The Angular Workshop

Tiberiu ‘Tibi’ Covaci

whoami

- Tiberiu ’Tibi’ Covaci
- Software engineer, > 25 years experience
- 14 years as Trainer, teaching Web technologies, and .NET
- Developer, Trainer, Mentor
- Microsoft Regional Director
- Father & Geek
- @tibor19

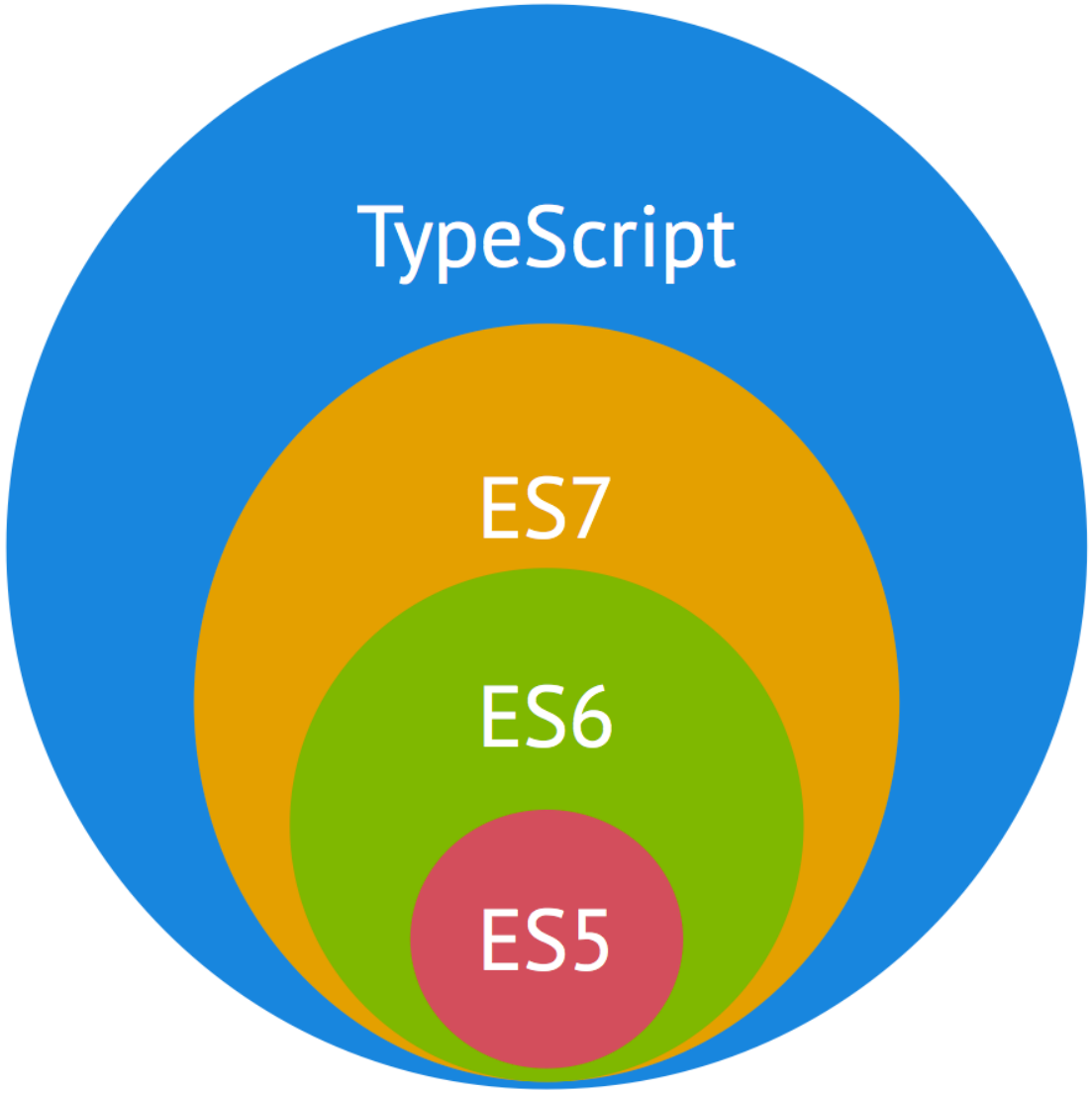
Agenda

- The History Lesson
- The Quick Start Guide
- Modules
- Components
- Services
- Out of the box stuff

The History Lesson

- First appeared in 2009
- A nice way to augment HTML
- Angular 2 announced in September 2014

TypeScript Overview



TypeScript Overview

- Basic Types
- Variable Declarations
- Interfaces
- Classes
- Functions
- Generics
- Enums
- Type Inference
- Type Compatibility
- Advanced Types
- Symbols
- Iterators and Generators
- Modules
- Namespaces
- Namespaces and Modules
- Module Resolution
- Declaration Merging
- Writing Declaration Files
- JSX
- Decorators
- Mixins
- Triple-Slash Directives

Basic Types

- Boolean
- Number
- String
- Array
- Tuple
- Enum
- Any
- Void
- Type assertions

Boolean & Number

```
let isDone: boolean = false;
```

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

Strings

```
let fullName: string = `Hello World!`;
let age: number = 27;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old in December.`
```

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old in December."
```

Array

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

Tuple

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

```
x[3] = "world"; // OK, 'string' can be assigned to 'string | number
'

console.log(x[5].toString()); // OK, 'string' and 'number' both hav
e 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'
```

Enum

```
enum Color {Red = 1, Green, Blue};
let colorName: string = Color[2];

alert(colorName);
```

Any

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn'
t check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on
 type 'Object'.
```

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Void

```
function warnUser(): void {
    alert("This is my warning message");
}
```

Type assertions

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

Variable Declarations

- var
- let
- const

Destructuring

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

```
first = input[0];
second = input[1];
```

Destructuring

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

```
first = input[0];
second = input[1];
```

Destructuring

```
// swap variables
[first, second] = [second, first];
```

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
f(input);
```

Destructuring

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

```
let [, second, , fourth] = [1, 2, 3, 4];
```

Destructuring

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
}
let {a, b} = o;
```

```
({a, b} = {a: "baz", b: 101});
```

Destructuring

```
let {a: newName1, b: newName2} = o;
```

```
let newName1 = o.a;
let newName2 = o.b;
```

Destructuring

```
function f({a, b = 0} = {a: ""}): void {
    // ...
}
f({a: "yes"}) // ok, default b = 0
f() // ok, default to {a: ""}, which then defaults b = 0
f({}) // error, 'a' is required if you supply an argument
```

Interfaces

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Interfaces

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

function createSquare(config: SquareConfig): {color: string; area: number} {
    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

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

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    if (result == -1) {
        return false;
    }
    else {
        return true;
    }
}
```

Classes

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter: Greeter;
greeter = new Greeter("world");
console.log(greeter.greet());
```

Functions

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x+y; };
```

Fat Arrow Functions

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // Notice: the line below is now a lambda, allowing us to capture 'this' early
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Type Inference

- Best common type
- Contextual Type

Type Compatibility

```
interface Named {
    name: string;
}

let x: Named;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Wonderland" };
x = y;
```

Advanced Types

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left side.
 */
function padLeft(value: string, padding: string | number) {
    // ...
}

let indentedString = padLeft("Hello world", true); // errors during compilation
```

Decorators

```
class Point {
    private _x: number;
    private _y: number;
    constructor(x: number, y: number) {
        this._x = x;
        this._y = y;
    }

    @configurable(false)
    get x() { return this._x; }

    @configurable(false)
    get y() { return this._y; }
}
```

Decorators

- A way to annotate your code
- A way to add metadata to your code
- They can be attached to
  - Classes
  - Methods
  - Accessors
  - Properties
  - Parameters
- It uses the @...() notation
  - It is just a function

Decorators

```
@decorator()
Class MyClass(){}

function decorator(target){
    // Do something to the target here
}
```

Modules

- EcmaScript 2015
  - A new way to scope your code
- TypeScript
  - Same as JavaScript which is effectively a scope.
- AngularJS/Angular
  - A way organize an application into cohesive blocks of functionality.

