DZone

An Angular Autocomplete From UI to DB

by Sven Loesekann · Apr. 30, 18 · Web Dev Zone · Tutorial

The MovieManager project is used to manage my collection of movies and uses Angular and Spring Boot with PostgreSQL to do it.

The project is used to show how to build the autocomplete search box for the movie titles. The autocomplete box uses Bootstrap 4 for styling and Angular on the front-end and Spring Boot with PostgreSQL on the backend. The focus is on how to use native Angular/RxJS together with Spring Boot and a relational database.

The Front-End

The front-end uses Angular with RxJS and Bootstrap 4 for styling. Angular and RxJS have the features to create an input and connect it to the backend REST service.

The result of the service is then displayed and updated in a div below the input after every keystroke. Bootstrap provides the CSS classes to style the components.

The front-end has three parts:

- The template for the div with an input tag and the output div.
- The component class with form control to connect the input tag with the service method.
- The service class that makes the REST request to the server.

In the Search Component, I've used the following template:

```
<div class="container-fluid">
    <div class="row">
...
        <div class="col-3">
            <div class="input-group-prepend">
                <span class="input-group-text" id="searchMovieTitle">Searchfor Movie Title<
                <input type="text" class="form-control" placeholder="title" aria-describedb
            </div>
            <div *ngIf="movieTitle.value" class="searchList">
                <span *ngIf="moviesLoading">Loading</span>
                <a class="dropdown-item" [routerLink]="['/movie',movie.id]" *ngFor="let mov
```

```
12              </div>
13          </div>
14          ...
15      </div>
16  </div>
```

Lines 1-2 create a Bootstrap container of variable width and a row.

Line 4 creates the Bootstrap column for the Autocomplete feature.

Lines 5-8 create an input group that displays a styled input with a label and links the Angular form control `movieTitle` to the input.

Line 9 creates the result div if something is typed in the input.

Line 10 shows the text 'Loading' if the service is retrieving the movies from the server.

Line 11 shows router links for the movies that have been loaded from the server. The `routerLink` is the route to the Movie Component with the movie id. `*ngFor` iterates over the movies that the server sends and `async` means that the movies are an observable whose data is displayed on arrival. The text of the link is the movie title.

In the search component I've used the following class:

```
@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
  styleUrls: ['./search.component.scss']
})
export class SearchComponent implements OnInit {

  generes: Genere[];
  movieTitle = new FormControl();
  movies: Observable<Movie[]>;
  movieActor = new FormControl();
  actors: Observable<Actor[]>;
  importMovies: Movie[] = [];
  importMovieTitle = new FormControl();
  actorsLoading = false;
  moviesLoading = false;
  importMoviesLoading = false;
  showMenu = false;
  moviesByGenere: Movie[] = [];
  moviesByGenLoading = false;

  constructor(private actorService: ActorsService, private movieService: MoviesService, pri

  ngOnInit() {
      this.actors = this.movieActor.valueChanges
        .debounceTime(400)
        .distinctUntilChanged()
28          .do(() => this.actorsLoading = true)
29          .switchMap(name => this.actorService.findActorByName(name))
30          .do(() => this.actorsLoading = false);
31        this.movies = this.movieTitle.valueChanges
32          .debounceTime(400)
33          .distinctUntilChanged()
34          .do(() => this.moviesLoading = true)
35          .switchMap(title => this.movieService.findMovieByTitle(title))
36          .do(() => this.moviesLoading = false);
37          this.userService.allGeneres().subscribe(res => this.generes = res);
38    }
```

Lines 1-6 create the Search component with the annotation and `OnInit` interface.

Line 9 creates the `movieTitle` form control for the user input.

Line 10 declares the movie observables for the server result.

Line 16 creates the `moviesLoading` boolean to show the loading text.

Line 22 declares the constructor and gets the `MovieService` and others injected by Angular.

Line 24 declares the `ngOnInit` method to initialize form controls with the services and set the genres.

