Busy Developer's Workshop: AngularJS

Ted Neward

Neward & Associates

http://www.tedneward.com | ted@tedneward.com

Credentials

Who are these guys?

- Ted Neward
  - **Principal -- Neward & Associates**
  - **Director -- Developer Relations, Smartsheet.com**
  - **Blog: http://blogs.tedneward.com**
  - **Writing: http://www.newardassociates.com/#/writing**
  - **Twitter: @tedneward**
  - **For more, see http://www.newardassociates.com/#/about**
- Garvice Eakins
  - **Senior Development Engineer, Smartsheet.com**
  - **Github: https://github.com/Garvice**

AngularJS (v2): The Workshop

How are we gonna do this?

Assumptions

We are assuming the following:

- You are comfortable with command-line tools and editor
- You are comfortable with HTML and web apps in general
- You are at least a little comfortable with "modern" JavaScript

  if not, consider Crockford's "JavaScript: The Good Parts"
- You know nothing about AngularJS or TypeScript

  except maybe that it exists and that "it's cool"

Objectives

Get up and running with AngularJS

- get the basics

  but certainly not an exhaustive exploration!
- explore some of the core concepts

  set you up to explore more on your own
- see "the Angular Way" up close and personal

  and avoid some of the mental minefields
- practice, practice, practice

Format

Lecture/lab format

- We will lecture for a bit
- You will code/debug/explore for a bit

  We will try to wander the room and help where possible
- Lather, rinse, repeat
- Turn in your evals and you are free!

Format

"The Rules"

- Pairing and interaction

  absolutely encouraged
- Googling/Binging/looking stuff up online

  absolutely encouraged
- Leaving the room during a "lab"

  absolutely encouraged (if necessary)
- Jumping ahead

  encouraged, but we can only help for "this" step

Format

"Requests"

- set the phone/volume/etc to vibrate

  basically, don't disturb the neighbors

- ask questions!

  if you're thinking it, so are other people

- ... but let's keep the focus to the here and now

  there's a lot to cover, so we need to keep focused

Lab App

What should we build today?

Lab App

Enter... NgJoke!

- an application to allow people to browse different jokes
  - **Chuck Norris jokes?**
  - **Dad jokes?**
  - **Good jokes? Bad jokes?**
  - **The world clearly needs a comprehensive repository**
- Basic master-detail CRUD application

  using an HTTP API for retrieval and storage

Resources

Online resources for this workshop include:

- Slides

  http://www.newardassociates/slides/Workshops/AngularJS.pdf

- Lab code (with branches at each step)

  https://github.com/Garvice/angular-labs

- Typescript reference/docs

  https://github.com/Microsoft/TypeScript
  https://github.com/Microsoft/TypeScript/blob/2.1/doc/TypeScript%20Language%20Specification.pdf

- AngularJS website documentation

  https://angular.io/docs/ts/latest

- AngularJS "cheat sheet"

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

Overview

TypeScript in a nutshell

Overview

TypeScript is Microsoft's entry into the field of JavaScript transpilers

- Statically typed superset of ECMAScript 2015
  - **Any ES code is legitimate TS code**
  - **Supports ES6 class-based OOP style**
  - **Entirely "friendly" to the ECMAScript ecosystem**
- Chosen language for Angular 2

Syntax

Starting with the core

Syntax

TypeScript is syntactically compatible with ECMAScript

- any legal ECMAScript code is legal TypeScript code
- you don't need to be an ECMAScript expert

... but it helps

Syntax

TypeScript is thus a C-family language

- curly braces denote code blocks
- (optional) semicolon terminator
- if/else/else if, switch, for, ...

all the usual suspects from a C-based language

Syntax

Variable declarations

- "var" declares a function-local variable
- "let" declares a block-local variable
- "const" declares a block-local value (immutable)
- prefer "let" and "const"

Hello, World

Local variables

```
var v_hello = "Hello";
let l_hello = "Hello";
const c_hello = "Hello";
```

Hello, World

Function scoping

```
// Function scope
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }
   return sum;
}
```

Syntax

Ambient declarations

- sometimes a variable comes from "out of nowhere"
  - **a la "document" or "window" in the browser**
  - **or "console" in the command-line**
- TypeScript allows for "ambient variable declarations"
  - **uses the "declare" keyword**
  - **uses "var" declaration by convention**

Basic Types

The atoms of the language

Basic Types

Supported TypeScript basic types

- any
- number, boolean, string
- void, null, undefined
- symbol

Basic Types

TypeScript does limited type inference

- local variable declarations
- function parameters

... but NOT a full-blown type-inferenced language

- compiler will require type annotations in a number of places
- rule of thumb: start without, add when needed or wanted

Basic Types

Basic Types

- Any: primeval root type, allows for unchecked types
- Number: numerics
- Boolean: true/false
- String: collections of 0-n characters
  - **fully Unicode-supported**
  - **single or double-quoted literals**
  - **'backtick' literals provide multi-line and ${}-style interpolation**

Basic Types

Basic types

```
let isDone: boolean = false;

let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

let color: string = "blue";
color = 'red';

let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`
```

Basic Types

Symbol type

- Symbol instances are unique names
- typically constructed based from strings
- typically used as keys for properties

  avoids accidental name-clashes across libraries/modules
- Symbol has a number of predefined Symbol instances

  hasInstance, iterator, match, replace, split, ...