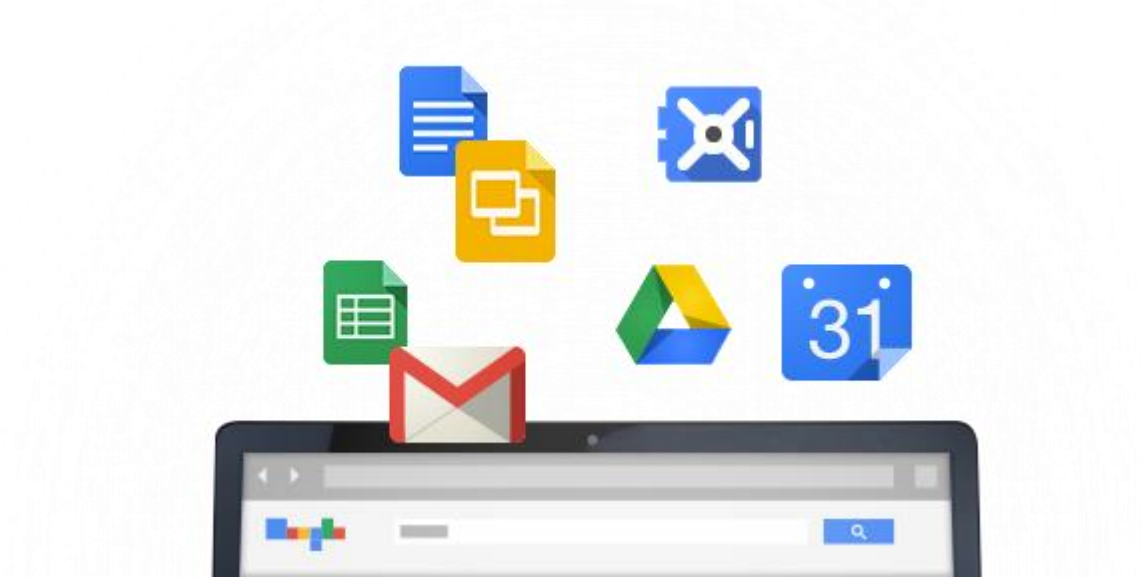
Modules

```
@NgModule {
    imports: [],
    providers: [],
    declarations: [],
    exports: [],
    bootstrap: []
}
```

