

DEVELOPING A COMPONENT-BASED APPLICATION WITH ANGULAR

DEVELOP HIGH QUALITY APPLICATIONS, FASTER

CARLO BONAMICO / carlo.bonamico@nispro.it / [@carbonamico](https://twitter.com/carbonamico) MATTEO PARODI

MODULE 00

Introduction

WHILE YOU ARE WAITING...

- download the labs from
 - <https://github.com/carlobonamico/angular-component-based>
 - angular2 branch

```
git clone -b angular2 https://github.com/carlobonamico/angular-component-based
```

or plain "Download Zip" from browser

DEVELOPING A COMPONENT-BASED APPLICATION WITH ANGULAR

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

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,...)

VISUAL STUDIO CODE

- Community has increased support for developing Angular applications.
- lots of useful extensions

VISUAL STUDIO CODE EXTENSIONS

<https://medium.com/frontend-coach/7-must-have-visual-studio-code-extensions-for-angular-af9c476147fd>

TOPICS

- How does our code become unmanageable? A practical example
- Issues and challenges in developing complex / large HTML5 applications
- How to identify application Components
- How to develop a simple Component in Angular
- 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
- Performance tips

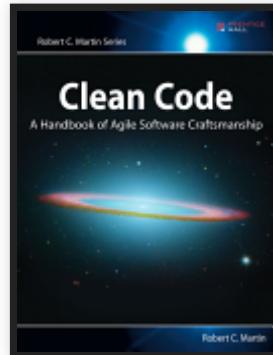
APPROACH

For each module, hands-on lab will include

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

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



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

Introduction to Typescript

PART 1 - INTRODUCTION TYPESCRIPT

TOPICS

- Syntax
- Pros of using
TypeScript

TYPESCRIPT

- Superset of Javascript
- very powerful
- Compile output is Javascript
- Java/C++/C# developers are more comfortable than with Javascript
- Valid Javascript code is valid Typescript code
- Easier to implement
 - modularity
 - robustness
 - readability
 - maintainability

TYPESCRIPT

- Errors at compile time
- Developer tools
 - Visual studio code and plugins
 - ... but you can use the IDE you want

CLASSES

```
class Persona {  
    constructor(private nome: string, private cognome: string) {  
  
    }  
  
    visualizzaNomeCognome() {  
        return this.nome + ' ' + this.cognome;  
    }  
}
```

INHERITANCE

```
enum Materie {Storia, Informatica, Matematica, Scienze};

class Studente extends Persona {
    materie: Materie[];

    constructor(nome, cognome) {
        super(nome, cognome);
    }
}
```

GENERICs

```
function identity<T>(arg: T): T { return arg; }

let output = identity<string>("myString");
```

FUNCTIONS

```
function buildName(firstName = "Jim", lastName?: string) {  
    if (lastName)  
        return firstName + " " + lastName;  
    else return firstName;  
}  
  
buildName();  
buildName("Jack");  
buildName("Jack", "Jones");
```

FAT ARROW FUNCTIONS

```
let inc = (x)=>x+1;  
  
onClickGood = (e: Event) => { this.info = e.message }
```

FAT ARROW FUNCTIONS: SCOPE

```
...
let a = this;
myFunction(function(value) {
  let b = this;
}) ;
...
```

Is a and b the same object?

```
...
let a = this;
myFunction((value)=>{
  let b = this;
}) ;
...
```

And now?

EXAMPLE: COMPILE ERRORS

```
var utente = {nome: "Mario", cognome: "Rossi", nome: "Carlo"};
var ora = Date().getTime();

document.onload = init();
function init() {
    var elemento = document.getElementById("myDiv");      elemento.innerHTML = "Test!";
}
```

try it on <http://www.typescriptlang.org/play/>

EXAMPLE: TYPE ERRORS

Without type checking

```
var x = "test";
function square(n) {
    return n*n;
}
var y = square(x);
```

With type checking

```
var x = "test";
function square(n:number) {
    return n*n;
}
var y = square(x);
```

try it on <http://www.typescriptlang.org/play/>

MODULE 01

Introduction to Angular

PART 1 - INTRODUCTION TO ANGULAR

TOPICS

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

ANGULAR KEY CONCEPTS

Re-implementation of the framework

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

ANGULAR

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 - 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
- also very clear if constant (custom-size=10) or bound property ([custom-size]=x * 10)
- 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

The same syntax can be used to bind either DOM properties of standard HTML5 components, or custom properties of Angular custom components

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

Full syntax <https://angular.io/docs/ts/latest/guide/template-syntax.html>

ANGULAR HELLO WORLD COMPONENT

See `labs/20-angular2/base-helloworld`

ASIDE - HOW TO BOOTSTRAP AN ANGULAR APPLICATION

From ng-app to AppModule

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

platformBrowserDynamic().bootstrapModule(AppModule);

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

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

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

LAB

Modify the Hello World example so that

- the Component class initialize a Date variable to the current Timestamp
- the html template displays it

COMMON UTILITY DIRECTIVES: NGCLASS

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">
```

FROM ANGULAR 1.5 TO ANGULAR - CONSTRUCTS

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

*NGFOR

```
<ul>
  <li *ngFor="let item of items">
    {{item.name}}
  </li>
</ul>
```

*NGIF

```
<button (click)="show = !show">{{show ? 'hide' : 'show'}}</button>
show = {{show}}
<br>
<div *ngIf="show">Text to show</div>
```

