Component creation

```
import {Component, Input, EventEmitter} from 'angular2/core'
@Component({
     selector: 'favorite',
     template: `<div (click)="onClick()">
                  {{ favorite ? "I like it" : "I don't like it" }}
                </div>
                `
})
export class Component {
  @Input() isFavorite = false;
  @Output() change = new EventEmitter();

  onClick() {
    this.isFavorite = !this.isFavorite();
    this.change.emit( { newValue : this.isFavorite } );
  }
}
```

Properties marked with **@Input()** and **@Output()** form the public API of our component. Users of our component can pass data to the component by setting the input properties and listening to its events. To fire an event, we use the **emit()** method of our **EventEmitter**, which can take an event object as parameter.

Component usage

```
import {FavoriteComponent} from './favorite.component'
@Component({
  selector: "componentUser"
  template: '<favorite [isFavorite]="true" (change)="onFavoriteChange($event)"></favorite >',
  directives: [FavoriteComponent]
})
export class ComponentUser {
  onFavoriteChange($event) {
    console.log("Favorite component fired change event: " + $event);
  }
}
```

To use a component, we need to import it, add it to our directives in order to let Angular know that it needs to render the component (otherwise, the tag is just output), and use it in our template.

We can pass data to the component by setting its input properties and handle any event that are declared as outputs. For each event, we also have access to the event object via the variable **$event**.

Service creation

```
import {Injectable} from 'angular2/core';
import {Http, Response} from 'angular2/http';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import {IUser} from './iuser';
@Injectable()
export class UserService {
  private url = 'http://jsonplaceholder.typicode.com/users';

  constructor(private http:Http) {}
  getUsers(): Observable<IUser[]> {
    return this.http.get(this.url)
        .map(response => response.json());
  }
}
```

Services are just regular typescript classes. **@Injectable()** is only required if the service itself depends on other services via dependency injection. However, it is a best practice to decorate all services with **@Injectable()**.

Service usage

```
import {Component, OnInit} from 'angular2/core';
import MyService from "./my-service.service";
@Component({
  providers: [UserService]
})
export class CourseComponent {
  users: Array<IUser> = undefined;
  constructor(userService: UserService) { }
  ngOnInit() {
    this.userService.getUsers().subscribe(users => { this.users = users; } });
  }
}
```

To use a service, we need to import it, reference it in our providers and pass a reference in a constructor, which lets Angular inject the service into our class.

Directive creation

```
import {Directive} from 'angular2/core';
import {ElementRef, Renderer} from 'angular2/core';
@Directive({
  selector: [myDirective],
  host: {
```

Directive creation (cont)

```
    '(onmouseenter)' : 'gettingFocus()',
    '(onmouseleave)' : 'leavingFocus()'
} })
export class MyDirective {
  constructor(el: ElementRef, renderer: Renderer) { }
  gettingFocus() {
    this.renderer.setElementStyle(this.el.nativeElement.style.color='red')
  }
  leavingFocus() { ... }
}
```

The host section captures DOM events and maps them to methods in our directive class.
To access and modify the DOM element, we inject the element reference (which can be used to get the native DOM element) and the renderer.

Bindings

Interpolation	{{ expression }}	{{ title }}
Elvis operator	{{ object?.nullableProperty }}	{{ book?.appendix?.pages }}
Property binding	[*property*]="expression"	
Class binding	[class.*className*]="expression"	<li [class.active]="isActive" />
ngClass binding	[ngClass]={'*className*':expression, ...}	li [ngClass]={ 'active':isActive, 'passive':!isActive } />
Style binding	[style.*styleName*]="expression"	<button [style.backgroundColor]="isActive ? 'blue' : 'gray'">
ngStyleBinding	[ngStyle]={*styleName*:expression, ...}	<button [ngStyle]={ backgroundColor:isActive ? blue : gray, color: canSave? 'white' : 'back' }/>
Event binding	(*event*)="expression"	<button (click)="onClick($event)">
Two-way binding	[(ngModel)]="*property*"	<input type="text" [(ngModel)]="firstName">

Templates

Show/hide element	[hidden]="expression"	<div [hidden]="courses.length == 0">
Add/remove element	*ngIf="expression"	<div *ngIf="courses.length > 0">
Add/remove multiple cases	[ngSwitch]="expression"; [ngWhen]="expression"; ngSwitchDefault	<div [ngSwitch]="viewMode"> <template [ngSwitchWhen]="'map'" ngSwitchDefault> ... </template> <template [ngSwitchWhen]="'list'"> ... </template>