Quick introduction to Angular

History

- Started in 2009 @ Google
- Built from experience with large web applications



ANGULARJS
by Google

Angular Goals

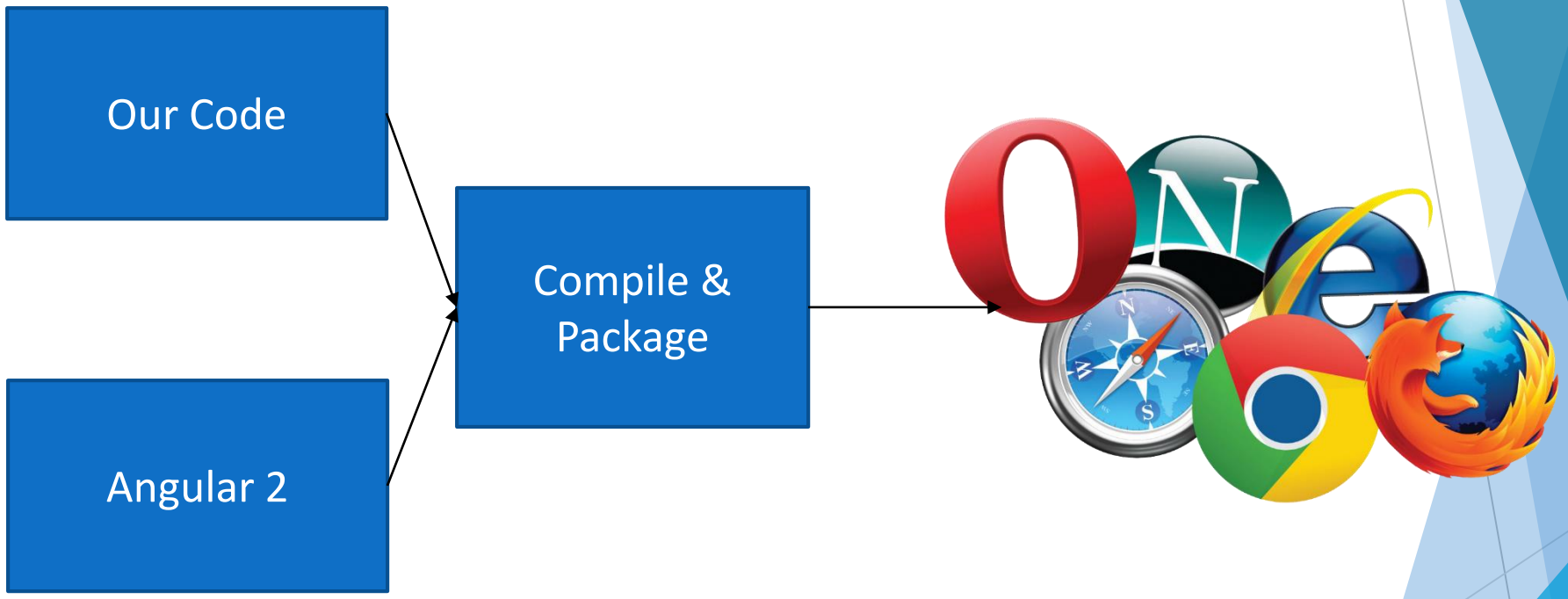
Separation of Concerns

Testability

Extensibility

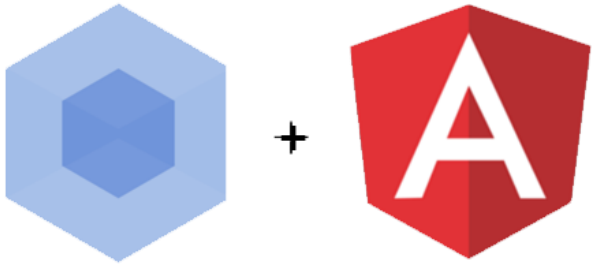
Forward Looking

Getting Off The Ground



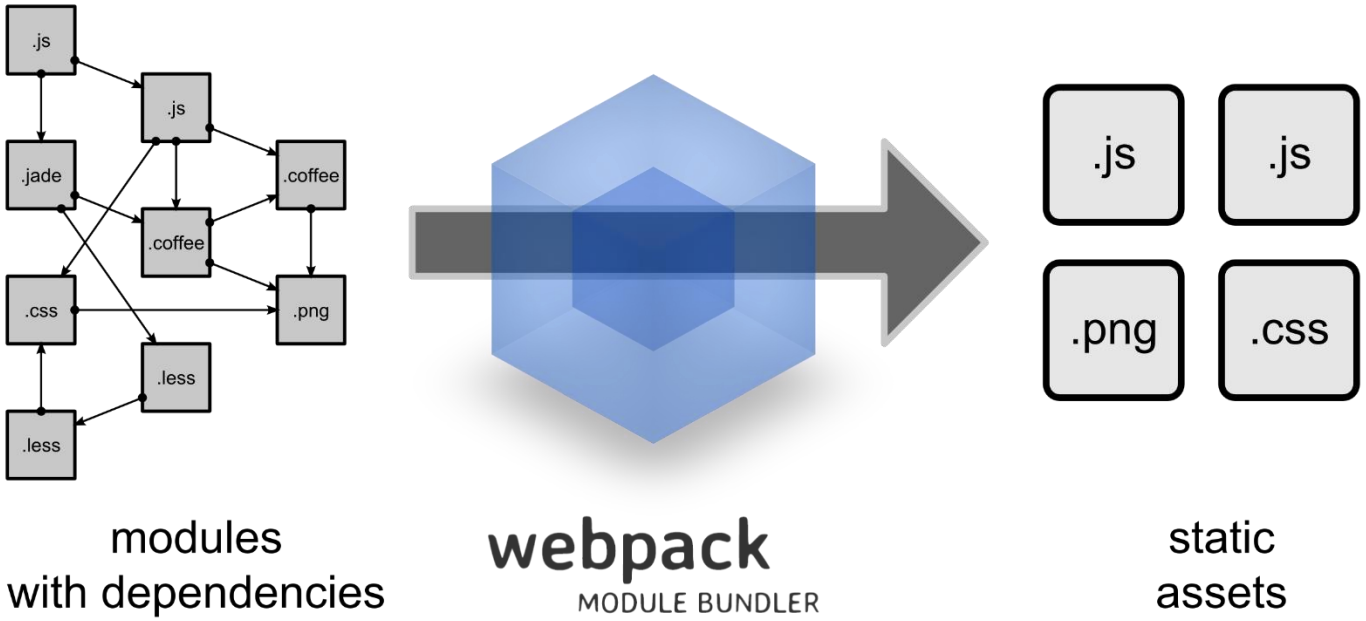
Getting Started Helpers

YEOMAN



Webpack and Angular 2
starter

What's Happening



Shell Page

```html
<html>
  <head>
    <meta charset="utf-8">
    <title>At The Movies</title>
    <link rel="icon" type="image/x-icon" href="/img/favicon.ico">
    <base href="/">
  </head>
  <body>
    <movie-app>Loading...</movie-app>
  </body>
</html>
```

Bootstrapping

- Initializes the Angular framework
  - Brings your application to life
  - You bootstrap a specific module

```
import {platformBrowserDynamic} from "@angular/platform-browser-dynamic";
import {MovieAppModule} from "./app/movieapp.module";

platformBrowserDynamic().bootstrapModule(MovieAppModule);
```

Modules

- Organize large applications into modules of functionality
  - Module object tells Angular how to execute module code

```
import {NgModule} from "@angular/core";
import {BrowserModule} from "@angular/platform-browser";
import {MovieAppComponent} from "./movieapp.component";

@NgModule({
    imports: [BrowserModule],
    declarations: [MovieAppComponent],
    bootstrap: [MovieAppComponent]
})
export class MovieAppModule {

}
```

Components

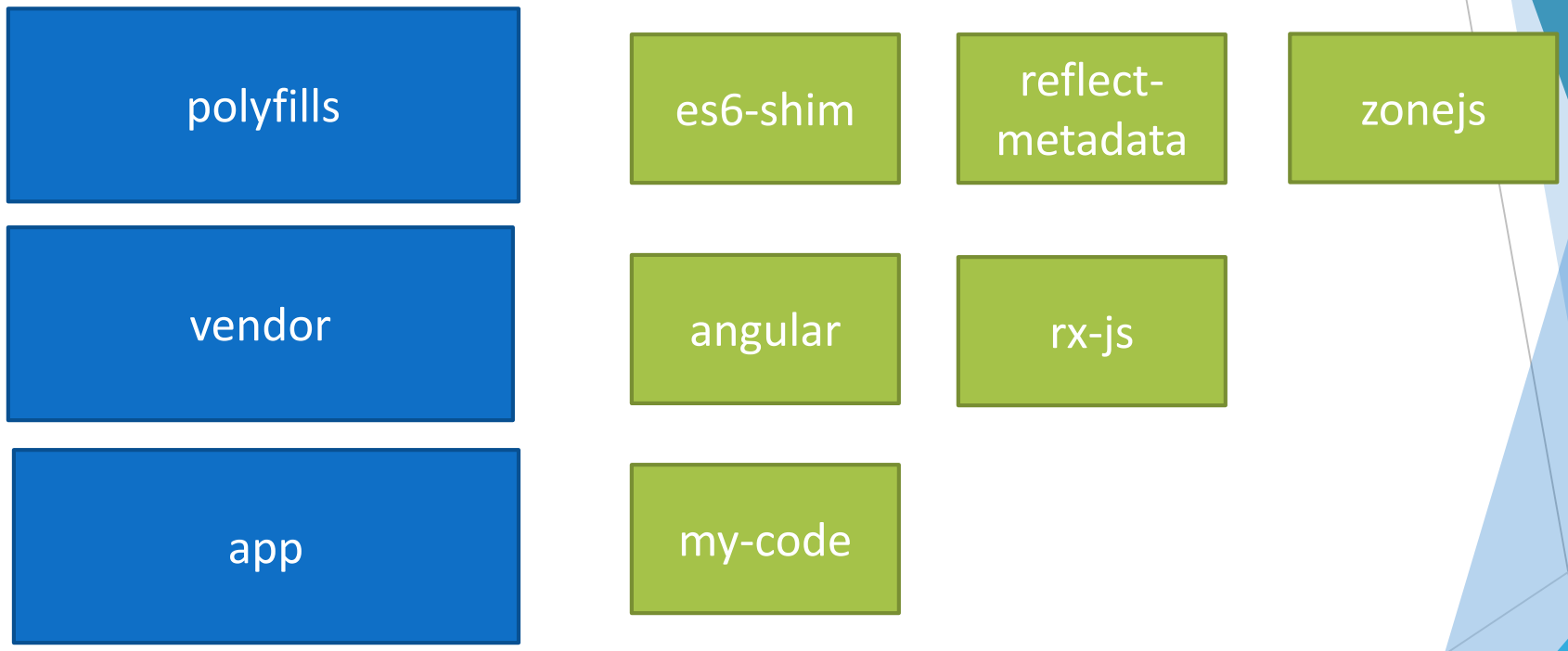
- Components define a custom HTML element
  - Data + behavior + template

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

@Component({
    selector: "movie-app",
    templateUrl: "./movieapp.component.html"
})
export class MovieAppComponent {

}
```

What's Being Loaded?



Directives

- Directives extend HTML
  - Add behavior or change the appearance of the DOM
  - Angular compiles markup to process directives
- You can build your own directives
  - In fact, this is encouraged

```
<div>

    <h1>{{title}}</h1>

    <div *ngIf="firstName">Hello, {{firstName}}</div>

</div>
```

Templates

- Responsible for presenting the model
  - Using directives and {{ interpolation }}

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

Models

- Plain Old JavaScript (2015)
- Component fields exposed for binding

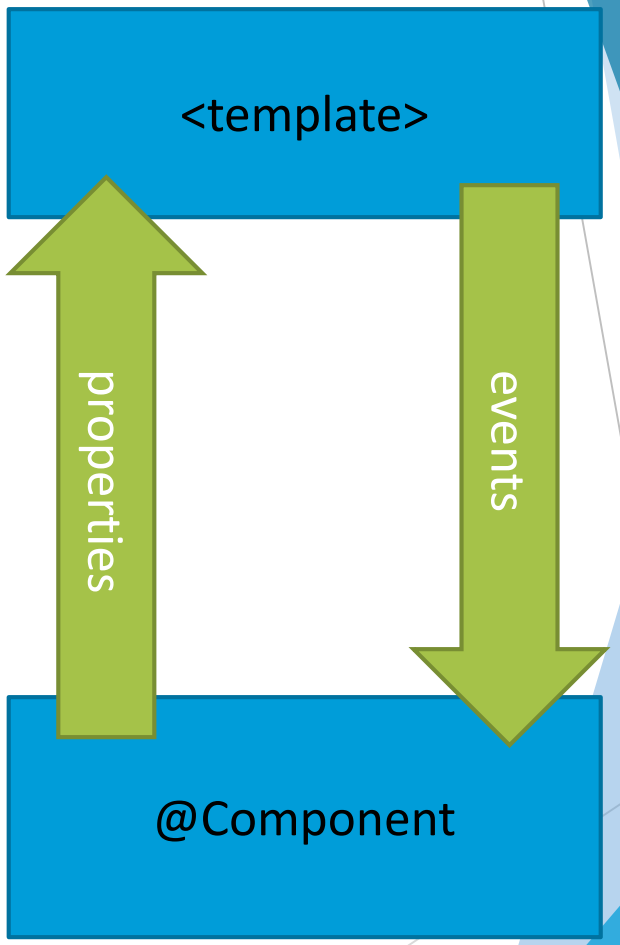
```
@Component({
    selector: "app",
    templateUrl: "/app/main.html"
})
export class App {