```
@Component({
  selector: 'ng-if-simple',
  template: '/components/ng-if-simple/ng-if-simple.html'
})
class NgIfSimple {
  show: boolean = true;
}
```

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 - 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>

NOW

we focus on Component APIs

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

Additional help from the official Cheat Sheet during the labs

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

NEXT STEPS

- Dependency Injection (DI)
- Services
- Http
- Routing
- RxJs and Observables

MODULE 01

Component Based Applications: Angular

PART 1 - INTRODUCTION TO ANGULAR

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

Real world apps are not easy

Bringing a complex application to production requires more than quickly bind form fields:

- decoupling unrelated parts
- preventing fragility in the face of changes
- keeping collaboration effective as team grows
- avoiding performance issues



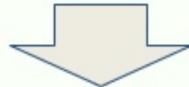
Angular gives us good tools



Avoiding bad practices

Often, as the application starts to grow, we see

- single large controller and HTML file per view
 - thousands of lines each
- significant repetition across views
 - same HTML fragments in many files
- “Spaghetti Binding”
 - creates dependencies between unrelated parts

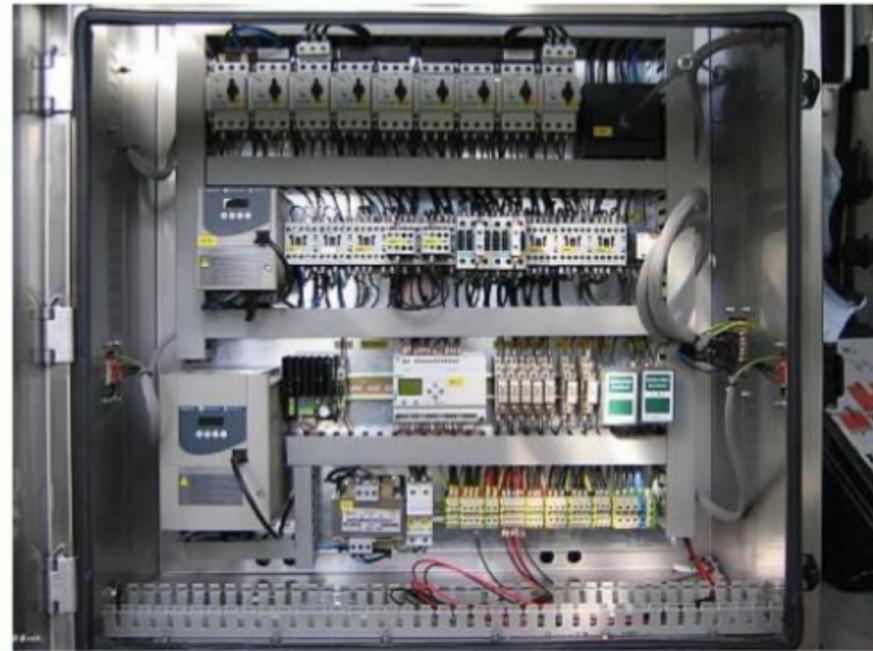
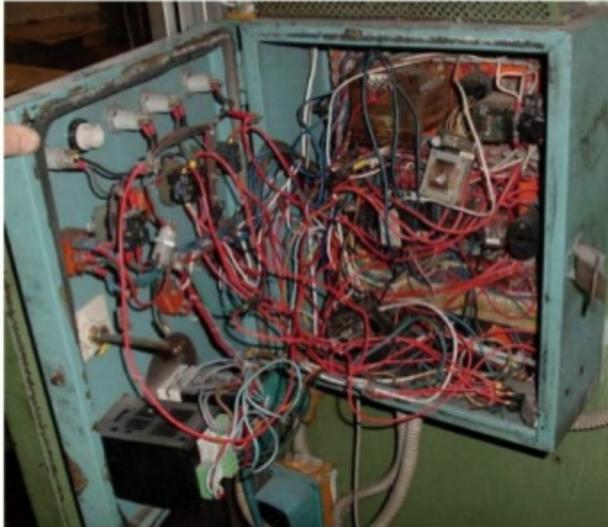


Code becomes **hard to navigate / risky to change**

We need **Software Engineering for the Frontend**, too



“Spaghetti Binding” vs Components



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 Component?

- Self-contained set of UI and logic
 - encapsulates a **specific behaviour**
 - provides an **explicit API**
- Since Angular 1.5, special kind of **Directive** with
 - UI in a **template**
 - Logic in a **controller**
 - some **metadata (inputs, outputs, ...)**
- Makes it **easier/cheaper to do a good design**
 - pushing the community towards **best practices**
- In Angular 2, the main application construct



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

Example App: NG Component Mail

The screenshot shows a mail application interface with the following components:

- Header:** "Angular Mail" on the left, search bar with placeholder "Search messages: js" and a "Search" button, and an email address "carlo.bonamico@gmail.com" on the right.
- Compose Button:** A red "Compose" button in the top-left corner of the main content area.
- Folder List:** On the left, under "Folder list", there are three items: "0 - Inbox" (highlighted with a red border), "1 - Trash", and "2 - Sent".
- Custom Folders:** Under "Custom folders", there are two items: "0 - Angular" and "1 - Typescript".
- Message List:** The main content area displays a list of 4 messages titled "Lista dei messaggi - 4 messages".
 - Message 0: From: sonia.pini@nispro.it Angular 1.5 (sent on 10/03/2016 12:00)
 - Message 1: From: carlo.bonamico@nispro.it Typescript (sent on 10/03/2016 12:00, highlighted with a yellow background)
 - Message 2: From: sonia.pini@nispro.it Flexbox how-to (sent on 10/03/2016 12:00)
 - Message 3: From: sonia.pini@nispro.it Re: ES6 tutorial (sent on 10/03/2016 12:00)
- Message Preview:** Below the message list, the selected message ("Typescript") is previewed.
 - From:** carlo.bonamico@nispro.it
 - To:** carlo.bonamico@gmail.com
 - Sent:** 10/03/2016 12:00
 - Action Buttons:** Reply, Forward, Delete
 - Text Area:** A large text area containing the message content, with character markers "a b c d e f" visible.

powered by Angular 1.5 and NIS s.r.l.

Full Source in <https://github.com/carlobonamico/angular-component-based>

Thinking in Components

The screenshot shows a user interface for an Angular Mail application. The interface is organized into several components:

- Search Panel:** Located at the top left, it includes a search bar with the placeholder "Search messages: js" and a "Search" button. This is labeled with the component name <search-panel>.
- User Information:** At the top right, it shows the email address "carlo.bonamico@gmail.com".
- Compose Button:** A red button labeled "Compose" is located in the top-left corner of the main content area.
- Folder List:** A sidebar on the left contains two sections: "Folder list" with items "0 - Inbox", "1 - Trash", and "2 - Sent"; and "Custom folders" with an item "0 - Angular" and a plus sign (+) button. This is labeled with the component name <folder-list>.
- Message List:** The main content area displays a list of messages titled "Lista dei messaggi - 4 messages". It includes navigation buttons "Prev <" and "Next >". The first message is highlighted with a yellow background and checked status, while others are greyed out. This is labeled with the component name <message-list>.
- Nav Actions:** Below the message list are buttons for "Reply", "Forward", and "Delete". This is labeled with the component name <nav-actions>.
- Message Viewer:** A detailed view of the selected message (From: carlo.bonamico@nispro.it, To: carlo.bonamico@gmail.com, Date: 10/03/2016 12:00) is shown. It includes a "Typescript" heading and a "Message Actions" section with buttons for "Reply", "Forward", and "Delete". This is labeled with the component name <message-viewer>.
- Message Actions:** Below the message viewer is a section labeled "<message-actions>" containing buttons for "Reply", "Forward", and "Delete".
- Page Footer:** At the bottom left, it says "Powered by Angular 1.5 and NIS s.r.l."

Two-way bindings pros and cons

- Very effective collaboration “in the small”
 - like when *all people in a room can talk to each other to solve a problem*
 - ng-model for bi-directional updates between form inputs and model
 - a validation expression can reference several model variables
- creates chaos “in the large”
 - think *all people in the building talking to any one else*



Advantages

Stronger Encapsulation (isolated scope + explicit binding)

- changing the component internal implementation has less impact on the rest of the application
- more decoupling, less regressions

Reusability (with parametrization)

- same component used in different contexts
 - <message-list> can display either folder messages and search results



Advantages

Better Collaboration

- less conflicts when team grows
- easier to test for regressions

More Clarity and Readability

- I can effectively use a component knowing only its API (the bindings section)
- the link to other component is very clear and explicit in the HTML



COMPONENT-BASED DEVELOPMENT WITH ANGULAR

TOPICS

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

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

- The only way is to do Components
- even the application is a component
- big syntax simplification
 - improved readability
 - less effort
- Typescript

ANGULAR COMPONENTS API

- declaring components @Component
- defining the component interface with inputs, outputs inside @Component annotation
- manage the component lifecycle with ngOnInit, ngOnChanges and ngOnDestroy
- linking components with each other

as always, embracing HTML

ASIDE - COMPONENTS VS HTML ELEMENTS

```
<nav class="page-left-menu">

    <section class="compose-toolbar">
        <button (click)="compose()" class="btn btn-danger">Compose</button>
    </section>

    <section class="folder-list">

        <folder-list
            [folders]="folders"
            [defaultFolder]="defaultFolder"
            (selected)="selectFolder($event.folder)">
        </folder-list>

    </section>
</nav>
```

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

```
import { Component } from '@angular/core';

@Component({selector: 'hello',
  template: "<h3>Hello World</h3>",
})
export class HelloWorldComponent {}
```

in the page

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

LAB 01

Create the <mail-logo> component

In the base application inside the labs folder you will find the mail-logo folder containing an empty scheleton of component

- the html template is already done
- complete the component definition on mail-logo.ts
- import the component in the app.ts file.

Remember: TEST the page at each step

TIPS

- use templateUrl into the Component annotation to address the right template
- import the component into app.ts

```
import {MailLogoComponent} from "components/mail-logo/mail-logo.ts"
```

- add the component to the module declarations (app.ts)

WHAT'S IN A COMPONENT?

- a selector, to reference it in HTML
 - a tag name in the simplest case
- some HTML
 - inline, with template
 - in an external file, with templateUrl
 - dynamic (inputs, outputs)
- the Component Class

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

Who is passing inputs to the other components? Role of the
MailView component

- 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

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

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"
})
export class MessageViewerComponent {
  @Input()
  message;
```

IN THE MAIL-VIEW.HTML

```
<div>

<section class="main-pane">
  <message-list
    [messages]="messages">
  </message-list>
</section>
</div>
```