Basic Types

Void type

- used to indicate lack of return
- technically, can instantiate a void instance

  not typically useful--can only receive null or undefined values

Basic Types

Null type

- the type of the "null" literal
- Null type is a subtype of all types (except Undefined)

  makes "null" a valid value for all primitive types
- type not directly referencable

Basic Types

Undefined type

- the type of the "undefined" literal
- Undefined type is a subtype of all types (except Null)
  - **makes "undefined" a valid value for all primitive types**
  - **language specifies this as the default uninitialized value**
- type not directly referencable

Basic Types

Types can be marked nullable or not-nullable

- "?" suffix on declaration adds nullability capability
- "!" suffix on usage requies non-nullability
- in 2.0, "null" and "undefined" become type names

makes it easy to declare nullable instances more explicitly w/unions

Basic Types

Basic types

```
// turn on --strictNullChecks
//let foo1: string = null; // Error!
let foo2: string | null = null; // OK

let strs: string[] | undefined;
let upperCased = strs!.map(s => s.toUpperCase());
  // Error! 'strs' is possibly undefined

let lowerCased = strs!.map(s => s.toLowerCase());
  // OK: 'strs' cannot be undefined
```

Basic Types

Type aliases

- "type" keyword can declare an alias for another type
- purely syntactic sugar

  meaning, this is just a name, not a new type

Basic Types

String literal types

- a curious form of string-based enumerations
- use "type" keyword and "|"-ed string literals

Basic Types

Basic types

```
type Music = "metal" | "punk" | "jazz" | "symphonic";
let m : Music = "metal"; // OK
//m = "rap"; // Error!
```

Simple Compound Types

Building molecules out of atoms

Simple Compound Types

Arrays: fixed-size collection of one particular type

- Syntax can either be
  - **traditional C-family "[]"**
  - **explicit genericized declaration of "Array<type>"**
- indexed access starting from 0

Simple Compound Types

Arrays and Tuples

```
let list1: number[] = [1, 2, 3];
let list2: Array<number> = [1, 2, 3];
  // These are functionally equivalent
```

Simple Compound Types

Union types

- similar to "union" types from C++/C

  done at the declaration; not a new type

- uses the vertical pipe character to denote "or"
- anything assignable to one of the union's members is accepted
- any property from any of the union's members are accessible

Simple Compound Types

Union types

```
let a : string | number = "a"; // OK
a = 10; // OK
//a = false; // Error!
```

Simple Compound Types

Intersection types

- represent values that simultaneously have multiple types

  a value of type "A & B" is a value that is both of type A and type B

- typically used for object types

  "string & number" effectively mutually-exclusive

- can be very useful for function signatures, however

Simple Compound Types

Intersection types

```
type F1 = (a: string, b: string) => void;
type F2 = (a: number, b: number) => void;

var f: F1 & F2 =
  (a: string | number, b: string | number) => { };
f("hello", "world");  // OK
f(1, 2);              // OK
//f(1, "test");         // Error!
```

Simple Compound Types

Tuples: fixed-size collection of any different types

- unnamed field names
- access using "indices" into fields
- tuples are structurally typed

Simple Compound Types

Arrays and Tuples

```
// Declare a tuple type
let tuple: [string, number];
tuple = ["hello", 10]; // OK
//x = [10, "hello"]; // Error!
console.log(tuple[0].substr(1)); // OK
//console.log(x[1].substr(1)); // Error! 'number' does not have 'substr'
```

Simple Compound Types

TypeScript supports "destructuring" declarations

- Locals can be declared as "parts" of a larger thing

  such as elements in a tuple or array
- doing so is essentially shorthand
- empty comma can ignore an element in the source
- ... ("spread") can capture the rest into one element

Simple Compound Types

Arrays and Tuples

```
let [first, , third] = list1; // from earlier
console.log(first); // "1"
console.log(third); // "3"

let [str, num] = tuple; // from earlier
console.log(str); // "hello"
console.log(num); // "10"

let numbers = [1, 2, 3, 4, 5];
let [head, ...tail] = numbers;
console.log(head); // "1"
console.log(tail); // "[2, 3, 4, 5]"
```

Simple Compound Types

Enumerated types

- represents bound set of possible values
- backed by numeric value

  usually starting at 0 and incrementing if not specified
- actual objects/types at runtime
- "const enum"s are compile-time computed for efficiency