    title: string
    firstName: string
    movies: Array<Movie>

    constructor() {
        this.title = "ng2";
        this.firstName = "Scott";
        this.movies = [
            new Movie("Star Wars", 120, 1979),
            new Movie("Jurrasic Park", 130, 1992),
            new Movie("SP", 300, 2014)
        ];
    }
}
```

Essence

- Components
  - No direct DOM manipulation
- Templates
  - No serious model manipulation



Template Syntax

- Display text using {{expressions}}
- Set properties using [expressions]
- Handle events using (expressions)
  - [] () deal with properties, not attributes
  - Expressions limit side effects and global scope

```
<div [style.color]="color">
    Hello!
    <button (click)="changeColor()">Change color</button>
</div>
```

Forms

- Support for validation and dirty flags
- Use ngModel in combination with:
  - ngSubmit
  - ngControl
  - ngForm

```
<form (ngSubmit)="saveEdits()">

    ...

    <button type="submit">Save</button>
</form>
```

Angular versus Unobtrusive JavaScript

- Everyone is using JavaScript
- Angular behaves the same across browsers
- Expressions not evaluated in global scope

```
<form (ngSubmit)="saveEdits()">

    ...

    <button type="submit">Save</button>
</form>
```

Hiding and Showing

- Several approaches
  - Use *ngIf
  - Bind to the hidden property
  - Bind to the style.display property

```
<div *ngIf="showDiv">Show div</div>
<div [hidden]="!showDiv">Show div</div>
<div [style.display]="showDiv? '' : 'none'">Show div</div>
<button (click)="showDiv = !showDiv">Toggle</button>
```

Styles

- Use the ngClass directive
  - Adds class names for truthy values

```
<ul>
    <li *ngFor="let item of items" [ngClass]="item.style()">
        {{ item.text }}
    </li>
</ul>
```

Debugging

- Source maps let you step through TypeScript

system.src.js | angular2.dev.js | main.ts | movie.ts!transpiled | movie.ts ×

```
1  export class Movie {
2      constructor(title: string, length: number, year: number, rating: number) {
3          this.title = title;
4          this.length = length;
5          this.year = year;
6          this.rating = rating;
7      }
8
9      get isGood() {
10         return this.rating > 4;
11     }
12
13     get isBad() {
```

Components and Directives

<custom-html>

</custom-html>

Components

- Base building blocks in Angular, the Sequel
- They contain
  - A template
  - A class
  - Metadata
  - Styles
  - Dependencies
- They must belong to a NgModule

Components

```
@Component({
    selector: …,
    template/templateUrl: …,
    styles/styleUrls: [],
    providers: []
})
```

Components

- Reusable, composable UI building blocks
- A template is required

```
<div *ngFor="let movie of movies">
    <movie-panel [movie]="movie"></movie-panel>
</div>
```

```
@Component({
    selector: "movie-panel",
    templateUrl: "./movie-panel.component.html"
})
export class MoviePanelComponent {
    @Input() movie: Movie;
}
```

@Input

- Declares a data-bound input property

```
export class MovieDisplay  {

    @Input("theMovie") movie;

}
```

```
<movie-display *ngFor="#movie of movies" [theMovie]="movie">
</movie-display>
```

@Output

- An event-bound output property

```
export class MovieDisplay  {

    @Input() movie : Movie
    @Output() ratingChange : EventEmitter<number> = new EventEmitter();


    changeRating() {
        this.ratingChange.emit(this.movie.rating + 1);
    }

}
```

```
<movie-display *ngFor="#movie of movies"
               [movie]="movie"
               (ratingChange)="onRatingChange(movie, $event)">
</movie-display>
```

Components versus Directives

- Directives require no template
  - But, can work with templates
- Components are directives, but with a template

```
@Directive({
    selector: '[unless]'
})
export class UnlessDirective {
  constructor(
        private templateRef: TemplateRef<any>,
        private viewContainer: ViewContainerRef)
    {

    }
```

Lifecycle

- Methods ngOnInit and ngOnDestroy invoked by framework
  - Optional interfaces are OnInit, OnDestroyAlso …
- Also …
  - DoCheck, OnChanges, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked

```
export class MovieDisplay implements OnInit, OnDestroy, OnChanges {

    ngOnInit() {
        console.log("MovieDisplay initialized")
    }

    ngOnDestroy() {
        console.log("MovieDisplay destroyed")
    }

    ngOnChanges(changes) {
        console.dir(changes);
    }
}
```

ngOnInit Note

- Put work inside ngOnInit instead of constructor
  - Easier to test a component

```
ngOnInit() {
    this.movieData.getAll()
        .subscribe(movies => this.movies = movies,
                   error => console.log(error));
}
```

Decorator Directives

- Attributes to add behavior to a DOM element

```
import {Directive, Input, ElementRef, Renderer} from "angular2/core";

@Directive({
    selector: "[high-light]",
    host: {
        "(mouseenter)": "onMouseEnter()",
        "(mouseleave)": "onMouseLeave()"
    }
})
export class HighLight {

    @Input("high-light") color: string;

    constructor(private element: ElementRef, private renderer: Renderer) {

    }

    onMouseEnter() {
        this.renderer.setElementStyle(this.element, "background-color", this.color);
    }

    onMouseLeave() {
        this.renderer.setElementStyle(this.element, "background-color", null);
    }

}
```

Directives with Content

- Use ng-content to transclude content from client template

```
import { Component } from 'angular2/angular2';

@Component({
  selector: 'tab',
  template: `
    <div [hidden]="!active">
      <ng-content></ng-content>
    </div>
  `
})
export class Tab {
  title: string;
  active = this.active || false;
}
```

Directives and ViewContainers

- Manage templates using a view container

```
@Directive({selector: '[ngIf]', inputs: ['ngIf']})
export class NgIf {
  private _prevCondition: boolean = null;

  constructor(private _viewContainer: ViewContainerRef, private _templateRef: TemplateRef) {}