Templates (cont)

Creating element several times	*ngFor="expression"	<li *ngFor="let course of courses, #i=index"> {{ (i+1) }} - {{ course }}

Built-in pipes

uppercase	{{ course.title | uppercase }} -> ANGULAR COURSE
lowercase	{{ course.title | lowercase }} -> angular course
number	{{ course.students | number }} -> 1,234 {{ course.rating | number:'2.2-2' }} -> 04.97
currency	{{ course.price | currency:'USD':true }} -> $99.95
date	{{ course.releaseDate | date:'MMM yyyy' }} -> Mar 2016
json	{{ course | json }} -> { name : "Angular", author: "Mosh" }

TODO: add pipe parameters

Pipe creation

```
import {Pipe, PipeTransform} from 'angular2/core';
@Pipe({ name: 'shorten' })
export class ShortenPipe implements PipeTransform {
  transform(value: string, args: string[]) {
    var limit = (args && args[0]) ? args[0] : 20;
    if (value)
      return value.substring(0, limit) + "...";
  }
}
```

Pipes are typescript classes that implement the **PipeTransform** interface and have an **@Pipe()** decorator. Each pipe has a name provided through the decorator. We can access arguments to the pipe via the **args** parameter in the **transform()** method.

Pipe usage

```
import {Component} from 'angular2/core';
import {ShortenPipe} from './shorten.pipe';
@Component({
  selector: 'my-app',
  template: '{{ longText | shorten:10 }},
  pipes: [ShortenPipe]
export class AppComponent {
  longText = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua."
  ...
}
```

To use a pipe, we need to import it, reference it in our pipes and then can apply it to any fields.

Todos

ngContent	vid 51

Form Control/ControlGroup Properties

value	Value of the input field
touched/untouched	Whether the field has been activated once
dirty/pristine	Whether the field value has been changed
valid	Whether the field value passes validation
errors	Validation errors for the field. May be null.

Template-driven Form (implicit controls)

```
<form #f="ngForm" (ngSubmit)="onSubmit(f.form)">
  <div class="form-group">
    <label for="name">Name</label>
    <!-- Input field -->
    <input ngControl="name" #name="ngForm"
      class="form-control"
      name="name" type="text"
      required minlength="3"
    />
    <!-- Error messages -->
    <div *ngIf="name.touched && name.errors">
      <div class="alert alert-danger" *ngIf="name.errors.required">
        Name is required.
      </div>
      <div class="alert alert-danger" *ngIf="name.errors.minlength">
        Name should be minimum {{  name.errors.minlength.requiredLength}} characters.
      </div>
    </div>
  </div>

  <!-- More input elements -->
  <button class="btn btn-primary" type="submit" [disabled]="!f.valid">
    Submit
  </button>
</form>
```

We need to create a ControlGroup for the form and a Control for each input field. This is done by assigning the attribute **ngControl="*nameOfControl*"**.
We can assign a local variable to the Control using ***#nameOfControl*="ngForm"**.
Implicitly generated Control objects only offer three validation rules: **required**, **minlength**, and **maxlength**.
Each input with an ngControl directive automatically gets CSS classes assigned, e.g., **ng-touched**, **ng-dirty**, **ng-invalid**.

Model-driven Form (explicit controls)

```
<form [ngFormModel]="form" (ngSubmit)="onSubmit(f.form)">
  <div class="form-group">
    <label for="name">Name</label>
    <!-- Input field -->
    <input ngControl="name" #name="ngForm"
      class="form-control"
      name="name" type="text"
    />
    <!-- Error messages -->
    <div class="alert alert-danger" *ngIf="!name.valid">
      Name is required.
    </div>
  </div>

  <!-- More input elements -->
  <button class="btn btn-primary" type="submit" [disabled]="!f.valid">
    Submit
  </button>
</form>
The typescript file:
import {Component} from 'angular2/core';
import {ControlGroup, Control, Validators} from 'angular2/common';
@Component({
  selector: 'my-form',
  templateUrl: 'my-form.component.html'
})
export class MyFormComponent {
  form = new ControlGroup({
     name : new Control('initial value', Validators.required)
    // one control for each input
  });
}
```

Model-driven forms are similar to template-driven ones, but we need to create a **ControlGroup** with a **Control** for each input explicitly in the typescript class. Also, the form is bound through the directive **[ngFormModel]**.