Simple Compound Types

Enums

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}
let dir = Direction.Down;

const enum GoodBeverages {
  DietCoke,
  VanillaShake,
  SingleMaltScotch
}
let drink = GoodBeverages.DietCoke; // 0
```

Flow Control

Decision-making and repetition

Flow Control

Looping and decision-making within code

- looping: for, for-in, for-of, while, do-while
- decision-making: if
- branching: break, continue, return, throw
- guarded code: try/catch/finally

Flow Control

for

- C-style "looping" construct
- syntax
  - **initialization expression**
  - **conditional expression**
  - **step expression**

Flow Control

for-in

- "looping" construct
- operates on list of keys on object
- serves as a way to inspect properties on an object

Flow Control

for-of

- "looping" construct
- operates on list of values on object
- mainly interested in the values of an object

Flow Control

for, for-in and for-of

```
for (let i=0; i<10; i++) {
    console.log(i); // 0, 1, 2, 3, 4 ... 9
}

let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";

for (let pet in pets) {
   console.log(pet); // "species"
}
for (let pet of pets) {
    console.log(pet); // "Cat", "Dog", "Hamster"
}
```

Flow Control

if

- C-style boolean conditional construct
- condition must be of boolean type
- optional "else" clause

Flow Control

break

- terminate execution in current scope

continue

- begin execution of next loop

Flow Control

return

- terminate execution of current function
- yield a value to caller

throw

- construct exception object
- terminate execution of current scope
- look for "catch" blocks in callers

Flow Control

try, catch, finally

- try denotes a "guarded block"
- catch denotes a block entered when an exception appears inside a guarded block
- finally denotes a block always executed regardless of how the guarded block is exited

Functions

Doing stuff

Functions

Functions structure

- "function" keyword
- (optional) name
- (optional) parameter list
  - **parameters can have type declarations**
  - **parameters can have default values**
  - **parameters can be declared optional**
  - **final parameter can be declared a "rest" parameter**

Functions

Functions

```
function add1(lhs : number, rhs : number) : number {
  return lhs + rhs;
}
let add2 = function(lhs = 0, rhs = 0) {
  return lhs + rhs; // return type inferred
}
let add3 = function(lhs: number, ...others : number[]) {
  let total = lhs;
  for (let i of others) { total += i }
  return total;
}
function add4(lhs : number, rhs? : number) : number {
  if (rhs)
    return lhs + rhs;
  return lhs;
}
```

Functions

Lambda structure

- parameter list
  - **types optional**
  - **parentheses required for more than one parameter**
- "arrow" ("=>")
- lambda body
  - **single expression requires no braces and no return**
  - **multiple expression requires braces and explicit return**

Functions

Lambdas

```
let add5 = function(lhs: number, ...others : number[]) {
  return others.reduce( (prev,curr) => prev + curr);
}
let add6 = (lhs, rhs) => lhs + rhs;
let add7 = (lhs, ...r) => r.reduce ( (prev, curr) => prev, lhs);
```

Functions

Functions have a type signature

- parameter list
  - **types (required)**
  - **names are helpful, but not part of the signature**
- arrow separator ("=>")
- return type (required)
- compiler can often infer types from usage

Functions

Functions

```
let add1t : (lhs : number, rhs : number) => number = add1;
let add2t : (l : number, r : number) => number = add2;
let add3t : (lhs : number, ...rest : number[]) => number = add3;
let add4t : (l : number, r? : number) => number = add4;
let add5t : (lhs : number, ...rest : number[]) => number = add5;
```

Classes

Enter the object

Classes

Classes

- TypeScript brings ECMAScript 2015 class declarations forward
- syntactically identical

  but then we can add type descriptors
- functionally equivalent

  assuming the type-checking passes
- "static" methods (no static fields)

  instead, use instance prototype for static storage

Classes

class

```
class Point {
  x: number;
  y: number;
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  add(other : Point) : Point {
    this.x += other.x;
    this.y += other.y;
    return this;
  }
  get distance () {
    return Math.sqrt(
      (this.x * this.x) + (this.y * this.y));
  }
}
let pt = new Point(5,12);
console.log(pt.distance); // 13
```

Classes

NOTE: This is NOT C++/Java/C#-style objects

- instances hold a runtime reference to the "type"
- "type" is better described as "type object"
- member lookup follows the prototype lookup chain

Classes

Prototype chain

```
let origin = new Point(0,0);
console.log(origin.toString());

console.log("Consulting origin...");
for (let member in origin) {
  console.log(member,"=",origin[member]);
}

console.log("Consulting origin prototype...");
let originP = Object.getPrototypeOf(origin);
for (let member of Object.getOwnPropertyNames(originP)) {
  console.log(member,"=",originP[member]);
}
```

Classes

Inheritance

- "B extends A"
- use super() to reference base constructor
- use super.method() to reference base methods

Classes

Inheritance