  set ngIf(newCondition /* boolean */) {
    if (newCondition && (isBlank(this._prevCondition) || !this._prevCondition)) {
      this._prevCondition = true;
      this._viewContainer.createEmbeddedView(this._templateRef);
    } else if (!newCondition && (isBlank(this._prevCondition) || this._prevCondition)) {
      this._prevCondition = false;
      this._viewContainer.clear();
    }
  }
}
```

Summary

- Angular encourages you to build components
  - Self describing UI building blocks
  - Extend HTML

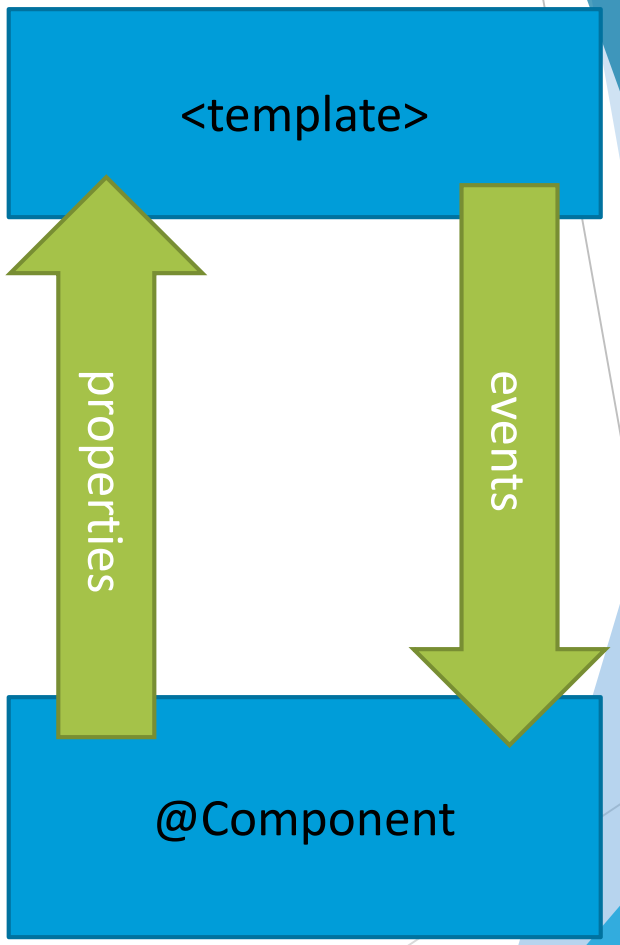
```
<movie-display *ngFor="#movie in movies">
</movie-display>
```

Templates

{{ binding }}

The Role of Templates

- Components
  - No direct DOM manipulation
- Templates
  - No serious model manipulation



Interpolation

- Expression evaluated against parent component
  - No access to global scope
  - Somewhat forgiving (errors logged to console)

```
<td>{{movie.title}}</td>
<td>{{movie.rating}}</td>
```

```
<td [textContent]="movie.title"></td>
```

```
TypeError: Unable to get property 'length' of undefined or null reference in [
    Number of movies: {{ movies.length }}
```

Template Expressions

- Object literals and array literals are allowed
- Statements only allowed for events
- No keywords (*new*, *if*, *class*, *function*)
- Angular uses a parser to understand expressions

```
<li><a [routerLink]="['']">List</a></li>
<li><a [routerLink]="['about', 'phone']">About</a></li>
```

Binding Overview

Event Binding

Property Binding

Attribute Binding

Two Way Binding

Property Binding

- [ ] or bind are property bindings
  - Binding to **properties** not **attributes**
- Also works with custom components and directives

```
<td>{{movie.title}}</td>
<td [textContent]="movie.title"></td>
<td bind-textContent="movie.title"></td>
```

Attribute Binding

- Not all attributes are backed by a property
  - aria-* attributes, for example

```
<button [attr.aria-label]="commandName">{{commandName}}</button>
```

Class and Style Binding

- Use ngClass and ngStyle
  - Special directives for special cases

```
<td [ngClass]="{ good:movie.isGood, bad:movie.isBad }>
    {{movie.rating}}
</td>
```

Class Binding versus NgClass

- Two approaches for binding classes and styles

```
<td [ngClass]="{ good:movie.isGood, bad:movie.isBad }>
    {{movie.rating}}
</td>
```

```
<td [class.good] = "movie.isGood"
    [class.bad] = "movie.isBad">
    {{movie.rating}}
</td>
```

Event Binding

- Use ( ) or *on with any DOM event*
- No special directives needed for new events

```
<button (click)="movie.increaseRating()">+</button>
<button on-click="movie.decreaseRating()">-</button>
```

$event

- Send DOM event information to event handler
- Directives can also emit events (more later)

```
<input type="text" [value]="title"
        (blur)="updateTitle($event)">
```

```
updateTitle(event) {
    this.title = event.target.value;
}
```

Two Way Binding

- Use [( )] with ngModel to synchronize an input with a model

```
<input [(ngModel)]="movie.title" type="text">
```

```
<input [value]="movie.title"
       (input)="movie.title = $event.target.value"
        type="text">
```

```
<input [ngModel]="movie.title"
       (ngModelChange)="updateTitle($event)"
        type="text">
```

Using Built-in Directives

- New binding syntax doesn't require as many low level directives
  - No need for ngMouseOver and ngClick, for example
- Ng2 directives are helpers
  - NgClass, for example

```
<td [ngClass]="{ good:movie.isGood, bad:movie.isBad }>
    {{movie.rating}}
</td>
```

```
<td [class.good] = "movie.isGood"
    [class.bad] = "movie.isBad">
    {{movie.rating}}
</td>
```

NgIf

- Add or remove element from DOM

```
<div *ngIf="isEditing">
    ... form
</div>
```

```
<div [style.display]="isEditing ? 'block'  : 'none'">
    ... form
</div>
```

How Things Work - <template>

- Defined in HTML5 specification
- Holds HTML fragments
  - Browser will not render fragments
  - Fragments are managed by scripts

```
<template>
    <div>
        ... form
    </div>
</template>
```

```
<template [ngIf]="isEditing">
    <div>
        ... form
    </div>
</template>
```

```
<div *ngIf="isEditing">
    ... form
</div>
```

NgSwitch

- Like a switch statement for DOM manipulation

```
<div [ngSwitch]="movie.rating">
    <span *ngSwitchCase="1">Bad</span>
    <span *ngSwitchCase="5">Great</span>
    <span *ngSwitchDefault>Ok</span>
</div>
```

NgFor

- Loop over a collection to clone a template once for each item
  - Provides index, last, even, odd for aliasing

```
<li *ngFor="let movie of movies; let i = index;">
   {{ i + 1 }} : {{ movie.title }}
</li>
```

```
<template ngFor let-movie let-i="index" [ngForOf]="movies">
    <li>
    {{ i + 1 }} : {{ movie.title }}
    </li>
</template>
```

Template Variables

- # creates a template reference variable
- Reference an element or directive

```
<div #output></div>
<button (click)="output.textContent='Clicked!'">
    Press Here
</button>
```

Null Conditional (Elvis) Operators

- Avoid dereferencing null and undefined values

```
<div>
    Showing {{movies.length}} movies
</div>
```

```
EXCEPTION: TypeError: Unable to get property 'length' of undefined or null reference in [
                Showing {{movies.length}} movies
        in MovieList@4:6]
```

```
<div>
    Showing {{movies?.length}} movies
</div>
```

Pipes

- Typically format a model value for display

```
<div>
    Current date is {{ today | date | uppercase }}
    Current time is {{ today | date:'shortTime'}}
</div>
```

Custom Pipes

- Extend PipeTransform and use @Pipe decorator
- Register pipe at module level

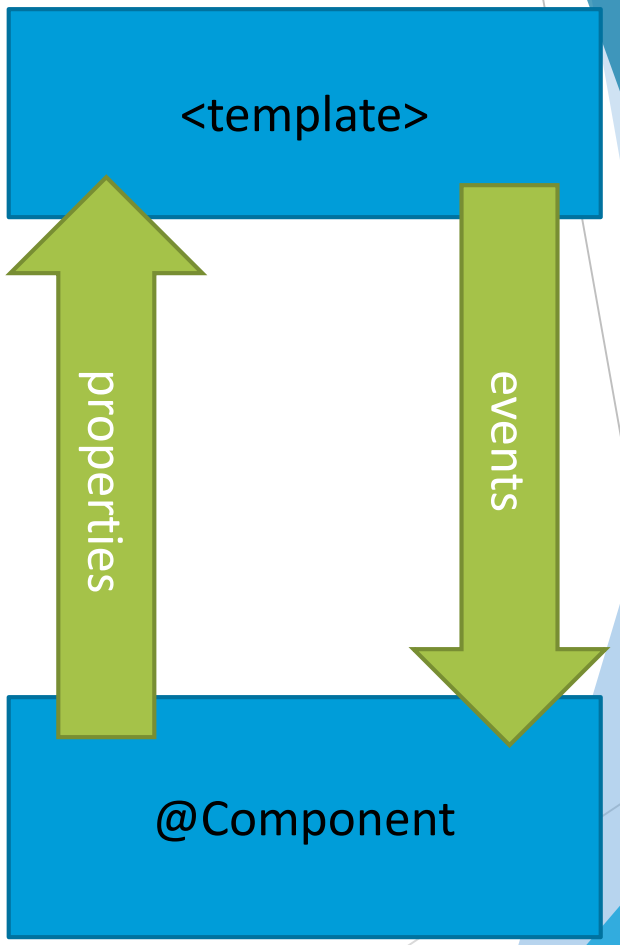
```
import {Pipe, PipeTransform} from "angular2/core";

