

Chapter 12: Creating directives and advanced components

1. Implementing search sessions:

We want to add meaning to the search bar in the nav bar

Edited nav bar component.html

```
</div>
<form id="searchForm" (ngSubmit) = "searchSessions(searchTerm)" class="navbar-form navbar-right" >
  <div class="form-group">
    <input type="text" [(ngModel)] = "searchTerm" name = "searchTerm" class="form-control" placeholder="Search Sessions" />
  </div>
</form>
```

Then nav bar component.ts

```
searchTerm: string = "";
foundSessions: ISession[]
constructor (public auth: AuthService, private eventService: EventService){
}

searchSessions(searchTerm) {
  this.eventService.searchSessions(searchTerm).subscribe(sessions => {
    this.foundSessions = sessions;
    console.log(this.foundSessions);
  })
}
```

Finally inside event service we added a search Sessions function

```
searchSessions(searchTerm: string) {
  var term = searchTerm.toLocaleLowerCase();
  var results: ISession[] = [];

  EVENTS.forEach(event => {
    var matchingSessions = event.sessions.filter(session => session.name.toLocaleLowerCase().indexOf(term) > -1);
    matchingSessions = matchingSessions.map((session: any) => {
      session.eventId = event.id;
      return session;
    })
    results = results.concat(matchingSessions);
  })

  var emitter = new EventEmitter(true);
  setTimeout(() => {
    emitter.emit(results);
  }, 100);
  return emitter;
}
```

Now we can see the found session in the console log:

```
▼ [{"...}] ⓘ
  0: {id: 1, name: "Using Angular 4 Pipes", presenter: "Peter Bacon Darwin", duration: 1, level: "Intermediate", ...}
    length: 1
    __proto__: Array(0)
```

2. Adding jQuery:

We want the search results to be shown in the UI, so we need jQuery for it. As bootstrap runs with jQuery.

So we create a jQuery service and a token for it and import it into app module .

```
import { TOATSR_TOKEN, Toastr, CollapsibleWellComponent, JQ_TOKEN } from './common/index'
import { AuthService } from './user/auth.service';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

let toastr:Toastr = window['toastr']
let jQuery = window['$']
```

```
c > app > common > ts jQuery.service.ts > JQ_TOKEN
1 import { InjectionToken } from '@angular/core'
2
3
4
5 export let JQ_TOKEN = new InjectionToken <object>('jQuery')
```

3. Creating a modal component

Created a simpleModal Component: check out simpleModal component.ts in common folder

Set the modal component inside nav bar html

```
<simple-modal closeOnBodyClick="true" elementId="searchResults" title="Matching Sessions">
  <div class="list-group">
    <a class="list-group-item" *ngFor="let session of foundSessions" [routerLink]="['/events', session.eventId]">{{session.name}}</a>
  </div>
</simple-modal>
```

Declared and imported simpleModal component into app Module as well, along with jQuery token.

```
DurationPipe,
SimpleModalComponent
],
providers:
[
  EventService,
  EventRouteActivator,
  {provide: TOATSR_TOKEN, useValue: toastr},
  {provide: JQ_TOKEN, useValue: jQuery},
  AuthService,
  {
    provide: EventRouteActivator,
    useClass: EventRouteActivator
```

4. **Template parse errors** originate from html files, either html components or embedded html elements. We can find the error inside the console

5. Creating Directives: Trigger directive

Created a modal Trigger directive and connected it to simple modal component and nav bar to show a matching box, which shows the search

modalTrigger directive:

```
rc > app > common > TS modalTrigger.directive.ts > ...
1  import { Directive, OnInit, Inject, ElementRef } from "@angular/core";
2  import { JQ_TOKEN } from "../jquery.service";
3
4  @Directive({
5    selector: '[modal-trigger]'
6  })
7  export class ModalTriggerDirective implements OnInit {
8    private el: HTMLElement;
9
10
11    constructor(ref: ElementRef, @Inject(JQ_TOKEN) private $ : any) {
12      this.el = ref.nativeElement;
13    }
14
15    ngOnInit() {
16      this.el.addEventListener('click', e => {
17        this.$('#simple-modal').modal({})
18      })
19    }
20
21  }
```

Simple modal component to connect ID

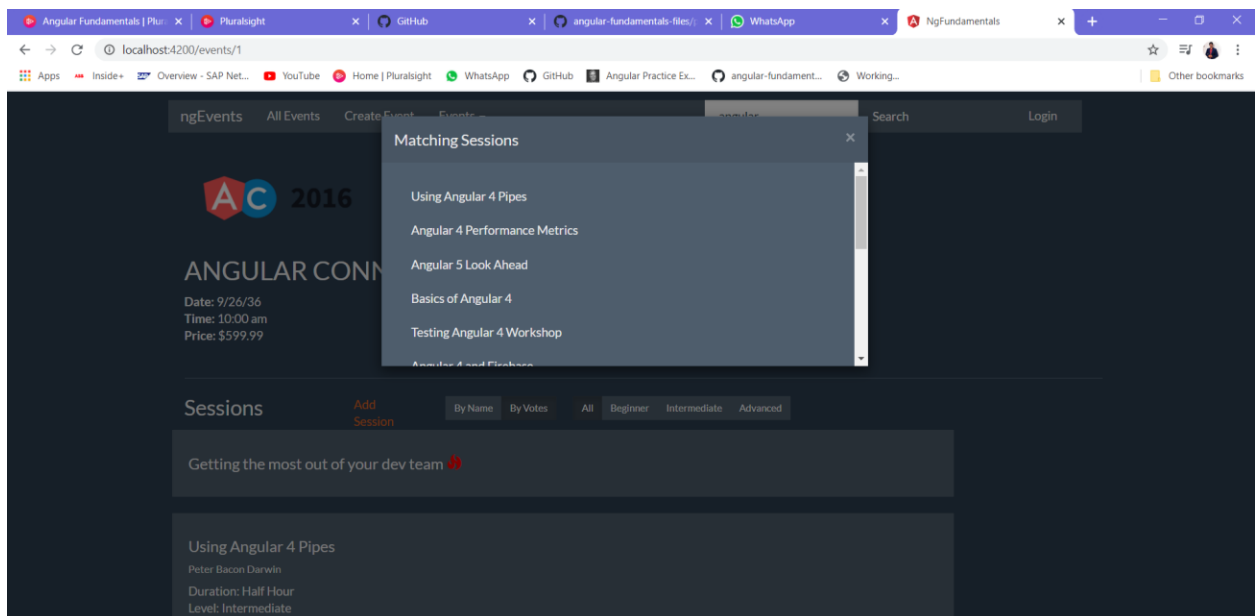
```
template: `
<div id="simple-modal" class="modal fade" tabindex="-1">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal"><span>&times;</span></button>
        <h4 class="modal-title">{{title}}</h4>
      </div>
      <div class="modal-body">
        <ng-content></ng-content>
      </div>
    </div>
  </div>
</div>
`,
styles: [
```