```
class ThreeDPoint extends Point {
  z: number;
  constructor(x, y, z) {
    super(x,y);
    this.z = z;
  }
  get distance () {
    let dist = super.distance;
    return Math.sqrt(
      (dist * dist) + (this.z * this.z));
  }
}

let p : Point = new ThreeDPoint(1, 1, 1);
console.log(p.distance);
```

Classes

Access control

- members can be marked
  - **public: accessible anywhere**
  - **private: accessible only within this class**
  - **protected: accessible only within this class and subclasses**
- access is checked at TypeScript-compile-time only

  no runtime enforcement

Classes

Polymorphic this

- "this" normally refers to the nominal type in which it is used
- as a return type, "this" refers to the type when it is used

  allows for late-bound typing to this
- extremely useful in fluent interfaces

Type Compatibility

Polymorphic this

Type Compatibility

Polymorphic this

Interfaces

Type-verified contracts

Interfaces

Interfaces

- declarations without implementation
- essentially a "contract" or "promise" of behavior
- works extremely well with TS's structural subtyping

Interfaces

Interfaces

```
interface LabelledValue {
    label: string;
}
function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}
let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Interfaces

Optional properties in Interfaces

- properties type-suffixed with "?" are optional
- optional properties do not need to be present
- test for presence with "if" test

Interfaces

Optional properties

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig) {
    let newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({color: "black"});
```

Interfaces

Interfaces can describe function types

- sometimes makes more readable types
- good to use instead of type aliases

Interfaces

Function interface

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}

let mySearch : SearchFunc;
mySearch = function(src: string, sub: string): boolean {
  return true; // search in O(1) time
}
```

Interfaces

Classes can "implement" interfaces

- all interface-declared members must be on class
  - **any missing members will be caught by compiler**
  - **... or class must not be instantiated**

Interfaces

Interfaces can describe indexable types

- meaning, we can use "[]" to access the type
- parameter can be either string or number

return type of number-indexer must be same or subtype of string-indexer

Lab: Domain Model

Building out our domain model (15 mins)

Lab: Domain Model

Step: Setup

- make sure everything is installed

  node --version
  npm --version

- install TypeScript on your machine

  npm install -g typescript
  you must be online for this command to work!

- create a new project (npm init) in a new directory

  don't worry too much about the details; we will be overwriting this with new values very soon

- create a "tsconfig.json"

  using content in next slide
  this will allow you to just "tsc" and run the results from the "out" directory

