {  
      console.log("x filter");  
      return value + "X";  
    }  
  
    fn.$stateful = true;  
    return fn;  
  });
```

- Filter **filter** is considered stateful and therefore will be invoked per each digest cycle

# Filters - Performance

42

- In general, try to avoid filters
- Angular is not able to optimize them well
  - ▣ Primitive values are optimized
  - ▣ Arrays are not
- Are quite limited from application perspective
- Prefer plain controller logic manipulation instead

# Summary

43

- Angular DOM compilation is encapsulated under a service named **\$compile**
- Companioned with its friends **\$parse** and **\$interpolate** the developer can implement DOM reusable logic