

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)

> data | json

Convert a JavaScript object into JSON string.

{{ json_expression | json : spacing}}

e.g.

```
<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="snippet | linky">

</div>

<div ng-bind-html="snippetWithSingleURL | linky:'_blank'">

</div>

<div ng-bind-html="snippetWithSingleURL | linky:'_self':

{rel: 'nofollow'}">

</div>

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:

amount | currency[:symbol]

```
{{ currency_expression | currency : symbol :
fractionSize}}
```

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_expression | date : format : timezone}}
```

e.g.

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

output:

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

array | filter :expression

Selects a subset of items from array.

Expression takes string |Object|function()

```
{{ filter_expression | filter : expression :
  comparator : anyPropertyKey}}
```

e.g.

```
ng-repeat ="friend in friends | filter :searchText"
```



By **rajanvora**

cheatography.com/rajanvora/

Published 15th December, 2021.

Last updated 15th December, 2021.

Page 1 of 11.

Sponsored by **Readable.com**

Measure your website readability!

<https://readable.com>

Filters (cont)

```
> <span id='number-default'>
  {{val | number}}
</span>
No fractions:
<span>
  {{val | number:0}}
</span>
Negative number:
<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 reactive (recommended for large forms)

Template-driven:

```
Import FormsModule in app.module.ts
Sample:
.ts
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-newuser',
  templateUrl: './newuser.component.html',
  styleUrls: ['./newuser.component.scss']
})
export class NewuserComponent implements
OnInit {
  // Data
```

Forms (cont)

```
> user = {username: "", email: "", password: ""};
constructor() {}
ngOnInit(): void {}
/**
 * Called when the user clicks in the "Register"
 * button.
 */
onSubmit() {
  console.log('User: ', this.user.username);
  console.log('Email: ', this.user.email);
  console.log('Password: ', this.user.password);
}
}
.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 -->
```



Forms (cont)

```
> <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 -->
<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">
```

Forms (cont)

```
> 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';
@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
  /*
```



Forms (cont)

```
> * 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.'
  },
  '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.'
```

Forms (cont)

```
> }
};

/**
* 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
  this.onValueChanged();
  // reset validation messages
}
/**
* Validate the form after a value change.
*/
onValueChanged() {
  if(!this.registerForm) { return; }
```



Forms (cont)

```
> // 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 and 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
// immediately with the new form errors.
this.formErrors[field] +=
messages[key] + ' ';
        }
    }
}
}
}
}
/**
 * Called when the user clicks the "Submit" button in the form
 */
onSubmit(){
    // Create a User object from the form data
```

Forms (cont)

```
> 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-bind[-html-unsafe]<exp res sio n"
ng-bind-template -
late="string{{expression}}string{{expression}}"
ng-change=" exp res sio n"
ng-checked# boo lea n"
ng-class[-even| -odd]&#x2013;ring lob jec t"
ng-click=exp res sio n"
ng-cloak=" boo lea n"
ng-controller=pla int ext "
ng-disabled# boo lea n"
<form| ng-formname="p lai nte xt">ng-form=" -
pla int ext "
ng-hide|show# boo lea n"
ng-href="plaintext{{string}}"
ng-include# str ing<ng -in clude>rc="st -
rin g"
onload="expression" autoscroll="e xpr ess -
ion ">
ng-init=" exp res sio n"
<input ng-pattern# /r ege x/ "
ng-min length ng-max length ng-req uired
<input ng-list=" del imi ter |re gex ">
<input type="c he c k ng-tru e- valúe=
pl ain te xt "
ng-false - v alue= ain te xt ">
ng-model=" exp res sio n"
ng-mousedown# exp res sio n"
ng-mouseenter# exp res sio n"
ng-mouseleave# exp res sio n"
ng-mousemove# exp res sio n"
ng-mouseover# exp res sio n"
ng-mouseup# exp res sio n"
<select ng-multiple
ng-non-bindable
ng-options# select [as label] [group by group]
for ([key,] value) in object |ar ray "
ng-pluralize |<n g-p lur abize= " num -
ber "
when="object" offset ="nu mbe r">
ng-reasonly# exp res sio n"
ng-repeat=" ([key,] value) in object |ar ray "
<option ng-selected# boo lea n">
ng-src=" str ing "
```

Directives (cont)

```
> 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 components) in your app.

Define routes in `app-routing.module.ts`
Instantiate the router in html as :

```
<router-outlet> </router-outlet>
```

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 (controller), 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 controller (event binding).

Two-way data binding, where using the notation `[(object)]`, a bidirectional relationship between the view and the controller is established, 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 reference 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 asynchronous programming paradigm where it exists an

entity called `Observable<T>`, which consists in a value of type `T` that changes over time.

Our application components can subscribe to this observable,

becoming observers by implementing a callback which will be triggered whenever the value changes.

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 services/ data-fetch
```

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.module.ts` file
To use it in any component of our app, you just have to ask for it in the constructor.

Promises


```
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
    meanwhile you can continue with the execution of
your program.
.servi ce.ts
@Injectable({
    pro vid edIn: 'root'
})
export class UserSe rvice {
```



By **rajanvora**
cheatography.com/rajanvora/

Published 15th December, 2021.
Last updated 15th December, 2021.
Page 7 of 11.

Sponsored by **Readable.com**
Measure your website readability!
<https://readable.com>

Promises (cont)

```
> constructor() {}
getUsers(): Promise<User> {
  return new Promise(
    function(resolve, reject){
      // get the data from some API...
      if(successful) {
        // Data was successfully retrieved
        resolve(result);
      } else {
        // There was an error retrieving the data
        reject(error);
      }
    }
  );
}

.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 response body encoded as an object of the specified class.

Sample:

Image a GET request to `http://localhost:1234/items` returns following JSON

```
[
  {
    "name": "Porcelain cup",
    "price": 9.99,
    "quantity": 20
  },
  {
    "name": "Photo frame",
    "price": 5.99,
    "quantity": 50
  }
]
```

create a model to capture the data

```
export class Item {
  name: string;
  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/'
```

HTTP Request (cont)

```
> /**
 * 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
  }
  }
  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']
  })
```

HTTP Request (cont)

```
> 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 Validations
5. Add Submit Validation Control
6. Add Dynamic Behaviors