IN THE COMPONENT DEFINITION

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

@Component({selector: 'message-list',
  templateUrl : "components/message-list/message-list.html",
  inputs : ["messages"]

})
export class MessageListComponent {
  messages;
  ...
}
```

This is automatically available as a `messages` field in the component instance

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

IN THE COMPONENT HTML

```
<div *ngFor="let message of messages">
</div>
```

THE MAIL-MESSAGE-LIST COMPONENT

Manages

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

LAB 02

Define the <message-viewer> component

In the base application inside the labs folder you will find the message-viewer folder containing an empty scheleton of component

Steps

- complete the message-viewer component to handle
 - message input
- handle click events
 - reply
 - forward
 - delete
- and log the function call

...

- import the component in the app.ts file.
- edit the mail message html into message-viewer.html
- complete the component definition in message-viewer.ts, passing in the message parameter
- use messages[0] from the mail-view component

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

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

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

OUTSIDE THE COMPONENT

We would like to be notified

```
<message-list
    [messages]="messages"
    (onCurrentMessageChanged)="selectCurrentMessage($event.message)"

    >
</message-list>
```

WE NEED TWO STEPS TO DO THIS

1) declare the event in the bindings

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

@Component({selector: 'message-list',
  templateUrl : "components/message-list/message-list.html"

})
export class MessageListComponent  {
  messages;
  currentMessageIndex = 0;
  currentMessage;

  @Output()
  onCurrentMessageChanged = new EventEmitter<any>();
```

This injects an onCurrentMessageChanged event callback in the component instance

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

```
setCurrentMessage(index)
{
    this.currentMessage = this.messages[index];
    this.currentMessageIndex = index;

    this.onCurrentMessageChanged.emit({
        message: this.currentMessage
    });
}
...
```

LET'S CHANGE TOGETHER THE MESSAGE-LIST BEHAVIOUR

adding onCurrentMessageChanged outputs

LAB 02 (2)

Declare the output events in `MessageViewerComponent.ts`

- reply
- forward
- delete

Handle the button click events and emit the events

In the parent html (`mail-view.html`)

- bind the events to the `MailViewComponent` class methods

HOW TO TRANSCLUDE CONTENTS

Like ng-transclude on AngularJS. Allows to inject DOM objects inside a component

```
<ng-content></ng-content>
```

EXAMPLE

```
import { Component, Input, Output } from '@angular/core';
@Component({
  selector: 'card',
  templateUrl: 'card.component.html',
})
export class CardComponent {
  @Input() header: string = 'this is header';
  @Input() footer: string = 'this is footer';
}
```

EXAMPLE

```
<div class="card">
  <div class="card-header">
    {{ header }}
  </div>

  <!-- single slot transclusion here -->
  <ng-content></ng-content>

  <div class="card-footer">
    {{ footer }}
  </div>
</div>
```

EXAMPLE

```
<h1>Single slot transclusion</h1>
<card header="my header" footer="my footer">
  <!-- put your dynamic content here -->
  <div class="card-block">
    <h4 class="card-title">You can put any content here</h4>
    <p class="card-text">For example this line of text and</p>
    <a href="#" class="btn btn-primary">This button</a>
  </div>
  <!-- end dynamic content -->
<card>
```

NG-CONTENT: SELECT ATTRIBUTE

```
...
<ng-content select="[card-body]"></ng-content>
...
```

```
<h1>Single slot transclusion</h1>
<card header="my header" footer="my footer">

<div class="card-block" card-body><!-- We add the card-body attribute here -->
    <h4 class="card-title">You can put any content here</h4>
    <p class="card-text">For example this line of text and</p>
    <a href="#" class="btn btn-primary">This button</a>
</div>

<card>
```

NG-CONTENT: SELECT ATTRIBUTE

```
...
<ng-content select=".card-body"></ng-content>
...
```

```
<h1>Single slot transclusion</h1>
<card header="my header" footer="my footer">

  <div class="card-block card-body">
    <h4 class="card-title">You can put any content here</h4>
    <p class="card-text">For example this line of text and</p>
    <a href="#" class="btn btn-primary">This button</a>
  </div>

</card>
```

MULTI-SLOT TRANSCLUSION

```
<div class="card">
  <div class="card-header">
    <!-- header slot here --&gt;
    &lt;ng-content select="card-header"&gt;&lt;/ng-content&gt;
  &lt;/div&gt;
  <!-- body slot here --&gt;
  &lt;ng-content select="card-body"&gt;&lt;/ng-content&gt;
  &lt;div class="card-footer"&gt;
    <!-- footer --&gt;
    &lt;ng-content select="card-footer"&gt;&lt;/ng-content&gt;
  &lt;/div&gt;
&lt;/div&gt;</pre>
```

OPTIONAL LAB

Refactor the message-viewer component removing the <h3> tag containing the title.

Create a new component named message-card with:

- An header section that will handle the message title.
- A ng-content section which will contain the message-viewer

...

HOW THE MAIL-VIEW SHOULD LOOK LIKE

```
<message-card [header]="currentMessage.title" *ngIf="currentMessage">
  <message-viewer
    [message]="currentMessage"
    (onDelete)="delete($event.message)"
    (onReply)="replyTo($event.message)"
    (onForward)="forward($event.message)"
  >
</message-viewer>
</message-card>
```

LIMIT OF NG-CONTENT

- It's not possible to use it on an
 `*ngFor`
- but there's a way: `ng-template`

[https://blog.angular-university.io/angular-ng-template-ng-container-
ngtemplateoutlet/](https://blog.angular-university.io/angular-ng-template-ng-container-ngtemplateoutlet/)

EXAMPLE

List with customizable content

```
@Component({
  selector: 'awesome-list',
  templateUrl: 'awesome-list.component.html',
})
export class CardComponent {
  @Input() itemTemplate: string = '';
}
```

```
<div class="list">
  <div *ngFor="let message of messages" class="list-item">
    <ng-container
      *ngTemplateOutlet="itemTemplate;context:ctx">
    </ng-container>
  </div>
</div>
```

COMPDOC

```
compodoc -p ./tsconfig.json -d docs demo-2.0/
```

```
compodoc -p ./tsconfig.json -d docs
```

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

FOLDERLIST COMPONENT

Declaring a Component

1. import Component Class in app.ts
2. add to declarations: [] section
3. completing the declaration
 - define metadata
 - complete html template
 - activate the selected class on click

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

...

Declaring outputs

1. Instantiate event Emitter
2. Handle click on folder
3. initially, just log it
4. Emit the event on click
5. Bind the event in the parent html template (mail view)
6. Implement the `selectFolder()` method in the parent component

LAB EXTRA

Pass an additional allow-create="true" parameter

LAB EXTRA (2)

Use templates to customize list

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

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

COMPONENT-BASED UI ARCHITECTURE

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

LET'S CHANGE TOGETHER THE MAIL-VIEW BEHAVIOUR

Bind the inputs and outputs

In the `mail-view.html` conditionally display the compose section
when reply / forward is selected

Add the message reply logic to `mail-view.ts`

- sender becomes to field
- Prepend "Re" to Subject
- Prefix body text with ">"

TEMPLATE-DRIVEN FORMS

- quick setup
- based on familiar ngModel directive
 - similar to Angular 1.x
- more difficult to dynamically add/modify fields

TEMPLATE-DRIVEN FORM RECIPE

- include `FormsModule` in the module imports: [] section
 - `import { FormsModule } from '@angular/forms';`
- add the `<form>` tag
- include name attribute for each `input` tag
 - e.g. `<input type="text" name="user_name">`
- add DataBinding to `input` tags
 - `[(ngModel)]="user.user_name"`
- Optionally add validations e.g. required or
 - `[required]="conditional_expression"`

ADVANCED FORMS FEATURES

- Give the form a name
 - <form #userForm="ngForm">
 - meaning this form has an id of userForm which will reference the "ngForm" directive instance in the controller
- Try printing it

```
 {{ userForm.valid }}  
<pre>{{ userForm.valid | json }}</pre>
```

