CODEMOTION WORKSHOP

DEVELOPING A COMPONENT-BASED APPLICATION WITH ANGULAR 1.5 AND ANGULAR 2.0

DEVELOP HIGH QUALITY APPLICATIONS, FASTER

CARLO BONAMICO / carlo.bonamico@gmail.com / @carlobonamico

MODULE 00

Introduction

WHILE YOU ARE WAITING...

- download the labs from
  - https://github.com/carlobonamico/angular-component-based

```
git clone https://github.com/carlobonamico/angular-component-based
```

or plain "Download Zip" from browser

DEVELOPING A COMPONENT-BASED APPLICATION WITH ANGULAR 1.5 AND ANGULAR 2.0

DEVELOP QUALITY APPLICATIONS, FASTER

In your first Angular project, the framework helped you quickly create an HTML5 app. But you now face new challenges as the UI complexity increases and more code moves to the front-end. The workshop shares Patterns and Best Practices on how to structure and implement complex, real-world Angular apps

- http://milan2016.codemotionworld.com/workshop/developing-a-component-based-application-with-angular-1-5-and-angular-2-0/

ABSTRACT

In your first Angular project, you have experienced first hand how Angular lets developers from any background quickly create HTML5 apps. However, as the UI complexity increases and more code moves to the front-end, you face new challenges such as how to manage huge Controllers, avoid application fragility and increase code reuse. In the workshop, Carlo shares his experience in developing several large scale Angular applications in the last two years, and proposes Patterns and Best Practices on how to structure and implement complex, real-world Angular apps with a Component-based approach.

AUDIENCE

AngularJS Developers who master the basics of the framework and would like to learn an effective approach to design and implement complex real-world Angular applications in a robust, modular and future-proof way

PREREQUISITES

- Practical experience in Javascript and AngularJS development (you should be able to write/compile/test/debug by yourself an AngularJS 1.x application including Data Binding, Controllers and Services).
- Working knowledge of AngularJS syntax, Controllers and Services is required as these topics will NOT be explained in the workshop.
- Knowledge of AngularJS Directive is useful, but not required.
- Basic knowledge of HTML5 and of the DOM.

HARDWARE AND SOFTWARE REQUIREMENTS

- Participants are required to bring their own laptop. The labs require an HTML5 Browser (Chrome or Firefox), text editor or IDE, supporting HTML5, CSS3, JavaScript.
- Modern Browser (Chrome - Firefox)

Text Editor (Sublime, Atom, Visual Studio Code,...)

- http://brackets.io/
- http://atom.io

and/or IDE (Eclipse, NetBeans, Intellij, Visual Studio,..)

- http://www.eclipse.org
- http://netbeans.org
- WebStorms / Visual Studio Community
- `python -m SimpleHTTPServer` vs https://code.google.com/p/mongoose/

TOPICS

- How does our code become unmanageable? A practical example
- Issues and challenges in developing complex / large HTML5 applications
- From huge controllers and "scope soup" to Component-based Uis
- How to identify application Components
- How to develop a simple Component in Angular 1.5
- Adding inputs to the Component through bindings
- Returning outputs through events and callbacks
- Lifecycle callbacks

TOPICS

- How to interconnect multiple collaborating Components to achieve complex UI interactions
- "Smart", "dumb" and "stateless" components
- When to use two-way DataBinding and when One-Way Data Flow
- refactoring
- From AngularJS 1.5 to Angular 2.0: syntax changes, but Component-based architecture remains
- How to incrementally upgrade an application from 1.5 to 2.0 with ng-upgrade
- Side

APPROACH

For each module, hands-on lab will include

- quizzes (which of these variants is better? trade-offs)
- interactive examples to complete and modify in an online IDE

KEY REFERENCES

- All Labs and links available at
  - https://github.com/carlobonamico/angular-component-based
- Clean Code: the book
  - https://books.google.it/books/about/Clean_Code.html?id=hjEFCAAAQBAJ

(images/CleanCode.png)

REVISING CORE JAVASCRIPT CONCEPTS

- Yakov Fain - Advanced Introduction to Javascript
  - https://www.youtube.com/watch?v=X1J0oMayvC0
  - http://enterprisewebbook.com/appendix_a_advancedjs.html
    - https://github.com/Farata/EnterpriseWebBook
    - https://github.com/Farata/EnterpriseWebBook_sources

MODULE 01

Component Based Applicataions: Angular 1.5

THE CHALLENGE

TOPICS

- How does our code become unmanageable? A practical example
- Issues and challenges in developing complex / large HTML5 applications
- Huge controllers and "scope soup"

FEATURE PRESSURE

WORKING PROTOTYPE != PRODUCTION-READY

WHAT OFTEN HAPPENS

- huge files
- deep interconnections between features
- cross-cutting mechanisms "spread" everywhere
- fragility
- risk of change increases
- productivity decreases over time

WHAT CAN WE DO ABOUT IT?