Connection from nav bar component:

```
<button class="btn btn-default" modal-trigger >
  Search
</button>
</form>
</div>
</div>
```

Imported the directive into app module under index.ts created for common folder.

Output of modal component and directive after clicking search with angular as searchTerm.



6. Binding an ID:

The modal we created restricts us from creating other modals called simple modal from the same component due to the name which cannot be used more than once.

By binding the modal to an ID, we can create multiple simple modals. To bind the modal to an ID, the following changes have to be made

In simpleModal Component:

```
template:
<div id="{{elementId}}" class="modal fade" tabindex="-1">
  <div class="modal-dialog">
    <div class="modal-content">
```

```

    })
    export class SimpleModalComponent {
      @Input() title: string;
      @Input() elementId: string;
    }

```

In nav bar html

```

    </div>
    <button class="btn btn-default" modal-trigger="searchResults" >
      Search
    </button>
  </form>
</div>
</div>
</div>

<simple-modal elementId="searchResults" title="Matching Sessions">
  <div class="list-group">
    <a class="list-group-item" *ngFor="let session of foundSessions" [routerLink]
  </div>

```

Edit the button class to show the element ID and simple modal to set an elementID

In modalTrigger directive:

```

directive({
  selector: '[modal-trigger]'

  port class ModalTriggerDirective implements OnInit {
    private el: HTMLElement;
    @Input('modal-trigger') modalId: string; // Input elements aren't allowed to have a hyp
  }

  constructor(ref: ElementRef, @Inject(JQ_TOKEN) private $ : any) {
    this.el = ref.nativeElement;
  }

  ngOnInit() {
    this.el.addEventListener('click', e => {
      this.$(`#${this.modalId}`).modal({})
    })
  }
}

```

Input the modal trigger as a valid element and set its reference inside ngOnInit().

Then Output is same as the previous lesson.

7. Routing to the same component

There was a bug, which wouldn't let us go to the page of the searched event but changes the url to that particular event. This happened because we didn't configure the app to route to the same component, which is the event details component.

We just change 2 lines of code to fix this bug : inside event-details component.ts

```
ngOnInit() {  
  this.route.params.forEach((params: Params) => {  
    this.event = this.eventService.getEvent(+params['id']); // this helps  
    this.addMode = false; // without this, the addmode will be true even w  
  })  
}
```

Now it can route to the desired page and the addmode = false help reset the add session functionality once we route to another set of event details.

8. Using @ViewChild decorator and added settings to it : refer GitHub for changes and commit

Chapter 13: More Components and Custom Validators

1. Intro
2. Creating a voting component: 2+3 together* ||
3. Adding voting Functionality:

We have the voters array in each session and we would like the current user to have to option to vote as well for the sessions and also see the amount of votes which are previously present.

In session list html : we add

```
<div class="col-md-1">  
  <upvote (vote)="toggleVote(session)" [count]="session.voters.length" [voted]="userHasVoted(session)"></upvote>  
</div>  
  
<div class="col-md-10">
```

Then we create an upvote component:

```
import { Input, Output, Component, EventEmitter } from "@angular/core";

@Component({
  selector: 'upvote',
  styleUrls: ['./upvote.component.css'],
  template: `
    <div class="votingWidgetContainer pointable" (click)="onClick()">
      <div class="well votingWidget">
        <div class="votingButton">
          <i *ngIf = "voted" class="glyphicon glyphicon-heart" ></i>
          <i *ngIf = "!voted" class="glyphicon glyphicon-heart-empty" ></i>
        </div>
        <div class="badge badge-inverse votingCount">
          <div>{{count}}</div>
        </div>
      </div>
    </div>
  `
})
export class UpvoteComponent {
  @Input() count: number;
  @Input() voted: boolean;
  @Output() vote = new EventEmitter();

  onClick() {
    this.vote.emit(!this.voted);
  }
}
```

Along with an **upvote** css

Then we link it with the **session list component**: for `userHasVoted` and `toggleVote()`

```
toggleVote (session:ISession){
  if(this.userHasVoted(session)){
    this.voterService.deleteVoter(session, this.auth.currentUser.userName);
  } else {
    this.voterService.addVoter(session, this.auth.currentUser.userName);
  }
  if(this.sortBy == 'votes')
    this.visibleSessions.sort(sortByVotesDesc);
}

userHasVoted(session: ISession) {
  return this.voterService.userHasVoted(session, this.auth.currentUser.userName)
}
```