Line 31 links the `movieTitle` form control with the `valueChanges` observable and the result Observable `movies` .

Lines 32-33 add a 400 ms timeout between requests to the backend and only send requests if the input has changed.

Line 34 sets the boolean that shows the loading text in the template.

Line 35 uses `switchMap` to discard the running requests and sends a new request to the service.

Line 36 sets the boolean that sets the loading text to false.

This is the movie service that manages all the movie data:

```
@Injectable()
export class MoviesService {
    private _reqOptionsArgs = { headers: new HttpHeaders().set( 'Content-Type', 'applicatio

    constructor(private http: HttpClient) { }

  ...

    public findMovieByTitle(title: string) :Observable<Movie[]> {
        if(!title) {
            return Observable.of([]);
        }
        return this.http.get('/rest/movie/'+title, this._reqOptionsArgs).catch(error => {
            console.error( JSON.stringify( error ) );
15                return Observable.throw( error );
16                });
17        }
18
19        ...
20  }
```

Lines 1-2 creates the `MoviesService` with the annotation.

Line 3 creates the HTTP headers.

Line 5 is the constructor that gets the HttpClient that's injected.

Line 9 declares the method `findMoveByTitle` that returns an `Observable` movie array.

Lines 10-12 return an empty `Observable` if the title is empty.

Lines 13-16 send the HTTP request to the server, catch server errors, log them, and throw an error.

The Backend

The backend is done in Spring Boot with JPA and PostgreSQL. It uses Spring Boot features to serve a REST interface and a service to manage the repositories and create transactions around service calls. The transactions are needed to support the importation and deletion of functions for movies in the MovieManager.

The repository uses JPA to create a query that finds the movie titles of the movies of the user that contains (case sensitive) the title string, that the user typed.

The REST service is created in this class:

```
1   @RestController
2   @RequestMapping("rest/movie")
3   public class MovieController {
4      @Autowired
5      private MovieManagerService service;
6
7        ...
8
9      @RequestMapping(value="/{title}", method=RequestMethod.GET, produces = MediaType.APPLICA
10     public ResponseEntity<List<MovieDto>> getMovieSearch(@PathVariable("title") String title
11        List<MovieDto> movies = this.service.findMovie(titleStr);
12        return new ResponseEntity<List<MovieDto>>(movies, HttpStatus.OK);
13     }
14
15  ...
16  }
```

Lines 1-3 declare the `MovieController` as `Restcontroller` with the `RequestMapping`.

Lines 5-6 have the `MovieManagerService` injected. This manages the transactions and is the common service for movie data.

Lines 10-11 set up the `RequestMapping` and get the title value of the request string.

Line 12 reads the movie list from the `MovieManagerService` for this title string.

Line 13 wraps the movie list in a `ResponseEntity` and returns it.

The main movie service is created in this class:

```
1  @Transactional
2  @Service
3  public class MovieManagerService {
4     @Autowired
5     private CrudMovieRepository crudMovieRep;
6     @Autowired
7     private CrudCastRepository crudCastRep;
8     @Autowired
9     private CrudActorRepository crudActorRep;
10    @Autowired
11    private CrudGenereRepository crudGenereRep;
12    @Autowired
13    private CrudUserRepository crudUserRep;
14    @Autowired
15    private CustomRepository customRep;
16    @Autowired
17    private AppUserDetailsService auds;
18
19   ...
20
21    public List<MovieDto> findMovie(String title) {
22       List<MovieDto> result = this.customRep.findByTitle(title).stream()
23          .map(m -> Converter.convert(m)).collect(Collectors.toList());
24       return result;
25    }
26
27   ...
28 }
```

Lines 1-3 declare the `MovieManagerService` with the Service annotation to make the class injectable and the Transactional annotation wraps a transaction around the service requests. The transactions are needed for importing or deleting movies in the DB.

Lines 14-15 get the `CustomRepository` injected. That is the repo for all the non-CRUD DB requests.