TO LEARN MORE

THINKING IN COMPONENTS

TOPICS

- From huge controllers and "scope soup" to Component-based Uis
- How to identify application Components

THINKING IN COMPONENTS

- Learn to split a single "View" or "Page" from the user perspective into a hierarchy of Components From huge controllers and "scope soup" to Component-based Uis
- How to identify application Components

WHAT IS A UI COMPONENT?

Very rough definition

- a part of an application / website which includes
  - UI elements
  - interaction logic
  - and (possibly) Business logic

More ideas?

LET'S TRY

- Let's focus on UI Components
- Analyze the http://www.trenitalia.com website

TIP: use a screen capture and annotation tool such as https://qsnapnet.com/

TYPES OF COMPONENTS

- UI Components
  - individual input / output widgets
  - more complex widgets
  - user-level features
  - entire "pages"
- Non-Graphical Components

LAB

- Identify key components in a typical WebMail application
- Analyze which components can be reused in multiple views
- Identify key inputs and outputs for each component
- Now go find even more components
- https://drive.google.com/drive/u/0/folders/0B-Bogp8tUho_bDh6SkFOMXEwa1E

ADVANTAGES IN SHORT

- More reuse
- more Encapsulation
- easier collaboration in the team

REFERENCES

COMPONENT-BASED DEVELOPMENT WITH ANGULAR

TOPICS

- Angular 1.5 Component model and API
- How to develop a simple Component in Angular 1.5

THE ISSUES

- up to Angular 1.4.x, developing Component-Based Applications was possible, but
  - NOT easy
  - required additional effort
  - "handcraft" a `directive` following a number of criteria

ENTER ANGULAR 1.5

- Components as (almost) first class citizens
- embed consolidated Best Practices into the framework
  - controllerAs
- big syntax simplification
  - improved readability
  - less effort
- goal: make creating components so easy that you want to do it everywhere
- Truly first class support in Angular2

ANGULAR 1.5 COMPONENTS API

- declaring components `angular.component()`
- defining the component interface with `bindings`
- manage the component lifecycle with `$onInit` `$onChange` and `$onDestroy`
- linking components with each other

as always, embracing HTML

ASIDE - COMPONENTS VS HTML ELEMENTS

PLNKR or demo

How can it possibly work?

SO, IN HTML

- custom nodes are
  - managed within the DOM
  - styled with CSS
  - processed with JS

Angular builds on that and tries to integrate its component model with HTML as much as possible

OUR FIRST COMPONENT

A minimal <hello></hello> component

```
angular.module("helloApp").component("hello",
  {
    template: "<h3>Hello World</h3>
  });
```

in the page

```
<body>
  <hello></hello>
</body>
```

LAB 01

Create the `<mail-logo>` component

Preliminary steps:

- create a `mailLogo` folder under `components`
- create a `mailLogo.html` within `mailLogo`
- create a `mailLogo.component.js` within `mailLogo`
- add a `<script>` reference to `mailLogo.component.js` in `index.html`

Steps:

- complete the component definition

Remember: TEST the page at each step

THE IMPORTANCE OF NAMING CONVENTIONS

- You already know about this
- even more important in Javascript where you have less support from the Type-system and language

WHAT'S IN A COMPONENT?

- a name, to reference it in HTML (with CamelCase to kebab-case convention)
- some HTML
  - inline, with `template`
  - in an external file, with `templateUrl`
  - dynamic (with a `function()`)
- an optional controller
  - aliased as `$ctrl` by default

BEYOND HELLO WORLD

This is already useful to reduce duplication in our pages, but to be useful, the component must be able to interact with the user and with the rest of the page

THE MAIN PAGE CONTROLLER

Role of the `MailController`

- interact with backend services
- provide data to the individual components
- coordinate page elements

A look at the code...