FORM STATE MANAGEMENT

- ngModel will automatically update the following control, form and css properties
 - form.valid -> ng-valid css class
 - form.field.valid -> ng-valid css class
 - form.field.invalid -> ng-invalid css class
 - and recursively compute them on forms and subforms
- Other
 - valid - invalid
 - dirty - pristine
 - touched - untouched

MAKE CSS NICE

```
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}
.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

CUSTOM VALIDATION MESSAGES

```
<input type="text" id="name"
       required
       [(ngModel)]="user.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
      class="alert alert-danger">
```

OTHER FEATURES

- Reset a form to initial state `userForm.reset()`
- Handle form submission (e.g. 'ENTER' key)
 - `<form (ngSubmit)="onSubmit()" #userForm="ngForm">`
- Prevent submission on invalid form

```
<button type="submit" class="btn btn-success" [disabled]="!userForm.form.valid">Submit</button>
```

LAB 04

Integrate the mail-composer component

Include the component in the app module

 Declare inputs

- draft

 Declare output events

- send, cancel, and optionally
 save

Include it in the parent html template

ADDITIONAL LAB

- play with validation: make subject required and with minimum 3 characters length.
- Disable the send button if the form is not valid
- Display a validation error message

#COMPONENTID

- As we saw using ngForm, We can assign an Id to a component and use it to access it's methods from the html.

```
<modal #errorModal>
  <div class="modal-content">
    <p>My modal content</p>
  </div>
</modal>
...
<button (click)="errorModal.open()">Open Modal</button>
```

MORE CONTROL ACCESS DOM FROM TS

- sometimes we need to access the DOM
- for instance to wrap a third party JS library in an angular component
- or to make some directives

ELEMENTREF

The following example prints the html of the component itself

```
import { AfterContentInit, Component, ElementRef } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>My App</h1>
    <pre>
      <code>{{ node }}</code>
    </pre>
  `
})
export class AppComponent implements AfterContentInit {
  node: string;

  constructor(private elementRef: ElementRef) { }

  ngAfterContentInit() {
    const tmp = document.createElement('div');
    const el = this.elementRef.nativeElement.cloneNode(true);

    tmp.appendChild(el);
  }
}
```

AN EXAMPLE OF DIRECTIVE

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

COMMON UTILITY DIRECTIVES: HOSTLISTENER

```
class CardHoverDirective {
  constructor(private el: ElementRef,
             private renderer: Renderer)
  // renderer.setStyle(el.nativeElement, 'backgroundColor', 'gray');
}

@HostListener('mouseover') onMouseOver() {
  let part = this.el.nativeElement.querySelector('.card-text')
  this.renderer.setStyle(part, 'display', 'block');
}
}
```

COMMON UTILITY DIRECTIVES: HOSTBINDING

```
class CardHoverDirective {  
  @HostBinding('class.card-outline-primary') private ishovering: boolean;  
  
  constructor(private el: ElementRef,  
             private renderer: Renderer) {  
    // renderer.setStyle(el.nativeElement, 'backgroundColor', 'gray');  
  }  
  
  @HostListener('mouseover') onMouseOver() {  
    let part = this.el.nativeElement.querySelector('.card-text');  
    this.renderer.setStyle(part, 'display', 'block');  
    this.ishovering = true;  
  }  
  
  @HostListener('mouseout') onMouseOut() {  
    let part = this.el.nativeElement.querySelector('.card-text');  
    this.renderer.setStyle(part, 'display', 'none');  
    this.ishovering = false;  
  }  
}
```

HOSTBINDING AND HOSTLISTENER EXAMPLES

<https://alligator.io/angular/hostbinding-hostlistener/>

<http://plnkr.co/edit/EgsmbXMN7s7YYDYIu9N8?p=preview>

OPTIONAL LAB

Use Host @HostBinding and @HostListener to

- On hovering
 - create a directive changing the background color of a folder on hovering
 - Showing an message under the folder list with the name of the folder

LIFECYCLE CALLBACKS

TOPICS

SIMPLIFY THE LIFECYCLE OF A COMPONENT

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

NGONINIT

Called when the component is initialized

```
@Component({selector: 'my-cmp', template: `...`})
class MyComponent implements OnInit {
  ngOnInit() {
    // ...
  }
}
```

NGONCHANGES

Is called right after the data-bound properties have been checked and before view and content children are checked if at least one of them has changed. The changes parameter contains the changed properties

```
@Component({selector: 'my-cmp', template: `...`})
class MyComponent implements OnChanges {
  @Input()
  prop: number;
  ngOnChanges(changes: SimpleChanges) {
    // changes.prop contains the old and the new value...
  }
}
```

NGONDESTROY

Is typically used for any custom cleanup that needs to occur when the instance is destroyed.

```
@Component({selector: 'my-cmp', template: `...`})
class MyComponent implements OnDestroy {
  ngOnDestroy() {
    // ...
  }
}
```

LAB 05

Into the message-list component

Implement the ngOnChanges callback

TO LEARN MORE

<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

<https://teropa.info/blog/2016/03/06/writing-an-angular-2-template-directive.html>

LAB 06

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

IMPROVE PRODUCTIVITY

<http://blog.mgechev.com/2017/04/23/angular-tooling-codelyzer-angular-cli-ngrev/>

ANGULAR CLI

<https://github.com/angular/angular-cli>

Let's try it

HOW TO CONFIGURE AN APPLICATION

- before AOT: using environment files
- after AOT:
- Using a custom js file in the index.html with some global variables to configure the application
- Using a server side configuration retrieved during the application startup.
- Using a json file published from the same server where the application is.

PACKAGING

- Angular-cli allows to create a build package for an entire application

```
ng build --prod
```

But... no methods to build a library... until yesterday!

PACKAGING A LIBRARY BEFORE ANGULAR 6

- A simple application you can use to make a library
<https://github.com/filipesilva/angular-quickstart-lib>
- a packager tool <https://github.com/dherges/ng-packagr>
- a Yeoman generator <https://github.com/jvandemo/generator-angular2-library>

PACKAGING A LIBRARY NOWADAYS

On an angular 6 project just run

```
ng generate library demo-components -p demo
```

- -p will put 'demo' as prefix on all the components on the library.

To build it

```
ng build demo-components
```

To release it on NPM

```
cd dist/demo-components  
npm publish
```

You can release the library also on your own NPM server (like Nesux for instance) configuring NPM for that

LET'S CREATE A NEW WORKSPACE TO TRY IT

<https://medium.com/@SirMaxxx/angular-6-creating-a-shareable-control-library-6a27f0ebe5c2>

```
ng g new demo-workspace -S
```

CREATE THE LIBRARY

```
cd demo-workspace  
ng g library my-components -p my
```

CREATE TWO APPLICATIONS

```
ng g application demo-website --prefix web  
ng g application demo-app --prefix app
```

Customize the app.component of each app to be able to distinguish them

CREATE A NEW COMPONENT IN THE LIBRARY

Create a new component

```
cd projects/my-components/src/lib  
ng generate component checkbox  
ng generate component textinput
```

... and build the library

```
ng build my-components
```

BEWARE: The cli will not export the components you created. You should do it manually

INSTALL THE LIBRARY ON ONE OF THE APPS

```
npm install dist/my-components
```

Add it on the module

```
import { MyComponentsModule } from 'my-components';

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

and use the widgets you made.

TRY THEM

```
ng serve demo-app  
ng serve demo-website
```

DEPENDENCY INJECTION RECIPE

- Only if you are using SystemJS and not an Angular-cli based application
- make sure annotation support is enabled

```
typescriptOptions: {  
  enableMetadataSupport: true  
}
```

in `tsconfig.json` and/or `system.conf.js`

DEPENDENCY INJECTION

- One of Angular's biggest features and selling points
- It allows us to inject dependencies in different components across our applications

PROVIDERS ON NgModule

LAZILY INSTANTIATED

Angular only instantiates a service when an application component depends on it.