We create a **voter service** in the same folder for this:

```

app / events / event-details / <voter-services / voter-service
import { Injectable } from "@angular/core";
import { ISession } from "../index";

@Injectable()
export class VoterService {
  deleteVoter(session: ISession, voterName: string) {
    session.voters = session.voters.filter(voter => voter !== voterName);
  }

  addVoter(session: ISession, voterName: string) {
    session.voters.push(voterName);
  }

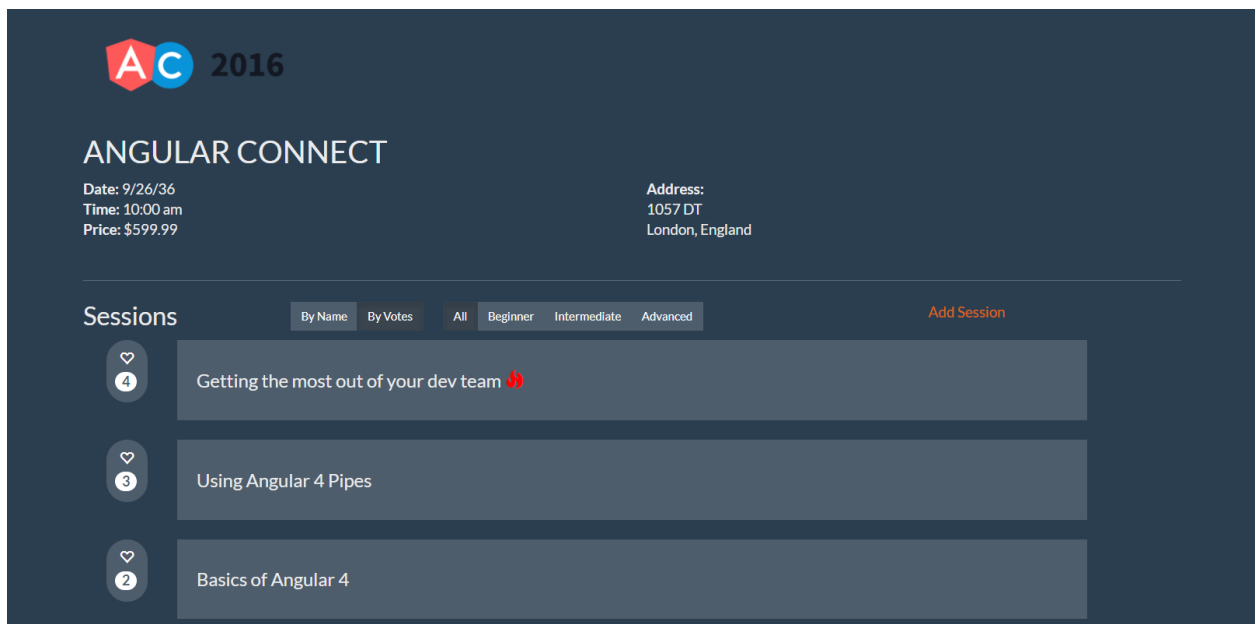
  userHasVoted(session: ISession, voterName: string) {
    return session.voters.some(voter => voter === voterName);
  }
}

```

We link up upvote component under declarations and voter service under providers in app module.

When we run the app after these changes, we need to refresh the page and enter into events url and login before seeing the desired output. Or else it shows a glitch

The **output** after all these changes is :



4. Hiding a functionality before authentication:

```

<div class="col-md-1">
  <div *ngIf = "auth.isAuthenticated()">
    <upvote (vote)="toggleVote(session)" [count]="session.voters.length" [voted]="userHasVoted(session)"></upvote>
  </div>
</div>

```

Adding the ngIf tag around the upvote tag allows the user to view the voting option only when logged in. With this we fix the glitch we ran into previously .

5. Using @Input Setters

Using this we change the color of the heart from white to red rather than the previous change: We edit the glyphicon color using a small function and set the color based on voted inside the upvote component rather than ngIf:

```
    <div class="well votingWidget">
      <div class="votingButton">
        <i class="glyphicon glyphicon-heart" [style.color] = "iconColor" ></i>
      </div>
      <div class="badge badge-inverse votingCount">
        <div>{{count}}</div>
      </div>
    </div>
  </div>
})
export class UpvoteComponent {
  @Input() count: number;
  @Input() set voted(val) {
    this.iconColor = val ? 'red' : 'white';
  }
  @Output() vote = new EventEmitter();
  private iconColor: string;

  onClick() {
    this.vote.emit({});
  }
}
```

6. Added a custom validator :

We added a custom validator to either take in the full address or an online URI before creating an event. Now the form can only be submitted only when either of the 2 are available . We created a custom validator and made changes to create-event. Component.html

```
import { Directive } from "@angular/core";
import { Validator, FormGroup, NG_VALIDATORS } from "@angular/forms";

@Directive({
  selector: '[validateLocation]',
  providers: [{provide: NG_VALIDATORS, useExisting: LocationValidator, multi: true}]
})
export class LocationValidator implements Validator {
  validate(formGroup: FormGroup): { [key: string]: any; } {
    let addressControl = formGroup.controls['address'];
    let cityControl = formGroup.controls['city'];
    let countryControl = formGroup.controls['country'];
    let onlineUrlControl = (<FormGroup>formGroup.root).controls['onlineUrl'];

    if((addressControl && addressControl.value && cityControl && cityControl.value && countryControl && countryControl.value) || (onlineUrlControl && onlineUrlControl.value)) {
      return null;
    } else {
      return {validateLocation: false};
    }
  }
}
```

Added some code into location formgroup and online url formgroup inside create event html.

Chapter 14: Communicating with server using HTTP, observables and Rx

1. Preparing to store data on the server:

First we install the server: Enter into the ng-fundamentals1 directory through cmd prompt and run **"npm install ngf-server -S"**

That installs the server and saves it in the same directory.

Installation is a success when we can see ngf-server inside package.json (under dependencies)

After that we create a file named proxy.conf.json in the root directory where package.json is present and enter in this code:

```
proxy.conf.json > {} /api > secure
1  {
2    "/api" : {
3      "target": "http://localhost:8808",
4      "secure": false
5    }
6  }
```

This states, if the url contains the term api, it should run at this particular host.

Then we wire up the server and the config file in the package.json (under scripts)

```
"ng": "ng",
"start": "ng serve --proxy-config proxy.conf.json",
"build": "ng build",
"test": "ng test",
"lint": "ng lint",
"server": "node node_modules/ngf-server/server.js",
"e2e": "ng e2e",
},
"private": true,
"dependencies": {
```

Now when we run **"npm start"** as usual and **"npm run server"** it runs with the proxy json file as well and the app runs as usual.