TIP

Separate Layout from components, to increase reuse

THE MAIL-MESSAGE-LIST COMPONENT

Manages

- display
- navigation within the list **current message** Next message action ** Previous message action

ADDING A CONTROLLER

```
angular.module("mailApp").component("messageList",
  {
    templateUrl: "components/messageList/messageList.html",
    controller: MessageListController //or inline function, if simple
    controllerAs: "messageListCtrl" //default is $ctrl
  });
```

WHERE TO PUT THE CONTROLLER

- In the same file, if simple
- in a separate `messageList.controller.js` file if more complex
- or agree on a standard convention for your team

MANAGING COMPONENT INPUTS

TOPICS

- Adding inputs to the Component through bindings

INPUT BINDINGS

If we want to reuse the component, for instance

- for the Inbox views
- for a single folder view
- for the search results

We need to separate

- where do we get the list of messages
- where this list is stored
- from how it is displayed and navigated

IN THE INDEX.HTML

```
<div ng-controller="MailController as mailCtrl">

  <section class="main-pane">
    <message-list messages="mailCtrl.messages"></message-list>
  </section>
</div>
```

IN THE COMPONENT DEFINITION

```
angular.module("mailApp").component("messageList",
  {
    ...
    bindings: {
      messages: "<messages" //or just "<" if the name is the same
    }
  });
```

This is automatically available as a `messages` field in the controller

```
if (this.messages.length >0)
  //doSomething
```

IN THE COMPONENT HTML

```
<div ng-repeat="message in messageListController.messages" >

</div>
```

LAB 02

Define the `<message-viewer>` component

Preliminary steps:

- create a `message-viewer` folder under `components`
- create a `message-viewer.html` within `message-viewer`
- create a `message-viewer.component.js` within `message-viewer`
- add a `<script>` reference to `message-viewer.component.js` in `index.html`

Steps:

- move the mail message html into `message-viewer.html`
- complete the component definition in `message-viewer.component.js`, passing in the `message` parameter
- link the two components in `message-list.html`

Remember: TEST the page at each step - **F12 is your friend**

ASIDE - SIMPLER PARAMETERS

With the @ binding

- Passed to the component on initialization
- can be computed dynamically, but are not watched by default

Typical examples:

- size
- themes or css styles

MANAGING COMPONENT OUTPUTS

TOPICS

- Returning outputs through events and callbacks

A COMPONENT CANNOT DO EVERYTHING BY HIMSELF

To implement complex logics, a component needs to interact with

- child components, such as...
- parent components, such as...
- sibling components, such as

SEPARATING RESPONSIBILITIES

- the `<message-list>` component is responsible for
  - displaying the list
  - navigating in the list
  - showing which element is selected

But what to do when the User selects a message can change in different Use Cases

So let's keep this OUT of the `message-list` component

MANAGING AN ACTION WITH BOTH INTERNAL AND EXTERNAL CONSEQUENCES

When a user selects a message, two different thing must take place:

- within the component, the current message must be outlined
- outside the component, other components must be notified of the selection and perform actions
  - enable buttons
  - update other views

IN THE COMPONENT

```
<div ng-click="messageListCtrl.select(message)"> {{message.subject}}} </div>
```

```
this.select = function (selectedMessage){
   this.currentMessage = selectedMessage;
}
```

OUTSIDE THE COMPONENT

We would like to be notified

```
<message-list
  messages="mailCtrl.messages"
  on-select="mailCtrl.messageSelected(message)"
>
</message-list>
```

WE NEED THREE STEPS TO DO THIS

1) declare the event in the bindings

```
angular.module("mailApp").component("messageList",
  {
    ...
    bindings: {
      onSelected: "&"
    }
  });
```

This injects an `onSelected` event callback in the controller instance

2) call the callback when the message is selected within the component

```
  this.select = function (selectedMessage){
     this.currentMessage = selectedMessage;
     this.onSelected(selectedMessage);
  }
```

THIS WILL NOT WORK, UNLESS YOU REMEMBER STEP 3

3) explicitely declare the event object

```
this.select = function (selectedMessage){
   this.currentMessage = selectedMessage;
   this.onSelected({
     message: selectedMessage
   });
}
```