Creating custom validators

```
import {Control} from 'angular2/common';
export class MyValidators {
  // synchronous validator
  static cannotContainSpace(control: Control) {
    if (control.value.indexOf(' ') < 0)
      return null; // valid
    return { cannotContainSpace: true }; // invalid
  }
  // asynchronous validator
  static shouldBeUnique(control: Control) {
    return new Promise((resolve, reject) => {
      // simulating server call
      setTimeout(function() {
        if (control.value == "duplicate")
          resolve({ shouldBeUnique: true}); // validation fails
        else
          resolve null; // validation passes
        },1000);
    });
  }
}
```

Validators are static methods that return null in case of a passing validation and an object with the name of the validation rule as key and any additional information as value. Asynchronous validators return a promise instead of the object directly. In case validation depends on several inputs, we can pass the **ControlGroup** instead of **Control** and add the validator to the form.

Performing validation on submit

```
...
signUp() {
  var result = authService.login(this.form.value); // call some authentication service
  if (result.error) {
    this.form.find('username').setErrors({
     invalidLogin: true
    });
  }
}
```

Dirty checking

```
import {CanDeactivate, ComponentInstruction} from 'angular2/router';
...
export class MyComponent implements CanDeactivate {
  form : ControlGroup;
  routerCanDeactivate(next: ComponentInstruction, prev: ComponentInstruction) : boolean {
    if (!this.form.dirty)
      return true;
    return confirm('You have unsaved changes. Are you sure to leave?');
  }
}
```

Reactive Extensions / Observables

Common imports	import {Observable} from 'rxjs/Rx'; import 'rxjs/add/operator/map';
Creating observable from control group	this.form.valueChanges
Creating observable from control	this.form.find("inputField").valueChanges
Creating empty observable	Observable.empty()
Creating observable from object	Observable.of(object)
Creating observable from array	Observable.fromArray([1,2,3])
Creating observable from range	Observable.range(1,3)
Creating timer using observable	Observable.interval(1000)
Creating delay using observable	Observable.delay(1000)

Fork/Joining observables	let observable1 : Observable<any> = Observable.of({ user: 'user', pass: 'pass'}).delay(1000);
	let observable2 : Observable<any> = Observable.of({ unreadMessages: 100}).delay(2000);
	let combined : Observable<any> = Observable.forkJoin(observable1, observable2);
	// returns [{user: "user", pass: "pass"}, {unreadMessages: 100}]

Error handling	observable.subscribe(x => console.log(x), e => console.error(e))
Retry in case of error	observable.retry(3)

Reactive Extensions / Observables (cont)

Catching errors	let failedObservable : Observable<any> = Observable.throw(new Error("Cannot reach server")); failedObservable .catch((err) => Observable.of("default data")) .subscribe(x => console.log(x))
Setting timeout	observable.timeout(200)
Notification on completion	// pass callback as third argument when subscribing observable.subscribe(x => console.log(x), e => console.error(e), () => console.log("finished"))

Angular only ships with a minimum of Rx features for performance reasons. To use more functions (e.g. "map"), we have to import it additionally to the Observable (import 'rxjs/add/operator/map';).

Routing

Setting up the base URL	// in index.html, directly after <head> tag <base href="/">

Implementing main router	import {RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS} from '@angular/router';
	...
	directives: [ROUTER_DIRECTIVES, ...]
	...
	...
	@RouteConfig([
	{ path: '/home', component: HomeComponent },
	{ path: '/user/:userId', component: UserComponent },
	...
	])
	
	constructor(private router: Router) {}
	
	ngOnInit() { // default route
	this.router.navigate(['/home']);
	}

Catch-all route	{ path: '*', redirectTo: ['/home'] }
Use route	<router-outlet></router-outlet>

Routing (cont)	
Create router links	import {ROUTER_DIRECTIVES} from '@angular/router'; ... directives: [ROUTER_DIRECTIVES] ... <a [routerLink]="['/home', params]">Go Home <a [routerLink]="['/user', { "userId": userId }]">Go to user page

Manually trigger route	import {Router} from '@angular/router';
	...
	this.router.navigate(['/home']);
	this.router.navigate(['/user', {userId : user.id }]);

Using parameterized router	<a [routerLink]="['/user', {userId: user.id}]">
	
	constructor(..., routeParams: RouteParams) {
	this.userId=routeParams.get("userId");