Angular Cheat Sheet Cheat Sheet

by rajanvora via cheatography.com/141179/cs/30219/

ng commands

ng new *app-name*

(--routing) add routing

ng add *@angular/material*

(install and configure the Angular Material library)

ng build

(build your application and places directory called "dist")

ng serve *(build and serve on 4200)*

ng serve -o *(automatically open the browser)*

ng serve -p 666 -o *(use port 666)*

ng generate component *csr*

or ng g c *csr*

ng g c *csr* --flat *(generate component without folder)*

ng g s *csr-data (generate service objects)*

ng g cl *models/csr*

(generates the csr class inside of the models folder)

ng g i *models/csr*

(generates the csr interface inside the models folder)

ng g p *shared/init-caps*

(generates the init-caps pipes)

Filters

Filters (cont)

> <pre id="default-spacing">
{{ {'name':'value'} | json:4 }}
</pre>
output:
{
"name": "value"
}

array | limitTo:limit

Creates a new array containing only a
specified number of elements in an array.

{{ limitTo_expression | limitTo : limit : begin}}

text | linky 1

Finds links in text input and turns them into html links.

* Requires ngSanitize Module

**

e,g,

<div ng-bind-html="snippetWithSingleURL | linky:'_self':
{rel: 'nofollow'}">

string | lowercase

Converts string to lowercase.

{{ lowercase_expression | lowercase}}

number | number*[:fractionSize]*

Formats a number as text.

If the input is not a number an empty string is returned.

{{ number_expression | number : fractionSize}}

e.g.

Default formatting:

{{val | number}}

No fractions:

{{val | number:0}}

Negative number:

amount | currency ***[:symbol]***

{{ curren cy_ exp ression | currency : symbol : fracti onS ize}}

e.g.

```
<span id="currency-no-fractions">
  {{amount | currency:"USD$":0}}
</span>
```

Formats a number as a currency (ie $1,234.56).

date | date ***[:format]***

{{ date_e xpr ession | date : format : timezone}}

e.g.

```
<span ng-non-bindable>
  {{1288323623006 | date:' MM/ dd/yyyy @ h:mma'}}
</span>
<span>
  {{'1288323623006' | date:' MM/ dd/yyyy @ h:mma'}}
</span>
```

output:

```
{{1288323623006 | date:' MM/ dd/yyyy @ h:mma'}}:
  12/ 15/ 202 1@1 1:40PM
```

array | filter :ex pre ssion

Selects a subset of items from array.

Expression takes string |O b j ec t|f un c t - ion()

{{ filter _ex pre ssion | filter : expression : comparator : anyPro per tyKey}}

e.g.

```
ng-rep eat ="friend in friends | filter :se arc - hTe xt"
```

data | json

Convert a JavaScript object into JSON string.

{{ json_e xpr ession | json : spacing}}

e.g.

Filters (cont)

```
> <span>
  {{-val | number:4}}
</span>
```

array | orderBy:predicate*[:reverse]*

Predicate is function(*)|string|Array. Reverse is boolean

*{{ orderBy_expression | orderBy :
expression : reverse : comparator}}*

e.g.

ng-repeat="friend in friends | orderBy:'-age'"

string | uppercase

Converts string to uppercase.

{{ uppercase_expression | uppercase}}

```
<h1>{{title | uppercase}}</h1>
```

Forms

In Angular, there are 2 types: template-driven(easier to use)
and reacti ve(rec omm ended for large forms)

Templa te- driven:

Import FormsM odule in app.mo dule.ts

Sample:

.ts

```
import { Component, OnInit } from '@angu -
lar /core';
    @Co mpo nent({
                sel ector: 'app-n ewu ser',
templa teUrl: './new use r.c omp one nt.h tml',
styleUrls: ['./ne wus er.c om pon ent.scss']
   })
    export class Newuse rCo mponent implements
OnInit {
// Data
user = {username: '', email: '', password: ''};
constr uctor() { }
ngOnIn it(): void { }
/**
* Called when the user clicks in the " Reg ist er"
* button.
*/
onSubmit() {
consol e.l og( 'User: ', this.u ser.us ern ame);
consol e.l og( 'Email: ', this.u ser.em ail);
consol e.l og( 'Pa ssword: ', this.u ser.pa ssw -
ord);
}
```