LAB 03

Implement the `<folder-list>` component

- receive the list of folders from the main MailController
- display it
- outline the current folder
- allow for selecting a folder
- notify the MailController, so that it can load the list of messages for that folder

LAB STEPS

Define the `<folder-list>` component

Preliminary steps:

- create a `folder-list` folder under `components`
- create a `folder-list.html` within `folder-list`
- create a `folder-list.component.js` within `folder-list`
- add a `<script>` reference to `folder-list.component.js` in `index.html`

Steps:

- move the UI in `folder-list.html`
- complete the component definition in `folder-list.component.js`
  - passing in the `folders` parameter
  - passing the `on-selected` callback
- link the components in `index.html`

Remember: TEST the page at each step - **F12 is your friend**

LAB EXTRA

Pass an additional `allow-create="true"` parameter

REUSE

Advantages:

- we can create multiple instances of the components linked to different data

READABILITY

When we look at the parent html (index.html or parent component)

- we clearly see the main UI structure
- we get an overview, not low-level details
- we clearly see how components are linked and interact

ENCAPSULATION

Changing the Controller or the template of a component has a much reduced risk of introducing regressions elsewhere

The robustness of the application increases if the components are smaller

See also the Clean Code principles on SRP and Class design

LIFECYCLE CALLBACKS

TOPICS

SIMPLIFY THE LIFECYCLE OF A COMPONENT

- Reduce boilerplate code
- perform actions only when it is best or needed

$ONINIT

$ONCHANGES

Example: display the count of unread messages

$ONDESTROY

Called when the scope of the component is

$POSTLINK

LAB 04

Develop the message-list component

Implement the $OnChanges callback

TO LEARN MORE

TOPICS

- How to interconnect multiple collaborating Components to achieve complex UI interactions

SEPARATION OF RESPONSIBILITIES

Component Design Principles

- minimize Coupling
- maximize Cohesion
- every component does one thing Well

COMPOSITION

If we apply this pattern at the application level,

Components form a hierarchy

We achieve complex behaviours by collaboration of many simpler components

EXAMPLES

COMPONENT-BASED UI ARCHITECTURE

- "Smart", "dumb" and "stateless" components

TWO WAY DATABINDING VS ONE-WAY DATAFLOW

- When to use two-way DataBinding and when One-Way Data Flow

- events vs outputs vs services

LAB 05

Integrate the mail-composer component

LAB 06

Integrate the mail-composer component with the reply button in message viewer

BONUS: CLEAN COMPONENTS

CONCEPT 1 - NAMING

-reading code vs writing code

- what is a good name?
- same but different: the importance of conventions

CONCEPT 3 - WHAT'S IN A GOOD FUNCTION?

- single responsibility
- separing inputs from outputs
- if you have to do 3 things, make 4 functions
- primitives and orchestrators

CONCEPT 4 - WHAT'S IN A GOOD CLASS? DESIGN PRINCIPLES

- Single Responsibility Principle
- collaborating with other classes
- composition vs inheritance (and the Open/Closed principle)
- Dependency Injection
- interfaces and the importante of Contracts

CLEAN CODE

- It cannot solve all development problems...
- But it can make them way more tractable!

DESIGN PRINCIPLES

Once we have got the basics covered, then we will need to understand the Software Dynamics

- vs the nature (and Laws) of Software

Take them into account => Design Principles

Basically, Common Sense applied to software design

Treat your code like your kitchen C.B., about 2013

IMPROVE OUR CODE

It takes a Deliberate approach and constant effort

To complicate is easy, to simplify is hard To complicate, just add, everyone is able to complicate Few are able to simplify Bruno Munari

READING CODE VS WRITING CODE

What is written without effort is in general
read without pleasure.