Lab: Domain Model

tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es2015",
        "strictNullChecks": true,
        "noImplicitAny": false,
        "outDir": "out",
        "sourceMap": false
    },
    "exclude": [
        "node_modules"
    ]
}
```

(Feel free to change the value of "outDir" to "." to run the code in the same directory as your .ts files)

Lab: Domain Model

Step: Vote

- create a new domain model class, Vote

  this will hold a "vote count" for a given Joke

- create vote.ts
- create class Vote
  - **constructor should take a number parameter**
  - **voteCount() should return that value**
  - **increment() method adds one to the value**
  - **decrement() method subtracts one**
- export this class

Lab: Domain Model

Step: Joke

- create a new domain model class, Joke

  this will be our core domain model

- call this file "joke.ts"
- create exported class Joke
  - **private "setup" of type string**
  - **private "punchline" of type string**
  - **private nullable field "lols" of type Vote**
  - **private nullable field "groans" of type Vote**
  - **private nullable field "source" of type string**

Lab: Domain Model

Step: Joke (methods)

- add methods to retrieve the groan count...

  groanCount

- ... upvote a groan

  addGroan

- ... and likewise for LOLs

  - **lolCount**
  - **addLol**

Lab: Domain Model

Step: Console app

- create a console app "app.ts"
- import Joke
- create a new Joke

  setup and punchline are up to you
- use console.log() to print the Joke to the command-line

  you can either print each value, or override Joke's toString() method
- transpile and run

  "tsc", then "node app"

Lab: Domain Model

Step: Joke ("database")

- add static const Joke instance called JOKE

  create a new Joke here
- refactor console app to print the static JOKE instance
- transpile and run

  "tsc", then "node app"

AngularJS Concepts

The Big Picture

Concepts

AngularJS defines some core building blocks

- Modules
- Components
- Templates
- Metadata
- Data binding
- Services
- Directives
- Dependency Injection

Concepts

Modules

- Angular apps are modular in design
  - **every app consists of at least one module**

    the "root module", normally called AppModule

Concepts

Components

An Angular class responsible for exposing data to a view and handling most of the view's display and user-interaction logic.

Concepts

Templates

A template is a chunk of HTML that Angular uses to render a view with the support and continuing guidance of an Angular directive, most notably a component.

Concepts

Metadata

- also known as "decorators"
- a mechanism for describing the "surface area" of a component or module

Concepts

Data binding

Applications display data values to a user and respond to user actions (clicks, touches, keystrokes). ... use data binding by declaring the relationship between an HTML widget and data source and let the framework handle the details.

Concepts

Services

For data or logic that is not associated with a specific view or that you want to share across components, build services. ... A service is a class with a focused purpose. You often create a service to implement features that are independent from any specific view, provide shared data or logic across components, or encapsulate external interactions.

Concepts

Directives

An Angular class responsible for creating, reshaping, and interacting with HTML elements in the browser DOM

Concepts

Dependency Injection

Dependency injection is both a design pattern and a mechanism for creating and delivering parts of an application to other parts of an application that request them. ... Angular developers prefer to build applications by defining many simple parts that each do one thing well and then wiring them together at runtime.

Concepts

Resources

- AngularJS Architecture

  https://angular.io/docs/ts/latest/guide/architecture.html
- AngularJS Glossary

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

AngularJS (v2)

Getting Started

Getting Started

To use AngularJS, you must have...

- NodeJS
- TypeScript

Installing these is usually pretty straightforward

- Install NodeJS (Homebrew, Chocolatey, apt-get, etc)
- npm install -g typescript

Getting Started

To use AngularJS, you must...

- Install the Angular CLI

  npm install -g @angular/cli

Lab: Scaffolding and Install Verification

Making sure all the infrastructure works (15 mins)

Lab: Scaffolding

Step: Setup

- install Angular-CLI

  npm install -g @angular/cli

- use the CLI to create the app in a new directory

  ng new jokeapp
  (this will create a new subdirectory, jokeapp)
  you must be online for this command to work!

- run tests

  ng test (or) npm test

- (optional) package the app for distribution

  ng build (or) npm run build

- run the app

  ng serve (or) npm start

- browse

  (browser window should automatically open)

Lab: Scaffolding

Step: Examination

- look around the root directory

  this is mostly all project plumbing, and not something we need to worry about

- drop into the src directory
- look at index.html

  marvel at all the stuff that we will never touch

- look at assets

  this is where statically-served files would go

Lab: Scaffolding

Step: Examination

- drop into the app directory

  this is where all of our code lives

- look at app.module.ts

  this is the "AppModule" and root component of the app

- look at app.component.*

  this is the first "viewable" component, AppComponent

Lab: Scaffolding

Step: Modify

- open up app.component.ts
- change constructor title assignment to "Hello, NG!"
- save file; app will reload itself
- the root directory is initialized as a Git repo

feel free to commit your changes!

Component-Oriented Programming

The Big Picture

Component-Oriented Programming

Components

> "Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system."
>
> **"Component Software", by Szyperski**

Component-Oriented Programming

Component environments (history)

- Microsoft COM/ActiveX/COM+
- EJB
- CORBA
- Delphi VCL
- VBX

Component-Oriented Programming

Properties of a component:

- it is a unit of independent deployment

  it encapsulates its consituent features

- it is a unit of third-party composition

  needs to encapsulate its implementation and interact with its environment by means of well-defined interface(s)

- has no (externally) observable state

  the component instance cannot be distinguished from other instances

Component-Oriented Programming

Properties of an object:

- it is a unit of instantiation

  it has a unique identity (usually by memory location)
- may have state and this can be externally observable

- encapsulates its state and behavior

Component-Oriented Programming

Components != objects

- components will often be made up of objects
- however, nothing in COP insists upon OOP
  - **procedural code can be structured as components**
  - **as could functional code**

Component-Oriented Programming

Components

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

"Component Software", by Szyperski

Modules

The Angular Way to build a cohesive block of code dedicated to a single purpose

Modules

Angular breaks everything into modules

- modules form the major component boundaries in Angular
  - **these are ES6 modules**
  - **loaded via module loaders (SystemJS, Webpack, etc)**
- modules provide definitions
  - **modules export types, functions, etc**
  - **other modules import the desired bits**
- Angular modules are "just modules"
  - **core Angular bits imported via scoped packages**
  - **@angular/core, @angular/common, @angular/forms, @angular/http**

Modules

Angular modules == ES6 modules

- use "export" to publicize classes
- using "import" brings module into scope
- collect all component-related modules into one module
  - **call this file "index.ts"**
  - **(re-)export the declared elements from the individual files**
  - **this is a "barrel" module**

Modules

Angular modules: a sampling

- App component (usually called AppModule)

  represents the "root component" of the app

- Directives

  represents a chunk of functionality

- Components

  a directive plus a view

- Pipes

  a tool to transform input in some particular manner

- Barrel

  a means to "roll up" declarations into one convenient syntax

Model/View/Controller

Separating display from logic

Model/View/Controller

MVC is a design pattern that appears frequently in GUI systems

- Model represents the state
- Controller represents the logic
- View represents the display/UI

Model/View/Controller

Origins lie in Smalltalk

- Models were objects
- Controllers were objects
- Views were objects
- Today, this is an idiom known as "Naked Objects"

Model/View/Controller

MVC showed up in a number of GUI systems

- Windows UI frameworks called it "Document/View"
- Java Swing built around MVC quite deeply
  - **tables, lists, all used models extensively**
  - **even the actual UI code was split along view/controller lines; AbstractButton defined basic button behavior, and deferred to another class to do the actual pixel-drawing work**
- iOS uses it (ViewControllers are controllers)

Model/View/Controller

In the original MVC...

- Models were often connected to more than one View
- Controllers coordinated view updates as model changed
- Models, Views and Controllers relatively loosely coupled
- Web adaptation of MVC changed this to a 1:1 View/Controller

Web views and controllers never reused

Model/View/Controller

Parting thoughts

- Models are not necessarily Domain Objects
  - **community is split over this one**
- Views should not contain "logic"
  - **except for logic that is UI-focused**
  - **community is split over this one too**

Model/View/Controller

MVC has spawned a lot of similar ideas/patterns

- Hierarchical model-view-controller
- Model-view-adapter
- Model-view-presenter
- Model View ViewModel
- Observer
- Presentation-abstraction-control
- Three- or n-tier architecture

Model/View/Controller

Resources

- For more details, see

  http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- Observer pattern comes from "Design Patterns" (GOF book)

  Gamma, Helm, Johnson, Vlissides
- Presentation-abstraction-control comes from "POSA 1"

  Buschmann, Meuneir, Rohnert, Sommerlad, Stal

Components

The Angular Way to build a class responsible for exposing data to a view and handling most of the view's display and user-interaction logic

Components

Component is a class and HTML view

- component should represent a logical atom of UI or work
- class provides component storage and logic
- HTML provides view UI/UX
- class uses metadata to provide additional information

  @Component decorator

Components

Empty component

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class TitleComponent { name = 'Angular'; }
```