To finish off we import HttpClientModule from @angular/common/http under imports in app module

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http'
5
6
7 let toastr:Toastr = window['toastr']
8 let jquery = window['$']
9
10 @NgModule({
11   imports: [
12     BrowserModule,
13     RouterModule.forRoot(appRoutes),
14     FormsModule,
15     ReactiveFormsModule,
16     HttpClientModule
17   ]
18 })
```

2. Moving Data storage to the server:

We make changes in event.service and event.resolver.ts to be able to move and get data from the server:

Event.service.ts

```
> app > events > shared > TS event.service.ts > EventService > updateEvent
1 | import { HttpClient } from '@angular/common/http'
2 | import { Injectable, EventEmitter } from '@angular/core'
3 | import { Observable, Subject, of } from 'rxjs'
4 | import { IEvent, ISession } from './event.model'
5 | import { catchError } from 'rxjs/operators'
6 | @Injectable()
7 | export class EventService{
8 |
9 |     constructor (private http:HttpClient){
10 |
11 |     }
12 |
13 |     getEvents():Observable<IEvent[]> {
14 |         return this.http.get<IEvent[]>('/api/events')
15 |         .pipe(catchError(this.handleError<IEvent[]>('getEvents', [])))
16 |     }
17 |
18 |
19 |     private handleError<T> (operation = 'operation', result?: T) {
20 |         return (error: any): Observable<T> => {
21 |             console.error(error);
22 |             return of(result as T);
23 |         }
24 |     }
25 | }
```

Now for event-resolver.ts

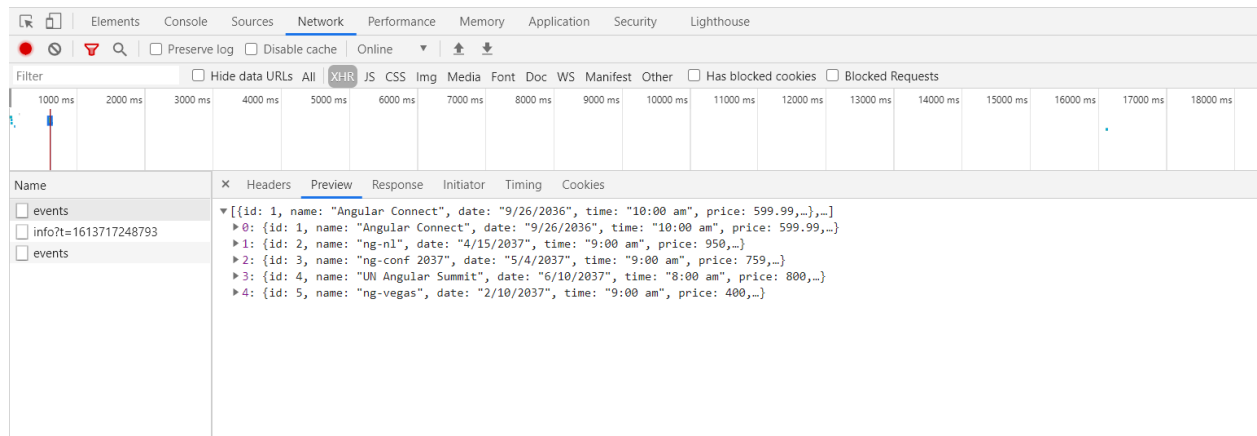
```
9
10 | }
11 |
12 | resolve() {
13 |     return this.eventService.getEvents()
14 | }
15 | }
```

We edit the resolve function which helps subscribe to the server.

To check whether data is coming from the server:

Go to console – network- XHR calls – events- preview

We see:



3. Listening to resolved data Changes:

Deleted event activator service, created event resolver service:

```
rc > app > events > ts event-resolver.service.ts > EventResolver > resolve
1 import { Injectable } from '@angular/core'
2 import { ActivatedRouteSnapshot, Resolve } from '@angular/router'
3 import { EventService } from './shared/event.service'
4 import { map } from 'rxjs/operators'
5
6 @Injectable()
7 export class EventResolver implements Resolve<any> {
8   constructor(private eventService:EventService) {
9
10  }
11
12   resolve(route: ActivatedRouteSnapshot) {
13     return this.eventService.getEvent(route.params['id'])
14   }
15 }
```

Edited event service `getEvent()` method and its call in `ngOnInit` in event-details component to resolve data once it comes from the server.

Made changes to `route.ts` to point to event-resolver, and added event-resolver in place of event activator in `index.ts` and app module.

`getEvent()` in event service:

```
getEvent(id:number):Observable <IEvent> {
  return this.http.get<IEvent>(`/api/events/${ id}`)
    .pipe(catchError(this.handleError<IEvent>('getEvents'))))
}
```

ngOnInit in event-details component.ts

```
ngOnInit() {  
  this.route.data.forEach((data) => {  
    this.event = data['event'];  
    this.addMode = false;  
  })  
}
```

Route.ts:

```
{path: 'events', component: EventsListComponent, resolve: {events: EventListResolver}},  
{path: 'events/:id', component: EventDetailsComponent, resolve: {event: EventResolver}},  
{path: 'events/session/new', component: CreateSessionComponent},
```

Now data is resolved as it comes from the server and all the functionalities we built earlier on work.

4. Using POST and PUT methods:

We implemented POST method in the event-service.ts file

```
saveEvent(event){  
  let options = { headers: new HttpHeaders({'Content-Type': 'application/json'})  
  return this.http.post<IEvent>('/api/events', event, options) // PUT method  
  .pipe(catchError(this.handleError<IEvent>('saveEvent')))  
}
```

We used it to saveEvent in the server. PUT method is usually used for updating events, but we don't have that functionality as we are using saveEvent to both save and update events.

We also made changes to where this event is called: in the event-details component, we subscribed it to listen to the server

```
saveNewSession (session:ISession) {  
  const nextId = Math.max.apply(null, this.event.sessions.map(s => s.id));  
  session.id = nextId + 1  
  this.event.sessions.push(session)  
  this.eventService.saveEvent(this.event).subscribe();  
  this.addMode = false  
}
```

Also update the saveEvent method call in create-event component.ts to subscribe to the server and update the server when an event is saved.