Forms (cont)

```
> }
```

.html

```
<div>
<form novalidate #registerForm="ngForm"
(ngSubmit)="onSubmit()">
<!-- User -->
<p>
<mat-form-field>
<input
matInput
placeholder="Username"
type="text"
[(ngModel)]="user.username"
name="username"
#username="ngModel"
required>
<mat-error
*ngIf=
"username.errors?.required">
Username is required
</mat-error>
</mat-form-field>
</p>
<!-- Email -->
<p>
<mat-form-field>
<input matInput
placeholder="Email"
type="email"
[(ngModel)]="user.email"
name="email"
#email="ngModel"
required>
<mat-error
*ngIf="email.errors?.required">
Email is required
</mat-error>
</mat-form-field>
</p>
<!-- Password -->
```

Forms (cont)

```
> <p>
<mat-form-field>
<input
matInput
placeholder="Password"
type="password" [
(ngModel)]="user.password"
name="password"
#password="ngModel"
required>
<mat-error
*ngIf="password.errors?.required">
Password is required
</mat-error>
</mat-form-field>
</p>
<p>
<button type="submit"
mat-button
[disabled]="registerForm.form.invalid">
Register user
</button>
          </p>
</form>
</div>
```

this example uses two way binding and templateRef (for registerForm)

Reactive Forms:

import ReactiveFormsModule in app.module.ts

all the form logic and validation are done controller

create a model

.ts

```
export class User {
  username: string;
  email: string;
  password: string;
}
```

.ts

```
import { Component, OnInit } from '@angular/core';
import { User } from '../shared/user';
```

Forms (cont)

```
> @Component({
  selector: 'app-newuser',
  templateUrl: './newuser.component.html',
  styleUrls: ['./newuser.component.scss']
})
export class NewuserComponent implements OnInit {
  // Register form
  registerForm: FormGroup;
   // form for submitting the new user
  myUser: User;
   // the user generated from the form
  @ViewChild('newuserForm') userFormDirective:
    FormGroupDirective;
  // reference to the form in the HTML template,
  // in order to perform validation
  /*
   * The errors being shown for each field.
   * The form will automatically update with
    * the errors stored here.
   */
   formErrors = {
     'username': '',
'email': '',
'password': ''
   }
  /*
    * Messages that will be shown in the
    * mat-error elements for each type of validation error.
    */
   validationMessages = {
'username': {
'required':
'Username is required.',
      'minlength':
'Username must be at
least 3 characters long.',
'maxlength':
'Username cannot be
more than 20 characters long.'
  },
```

Forms (cont)

```
>    'password': {
'required':
'Password is required.',
'minlength':
'Password must be at
least 8 characters long.'
   },
   'email': {
'required':
'Email is required.',
'email':
'Email not in valid format.'
   }
};

/**
* Inject a FormBuilder for creating a FormGroup.
*/
constructor(private fb: FormBuilder) {
  this.createForm();
}

/**
* Create the comment form with the injected FormBuilder.
*/
createForm(){
   this.registerForm = this.fb.group({
   username: ['',
      [Validators.required,
       Validators.minLength(3),
       Validators.maxLength(20)] ],
   password: ['',
     [Validators.required,
      Validators.minLength(8)] ],
   email: ['',
    [Validators.required,
     Validators.email] ],
});
this.registerForm.valueChanges.subscribe(
   data => this.onValueChanged());
// every time a value changes inside the form, the
// onValueChanged() method will be triggered
```

Forms (cont)

```
> this.onValueChanged();
// reset validation messages
}
/**
* Validate the form after a value change.
*/
onValueChanged() {
   if(!this.registerForm) { return; }
   // in case the form hasn't been created yet
   const form = this.registerForm;
      // the form values are constantly changing,
     // that's why we have to take a snapshot
// Validate the form
   for (const field in this.formErrors) {
     // Iterate the form field by field
     if (this.formErrors.hasOwnProperty(field)) {
this.formErrors[field] = '';
// clear previous error message (if any)
const control = form.get(field);
if (control && control.dirty
&& !control.valid) {
// If this form field has been
// touched an it's not valid
const messages =
this.validationMessages[field];
for (const key in control.errors) {
if (control.errors.hasOwnProperty(key)) {
// Add the corresponding error messages
// to the array of form errors.
// The form mat-error elements will update
// immediatly with the new form errors.
this.formErrors[field] +=
messages[key] + ' ';
                  }
}
   }
      }
   }
}
/**
```