Components

@Component decorator

- selector: the tag used in the HTML
- template: the HTML for the view
- templateUrl: the file containing the HTML view

  only one of template/templateUrl is used
- ... more

  https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html

Components

Lifecycle methods

- callbacks from Angular to inform component of events

  methods can be implemented via TS interfaces
- ngOnInit/ngOnDestroy

  prefer initialization in OnInit
- ngOnChanges: fires on any property change
- ngDoCheck: detect and act upon changes that Angular doesn't catch on its own
- ngAfterViewInit, ngAfterViewChecked

  called after view is initialized and checked
- ngAfterContentInit, ngAfterContentChecked

Metadata

The Angular Way to use ES6 decorators to provide data about types

Metadata

Angular makes heavy use of decorators to provide component metadata

- these provide additional information to the Angular environment

  never used by the developer/user

- decorators are always ES6 functions

  never forget the () when using them

Metadata

List of commonly-used decorators

- NgModule
- Component

Metadata

Metadata can also be applied to fields

- @Input(): property that can be bound from the view
- @Output(): event that can be bound from the view

Lab: Components

Building out our first component (30 mins)

Lab: Components

Step: Modify AppComponent

- app.component.ts: Change title

  set title to "NgJokes"

- app.component.html: Add "app-joke" tags

  starting and ending: "<app-joke></app-joke>"

Lab: Components

Step: Generate JokeComponent

- use Angular CLI to generate "JokeComponent"

  ng generate component Joke
  (adding "-d" will do a dry run, making no file changes)

Lab: Components

Step: Migrate and Import

- copy Joke and Vote code over to "src/app/joke" directory

  call them "joke.model.ts" and "vote.model.ts" if you like

- joke.component.ts:

  import Joke

Lab: Components

Step: Add Joke to the JokeComponent

- joke.component.ts:
  - **store a Joke as a field in the JokeComponent ("joke")**
  - **assign to "joke" the static Joke instance**
- joke.component.html:
  - **create two "div" regions**
  - **inside of the first, put "{{joke.setup}}"**
  - **inside of the second, put "{{joke.punchline}}"**

Lab: Components

Step: Check

- app.module.ts: Ensure JokeComponent is referenced
  - **imports**
  - **AppModule declarations array**

Templates

Laying out the viewable parts

Templates

View syntax for Angular

- templates can appear in metadata or standalone files
  - **use "template" for inline metadata declarations**
  - **use "templateUrl" for standalone files**
- contents contain "raw" HTML plus Angular expressions
  - **some HTML elements won't make sense in a template**
  - **some non-HTML constructs will appear (components)**
  - **expressions evaluated in a variety of ways**
- note that syntax is a little unusual in places

  more on this later

Templates

View syntax: Data binding

- three forms
  - **data source -> view target**
  - **view target -> data source**
  - **two-way/bidirectional**
- each has some different syntax

Templates

View syntax: Source-to-View data binding

- {{ }} (interpolation) syntax (data)
- [] property binding syntax (targets)

  this will re-evaluate as the value changes
- note that either syntax can often be used (for strings)
- note that for non-strings, property binding syntax is required

Templates

View syntax: Interpolation

- {{ }} surrounds a valid template expression

  typically a component or model property

- evaluated and result dropped in place

Templates

Template expressions

- expressions produce a value
- expressions have some restrictions (no side effects)
  - **no assignments (= += -= etc)**
  - **no "new"**
  - **no chained expressions (single-expressions only)**
  - **no increment/decrement operations (++ --)**
- expressions are limited in scope

  to the component instance and/or template's context
- keep these simple!