```

    saveEvent(formValues) {
      this.eventService.saveEvent(formValues).subscribe(() => {
        this.isDirty = false
        this.router.navigate(['/events'])
      });
    }
  }

```

5. Using QueryString parameters:

We implemented querystring parameters to implement the searchSession method in a simpler manner.

```

searchSessions(searchTerm: string): Observable<ISession[]> {
  return this.http.get<ISession[]>('/api/sessions/search?search=' + searchTerm)
    .pipe(catchError(this.handleError<ISession[]>('searchSessions')))
}

```

The searchSession is called from the nav-bar component.ts:

```

searchSessions(searchTerm) {
  this.eventService.searchSessions(searchTerm).subscribe(sessions => {
    this.foundSessions = sessions;
  });
}

```

The input parameters already fit and we subscribe it to the server to listen and search.

6. Using DELETE method:

We implemented the delete method in the voter service to add and delete votes:

```

@Injectable()
export class VoterService {
  constructor(private http: HttpClient) {}

  deleteVoter(eventId: number, session: ISession, voterName: string) {
    session.voters = session.voters.filter(voter => voter !== voterName);

    const url = `/api/events/${eventId}/sessions/${session.id}/voters/${voterName}`;
    this.http.delete(url)
      .pipe(catchError(this.handleError('addVoter')))
      .subscribe();
  }

  addVoter(eventId: number, session: ISession, voterName: string) {
    session.voters.push(voterName);

    const options = { headers: new HttpHeaders({'Content-Type': 'application/json'}) };
    const url = `/api/events/${eventId}/sessions/${session.id}/voters/${voterName}`;
    this.http.post(url, {}, options)
      .pipe(catchError(this.handleError('addVoter')))
      .subscribe();
  }
}

```

To make the eventId be recognized by the functions, we made edits to event-details.html where session-list component is called:

```
<session-list [eventId] = "event?.id" [filterBy]="filterBy" [sortBy]="sortBy" *ngIf="!addMode" [sessions]="event?.sessions"></session-list>  
<create-session *ngIf="addMode" (saveNewSession)="saveNewSession($event)" (cancelAddSession)="cancelAddSession()"></create-session>
```

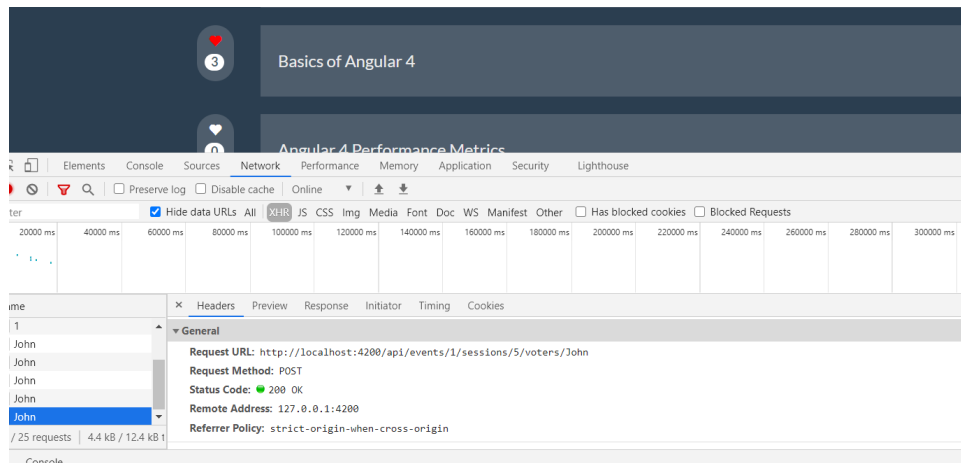
We added an eventId attribute to bind to the event.id and also added this as an input to the function inside session-list component.ts:

```
toggleVote (session:ISession){  
    if(this.userHasVoted(session)){  
        this.voterService.deleteVoter(this.eventId, session, this.auth.currentUser.userName);  
    } else {  
        this.voterService.addVoter(this.eventId, session, this.auth.currentUser.userName);  
    }  
    if(this.sortBy == 'votes')
```

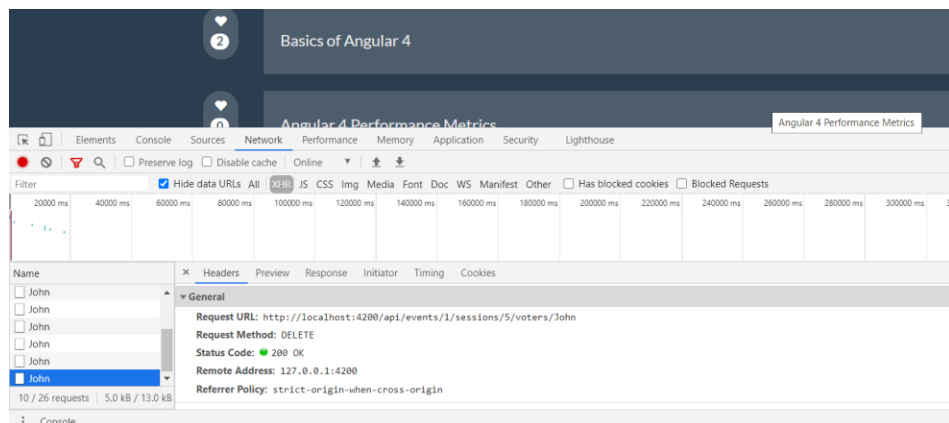
We declared eventId using @Input inside the session-list component to use it.

So the DELETE method was used in deleteVoter and POST method was used in addVoter, the output is shown in the console:

If we add a vote:



Once we remove that vote:



7. Integrating Authentication to the server:

In the auth.service.ts we made changes to loginUser :

```
@Injectable()
export class AuthService{

  constructor(private http:HttpClient) {}

  currentUser:IUser
  loginUser (userName:string, password: string){

    let loginInfo = { username: userName, password: password};
    let options = { headers: new HttpHeaders({ 'Content-Type': 'application/json'})};

    return this.http.post('/api/login', loginInfo, options)
      .pipe(tap(data => {
        this.currentUser = <IUser>data['user'];
      }))
      .pipe(catchError(err => {
        return of(false)
      })))
  }
}
```

Used a post method to send login credentials to server and to validate and send a response.

And then inside the login component.ts : to subscribe to receive a response and check whether invalid or not.