Forms (cont)

```
> * Called when the user clicks the "Submit" button in the form
*/
onSubmit(){
  // Create a User object from the form data
this.myUser = this.registerForm.value;
  // TODO: send the form data to somewhere else
  // Reset the form
        this.registerForm.reset({
username: '',
email: '',
password: ''
});
this.userFormDirective.resetForm();
}
}
```

.html

```
<div>
  <form
    novalidate
    [formGroup]="registerForm"
    #newuserForm="ngForm"
    (ngSubmit)="onSubmit()">
    <p>
<mat-form-field>
<input
matInput
formControlName="username"
placeholder="Username"
type="text"
required>
<mat-error
*ngIf="formErrors.username">
{{formErrors.username}}
</mat-error>
</mat-form-field>
</p>
<p>
<mat-form-field>
<input
matInput
```

Forms (cont)

```
> formControlName="password"
placeholder="Password"
type="password"
required>
<mat-error
*ngIf="formErrors.password">
{{formErrors.password}}
</mat-error>
  </mat-form-field>
</p>
<p>
<mat-form-field>
<input
matInput
formControlName="email"
placeholder="Email"
type="email"
required>
<mat-error
*ngIf="formErrors.email">
{{formErrors.email}}
</mat-error>
      </mat-form-field>
  </p>
  <button type="submit"
    [disabled]="registerForm.invalid"
    mat-raised-button
    color="primary">
Submit
  </button>
  </form>
</div>
```

Directives

```
ng-app="plaintext"
ng-bin d[- htm l-u nsafe]=exp res sio n"
ng-bin d-t emp -
late="string{{expression}}string{{expression}}"
ng-change=" exp res sio n"
ng-che cked= boo lea n"
ng-cla ss[ -ev en| -odd]=r ing |ob jec t"
ng-[db l]c lick=exp res sio n"
```

Cheatography

Angular Cheat Sheet Cheat Sheet

by rajanvora via cheatography.com/141179/cs/30219/

Directives (cont)

> **ng-cloak=**"boolean"
ng-controller="plaintext"
ng-disabled="boolean"
<form|ng-form name="plaintext"> | **ng-form=**"plaintext"
ng-hide|show="boolean"
ng-href="plaintext{{string}}"
ng-include="string"|**<ng-include** src="string"
onload="expression" autoscroll="expression">
ng-init="expression"
<input **ng-pattern=**"/regex/"
ng-minlength ng-maxlength ng-required
<input **ng-list=**"delimiter|regex">
<input type="checkbox" **ng-true-value=**"plaintext"
ng-false-value= "plaintext">
ng-model="expression"
ng-mousedown="expression"
ng-mouseenter="expression"
ng-mouseleave="expression"
ng-mousemove="expression"
ng-mouseover="expression"
ng-mouseup="expression"
<select **ng-multiple**>
ng-non-bindable
ng-options="select [as label] [group by group]
for ([key,] value) in object|array"
ng-pluralize|<ng-pluralize count="number"
when="object" offset="number">
ng-readonly="expression"
ng-repeat="([key,] value) in object|array"
<option **ng-selected=**"boolean">
ng-src="string"
ng-style="string|object"
ng-submit="expression"
ng-switch="expression"|**<ng-switch** on="expression">
ng-switch-when= "plaintext"
ng-switch-default
ng-view|<ng-view>
ng-bind-html="expression"

Routing

With routing, you can introduce navigation between screens
(actually, between Angular compon ents) in your app.
Define routes in **app-ro uti ng.m od ule.ts**
Instan tiate the router in html as :
<ro ute r-o utl et> </r out er- out let>
To redirect the user to a defined route
use the **routerLink** directive

Data Binding

Angular components are defined in three files:
an HTML file for the layout (view),
a TypeScript file for the logic (contr oller), and a CSS file for the style.
One-way data binding is the mechanism for rendering in the
view objects defined in the controller (property binding)
and for allowing the view to call methods in the con troller (event binding).
Two-way data binding, where using the notation [(obje ct)],
a bidire ctional relati onship between the view and
the controller is establ ished, so any changes on the bound
object from the controller will be reproduced in the
view and vice versa.

Structural Directives

Structural directives allow the developers to include
some code logic inside the HTML template in a very quick
and easy way in order to determine when and how many
times an HTML element has to be rendered

Template-Reference Variables

Inside the template of a component, we can assign a
ref erence to an HTML element so we can access its content
from other elements inside the DOM.