Templates

Template statements

- statements are not expressions
  - **do not have to produce a value**
  - **may produce side effects (generally desirable, in fact)**
- supports expressions, plus...
  - **assignment**
  - **chaining expressions (use of ; or ,)**
- still not allowed:
  - **no "new"**
  - **no increment/decrement**
  - **no operator assignment (+= -=)**
  - **no bitwise operators (| &)**
- keep these simple!

Templates

Tangent: HTML attribute vs DOM property

- attributes are defined by HTML; properties defined by the DOM
- attributes initialize DOM properties once
- property values change over time; attributes don't
- attributes != properties, even when they have the same name

Templates

View syntax: Property binding

- take the result of an expression, and bind it to a property
- element property, component property, directive property
- square-brackets used to capture the target
- one-way binding ("binding in")

Templates

View syntax: Event binding

- take an event source (target event), invoke a template statement
- event source must be surrounded in round-brackets
- one-way binding ("binding out")
- requires an EventEmitter<T> in the component
- invoking emit() passes the event out to any bound code
  - **parameter to emit() is passed to the target**
  - **parameter type is the genericized "T" parameter**

Templates

View syntax: Bidirectional binding

- uses both square- and round- brackets to surround the target
- establishes a round-trip cycle if certain conventions are followed
  - **property "x"**
  - **eventEmitter "xChange"**
- syntactic sugar equivalence:
  - **<my-sizer [(size)]="fontSizePx"></my-sizer>**
  - **<my-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></my-sizer>**

Lab: Data binding

Exploring data binding options (15 mins)

Lab: Data binding

Step: JokeComponent refactoring

- joke.component.ts:
  - **add four fields, one for each of setup, punchline, lols and groans**
  - **in constructor, assign to fields the corresponding values from Joke**
- joke.component.html:
  - **add two more "divs", one for each of LOLs and groans**
  - **each "div" should use a label for the data**
  - **each "div" should also use an input (type="number")**
  - **use two-way data binding on ngModel for the lols and groans fields**

Lab: Cross-component communication

Passing data between components (30 mins)

Lab: Cross-component communication

Step: Generate UpvoteComponent

- ng generate component Upvote

Lab: Cross-component communication

Step: Create view

- upvote.component.ts:

  add field "votes" of type number

- upvote.component.html:

  add a "span" with string-interpolated "votes"

Lab: Cross-component communication

Step: Accept input

- upvote.component.ts:

  decorate "votes" with @Input decorator

- joke.component.html:

  •inside the "lols" label, replace the two-way binding with "app-upvote"

  •use property binding to bind the Upvote's "votes" to the Joke's "lols"

Lab: Cross-component communication

Step: Wire up Upvote event binding

- upvote.component.ts:

  add an "incrementVoteCount" method to bump vote count

- upvote.component.html:

  •add a "span" that contains an "up-arrow"
  •▲
  •set the span's "click" to invoke "incrementVoteCount" as an event-binding

Lab: Cross-component communication

Step: Wire up Upvote-to-Joke event binding

- upvote.component.ts:
  - **add a field "onVoteIncremented" of type EventEmitter**
  - **in incrementVoteCount, call onVoteIncremented.emit()**
- joke.component.ts:

  add methods "incrementGroan" and "incrementLol" to increment vote counts in Joke instance
- joke.component.html:

  add event-binding for "onVoteIncremented" in app-vote to call incrementLol() or incrementGroan()

Angular Directives

Building Angular syntax

Directives

Directives are modules that perform actions

- actions are usually not visual

  directive + visual (usually a template view) = component

- custom components can interact with components

Directives

A "spy" directive

```
// Spy on any element to which it is applied.
// Usage: <div mySpy>...</div>
@Directive({selector: '[mySpy]'})
export class SpyDirective implements OnInit, OnDestroy {
  constructor(private logger: LoggerService) { }
  ngOnInit()    { this.logIt(`onInit`); }
  ngOnDestroy() { this.logIt(`onDestroy`); }
  private logIt(msg: string) {
    this.logger.log(`Spy #${nextId++} ${msg}`);
  }
}
```

Directives

View syntax: Built-in structural directives

- NgIf
  - **boolean decision-making**
  - **example: "<section *ngIf="showSection">"**
- NgFor
  - **iteration through collection**
  - **example: "<li *ngFor="let item of list">"**
- NgSwitch
  - **multi-branch decision-making**
  - **NgSwitchCase**
  - **NgSwitchDefault**