INJECTED IN COMPONENTS AUTOMATICALLY

Each component dependent on a service gets a reference to the single instance generated by the service factory.

DECLARE SERVICE TO BE INJECTED

```
import 'Injectable' from  
  
@Injectable  
class MessageService {  
  
}
```

ADD SERVICE TO PROVIDERS SECTION

```
providers: [MessageService]
```

REFERENCE SERVICE IN COMPONENT CONSTRUCTOR

```
constructor(private messageService: MessageService)
```

EXAMPLE

```
@NgModule({
  imports:      [ BrowserModule, FormsModule, HttpModule ],
  declarations: [ MailViewComponent, MailLogoComponent, MessageListComponent, CommonStarComponent
    ],
  providers: [LogService, MessageService, FolderService],
  bootstrap:   [ MailViewComponent ]
})
export class AppModule { }
```

LogService, MessageService, FolderService are singletons and injectable on every component inside the app

ANOTHER WAY TO PROVIDE SERVICES

FROM ANGULAR 6

Instead of add them to the providers array of the module it's possible to declare directly on the @Injectable annotation

```
@Injectable({  
  providedIn: 'root',  
})
```

PROVIDERS ON COMPONENTS

LAZILY INSTANTIATED

Angular only instantiates a service when an application component depends on it.

INJECTED IN COMPONENTS AUTOMATICALLY

Each component has it's own instance of the provider

PROVIDERS OR VIEWPROVIDERS

- ViewProvicers are local for the declaring component
- Providers are visible also for the declaring component's children

EXAMPLE 1

```
@Component({
  selector: 'greet',
  viewProviders: [
    Greeter
  ],
  template: `<needs-greeter></needs-greeter>`
})
class GreetComponent {
```

Each instance of <greet> has its own Greeter instance. If also <needs-greeter> needs Greeter, it will have its own instance too

EXAMPLE 2

```
@Component({
  selector: 'greet',
  providers: [
    Greeter
  ],
  template: `<needs-greeter></needs-greeter>`
})
class GreetComponent {
```

Each instance of <greet> has its own Greeter instance. If also <needs-greeter> needs Greeter, it will share the same instance of the parent component.

LET'S CREATE A SERVICE TOGETHER

- Create a TemplateService class
- Create the getReplyTemplate() method
- Replace the old code with the new method call
- import the service using DI

LAB

- Create the FolderService class
- methods
 - getCustomFolders()
 - getFolders()
- Replace the hard-coded folder lists with the new method calls
- Add the FolderService class as a Provider in the module
- Inject it in the component constructor
- Call its methods

ADVANCED DI

- You can inject Parent components inside child components
- You can inject services into other service

INJECT PARENT COMPONENTS INSIDE CHILD COMPONENTS

```
@Component({  
  selector: 'my-comp'  
  ...  
})  
class MyComponent {  
  constructor(@Optional() private container:MyContainer) {  
  
  }  
  
  handleClick = function(){  
    if(this.container){  
      this.container.foo();  
    }  
  }  
}
```

@INJECTABLE

- Injecting Services into Services

LAB

Create the LogService and inject it into the other services

Injector Hierarchy

<https://angular.io/docs/ts/latest/guide/dependency-injection.html>

<https://blog.thoughttram.io/angular/2015/05/18/dependency-injection-in-angular-2.html>

INJECTION TOKENS

When you register a provider with an injector, you associate that provider with a dependency injection token. The injector maintains an internal token-provider map that it references when asked for a dependency.

```
export const TITLE = new InjectionToken<string>('title');

...
providers: [
  ...
  LocalStorageService,
  { provide: HeroService,    useClass:    HeroService },
  { provide: TITLE,         useValue:     'Hero of the Month' },
  { provide: LoggerService, useClass:    DateLoggerService },
  { provide: MinimalLogger, useExisting:  LoggerService },
  { provide: ShiftService,   useFactory:   shiftServiceFactory,
    deps: [HttpClient, SettingsService]
  }
]
...

export function shiftServiceFactory (http: HttpClient, settingsService:SettingsService) {

  if (environment.production) {
    return new ShiftServiceImpl(http, settingsService);
  }
}
```

INJECTION TOKENS

On the component

```
constructor(@Inject(TITLE) public title: string) {}
```

ADVANCED REFERENCES

<https://blog.thoughttram.io/angular2/2015/11/23/multi-providers-in-angular-2.html>

INTEGRATION WITH THE REST BACKEND

MAKING HTTP CALLS WITH PROMISES

WHAT'S A PROMISE

```
this.http.get(this.heroesUrl)
  .toPromise()
  .then(this.extractData)
  .catch(this.handleError);
```

CREATING A PROMISE

```
new Promise((resolve, reject) => {
  if (success)
    resolve(42);
}) ;
```

And resolving it later

HTTP CLIENT

- <https://angular.io/docs/ts/latest/guide/server-communication.html>

SETUP

- import

```
import { HttpClientModule } from '@angular/common/http'
```

- add HttpClientModule to main NgModule

GET

```
import { HttpClient }           from '@angular/common/http';

@Injectable()
export class HeroService {
  private heroesUrl = 'api/heroes'; // URL to web API
  constructor (private http: HttpClient) {}
  getHeroes() {
    return this.http.get<Hero[]>(this.heroesUrl).toPromise();
  }
}
```

ON THE COMPONENT

```
this.heroService.getHeroes()
  .then(this.extractData)
  .catch(this.handleError);

private extractData(res:any) {
  //do what you want with the data retrieved
}
```

TYPESCRIPT STRING TEMPLATE

- Composing URLs and strings is easier using string templates

```
let var1 = 1;
let var2 = 'my_var_2';

let url = `resource/${var1}/${var2}`;
```

TYPESCRIPT STRING TEMPLATE

- What you should do:
 - urls
 - error and alert messages
- What you should NOT do:
 - html injection

REAL LIFE EXAMPLE

```
import { HttpClient }           from '@angular/common/http';

@Injectable()
export class HeroService {
  constructor (private http: HttpClient) {}
  getHero(id:string) {
    let url = `resource/${id}`;
    return this.http.get<Hero>(url).toPromise();
  }
}
```

HANDLING ERRORS

```
private handleError (error: Response | any) {
  // In a real world app, you might use a remote logging infrastructure
  let errMsg: string;
  if (error instanceof Response) {
    const body = error.json() || '';
    const err = body.error || JSON.stringify(body);
    errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
  } else {
    errMsg = error.message ? error.message : error.toString();
  }
  console.error(errMsg);
}
```

CALLING THE SERVICE

```
this.heroService.getHero(id)
  .then(this.extractData)
  .catch(this.handleError);
```

MORE ON HTTP: POST

```
create(name: string): Observable<Hero> {
  let headers = new Headers({ 'Content-Type': 'application/json' });
  let options = new RequestOptions({ headers: headers });

  let body = {
    newName: name
  };

  return this.http.post<any>(this.heroesUrl, body, options).toPromise();
}
```

LAB: ADD HTTP CLIENTS CALLING MOCK REST DATA

Include the HttpClientModule in the main module

Import HttpClient service in MailService

Inject HttpClient service instance in the MailService constructor

Create the messages.json mock data file in the root project folder (or
better in a dedicated data folder)

Implement the http GET call

TIP

Encapsulate Backend Calls in a Service Layer

CUSTOM FORM ELEMENTS

CREATE A CUSTOM FORM ELEMENT

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

import {
  NG_VALUE_ACCESSOR,
} from '@angular/forms';

@Component({
  selector: 'form-text',
  template: `
    <div>
      <input type="text" [(ngModel)]="value" />
    </div>
  `,
  providers: [
    {provide: NG_VALUE_ACCESSOR, useExisting: forwardRef(() => FormTextComponent), multi: true}
  ],
})
export class FormTextComponent {}
```

IMPLEMENT CONTROLVALUEACCESSOR INTERFACE