@Pipe({name:"stars"})
export class StarPipe implements PipeTransform
{
    transform(value: number, args: string[]) {
        return "*".repeat(value);
    }
}
```

```
@NgModule({
  declarations: [
    ...
    StarPipe,
    ...
  ],
```

Summary

- Templates present the component model
  - Data flows to the template
  - Events flow to the component
- Directives are the underlying building block

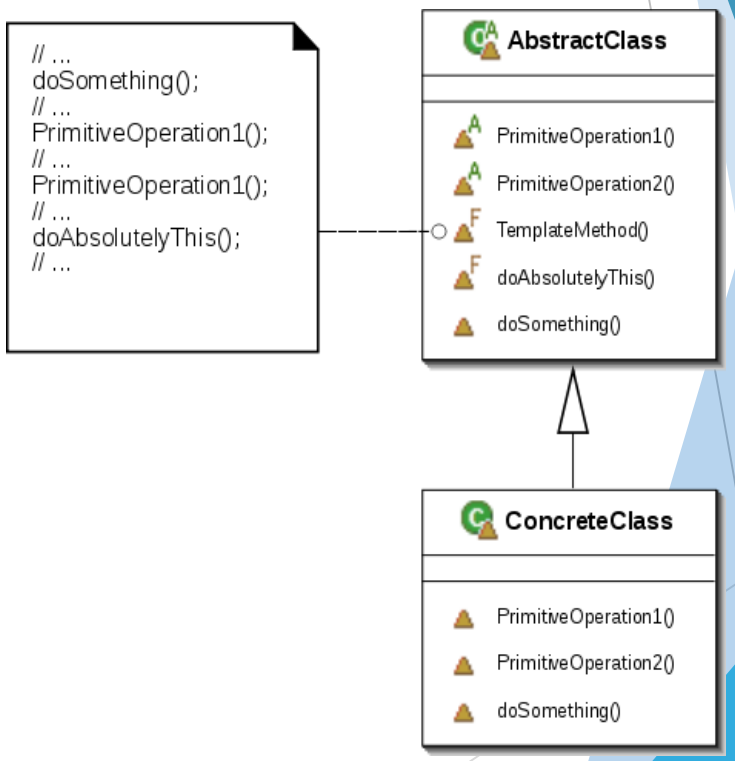


Services

- A piece of code (class) that encapsulates reusable functionality
- Can be injected via DI
- Can have dependencies injected via DI
  - Decorate them with @Injectable()

Inversion of Control

- Increase modularity and extensibility
  - Factory pattern
  - Template method pattern
  - Strategy pattern
  - Service locator pattern
  - Dependency injection*



"Template Method UML" by Giacomo Ritucci

Dependency Injection

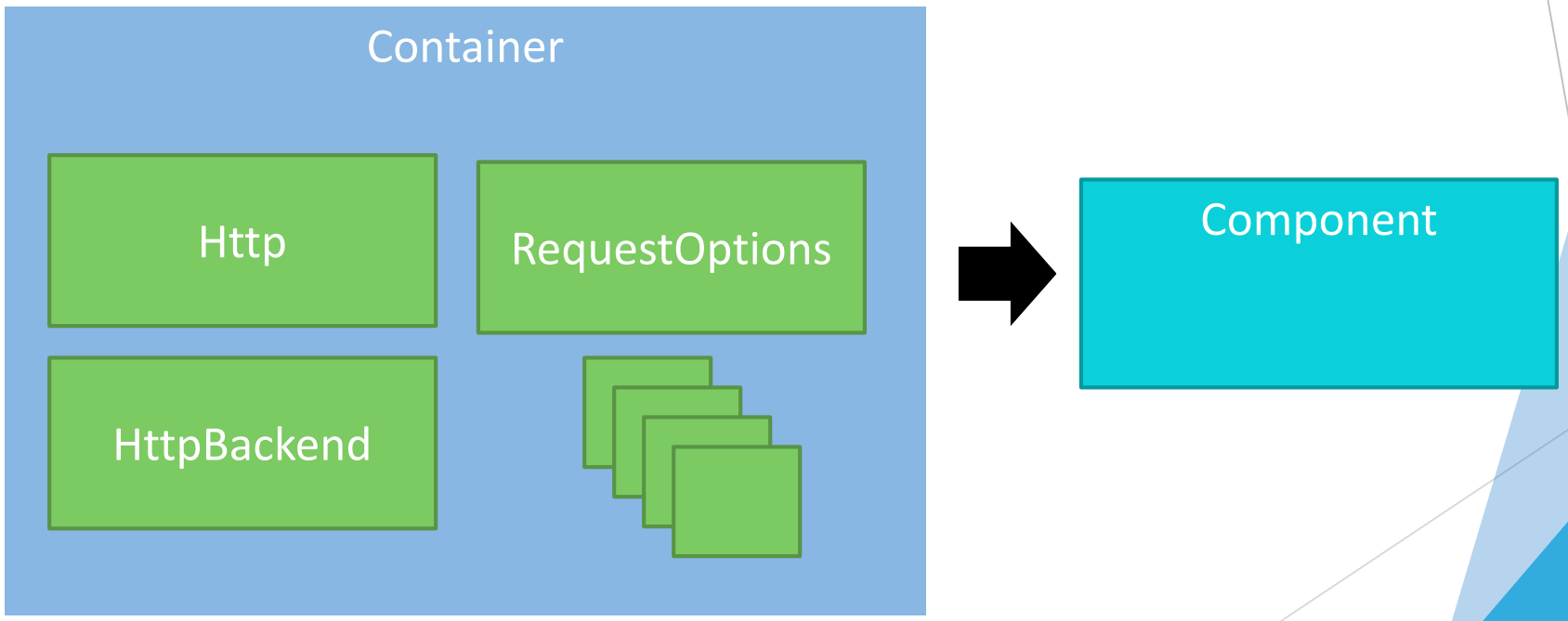
- Object asks for dependencies instead of creating them

```
http: Http
constructor() {
    let options = new RequestOptions();
    let backend = new HttpBackend();
    this.http = new Http(backend, options);
}
```

```
http: Http
constructor(http: Http) {
    this.http = http;
}
```

DI and IoC

- Dependency injection enabled by IoC container
  - Container knows how to build and assemble objects
  - Container knows how to analyze dependencies



The Angular Injector

- The injector is available after bootstrapping
  - Many core services come pre-configured
  - Need to register custom services

```
constructor(injector: Injector) {
    let router = injector.get(Router);
}
```

Providers

- The injector requires a provider for every service
  - Can use built-in providers that rely on class definitions
- Providers registered at the component level
  - Components can register unique providers for children
- Injector uses provider only once
  - Single instance

```
@Component({
    selector: "app",
    templateUrl: "/app/app.html",
    directives: [ROUTER_DIRECTIVES],
    providers: [MovieService]
})
```

Metadata

- Injector relies on metadata to analyze dependencies