Directives

View syntax: Built-in attribute directives

- NgClass: add and remove a set of CSS classes
- NgStyle: add and remove a set of HTML styles
- NgModel: two-way data binding to an HTML form element

Lab: Master list

Displaying a list of Jokes (30 mins)

Lab: Master list

Step: Generate the JokeListComponent

- ng generate component JokeList

Lab: Master list

Step: Create the list of Jokes to use

- joke.model.ts:

  add a static array of Jokes, call it JOKE_DB

Lab: Master list

Step: Build the JokeList view

- joke-list.component.ts:

  add a local field ("jokes") that holds Joke.JOKE_DB

- joke-list.component.html:

  •use "ngFor" to iterate through the array of jokes
  •use "app-joke" and bind the current joke to the "joke" in JokeComponent
  •(this will require marking the "joke" field in JokeComponent with the Input() decorator, by the way)

Lab: Master list

Step: Use the JokeList

- app.component.html:

  change the body of the app to "<app-joke-list"

Lab: Master list

Reminder: Import all necessary types as you need them

Router

The Angular Way of configuring and managing the entire view navigation process

Router

Routing is the "rearrangement" of the view

- for master-detail scenarios...
  - **one view is the master list of data elements**
  - **another is the detailed view of an individual elements**
- these usually want to be matched up against URL patterns

  which standard components wouldn't provide
- ... and we don't want to write standalone HTML files

  loss of scope, additional round trips, etc

Router

Enter @angular/router

- module to provide routing/navigation support
- entirely optional

Router

Routing consists of several key parts:

- Routes configuration (typically in AppModule)
  - **establishing URL patterns (paths) and components**
  - **additional elements (data to be passed, etc)**
- NgModule configuration

  pass to "imports" metadata parameter
- "router-outlet" placeholder for view
- "router-link" connects to target route path
- route parameters can be consumed in target components

Router

Example Routes configuration

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Router

Example routing template (nav bar)

```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis
Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```

Router

Routes are taken on first-match basis

```
{ path: '**', component: PageNotFoundComponent }
```

PageNotFoundComponent

```
import { Component } from '@angular/core';
@Component({
  template: '<h2>Page not found</h2>'
})
export class PageNotFoundComponent {}
```

Router

Routes also can redirect

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes',        component: HeroListComponent },
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

Lab: Routing

Putting master/detail routes into URLs

Lab: Routing

Step: Add routing module

- app.module.ts:
  - **import RouterModule and Routes from "@angular/router"**
  - **define a Routes array (usually called "appRoutes")**
  - **add a path/component pair for 'joke-list'/JokeListComponent**
  - **add a path/component pair for 'joke'/JokeComponent**
  - **add RouterModule.forRoot() to imports decorator**
- app.component.html:
  - **add "router-link" to "h1", pointing to "/joke-list"**
  - **replace app-joke-list with router-outlet**

Lab: Routing

Step: Add routing module

- joke-list.component.ts:
  - **take a Router as a constructor parameter**
  - **create a "showJoke()" method taking a Joke parameter**
  - **in showJoke(), instantiate a NavigationExtras**
  - **set queryParams to include params from the Joke**
  - **call the Router's navigate() method to 'joke', with the NavigationExtras**

Services

The Angular Way of providing data or logic that is not associated with a specific view

Services

Services in Angular provide pure logic

- typically around data access of one form or another
- essentially, functional behavior

Services

Services are "dependency-injected"

- Angular will pass the service in to the component

this makes services easier to mock for testing purposes

Services

Service implementation

- standard TS class
- decorated with @Injectable metadata
- beyond that, "just" a class; provide methods, fields, etc as normal
- typically services will want to use Promises or RxJS

Services

Defining a service

```
import { Injectable } from '@angular/core';
import { Hero } from './hero';
import { HEROES } from './mock-heroes'; // const array
@Injectable()
export class HeroService {
  getHeroes(): Promise<Hero[]> {
    return Promise.resolve(HEROES);
  }
}
```

Services

Service usage

- component declares the dependency via two mechanisms:
  - **constructor params**
  - **providers metadata parameter**
- Angular will see the providers declaration and match it against the appropriate parameter in the constructor
- note that services will not be available prior to ngOnInit()

Services

Using a service

```
import { Component, OnInit } from '@angular/core';
import { Hero } from './hero';
import { HeroService } from './hero.service';
@Component({
  selector: 'my-app', template: `...`,
  providers: [HeroService]
})
export class AppComponent implements OnInit {
  heroes: Hero[];
  constructor(private heroService: HeroService) { }
  getHeroes(): void {
    this.heroService.getHeroes().then(heroes => this.heroes = heroes);
  }
  ngOnInit(): void {
    this.getHeroes();
  }
}
```

Summary

Wrapping up (for now)

Summary

AngularJS is...

- a highly-opinionated framework
- a highly-productive framework

  once you agree to the "Angular Way" of doing things
- growing in popularity
- a very reasonable choice for your web v.Next applications