```
interface ControlValueAccessor {  
  writeValue(obj: any): void  
  registerOnChange(fn: any): void  
  registerOnTouched(fn: any): void  
  setDisabledState(isDisabled: boolean): void  
}
```

IMPLEMENT CONTROLVALUEACCESSOR INTERFACE

```
import {ControlValueAccessor} from '@angular/forms';

export class FormTextComponent implements ControlValueAccessor {
  private innerValue: string;

  private changed = new Array<(value: string) => void>();
  private touched = new Array<() => void>();

  get value(): string {
    return this.innerValue;
  }

  set value(value: string) {
    if (this.innerValue !== value) {
      this.innerValue = value;
      this.changed.forEach(f => f(value));
    }
  }

  writeValue(value: string): void {
    this.innerValue = value;
  }

  registerOnChange(fn: any): void {
    this.changed.push(fn);
  }

  registerOnTouched(fn: any): void {
    this.touched.push(fn);
  }
}
```

CUSTOM VALIDATION 1

```
@Directive({
  selector: '[forbiddenName]',
  providers: [{provide: NG_VALIDATORS, useExisting: forwardRef(() => ForbiddenValidatorDirective), multi: true}]
})
export class ForbiddenValidatorDirective implements Validator {
  @Input() forbiddenName: string;

  validate(control: AbstractControl): {[key: string]: any} {
    return this.forbiddenName ? this.forbiddenNameValidator(new RegExp(this.forbiddenName, 'i')) : null;
  }

  forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
    return (control: AbstractControl): {[key: string]: any} => {
      let forbidden = nameRe.test(control.value);
      return forbidden ? {'forbiddenName': {value: control.value}} : null;
    };
  }
}
```

```
<input id="name" name="name" class="form-control"
       required minlength="4" forbiddenName="bob"
       [(ngModel)]="hero.name" #name="ngModel" >
```

CUSTOM VALIDATION 2

```
import { Directive, forwardRef } from '@angular/core';
import { NG_VALIDATORS, FormControl } from '@angular/forms';

function validateEmailFactory(emailBlackList: EmailBlackList) {
  return (c: FormControl) => {
    let EMAIL_REGEXP = /^[a-zA-Z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-zA-Z0-9]([a-zA-Z0-9-]*[a-zA-Z0-9])?(\.[a-zA-Z0-9-]*[a-zA-Z0-9])*/g;
    return EMAIL_REGEXP.test(c.value) ? null : {
      validateEmail: {
        valid: false
      }
    };
  };
}

@Directive({
  selector: '[validateEmail][ngModel]',
  providers: [
    { provide: NG_VALIDATORS, useExisting: forwardRef(() => EmailValidator), multi: true }
  ]
})
```

REFERENCES

<https://blog.thoughtram.io/angular/2016/03/14/customValidators-in-angular-2.html>

git merge --no-ff mattex83-angular2 Angular - Advanced topics

PART 3 - ADVANCED ANGULAR RECIPES

SOME REFERENCES ABOUT ANGULAR BASICS

Angular 4 with Yakov Fain <https://www.youtube.com/watch?v=k8r76d8QzXs>

TypeScript with Dan Wahlin https://www.youtube.com/watch?v=4xScMnaasG0&feature=em-subs_digest-vrecs

Packaging Angular https://www.youtube.com/watch?v=uIICbsPGFIA&feature=em-subs_digest-vrecs

OBSERVABLES AND REACTIVE PROGRAMMING

BEFORE WE START

Angular 6 came with RxJS 6 which breaks backward compatibility. To make it work on legacy project you should use rxjs-compat

```
npm install rxjs@6 rxjs-compat@6 --save
```

WHAT'S AN OBSERVABLE

Observable == stream of events

- published by some source
- observed/subscribed by one or more functions

To listen for events in this stream, subscribe to the Observable.

The introduction to Reactive Programming you have been missing
(Andre Staltz) <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>

Functional Programming you already know (Kevlin Henney)
<https://www.youtube.com/watch?v=lNKXTlCOGEc>

OBSERVABLE'S FUNCTIONS

- `next(value)`: called every time a new event is generated
- `error(err)`: called when an error occurs. It will terminate the observer (so no more events may be received)
- `complete()`: called when the observer complete the procedure (so no more events may be received).

LET'S TRY SOMETHING

<https://jsfiddle.net/8ptLorpj/>

CONSUMING EVENTS

```
source.subscribe(  
    event => { console.log("Event received "+event); },  
    error => this.errorMessage = <any>error);
```

Remember to unsubscribe the stream at the end to prevent
memory leaks

WITHOUT USING => FUNCTIONS

```
var observer = {
  next: function(value) {
    console.log(value);
  },
  error: function(error) {
    console.error(error);
  },
  complete: function() {
    console.log('Observer completed');
  }
};

Rx.Observable.fromEvent(button, 'click')
.subscribe(observer)
```

<https://jsfiddle.net/kq1cr1ns/>

CREATING AN OBSERVABLE

A stream containing a single value

```
var source = Rx.Observable  
  .of({name: "Carlo"});
```

A number sequence as a stream

```
var source = Rx.Observable  
  .range(1, 10);
```

Or from an Array

```
var source = Rx.Observable.from(array);
```

- <https://github.com/ReactiveX/rxjs>
- <http://reactivex.io/rxjs/>
- <http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html>
- <http://reactivex.io/documentation/observable.html>

CREATING AN OBSERVABLE MANUALLY

```
Rx.Observable.create(function(obs) {
  obs.next('Value emitted');
  setTimeout(function() {
    obs.complete();

  }, 2000);
  //obs.error('Error');
  obs.next('Value emitted 2');
}) .subscribe(observer)
```

<https://jsfiddle.net/agjemjxk/>

UNSUBSCRIBE

```
var sub1 = Rx.Observable.fromEvent(button, 'click')
.subscribe(observer)

setTimeout(function() {
  sub1.unsubscribe();
}, 3000)
```

<https://jsfiddle.net/9q297vo7/>

https://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/creating_and_querying_observers.html

A TIMER / SCHEDULER

```
var source = Rx.Observable.timer(  
  5000, /* 5 seconds */  
  1000 /* 1 second */)  
  .timestamp(); //adds the timestamp to generated events  
  
var subscription = source.subscribe(  
  x => console.log(x.value + ': ' + x.timestamp));
```

PUBLISH/SUBSCRIBE WITH SUBJECTS

A subject is an observer that can be controlled

- <http://reactivex.io/documentation/subject.html>

USING SUBJECTS

```
subject.subscribe({
  next:function(value) {
    console.log('1 subscriber ' + value);
  },
  error:function(error) {
    console.error(error);
  },
  complete:function() {
    console.log('completed');
  }
});

subject.subscribe({
  next:function(value) {
    console.log('2 subscriber ' + value);
  }
});

subject.next('test');
```

<https://jsfiddle.net/w6jy1fyx/1/>

COMBINING AND PROCESSING STREAMS: OPERATORS

- <http://reactivex.io/documentation/operators.html>
- <https://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html>
- <https://netbasal.com/rxjs-six-operators-that-you-must-know-5ed3b6e238a0>
- <http://reactive.how/>
- <https://rxviz.com/>

MAP OPERATOR

```
var observable = Rx.Observable.interval(1000);

var observer = {
  next: function(value) {
    console.log(value);
  }
};

observable.map(
  (x) => 'Value is ' + x
).subscribe(observer);
```

<https://jsfiddle.net/zv1yt9bq/>

THROTTLETIME OPERATOR

```
var observable = Rx.Observable.interval(1000);

var observer = {
  next: function(value) {
    console.log(value);
  }
};

observable.map(
  (x) => 'Value is ' + x
).throttleTime(3000).subscribe(observer);
```

<https://jsfiddle.net/zv1yt9bq/>

LET'S SEE IT IN ACTION

- <http://rxmarbles.com/>

ANOTHER EXAMPLE

- Event
Demo

<https://github.com/carlobonamico/angular-event-streams-lab>

LAB

Use JSFiddle at <https://jsfiddle.net/8ptLorpj/> to try ReactiveX

- Create an observable using one of the one available on the documentation
- Create two or more kind of observers subscribing the observable with different operators (even chained).

For instance the first observer using a map operator, the second using a debounce and a filter.

MANAGING HTTP CALLS WITH RXJS

```
import { Http, Response }           from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/map';

@Injectable()
export class HeroService {
  private heroesUrl = 'api/heroes'; // URL to web API
  constructor (private http: Http) {}
  getHeroes(): Observable<Hero[]> {
    return this.http.get(this.heroesUrl)
      .map(this.extractData)
      .catch(this.handleError);
  }

  extractData(res: Response) {
    let body = res.json();
    return body || {};
  }
}
```

MANAGING ERRORS

```
private handleError (error: Response | any) {
  // In a real world app, you might use a remote logging infrastructure
  let errMsg: string;
  if (error instanceof Response) {
    const body = error.json() || '';
    const err = body.error || JSON.stringify(body);
    errMsg = `${error.status} - ${error.statusText || ''} ${err}`;
  } else {
    errMsg = error.message ? error.message : error.toString();
  }
  console.error(errMsg);
  return Observable.throw(errMsg);
}
```

ON THE OTHER SIDE...