```
...mouseoverLogin
loginInvalid = false
constructor (private authService: AuthService, private router:Router){

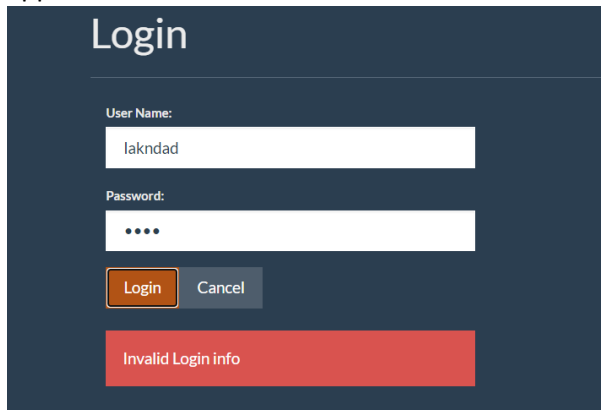
}

login(formvalues) {
  this.authService.loginUser(formvalues.userName, formvalues.password)
  .subscribe(resp => {
    if(!resp){
      this.loginInvalid = true;
    }
    else {
      this.router.navigate (['events'])
    }
  })
}
```

If response is set to invalid and it is received by login component.ts, then login component uses it as an ngIf to show some text:

```
<br>
<div *ngIf ='loginInvalid' class="alert alert-danger"> Invalid Login info</div>
</div>
```


The app now doesn't take in random values and only preset users can access the application:



The password hasn't been set, so any password will do, but only if the username is "Johnpapa" will the login form let the user enter.

8. Persisting Authentication status across refresh pages:

After every page refresh, the user has to login again, to fix this we added a `checkAuthenticationStatus` method to the `events-app.component.ts` (the root file)

```
14
15 })
16 export class EventsAppComponent {
17
18   constructor(private auth: AuthService) {}
19
20   ngOnInit () {
21     this.auth.checkAuthenticationStatus();
22   }
23
24 }
25
```

And declared the method in `AuthService`:

```
//-----
checkAuthenticationStatus() {
  this.http.get('/api/currentIdentity')
    .pipe(tap(data => {
      if(data instanceof Object) {
        this.currentUser = <IUser>data;
      }
    })))
    .subscribe(); // either subscribe or tap can be use here, so check
}
```

We used the GET method to get the user status and inform the browser that the user is still logged in. So now after every refresh, the user is still logged in.

9. Saving user data to server:

In the profile component, where we edit the user name and save, and we receive a toast message which says “ profile saved ”, we made a few changes where it listens to the server before it shows the toast message.

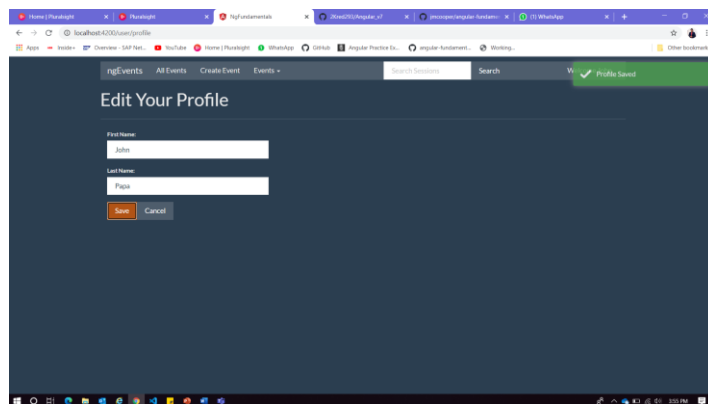
Also we added some code in Auth service in updateUser method to update the user name on the server using PUT method.

```
// updateUser(firstName:string, lastName:string){  
  this.currentUser.firstName = firstName  
  this.currentUser.lastName = lastName  
  
  let options = { headers: new HttpHeaders({ 'Content-Type': 'application/json'})};  
  
  return this.http.put(`/api/users/${this.currentUser.id}`, this.currentUser, options);  
}
```

In the saveProfile method in profile component, we made sure the app listens for the server before showing the toast. So we make it subscribe to the server and on any response, it shows the message.

```
}  
saveProfile(formValues) {  
  if(this.profileForm.valid){  
    this.authService.updateCurrentUser(formValues.firstName, formValues.lastName)  
      .subscribe(() => {  
        this.toastr.success('Profile Saved');  
      })  
  }  
}
```

Now when the username is updated and the name on the nav bar changes, the user is still logged , even after refreshes:



10. Implementing Logout:

First we created a logout button in the profile component.html

```
<button type="button" (click) = "cancel()" class="btn btn-default">Cancel</button>
<button type="button" (click) = "logout()" style="float:right" class="btn btn-warning">Logout</button>
</form>
</div>
</div>
```

Then we linked it to the profile.component.ts using the logout() function on click, where we implemented the logout function call:

```
logout() {
  this.authService.logout().subscribe(() => {
    this.router.navigate(['/user/login']);
  })
}
```

Here we subscribe the the logout method in the authService service and wait for the response and on response, navigate to the login page.

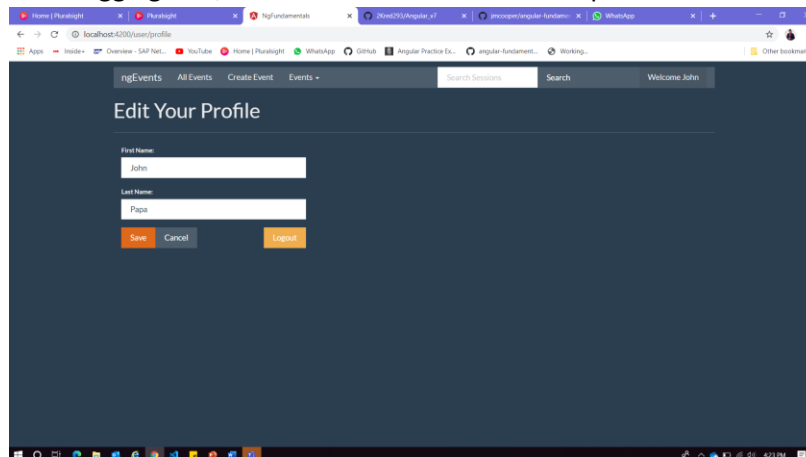
auth. service.ts:

```
logout() {
  this.currentUser = undefined;

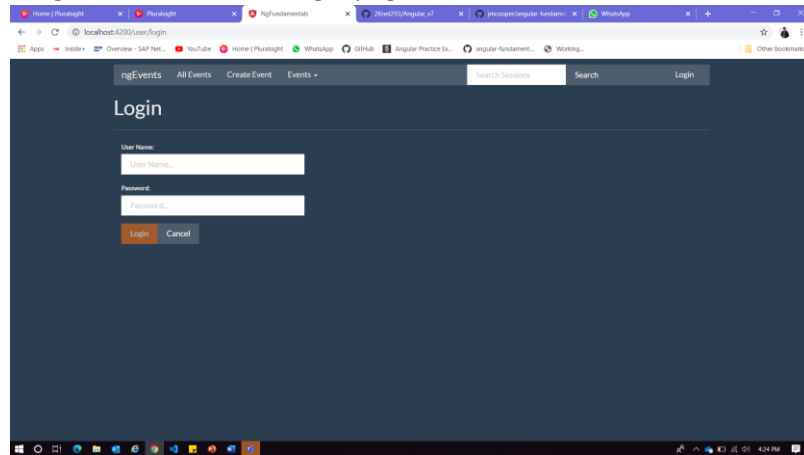
  let options = { headers: new HttpHeaders({ 'Content-Type': 'application/json' }) };
  return this.http.post('/api/logout', {}, options);
}
//-----
```

Here we informed the client that the user has logged out via the first line and the other lines of code inform the server that the user has logged out using the post method. Here we passed the body as empty as we need shown in the console during logout.

After logging out , the user name doesn't show up in the nav bar even after refresh.



After log out: redirected to login page:



Chapter 15: Unit Testing your Angular Code

Good Unit Tests

- Fast
- Cheap to Write
- Single State Change
- Assert 1 Thing
- Doesn't Cross Process Boundaries
- Reliable

Isolated vs Integrated Tests

Isolated Tests

- Test Class Only - No Template
- Constructed in Test
- Simple
- Best for Services & Pipes
- Appropriate for Components & Directives

Integrated Tests

- Test Class & Template
- Constructed by Framework
- Complex
- Mainly Used for Components & Directives
- Sometimes Used for Services
- Deep or Shallow

1. Installing Karma:

Karma is installed by the CLI at the start and it tests the code when the command “ng test” is run.

Any files with the **extension “spec.ts”** will be picked up by Karma and tested.

An example test file and its output are:

```
src > app > TS test.spec.ts > ...
1 describe('first test', () => {
2   it('should be true', () => {
3     expect(true).toBe(true);
4   })
5   it('should be false', () => {
6     expect(false).toBe(false);
7   })
8 })
```

test.spec.ts

```

at Proxyzonespec.push../node_modules/zone.js/dist/zone-testing.js.Proxyzonespec.  

Chrome 88.0.4324 (Windows 10.0.0): Executed 1 of 1 (1 FAILED) ERROR (0.02 secs / 0.232 secs)  

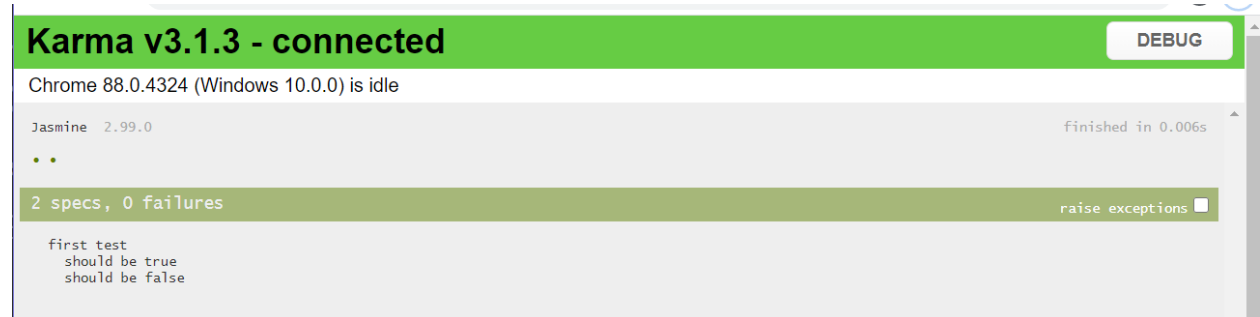
Chrome 88.0.4324 (Windows 10.0.0): Executed 2 of 2 SUCCESS (0.007 secs / 0.002 secs)  

TOTAL: 2 SUCCESS  

TOTAL: 2 SUCCESS

```

The cmd line output



Karma output in the browser

2. Unit Testing services (deleteVote) and Testing mock calls (addVote)

We use voter service as a test subject. So we create a file called voter.service.ts and run tests to check whether the add and deletevoter methods are working properly.

For example, the addVoter method:

```

describe('addVoter', () => {

  it('should call http.post with the right URL', () => {
    var session = { id: 6, voters: ['john']};
    mockHttp.post.and.returnValue(of(false));

    voterService.addVoter(3, <ISession>session, "joe");

    expect(mockHttp.post).toHaveBeenCalledWith('/api/events/3/sessions/6/voters/joe', {}, jasmine.any(Object))
  })
})

```

Inside the voter service: we can see the addVoter method:

```

addVoter(eventId:number ,session: ISession, voterName: string) {
  session.voters.push(voterName);

  const options = { headers: new HttpHeaders({'Content-Type': 'application/json'})};
  const url = `/api/events/${eventId}/sessions/${session.id}/voters/${voterName}`;
  this.http.post(url, {}, options)
    .pipe(catchError(this.handleError('addVoter')))
    .subscribe();
}

```

If we compare, we can see a few similarities of how a mock http call has been run .

Similarly, in the same file, we can see the mock call and test being done for deleteVoter method as well. If it's a successful test, the out in the cmd prompt shows:

```

Chrome 88.0.4324 (Windows 10.0.0): Executed 4 of 4 SUCCESS (0.008 secs / 0.003 secs)
TOTAL: 4 SUCCESS
TOTAL: 4 SUCCESS
Chrome 88.0.4324 (Windows 10.0.0): Executed 5 of 5 SUCCESS (0.01 secs / 0.005 secs)
TOTAL: 5 SUCCESS
TOTAL: 5 SUCCESS

```

It shows a success to all the tests which have been conducted.

3. Testing components with Isolated tests:

We tested a component here, session-list.component.ts and the ngOnChanges method:

```

ngOnChanges() {
  if(this.sessions) {
    this.filterSessions(this.filterBy);
    this.sortBy === 'name' ? this.visibleSessions.sort(sortByNameAsc) : this.visibleSessions.sort(sortByVotesDesc);
  }
}

```

We created a session-list.component.spec.ts to check the filter and sortBy functionalities:

```

import { SessionListComponent } from './session-list.component'
import { ISession } from '../shared/event.model'

describe('SessionListComponent', () => {
  let component: SessionListComponent;
  let mockAuthService, mockVoterService;

  beforeEach(() => {
    component = new SessionListComponent(mockAuthService, mockVoterService);
  })

  describe('ngOnChanges', () => {

    it('should filter the session correctly', () => {
      component.sessions = <ISession[]>[{name: 'session 1', level: 'intermediate'},
        {name: 'session 2', level: 'intermediate'},
        {name: 'session 3', level: 'beginner'}];
      component.filterBy = 'intermediate';
      component.sortBy = 'name';
      component.eventId = 3;

      component.ngOnChanges();

      expect(component.visibleSessions.length).toBe(2);
    })
  })
})

```

In the image we check the filterby functionality. So we set the filterBy component to “intermediate” and expect the length of the sorted array to be ‘2’. In describe, we create the inputs of the component and then call the function and then expect an output. If its successful, the cmd line shows it, else throws an error.

Chapter 16: Testing Angular components with Integrated tests

1. Setting up for Integrated tests:

In integrated tests, we don't create components, we create fixtures which are wrappers around the component. It gives access to a few features which we usually wouldn't have such as change detection or dependency injection etc

Difference between Unit tests and Integrated tests

Difference between Isolated tests and integrated tests

Difference between deep tests and Shallow tests

In Integrated testing, anything under `ngOnInit` might be called on its own while testing by `ngOnChanges` will be called only when declared in the `integrated.spec.ts` file.

Shallow integration tests are the tests which we use to test a component but not its children, here we wanted to test the session-list component, but not the upvote or collapsible-well components

`NO_ERRORS_SCHEMA` tells the compiler not to worry about the html elements which it doesn't recognize, it is used with shallow integration tests. So it doesn't run into errors when it doesn't recognize the upvote component for example

We renamed `session-list.component.ts` into `session-list.component.isolated.ts` and created `session-list.component.integrated.ts`

Chapter 17: Taking an Angular app to production

Linting, creating builds and deploying

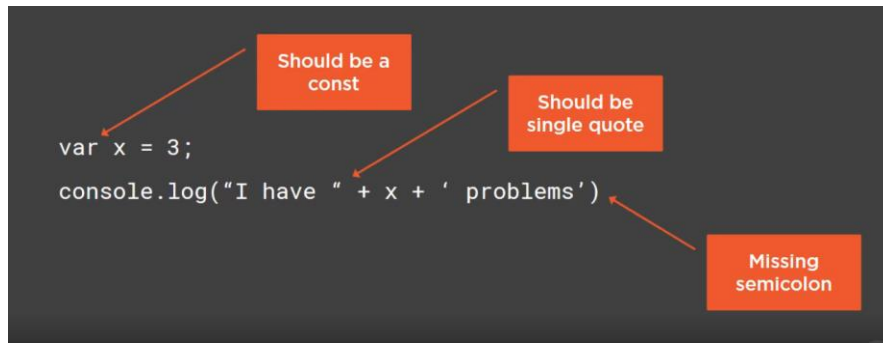
Linting: Pointing out and fixing small problems in the code (finding lints)

What Is Linting
Really?

Pointing out and fixing potential problems

Mostly coding style changes

- Missing semicolons
- Double vs. single quotes
- Long lines
- Etc.



Generates inconsistency and corrects errors for us

View -> extensions -> TSLint -> install = inside VSCode

From Command Line: "ng lint"

TSLint will find and fix simple errors

Going to Production:

ng build :

ng build
Command

Produces a deliverable code package

Production optimizations

- Development mode off
- Bundling
- Minification
- Tree shaking
- Dead code elimination
- Asset inlining
- Executes AOT

AOT
Benefits

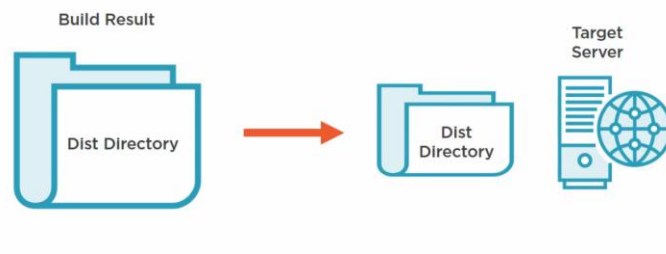
Faster rendering

Smaller Angular framework download

Detect template errors

Better security

Simple copy deployment:



Difference between
ng build and npm start – in dev mode vs prod mode