Reading Users

This lesson will show you how to implement the Reading Users operation. It goes over all four aspects of the implementation and includes HTML, typescript, UserService, and back-end code.

WE'LL COVER THE FOLLOWING

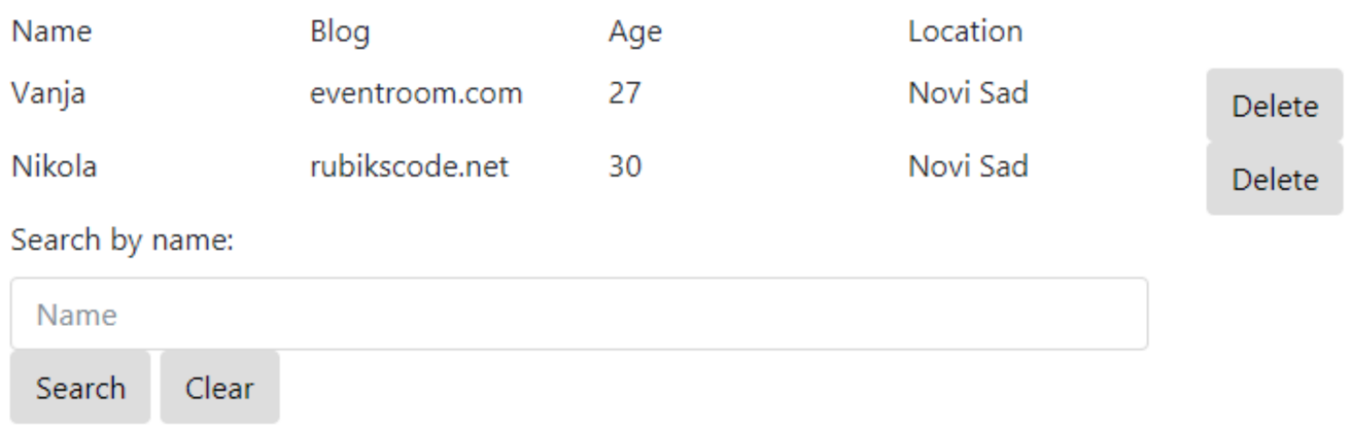
Let's follow the pattern established in the previous lesson and make changes to the HTML.

We want to analyze the implementation of reading users from the database and display them in the browser.

There are two situations we want to cover:

- reading of all users
- filtering users by some criteria (search option).

Basically, this section of the screen:



Name	Blog	Age	Location	
Vanja	eventroom.com	27	Novi Sad	Delete
Nikola	rubikscode.net	30	Novi Sad	Delete

Search by name:

Reading Users: HTML Code

Here is the HTML:

```
<div class="col-sm-8">
        <div class="row">
                <div class="col-sm-2">
                    <label for="Name">Name</label>
                </div>
                <div class="col-sm-2">
                    <label for="Blog">Blog</label>
                </div>
                <div class="col-sm-2">
                    <label for="Age">Age</label>
                </div>
                <div class="col-sm-2">
                    <label for="Location">Location</label>
                </div>
        </div>
        <div class="row" *ngFor="let user of users" (click)="setEditUser(user)">
            <div class="col-sm-2">
                {{user.name}}
            </div>
            <div class="col-sm-2">
                {{user.blog}}
            </div>
            <div class="col-sm-2">
                {{user.age}}
            </div>
            <div class="col-sm-2">
                {{user.location}}
            </div>
        </div>

    <div class="row">
        <div class="col-sm-8">
            <label for="searchByName">Search by name:</label>
            <input class="form-control" type="text" placeholder="Name" [(ngModel)]="searchCri
            <button class="btn btn-default" (click)="getUsers()">Search</button>
            <button class="btn btn-default" (click)="clearSearch()">Clear</button>
        </div>
    <div>
```

/mean_frontend/src/app/users/users.component.html

`*ngFor`

Pay attention to the `*ngFor` attribute of the *second* `div` (**line 16**).

This attribute allows us to iterate through all of the elements of some *array* or *list* and create the *DOM* element for it.

We are iterating through the `users` array, which we mentioned in the previous lesson, and creating a row for it.