```
this.heroService.getHeroes()
  .subscribe(
    heroes => this.heroes = heroes,
    error => this.errorMessage = <any>error);
```

NOW I SEE WHY...

Creating a smart Autocomplete widget

UI

```
@Component ({  
  selector: 'my-wiki-smart',  
  template: `  
    <h1>Smarter Wikipedia Demo</h1>  
    <p>Search when typing stops</p>  
    <input #term (keyup)="search(term.value)" />  
    <ul>  
      <li *ngFor="let item of items | async">{{item}}</li>  
    </ul>`,  
  providers: [ WikipediaService ]  
})
```

Async pipe?

ASYNC PIPE

<https://angular.io/docs/ts/latest/api/common/index/AsyncPipe.html>

- The async pipe subscribes to an Observable or Promise and returns the latest value it has emitted.
- When a new value is emitted, the async pipe marks the component to be checked for changes.
- When the component gets destroyed, the async pipe unsubscribes automatically to avoid potential memory leaks.

<https://blog.thoughttram.io/angular/2016/01/07/taking-advantage-of-observables-in-angular2-pt2.html>

PIPING REQUESTS

```
export class WikiSmartComponent implements OnInit {
  items: Observable<string[]>;
  constructor (private wikipediaService: WikipediaService) {}
  private searchTermStream = new Subject<string>();
  search(term: string) { this.searchTermStream.next(term); }
  ngOnInit() {
    this.items = this.searchTermStream
      .debounceTime(300)
      .distinctUntilChanged()
      .switchMap((term: string) => this.wikipediaService.search(term));
  }
}
```

PIPEBLE OPERATORS

BEFORE

```
source
  .map(x => x + x)
  .mergeMap(n => of(n + 1, n + 2)
    .filter(x => x % 1 == 0)
    .scan((acc, x) => acc + x, 0)
  )
  .catch(err => of('error found'))
  .subscribe(printResult);
```

PIPEBLE OPERATORS

NOW

```
source.pipe(  
    map(x => x + x),  
    mergeMap(n => of(n + 1, n + 2).pipe(  
        filter(x => x % 1 == 0),  
        scan((acc, x) => acc + x, 0),  
    )),  
    catchError(err => of('error found')),  
) .subscribe(printResult);
```

PIPING REQUESTS

```
export class WikiSmartComponent implements OnInit {
  items: Observable<string[]>;
  constructor (private wikipediaService: WikipediaService) {}
  private searchTermStream = new Subject<string>();
  search(term: string) { this.searchTermStream.next(term); }
  ngOnInit() {
    this.items = this.searchTermStream.pipe(
      debounceTime(300),
      distinctUntilChanged(),
      switchMap((term: string) => this.wikipediaService.search(term))
    )
  }
}
```

EXPLAINING THE EXAMPLE

- debounceTime waits for the user to stop typing for at least 300 milliseconds.
- distinctUntilChanged ensures that the service is called only when the new search term is different from the previous search term.
- the switchMap calls the WikipediaService with a fresh, debounced search term and coordinates the stream(s) of service response.

HOW DO I...

https://xgrommx.github.io/rx-book/content/getting_started_with_rxjs/creating_and_querying_observers.html

https://xgrommx.github.io/rx-book/content/which_operator_do_i_use/instance_operators.html

CREATING A CUSTOM OPERATOR

LAGACY

```
Observable.prototype.userDefined = () => {
  return new Observable((subscriber) => {
    this.subscribe({
      next(value) { subscriber.next(value); },
      error(err) { subscriber.error(err); },
      complete() { subscriber.complete(); },
    });
  });
};

source$.userDefined().subscribe();
```

CREATING A CUSTOM OPERATOR

RXJS 6

```
const userDefined = <T>() => (source: Observable<T>) => new Observable<T>((subscriber) => {
  this.subscribe({
    next(value) { subscriber.next(value); },
    error(err) { subscriber.error(err); },
    complete() { subscriber.complete(); },
  });
}) ;

source$.pipe(
  userDefined(),
)
.subscribe();
```

LAB

- call the Http service using Observable instead of Promises

```
this.messageService.getHttpObservableMessages()
  .subscribe(
    res => this.setMessages(res.json()),
    error => this.manageError(error)
  );
```

LAB

- Add a search field to the application.
- Use operators (and a subject) to send requests with a debounce time of 2 seconds.

ROUTING

WHAT IS ROUTING

- Wikipedia Example
- SPA Example

TYPES OF ROUTING

- Url-based
- State-based
- Component-based

<https://angular-2-training-book.rangle.io/handout/routing>

INSTALLING THE ROUTER

Add base URL to index.html

```
<base href="/">
```

Can also use dynamic script to automatically adapt

Add dependency to project

```
npm install -g @angular/router --save
```

DEFINING ROUTES

```
import { Routes, RouterModule } from '@angular/router';

const appRoutes: Routes = [
  { path: 'messages', component: MessageListComponent },
  { path: 'message/:id', component: MessageDetailComponent },
  {
    path: 'inbox',
    component: MessageListComponent,
    data: { title: 'Inbox', folder: 'inbox' }
  },
  { path: '',
    redirectTo: '/inbox',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

INCLUDING THE ROUTERMODULE DEPENDENCY

```
@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes)
    // other imports here
  ],
  ...
})
export class AppModule { }
```

ROUTED COMPONENTS

A component which receives parameters and state from the current route

ROUTER-OUTLET

To be put:

- in the high-level main component, defining the overall layout
`<router-outlet></router-outlet>`
- in sub-components for child routes
- can have multiple outlets with names

```
<router-outlet name="popup"></router-outlet>
```

CREATING LINKS

```
<nav>
  <a routerLink="/messages" routerLinkActive="active">Messages</a>
  <a routerLink="/inbox" routerLinkActive="active">Inbox</a>
</nav>
```

PASSING PARAMETERS

Path Parameters

```
{ path: 'message/:id', component: MessageDetailComponent },
```

```
this.router.navigate(['/message', message.id]);
```

READING PARAMETERS IN COMPONENTS

With Snapshot

```
ngOnInit() {  
  // (+) converts string 'id' to a number  
  let id = +this.route.snapshot.params['id'];
```

READING PARAMETERS WITH AN OBSERVABLE

```
ngOnInit() {  
    this.sub = this.route.params.subscribe(params => {  
        this.id = +params['id']; // (+) converts string 'id' to a number  
  
        // In a real app: dispatch action to load the details here.  
    });  
}  
  
ngOnDestroy() {  
    this.sub.unsubscribe();  
}
```

LAB

- Create a new application with angular-cli
- Generate 3 components:
 - main-page
 - detail-page with an id parameter
 - about-page
- put the <router-outlet> on the app.component

...

...

- define routes configuration

```
const appRoutes: Routes = [{  
    path: 'details',  
    component: DetailPageComponent  
}, {  
    ...  
}];
```

in @NgModule

```
imports: [  
    RouterModule.forRoot(appRoutes)  
,  
    ...  
],
```

...