Samuel Johnson

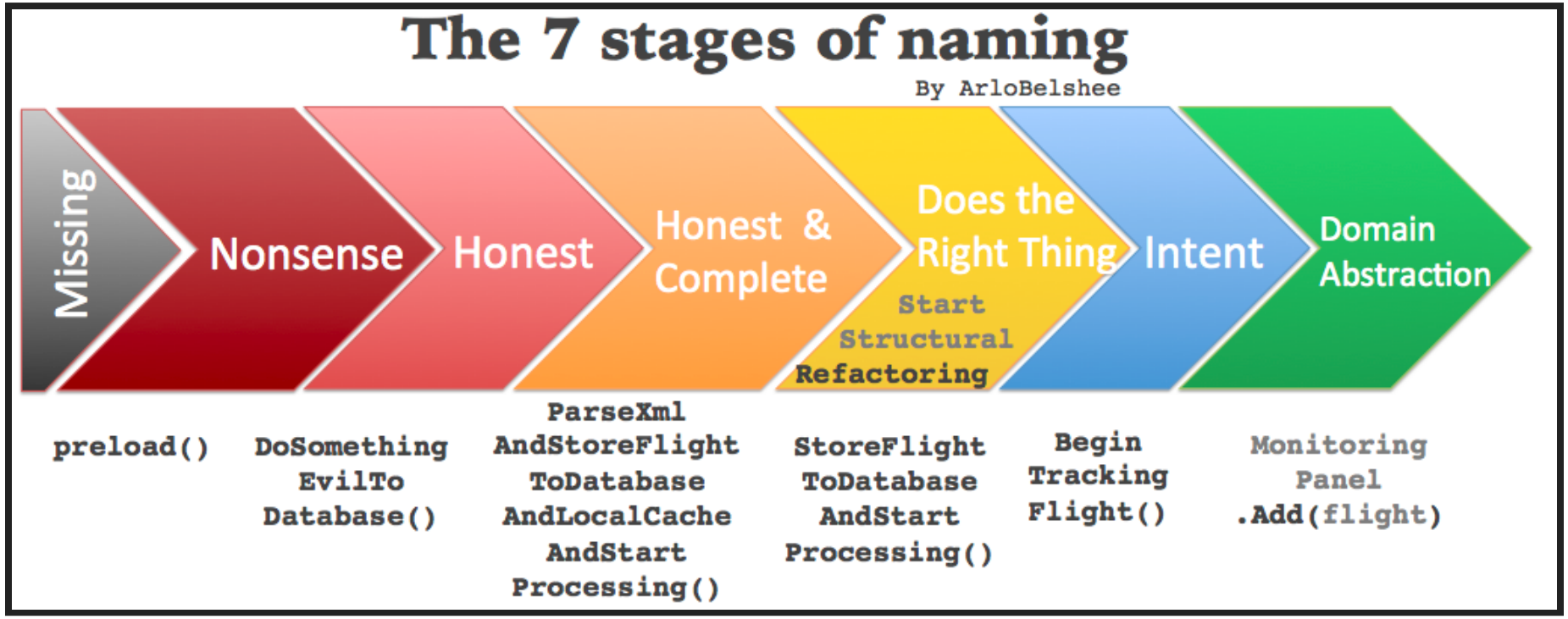
Most code is written once, but read

- every time you need to fix a bug
- to add new features
- by other developers
  - including your future self

WHAT IS A GOOD NAME?

- Ideas?

WHAT IS A GOOD NAME



- nonsense
- honest
- honest & complete
- does the right thing
- intent
- domain abstraction

http://lowellvnfalso.blogspot.it/p/infographics.html

http://llewellynfalco.blogspot.it/p/infographics.html

SINGLE RESPONSIBILITY

Each function should do 1 thing

Or even better, have a single responsibility

- and reason to change

HOW TO FIND RESPONSIBILITIES?

Ask yourself questions...

- What?
- Who?
- When?
- Why?
- Where?

And put the answer in different sub-functions

INPUTS VS OUTPUTS

- make inputs clear
- limit / avoid output parameters

3 THINGS, 4 FUNCTIONS

PRIMITIVES, ORCHESTRATORS, LEVEL OF ABSTRACTION

- Primitives: small, focused, typically use-case independent
- Orchestrators: implement use-cases by combining primitives
- rinse and repeat over multiple levels of abstraction
- benefits:
  - more reusable
  - easier to test

SINGLE RESPONSIBILITY PRINCIPLE

Have you ever seen your grandmother put dirty clothes in the fridge?

Or biscuits in the vegetable box?