```
MovieList = __decorate([
  core_1.Component({
    selector: "movie-list",
    templateUrl: "/app/list/movie-list.html",
    directives: [router_1.ROUTER_DIRECTIVES]
  }),
  __metadata('design:paramtypes',
    [(typeof (_a = typeof MovieService_1.MovieService !== 'undefined' &&
                    MovieService_1.MovieService) === 'function' && _a) || Object,
     (typeof (_b = typeof router_1.Router !== 'undefined' &&
                    router_1.Router) === 'function' && _b) || Object])
], MovieList);
```

Decorators

- Only decorators can add the metadata required for injection
  - @Component
  - @Injectable
  - Any other decorator you might need on a service

```
@Injectable()
export class MovieService {

    constructor(http: Http) {
        // ...
    }

}
```

Providers Revisited

- Different techniques for registering providers
  - useClass
  - useFactory
  - useValue

```
providers: [MovieService]
```

```
providers: [provide(MovieService, { useClass:MovieService})]
```

```
providers: [new Provider(MovieService, { useClass:MovieService})]
```

Factories

- Take control of the service creation
  - Can specify dependencies with the “deps” property

```
providers: [provide(MovieService, {
    useFactory:() => new MovieService()
})]
```

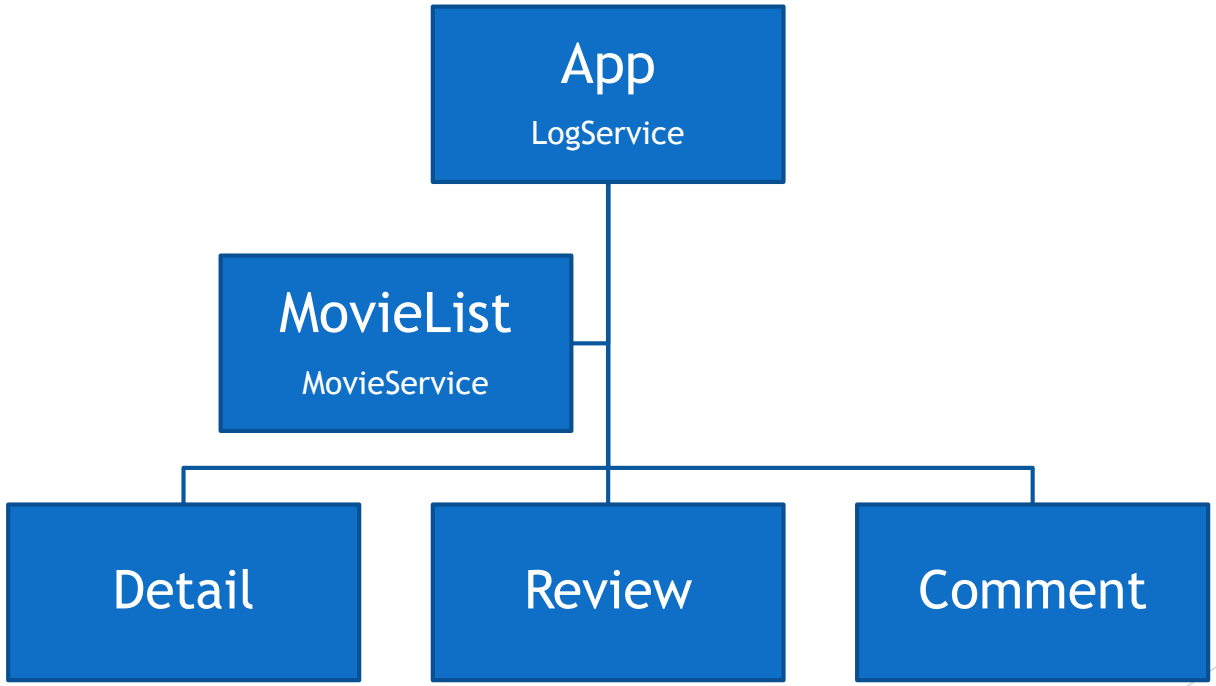
Values

- Useful for configuration values
  - Paths
  - Magic numbers
- Also can register existing instance of a service

```
providers: [
    provide(MovieService, { useFactory:() => new MovieService() }),
    provide("apiUrl", { useValue: "/api "})
]
```

Hierarchy

- Injectors follow the component hierarchy
  - Providers create new instances for current and child components
  - Child injectors search up the tree



Http

- Makes HTTP requests
  - get, put, post, delete
- Returns an Observable from RxJs

```
getAll() : Observable<any> {
    return this.http.get("http://otc-movies.azurewebsites.net/movies")
                .map(r => r.json())

}
```

RxJs

- Compose events and async data into queryable sequences
- Many operators available operators
  - http://reactivex.io/documentation/operators.html

```
var input = document.getElementById('input');
var dictionarySuggest = Rx.Observable.fromEvent(input, 'keyup')
.map(function () { return input.value; })
.filter(function (text) { return !!text; })
.distinctUntilChanged()
.debounce(250)
.flatMapLatest(searchWikipedia)
.subscribe(
    function (results) {
    list = [];
    list.concat(results.map(createItem));
    },
    function (err) {
    logError(err);
    }
);
```

Remember To Provide HTTP

- Can do this at the root level
  - Sets up XHRBackend

```
bootstrap(App, [
    ROUTER_PROVIDERS,
    HTTP_PROVIDERS,
    MovieService,
    provide(LocationStrategy, {useClass: HashLocationStrategy})
]);
```

Working with Headers

- Instantiate the Headers class

```
import {Headers} from 'angular2/http';

let headers = new Headers({
  'X-My-Custom-Header': 'Angular'
});

headers.append('Content-Type', 'image/jpeg');
```

Low Level Work

- Use Request and Request options

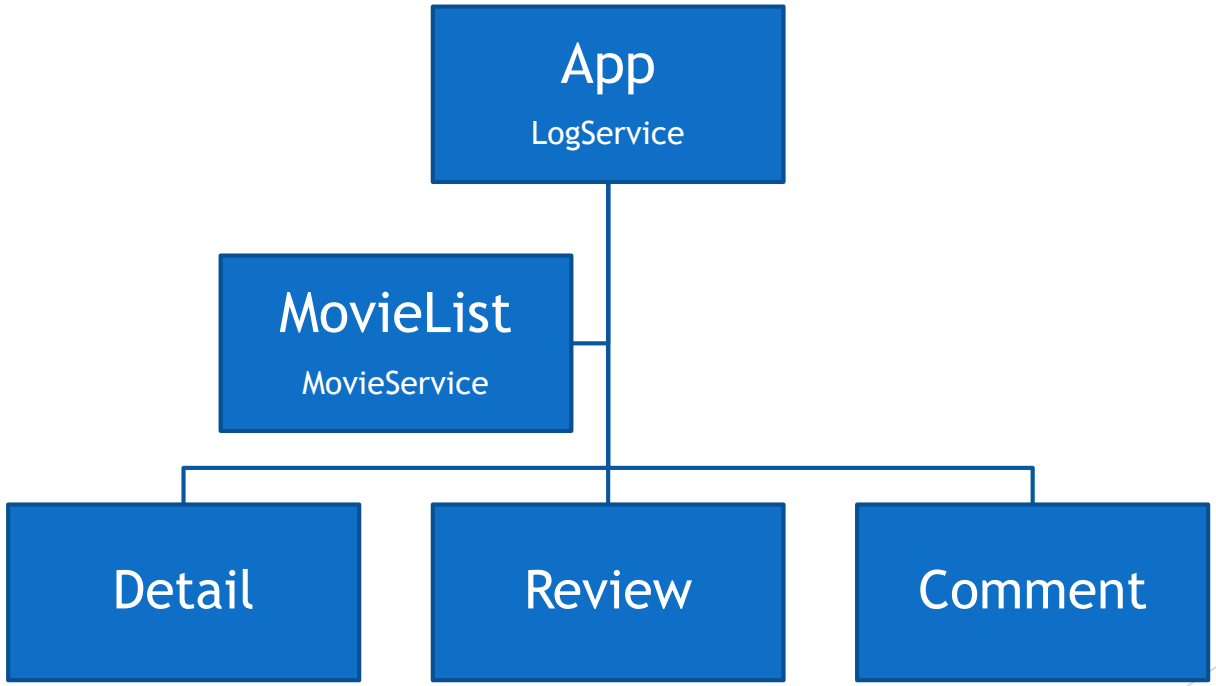
```
let options = new RequestOptions();
options.url = "/api/movies";
options.method = RequestMethod.Post;
options.headers = headers;
options.body = JSON.stringify({title: "Star Wars"});

request = new Request(options);

http.request(request);
```

Summary

- Injector -> Provider -> Service



Routing

<router-outlet>

Why Routing?

- Break large application into small components
- Provide more features
  - Deep linking
  - Bookmarks
  - History

Prepare for Routing

- Router features are in @angular/router
- Set a base href

