Inserting Users

This lesson explains how to insert a user into the database. It focuses on the UserService class and the backend implementation of the insertion operation.

WE'LL COVER THE FOLLOWING

In the previous lesson, we looked at the typescript and HTML files. Now, let's take a look at the implementation of `UserService` and the back-end implementation for the insertion of a user.

`UserService`

This class is in charge of communication with the *back-end* through the *Web API* and, because of that, it has to use *Angular's HTTP* service.

Example

Here is the code in the */mean_frontend/src/app/services/users.service.ts* file:

```
import { Injectable } from '@angular/core';
import { Http, URLSearchParams } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import { Subject } from 'rxjs/Subject';
import { User } from '../model/user';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';

@Injectable()
export class UserService {

    constructor(private http: Http) {
    }
```

```
    insertNewUser(user:User): Observable<any>{
        return this.http.post("http://localhost:3000/insertUser", 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

So, this class sends an `HTTP POST` request on *`http://localhost:3000/insertUser`*, and passes the `user` object that was passed from the `UserComponent`. This is where our back-end will listen and insert the data into the database.

Insert User: Backend Code

Let's jump to the back-end API implementation in *index.js* in the *routes* folder.

```
var express = require('express');
var User = require('../model/user');
var mongoose = require('mongoose');

var router = express.Router();

router.post('/insertUser', function(req, res, next) {
  var newUser = new User(req.body);
  newUser._id = mongoose.Types.ObjectId();

  newUser.save(function(err) {
    if (err) {
      console.log("not saved!");
      res.status(400);
      res.send();
    }

    console.log("saved!");
    res.send({ id : newUser._id });
  });
});
```

/mean_backend/routes/index.js

Explanation

Here:

- We used the `router` feature of the `express` web framework (**line 1**).
- Then, we defined the function that expects the `HTTP POST` request on the `/insertUser` link (**line** 7).
- Then, we make full use of `mongoose` to do the following:

First:

- We use the data passed in the body of `POST` request to create a new object using the `User` model (**line 8**).
- Then that data is stored in the database, using the `save()` function of the created object (**line 11**).

The process we just went through will be our pattern for the next features. We can break it down into four steps:

- Implement changes necessary in HTML of the component
- Implement changes necessary in TypeScript of the component
- Implement the call to the REST API in the service
- Implement REST API on the backend

Implementation

Now, let's execute the code for *Inserting the 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);
}
```

> **Note:** You'll see the message, "Your app can be found at" with a URL given at the bottom of the above coding widget. If you click on it before running the code, the error, "Your app refused to connect." will show up as the app has not yet been set up.

The application will appear in the output tab shortly after you click the RUN button. You will be able to view the "saved" message on the terminal once you insert a user into the database.

At this point, you won't be able to view the user that you inserted into the database. In order to learn how to read from the database, you will have to go to the next lesson which will teach you about *Reading Users* operation.