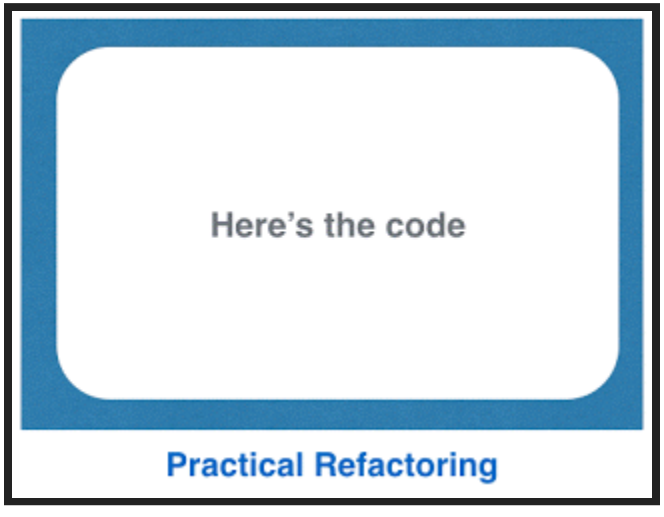
So, why to we do this all the time in our code?

SINGLE RESPONSIBILITY PRINCIPLE

Responsibility == reason to change

FROM BAD TO GOOD

Incremental transformation



IN STEPS

- Each step should not change the functional properties of the system
- and improve the non-functional ones
- separate adding features from refactoring
  - don't do both in the same step

THE BOY SCOUT RULE

Leave the campsite a little better than you found it

Every time you touch some code, leave it a little better

The power of compounding many small changes *in the same direction*

- 1% time

MORE PRACTICE AND KATAS

- http://codekata.com/
- https://www.industriallogic.com/blog/modern-agile/

IMPROVEMENT CULTURE

- https://codeascraft.com/2012/05/22/blameless-postmortems/

LEARNING TO LEARN

- Kathy Sierra
- https://www.youtube.com/watch?v=FKTxC9pl-WM

MODULE 02

Moving to Angular 2

PART 2 - COMPONENTS WITH ANGULAR 2

ENTER ANGULAR 2

TOPICS

- From AngularJS 1.5 to Angular 2.0: syntax changes,
- but Component-based architecture remains

ANGULAR2 KEY CONCEPTS

Re-implementation of the framework

- build on best practices
- target a wider range of platforms
  - web
  - desktop
  - mobile
- performance
- improve tooling

ANGULAR 2

Strongly Component-based

- no more controllers, scopes
- hierarchical Dependency Injection
- configurable change detection
  - dirty checking (zone.js)
  - Observables
  - immutables - based
  - custom
- generalized asynchronous handling (RxJs)
  - more general than Promises

FROM ANGULAR 1.5 TO ANGULAR 2 - SYNTAX

In an html template

Model-to-view binding

```
<div [hidden]="results.length >0">No results</div>
```

View-to-model binding with events

```
<button (click)="ctrl.send()">
```

Two-way binding

```
<input [(ngModel)]="ctrl.userName">
```

WHAT THE...

Initial surprise, but you get used to it.

- very clear if input or output binding
- automatically works with all DOM events and properties
  - without requiring ad-hoc directives such as `ng-show`
  - also works with Web Components, css classes

EXAMPLE WITH A CUSTOM COMPONENT

```
    <message-list [list]="messages" (selected)="select(message)">
```

FROM ANGULAR 1.5 TO ANGULAR 2 - CONSTRUCTS

- {{expression}}
- filters -> pipes
- ng-controller -> @Component classes
- angular.component -> @Component classes
- attribute directives -> same!
- ng-repeat -> *ngFor
- ng-if -> *ngIf

Modules

- angular.module -> @NgModule classes

https://angular.io/docs/ts/latest/cookbook/a1-a2-quick-reference.html

https://angular.io/resources/live-examples/cb-a1-a2-quick-reference/ts/plnkr.html

FROM ANGULAR 1.5 TO ANGULAR 2 - COMPONENTS

The key concepts and approach stay the same

- minor syntax changes
- component configuration in Metadata
  - as @Component annotations in Typescript
  - as fluent DSL in ES5 - ES6

https://angular.io/docs/ts/latest/guide/cheatsheet.html

ANGULAR2 HELLO WORLD COMPONENT

ASIDE - HOW TO BOOTSTRAP AN ANGULAR2 APPLICATION

From ng-app to AppModule

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

COMMON DIRECTIVES

From

```
<div ng-class="{active: isActive}">
<div ng-class="{active: isActive,
                  shazam: isImportant}">
```

To

```
<div [ngClass]="{active: isActive}">
<div [ngClass]="{active: isActive,
                 shazam: isImportant}">
<div [class.active]="isActive">
```