```
<head>
  <meta charset="utf-8">
  <title>At The Movies</title>
  <link rel="icon" type="image/x-icon" href="/img/favicon.ico">
  <base href="/">
</head>
```

Bootstrap for Routing

- Need a top level route configuration

```
import {provideRouter, RouterConfig} from "@angular/router";
import {MovieListComponent} from "./list/movie-list.component";
import {MovieAboutComponent} from "./about/movie-about.component";

export const routes: RouterConfig = [

    { path: "", component:MovieListComponent },
    { path: "about", component:MovieAboutComponent }

];

export const APP_ROUTER_PROVIDERS = [
    provideRouter(routes)
];
```

```
bootstrap(MovieAppComponent, [APP_ROUTER_PROVIDERS]);
```

The Navigation Shell

- Top level component becomes a shell
  - Use router-outlet directive to plugin routed components
  - Use routerLink to build anchor tags based on route config

```
<div>

    <h1>{{title}}</h1>

    <a [routerLink]="['List']">Movie List</a>
    <a [routerLink]="['About']">About</a>

    <router-outlet></router-outlet>

</div>
```

Working with Parameters

- :tokens are placeholders
  - Can use multiple tokens

```
export const routes: RouterConfig = [

    { path: "", component: MovieListComponent },
    { path: "about", component: MovieAboutComponent },
    { path: "detail/:id", component: MovieDetailComponent },
    { path: "**", redirectTo: "" }

];
```

Passing Parameters

- Pass an object for parameters in routerLink array

```
<a [routerLink]="['detail', movie.id]"
    class="btn btn-default">
    <i class="glyphicon glyphicon-zoom-in"></i>
</a>
```

Receive Parameters

- Use ActivatedRoute
  - Snapshot or subscribe

```
constructor(private route: ActivatedRoute,
            private movieData: MovieData) {

            }

ngOnInit() {
    let id = +this.route.snapshot.params["id"];
    this.movie = this.movieData.getById(id);
}
```

Programmatically Route

- Inject a Router and use navigate

```
<button class="btn btn-primary" (click)="goToList()">
    Back to the list
</button>
```

```
goToList() {
    this.router.navigate([""]);
}
```

Child Routes

- Useful for composite Uis
  - Dashboards, tree views, nested complexity

```
export const aboutRoutes: RouterConfig = [
    {
        path: "about",
        component: MovieAboutComponent,
        children: [
            { path: "phone", component: MovieAboutPhoneComponent},
            { path: "location", component: MovieAboutLocationComponent}
        ]
    }
];
```

Route Guards

- canActivate
- canDeactivate

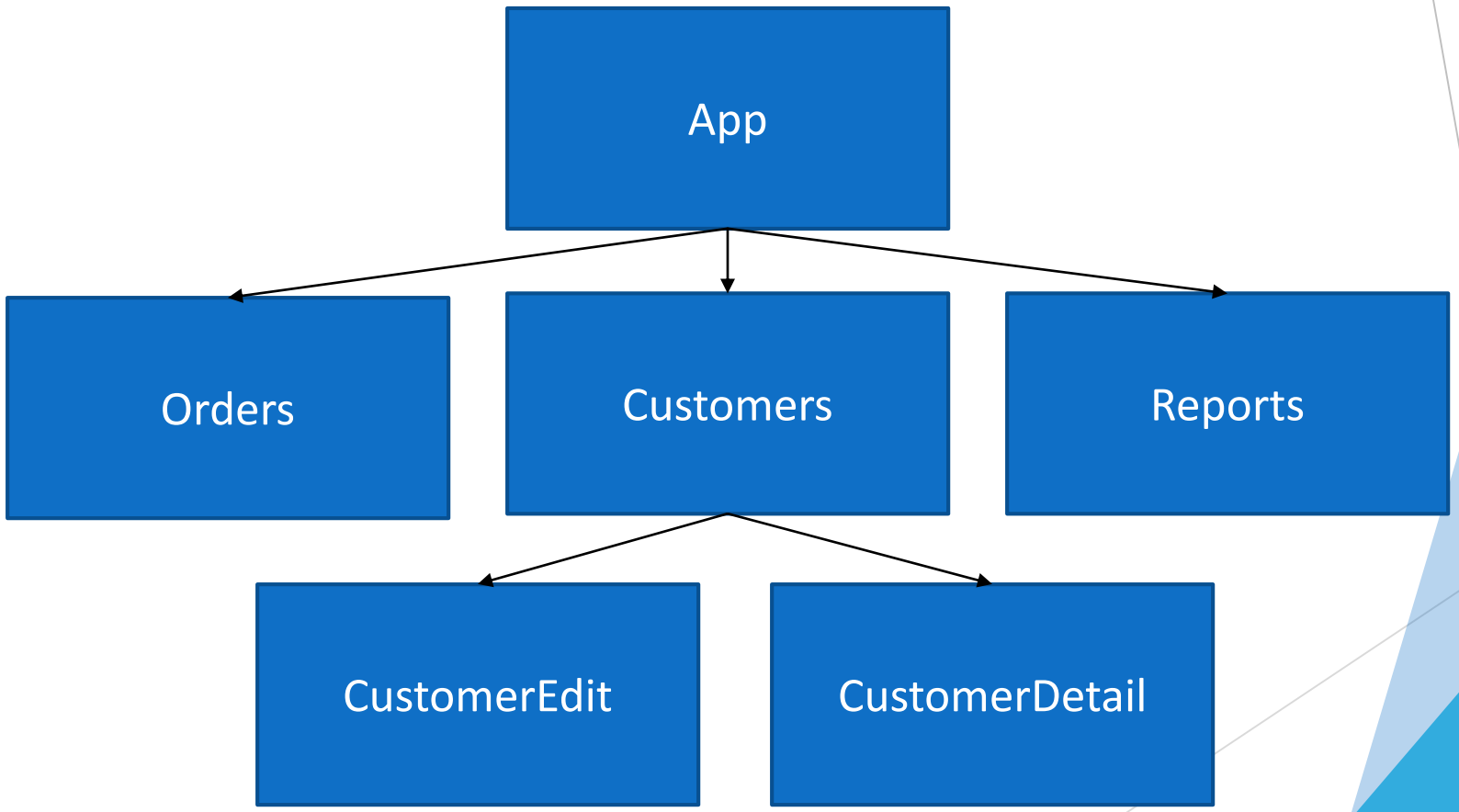
```
export class AboutGuard implements
      CanActivate, CanDeactivate<MovieAboutComponent> {

  canActivate() {
    console.log('canActivate');
    return true;
  }

  canDeactivate(component) {
    return component.canDeactivate();
  }
}
```

Summary

- Routing builds big applications from small pieces



Thank You!

Don’t forget to write! ☺

@tibor19