Lines 21-22 declare the `findMovie` method that is used by the `RestController` and gets the movies for the title string from the `customRepo`.

Lines 23-24 convert the List of Movie entities in Movie Dtos that can be serialized by the `RestController` and returns the Dto List.

The custom repository is created in this class:

```
1  @Repository
2  public class CustomRepository {
3      @PersistenceContext
4      private EntityManager em;
5      @Autowired
6      private CrudUserRepository crudUserRep;
7
8      ...
9
10     public List<Movie> findByTitle(String title) {
11        User user = getCurrentUser();
12        List<Movie> results = em.createQuery("select e from Movie e join e.users u where e.t
13        return results;
14     }
15
16     ...
17 }
```

Line 1-2 declare the CustomRepository class and and add the Repository annotation to make the repository injectable.

Line 3-4 get the EntitiyManager of Jpa injected.

Line 5-6 get the CrudUserRepository injected to access the current logged in user. The current user is needed for the multi user support.

Line 10-11 declare the findByTitle method that is used in the MovieManagerService and get the current user.

Line 12-13 queries the EntityManager for the movies where the movie title contains the title string and the current user is the owner of the movie. For that the Movie and User entities need to be joined and then filtered by movie title and userId. Then the result is returned.

Summary

Angular and RxJs have the features to create an autocomplete in a few lines of code. Features like timeout between keystrokes, no double requests, and the discarding of stale results are supported by native features. The form control linked to the `valueChanges` observable does the updating of the results in the div.

DZone has a nice article that describes Angular and some of its features. A lot of the features like dependency injection, static typing, and annotations are familiar to Java devs and make Angular easy to learn. That makes Angular a very good framework for Java devs that are interested in front-end development.

The backend is a small REST Controller with a service and a repository with JPA. Spring Boot makes it possible to create the backend with few lines of code and a few annotations.

Golang vs Node.js

by Ruchi V · May 05, 18 · Web Dev Zone · Opinion

Node.js vs Golang has been the theme of many internet wars. Going by the numbers, the end to this debate seems nowhere in sight. However, we decided to take the plunge and do a bit of research on it. To come up with a more accurate verdict on Node.js vs Golang, we read up a bit on it. So here goes.

No search on the internet can begin without a search on Quora. Here's what we found; it points to how developers see Golang as an excellent replacement choice for Node.js. We have a completely unbiased view on this! Let's try and make sense of what is happening and why developers, as well as businesses, are starting to choose Golang over the popular Node.js.

Node.js Performance:

Firstly, anyone who has used both the languages would realize that CPU performance or memory bound tasks are slow with Node.js. Node.js is based on JavaScript, an interpreted language. Interpreted languages are slower than most compiled languages. With Node's dynamically typed nature, it does not reach the raw performance that Go can achieve. In contrast, Golang's performance is similar to C or C++(C is another compiled language). Only in cases of network communication or database interaction can Node equal Go in performance.

Concurrent and Scalable:

This is another aspect in Go vs Node where Golang beats most of the modern computing languages. Node is no exception to this rule. Golang is scalable due to its "goroutines." Goroutines help multiple threads perform simultaneously. Also, execution of parallel tasks is efficient and reliable. As Node.js is single-threaded, instructions are executed in sequence. This limits its ability during massive scaling when a lot of parallel processes are executed at the same time.

Language Maturity:

Any comparison between the two languages would also be incomplete without a look at maturity. Golang is quite robust and mature for its age, while for Node the changing API becomes a cause for API problems for developers who are writing and using Node modules.

To be fair to both the languages, it is not as though Node.js is going out of "business" anytime soon, but, when it comes to developing "business" solutions, Golang is the best choice. Golang's performance is lightning fast, its goroutines allow for great scalability and concurrency, and it helps build way more robust applications. When you consider Node.js vs Golang, Golang is the wiser choice. Thus the verdict is out on Go vs Node.

Topics: GOLANG, NODE.JS, WEB DEV, CONCURRENCY, SCALABILITY