Updating Users

This lesson shows you how to implement the Updating Users operation. It goes over all four aspects of the implementation and includes html, typescript, UserService, and back-end code.

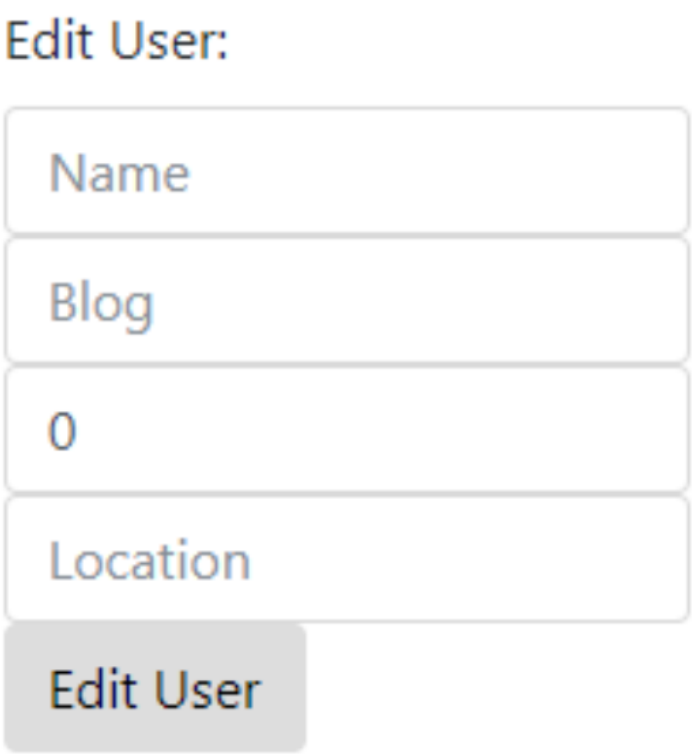
WE'LL COVER THE FOLLOWING

While implementing the reading users feature, you may have noticed that each `div` we created for every element of `users` array had a (`click`) attribute assigned to it.

It defined that the function `setEditUser()` will be called once a certain row has been clicked.

`setEditUser()`

This function will map that information from clicked row to the field `editUser`. This information is mapped to the next section of the screen, using `[(ngModel)]`:



Updating Users: HTML Code

The HTML behind this section looks like this:

```
<div class="form-group col-sm-2">
    <label for="editUser">Edit User:</label>
    <input class="form-control" type="text" placeholder="Name" [(ngModel)]="this.editUser.nam
    <input class="form-control" type="text" placeholder="Blog" [(ngModel)]="this.editUser.blo
    <input class="form-control" type="text" placeholder="Age" [(ngModel)]="this.editUser.age"
    <input class="form-control" type="text" placeholder="Location" [(ngModel)]="this.editUser
    <button class="btn btn-default" (click)="updateUser()"> Edit User </button>
</div>
```

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

The above code is very similar to the section of the code that handles the inserting of the users.

The difference is that this section maps its data to the `editUser` field (**lines 2-6**), and the button is wired to call the `updateUser()` function (**line** 7).

Updating Users: TypeScript Code

In the background it is managed like this:

```
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 {

  editUser: User;

  constructor(
    private userService: UserService
  ) { }

  ngOnInit() {
    this.editUser = User.CreateDefault();
  }

  updateUser(user:User) {
    this.userService
    .updateUser(this.editUser)
    .subscribe(
      data => {
         var index = this.users.findIndex(item => item._id === this.editUser._id);
         this.users[index] = this.editUser;
         this.editUser = User.CreateDefault();

         console.log("Added user.");
      }
    )
  }

  setEditUser(user: User){
    this.editUser = new User(user._id, user.name, user.age, user.location, user.blog);
  }
}
```

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

As expected:

- `setEditUser()` **(line 35)** creates a `new User` object based on information in the clicked row.
- `updateUser()` **(line 21)** calls the *service* function and passes information gathered from the HTML.

Updating Users: Service Implementation

The Service handles this information in a similar manner as before, by calling

the *HTTP POST* request.

```
@Injectable()
export class UserService {

    constructor(private http: Http) {
    }

    updateUser(user:User): Observable<any>{
        return this.http.post("http://localhost:3000/updateUser", user)
            .map((res:any) => {
                return res.json();
            })
            .catch((error:any) => {
                return Observable.throw(error.json ? error.json().error : error || 'Server er
            });
    }
}
```

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

Updating Users: Backend Code

In the same manner, as for previous operations, the back-end will handle the information that is passed:

```
router.post('/updateUser', function(req, res, next) {
  var user = new User(req.body);

  User.update({_id : user.id}, user, function(err) {
    if (err) {
      console.log("not updated!");
      res.status(400);
      res.send();
    }

    console.log("updated!");
    res.send({status: 'ok'});
  });
});
```

/mean_backend/routes/index.js

Implementation

Now, let's execute the code for the *Updating users* operation:

```
#!/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);
}
```

Updating a User

When you ran the code above you must have seen the *Edit User* option. Now, follow these steps to edit a user:

- Insert a new User.
- In the case of more than one entry, use the search option to find the entry to edit; or, simply scroll through to find it.
- Once the entry for that user is found, click on its row.
- You should see the contents of that row appear in the Edit User fields.
- Change whichever field you want to change.
- Click the Edit user option.

- Look up the entry again, you should be able to see the change you made.

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