FOR NOW

we focus on Component APIs

Additional concepts (module bundling, etc) are needed before going into production

PASSING INPUTS TO COMPONENTS

```
import {Component} from '@angular/core';
import {Input, Output, EventEmitter} from '@angular/core';

@Component({selector: 'message-viewer',
    templateUrl : "components/message-viewer/message-viewer.html",
    inputs : ["message"],
    outputs: ["onReply","onForward","onDelete"],
    directives: ["common-star"]
})
export class MessageViewerComponent  {
    message;
```

PROPAGATING OUTPUT FROM COMPONENTS

```
@Component({
  moduleId: module.id,
  selector: 'app-confirm',
  templateUrl: 'confirm.component.html'
})
export class ConfirmComponent {
  @Input() okMsg = '';
  @Input('cancelMsg') notOkMsg = '';
  @Output() ok = new EventEmitter();
  @Output('cancel') notOk = new EventEmitter();
  onOkClick() {
    this.ok.emit(true);
  }
  onNotOkClick() {
    this.notOk.emit(true);
  }
}
```

WITHOUT TYPESCRIPT

```
ConfirmComponent.annotations = [
  new ng.core.Component({
    selector: 'app-confirm',
    templateUrl: 'app/confirm.component.html',
    inputs: [
      'okMsg',
      'notOkMsg: cancelMsg'
    ],
    outputs: [
      'ok',
      'notOk: cancel'
    ]
  })
];
function ConfirmComponent() {
  this.ok    = new ng.core.EventEmitter();
  this.notOk = new ng.core.EventEmitter();
}
ConfirmComponent.prototype.onOkClick = function() {
  this.ok.emit(true);
}
```

WHAT IF I DO NOT WANT TYPESCRIPT?

https://angular.io/docs/ts/latest/cookbook/ts-to-js.html

```
HeroComponent.annotations = [
  new ng.core.Component({
    selector: 'hero-view',
    template: '<h1>{{title}}: {{getName()}}</h1>'
  })
];
```

OR WITH DSL

```
app.HeroDslComponent = ng.core.Component({
    selector: 'hero-view-dsl',
    template: '<h1>{{title}}: {{getName()}}</h1>',
  })
  .Class({
    constructor: function HeroDslComponent() {
      this.title = "Hero Detail";
    },
    getName: function() { return 'Windstorm'; }
  });
```

ROUTED COMPONENTS

HIERARCHICAL DI

NGUPGRADE

- How to incrementally upgrade an application from 1.5 to 2.0 with ng-upgrade

PERFORMANCE

- Side

LAB - MANUAL MIGRATION

A final lab will demonstrate porting the application to Angular 2.0.

ANGULAR 2 - TO PROBE FURTHER

Dependency Injection (DI) Services Http Routing RxJs and Observables

MODULE

References

TO LEARN MORE

- Online tutorials and video trainings:
  - https://cleancoders.com
- Full lab from my Codemotion Workshop
  - https://github.com/carlobonamico/clean-code-design-principles-in-action

HOW TO CONTINUE BY YOURSELF: REFERENCES FOR FURTHER LEARNING

- Principles of Package Design
  - http://www.objectmentor.com/resources/articles/Principles_and_
- More on TDD
  - http://matteo.vaccari.name/blog/tdd-resources
- Modern Agile
  - https://www.industriallogic.com/blog/modern-agile/
- Lean, Quality vs Productivity and DevOps
  - http://itrevolution.com/books/phoenix-project-devops-book/

JAVASCRIPT

- http://humanjavascript.com/
- http://javascript.crockford.com/
- http://yuiblog.com/crockford/
- Free javascript books
- http://jsbooks.revolunet.com/

THANK YOU

- Other trainings
  - https://github.com/carlobonamico/
- My presentations
  - http://slideshare.net/carlo.bonamico
- Follow me at @carlobonamico / @nis_srl
- Contact me carlo.bonamico@gmail.com / carlo.bonamico@gmail.com

#

THANK YOU FOR YOUR ATTENTION

CARLO BONAMICO

@CARLOBONAMICO / @NIS_SRL

CARLO.BONAMICO@GMAIL.COM

CARLO.BONAMICO@GMAIL.COM

HTTP://MILANO.CODEMOTIONWORLD.COM

HTTP://TRAINING.CODEMOTION.IT/