RxJS

```
A library for reactive programming in JS, an
  asy nch ron ous -pr ogr amming paradigm where it
exists an
  entity called Observ abl e<T >, which consists
in a value
 of type T that changes over time.
Our applic ation components can subscribe to this
observ able,
  bec oming observers by implem enting a callback
which will be
  tri ggered whenever the value changes.
```

RxJS (cont)

> The main method of observable objects is subscribe(data => {}), which enables us to ask Angular to notify us whenever the data changes.
Other interesting functions: map, pipe, filter, delay..

Services

Components without UI

ng g s servic es/ dat afetch

Tell Angular to inject this service in all of the app components that ask for it, so let's add it to the providers section of the **app.mo dule.ts**file

To use it in any component of our app, you just have to ask for it in the constr uctor.

Promises

Promises (cont)

> **.ts (component that consumes the service)**

```
@Component({
  selector: 'app-users',
  templateUrl: './users.component.html',
  styleUrls: ['./users.component.scss']
})
export class UsersComponent implements OnInit {
  constructor(private userService: UserService) {}
  myUsers: User[];
  getUsers(){
this.userService.getUsers()
.then(users => this.myUsers = users)
            // the Promise was resolved
.catch(error => console.log(error))
// the Promise was rejected
}
  ngOnInit(): void {}
}
```

HTTP Request

The HttpClient class underlies the JavaScript XMLHttpRequest
  object and returns an observable with the server -re sponse
 body encoded as an object of the specified class.
Sample:
Image a GET request to http:/ /lo cal hos t:1 -234 /items
  returns following JSON

```
[
{
" nam e": " Por celain cup",
" pri ce": 9.99,
" qua nti ty": 20
},
{
" nam e": " Photo frame",
" pri ce": 5.99,
" qua nti ty": 50
}
]
create a model to capture the data
export class Item {
name: string;
```

```
These are JS mechanism for async programming,
where a
  pending value is returned, which might be
available soon
  (re solve) or never (reject)
Promises allow you to specify what to do when the
answer to
 your request arrives or when something goes
wrong, and
  mea nwhile you can continue with the execution of
your program.
.servi ce.ts
@Injec table({
      pro vid edIn: 'root'
})
export class UserSe rvice {
      con str uctor() {}
      get Use rs(): Promis e<U ser> {
            return new Promise(
                    fun cti on( res olve, reject){
                           // get the data from
some API...
                            if( suc ces sful) {
                                   // Data was
succes sfully retrieved
                                    res olv e(r -
esult);
                           } else {
                                   // There was an
error retrieving the data
                                    rej ect (er -
ror);
                           }
});
       }
}
```

HTTP Request (cont)

```
> price: number;
quantity: number;
}
After importing HttpClientModule in app.module.ts
.service.ts
import { Injectable, Inject } from '@angular/core';
import { Item } from '../shared/item'
import { Observable } from 'rxjs';
import { map, catchError } from 'rxjs/operators';
import { HttpClient } from '@angular/common/http';
@Injectable({
providedIn: 'root'
})
export class ItemService {
baseURL = 'http://localhost:1234/'
/**
* Injects an HTTPClient and the BaseURL
* into the service.
* @param http HTTPClient used for making
* HTTP requests to the backend.
*/
constructor(private http: HttpClient) { }
/**
* Return the list of items from the API,
* as an array of Item objects
*/
getItems(): Observable<Item[]> {
return this.http.get<Item[]>
(this.baseURL + 'items');
// make the HTTP GET request
}
/**
* Send a new item to the API, as
* an Item object
*/
addItem(item: Item): Observable<Item> {
return this.http.post<Item>
(this.baseURL + 'items', item);
// make the HTTP POST request
}
```

HTTP Request (cont)

```
> }
component to consume the service:
.ts
import { Component, OnInit } from '@angular/core';
import { ItemService } from '../services/item';
import { Item } from '../shared/item';
@Component({
selector: 'app-items',
templateUrl: './items.component.html',
styleUrls: ['./items.component.scss']
})
export class ItemsComponent implements OnInit {
constructor(private itemService: ItemService) {}
myItems: Item[];
ngOnInit(): void {
this.itemService.getItems()
.subscribe(items => this.myItems = items,
// any time the value of the Observable
// changes, update the myItems object
error => console.log(error));
// if there is an error, log it to the
// console
}
}
```

Workflow

```
Steps to creating a reactive form:
1. Create the Domain Model
2. Create the Controller with references to View
3. Create the View
4. Add Valida  tions
5. Add Submit Validation Control
6. Add Dynamic Behaviors
```