...

- add a navigation menu to reach two detail-page and the about-page (start with the last one)

```
<a routerLink="/details" routerLinkActive="active">Messages</a>
```

- Print the id Parameter on detail-page

```
class DetailPageComponent {  
constructor(route: ActivatedRoute) {  
  
let id = route.snapshot.params.id
```

QUERY PARAMETERS

Use the [queryParams] directive along with [routerLink]

```
<a [routerLink]=["/details"] [queryParams]={ id: 99 }>Go to Id 99</a>
```

Navigate programmatically using the Router service:

```
goToPage(pageNum) {
  this.router.navigate(['/details'], { queryParams: { id: pageNum } });
}
```

RESOLVERS

<https://blog.thoughtram.io/angular/2016/10/10/resolving-route-data-in-angular-2.html>

```
@Injectable()
export class MessageDetailResolver implements Resolve<Message> {
  constructor(private messageService: MessageService, private router: Router) {}
  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Message> {
    let id = route.params['id'];

    return this.messageService.getHttpObservableMessages(id); //returns an observable
  }
}
```

RESOLVERS ON ROUTING CONFIGURATION

```
export const AppRoutes: Routes = [
  ...
  {
    path: 'message/:id',
    component: MessageDetailComponent,
    resolve: {
      message: MessageDetailResolver
    }
  }
];
```

RESOLVERS ON COMPONENT

```
@Component()
export class MessageDetailComponent implements OnInit {

  message;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.message = this.route.snapshot.data['message'];
  }
}
```

GUARDS

Determine if navigation is possible

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
    console.log('AuthGuard#canActivate called');
    return true;
  }
}
```

GUARDS FOR SECURITY

An example: authentication

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ],
      }
    ]
  }
];
```

<http://plnkr.co/edit/sRNxfXsbcWnPU818aZsu?p=preview>

GUARDS FOR UI (E.G. UNSAVED CHANGES)

```
import { Injectable }      from '@angular/core';
import { CanDeactivate }   from '@angular/router';
import { Observable }     from 'rxjs/Observable';
export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate) {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

CHILD ROUTES

When they must be accessible only within other routes and not by themselves.

```
const crisisCenterRoutes: Routes = [
  {
    path: 'messages',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: MessageListComponent,
        children: [
          {
            path: ':id',
            component: MessageDetailComponent
          },
          {
            path: 'help',
            component: HelpComponent
          }
        ]
      }
    ]
}
```

LAB

- clone the routing demo workspace

<https://github.com/mattex83/routing-demo>

... or continue using the application you made for the routing labs.

- git checkout empty-app
- Create three components <app-login>, <app-not-found> and <app-main>
- Use a guard to activate the <app-main> component when the user credentials are good ** just simulate it with fixed values
- put the <router-outlet> component instead a <app-root> component

LAZY LOADING OF A MODULE

<https://angular-2-training-book.rangle.io/handout/modules/lazy-loading-module.html>

<https://plnkr.co/edit/vpCqRHDAj7V6mlN1AknN?p=preview>

LAB

Refactor the application:

- keep <app-login> component and <app-not-found> component in the application module
- put other component and eventually its dependencies into a new module called *BodyModule*
- Lazy load the module when the navigation is performed

I18N

HOW THE PROCESS WORKS?

- Mark static text messages in your component templates for translation.
- An angular i18n tool extracts the marked messages into an industry standard translation source file.
- A translator edits that file, translating the extracted text messages into the target language, and returns the file to you.
- The Angular compiler imports the completed translation files, replaces the original messages with translated text, and generates a new version of the application in the target language.

YOU NEED TO BUILD AND DEPLOY A SEPARATE VERSION OF THE APPLICATION FOR EACH SUPPORTED LANGUAGE

THE ALTERNATIVE

NGX TRANSLATE <https://github.com/ngx-translate/core>

- Allows changing language at runtime
- same approach of translation service in AngularJS

I18N CUSTOM ATTRIBUTE

```
<h1 i18n>Hello i18n!</h1>
```

ADD A DESCRIPTION

```
<h1 i18n="An introduction header for this sample">Hello i18n!</h1>
```

ADD A MEANING

```
<h1 i18n="site header|An introduction header for this sample">Hello i18n!</h1>
```

ADD AN ID

On the generated xml file we will have

```
<trans-unit id="ba0cc104d3d69bf669f97b8d96a4c5d8d9559aa3" datatype="html">
```

the id may change on every export procedure, to avoid it

```
<h1 i18n="@@introductionHeader">Hello i18n!</h1>
```

the trans-unit will become

```
<trans-unit id="introductionHeader" datatype="html">
```

THIS WORKS ONLY WITH TEXT INSIDE TAGS?

What if I don't want to add a tag with the text inside?

```
<ng-container i18n>I don't output any element</ng-container>
```

HANDLE SINGLE OR PLURAL

```
<span i18n>{wolves, plural, =0 {no wolves} =1 {one wolf} =2 {two wolves} other {a wolf pack}}</span>
```

HOW TO EXTRACT I18N FILE

```
ng xi18n -op locale -of messages.en.xlf
```

HOW TO BUILD AN I18N APP

```
ng build --prod --i18n-file locale/messages.it.xlf --locale it --i18n-format xlf
```

MY PREFERRED WAY TO DO I18N

<https://github.com/nginx-translate/core>

LAB

- Add i18n feature into the routing app.
- extract the xlf file, compile the italian translation and build an
italian version of the application

ANGULAR 2 - TO PROBE FURTHER

Change Detection https://angular-2-training-book.rangle.io/handout/change-detection/angular_1_vs_angular_2.html

<https://juristr.com/blog/2016/04/angular2-change-detection/>

In Memory API <https://plnkr.co/edit/sLi9tWbekqAbxqwZEwkX?p=preview>

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
- separating 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 importance 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

The 7 stages of naming

By ArloBelshee

Missing

Nonsense

Honest

Honest &
Complete

Does the
Right Thing

Start
Structural
Refactoring

Intent

Domain
Abstraction

preload()

DoSomething
EvilTo
Database()

ParseXml
AndStoreFlight
ToDatabase
AndLocalCache
AndStart
Processing()

StoreFlight
ToDatabase
AndStart
Processing()

Begin
Tracking
Flight()

Monitoring
Panel
.Add(flight)

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

<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?

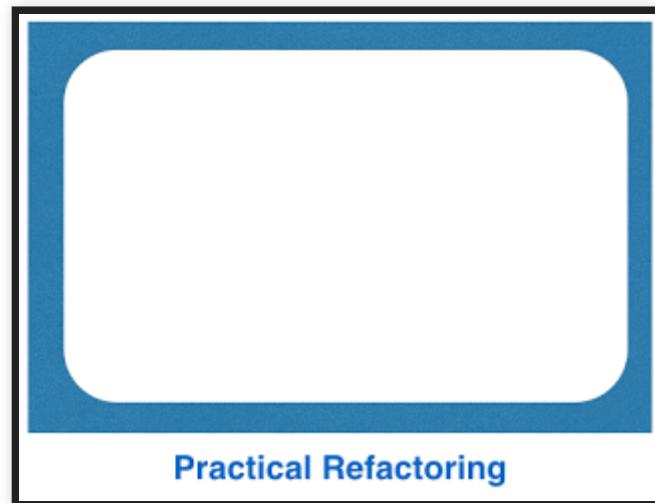
Or biscuits in the vegetable box?

So, why do 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

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@nispro.it /
carlo.bonamico@gmail.com

##

THANK YOU FOR YOUR ATTENTION

CARLO BONAMICO

@CARLOBONAMICO / @NIS_SRL

CARLO.BONAMICO@NISPRO.IT

CARLO.BONAMICO@GMAIL.COM

[HTTP://MILANO.CODEMOTIONWORLD.COM](http://MILANO.CODEMOTIONWORLD.COM)

[HTTP://TRAINING.CODEMOTION.IT/](http://TRAINING.CODEMOTION.IT/)