Reading Users: TypeScript Code

Take a look at the code that accompanies this in the TypeScript:

```
import { Component, OnInit, ViewEncapsulation  } from '@angular/core';
import { User } from '../model/user';
import { UserService} from '../services/users.service';

@Component({
  selector: 'users',
  templateUrl: './users.component.html',
})
export class UserComponent implements OnInit {

  users: User[];
  searchCriteria: string;

  constructor(
    private userService: UserService
  ) { }

  ngOnInit() {
    this.searchCriteria = '';
    this.getUsers();
  }

  getUsers(){
    this.userService.getUsers(this.searchCriteria)
    .subscribe(
      data => {
         this.users = [];
         data.forEach(
           element => {
             var newUser = new User(element._id,
                                   element.name,
                                   element.age,
                                   element.location,
                                   element.blog);
             this.users.push(newUser);
```

```
          })
      })
    }
     clearSearch(){
      this.searchCriteria = '';
      this.getUsers();
    }
  }
```

/mean_frontend/src/app/users/users.component.ts

Explanation

Note the usage of the `searchCriteria` field (**line 12**) which is used for *search* functionality. This is filtered information that we will pass to our backend.

Based on this information, the back-end will either return all users from the database, or just the ones that match certain conditions.

That is why, when the initialization of this component is done, the value of this field has been set to *empty string* (**line 19**), and the `getUsers()` function has been called (**line 24**). This is the function in charge of passing information down to the service.

Now, this function does one more important task; it processes data received from the back-end and stores them in the `users` array (**lines 28-35**).

To sum it up, once this component has been initialized:

- A request with empty conditions will be passed.
- In return it will send all users from the database.
- Once the information reaches the component, it will fill its internal `users` array with it.
- Then, for each element in this array, the appropriate DOM element will be created.

Reading Users: Service Implementation

Next on our list is service implementation (i.e., implementation of the web API call).

Here is the section of code that covers this feature:

```
@Injectable()
export class UserService {


    constructor(private http: Http) {
    }

    getUsers(searchCriteria:any) : Observable<User[]>{
        let params: URLSearchParams = new URLSearchParams();
        params.set('name', searchCriteria);

        return this.http.get("http://localhost:3000/getUsers", { search: params })
                .map((res:any) => {
                    return res.json();
                })
                .catch((error:any) => {
                    return Observable.throw(error.json ? error.json().error : error || 'Serve
                });
    }
}
```

/mean_frontend/src/app/services/users.service.ts

Once again, we used the *HTTP Angular service*, but this time, we have sent a `GET` request with conditions for *users* search, if there are any.

Reading Users: Backend Code

On the other end, the back-end implementation also handles this request:

```
var router = express.Router();

router.get('/getUsers', function(req, res, next) {
  var searchQuery = {};

  if(req.query.name)
    searchQuery = { name: req.query.name };

  User.find(searchQuery, function(err, users){
    if (err) {
      res.status(400);
      res.send();
    }

    console.log("returning all the users.");
    res.send(users);
  })
});
```

/mean_backend/routes/index.js

Explanation

Same as before, we have harnessed the features that `Mongoose` gave us.

We used the `find()` function of the `Users` model (**line 9**), which receives the *filter object* as the first parameter.

We are only implementing "filtering by `name`" in this case, but you get the picture of how this implementation could be extended to support more features.

Implementation

Now, let’s execute the code for the *Reading users* operation, below:

```
#!/usr/bin/env node

/**
 * Module dependencies.
 */

var app = require('../app');
var debug = require('debug')('mongodbnode:server');
var http = require('http');
var mongoose = require('mongoose');

var mongoDB = 'mongodb://127.0.0.1/blog';
mongoose.connect(mongoDB, {
  useMongoClient: true
});

//Get the default connection
var db = mongoose.connection;

//Bind connection to error event (to get notification of connection errors)
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

/**
 * Get port from environment and store in Express.
 */

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

/**
 * Create HTTP server.
 */

var server = http.createServer(app);

/**
 * Listen on provided port, on all network interfaces.
 */

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);

/**

 * Normalize a port into a number, string, or false.
 */

function normalizePort(val) {
  var port = parseInt(val, 10);

  if (isNaN(port)) {
    // named pipe
    return val;
  }

  if (port >= 0) {
    // port number
    return port;
  }

  return false;
}

/**
 * Event listener for HTTP server "error" event.
 */

function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }

  var bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;

  // handle specific listen errors with friendly messages
  switch (error.code) {
    case 'EACCES':
      console.error(bind + ' requires elevated privileges');
      process.exit(1);
      break;
    case 'EADDRINUSE':
      console.error(bind + ' is already in use');
      process.exit(1);
      break;
    default:
      throw error;
  }
}

/**
 * Event listener for HTTP server "listening" event.
 */

function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  debug('Listening on ' + bind);
}
```

You will be able to view all the entries of the database now.

> Try inserting more entries into the database and then search for them using the *name* property.

This was the complete implementation, including the front-end and the back-end of the *Reading Users* operation. In the next lesson, we will discuss the *Updating Users* operation.