# HOW TO BUILD A USER AUTHENTICATED TO-DO LIST

We're going to build a mini-application that includes an authentication module and a module where we'll be able to add and change the status of a to-do list.

## Technologies*

HTML5
CSS3
AngularJS v1.3.15
MongoDB v2.0.43
NodeJS v0.12.6
Express v4.13.3
Json Body Parser v1.14.0

\* this tutorial is color coded to help you identify the technologies we're referring to.

# REGISTRATION

## Front-end Stuff

Inside our main directory, we're going to start by creating an HTML file which we'll use as our home page ("index.html").

This page will have our registration form with 3 input fields: name, password and password confirmation; and a submit type button.

```html
<form>
        <!-- username -->
        <label for="username">username: </label>
        <input type="text" id="username" name="username" placeholder="john_smith"/>

        <!-- password -->
        <label for="password">password: </label>
        <input type="password" id="password" name='password' placeholder='***********'/>

        <!-- repeat password -->
        <label for="passwordconfirm">repeat password: </label>
        <input type="password" id="passwordconfirm" name='confirm password'
placeholder='***********'/>

        <!-- submit registration form -->
        <button type="submit">Register</button>
</form>
```

Below our registration form, we're going to include a div with a button that will take us to the login page ("login.html"), which we'll build later.

```html
<div class="login">
        <p class="login-message">Already registered?</p>
        <a href="login.html">
                <button type="button" name="login">Login</button>
        </a>
</div>
```

At this point, we're going to start using AngularJS.

I'll include Angular using a Google hosted library (you can find it here) but you can also download and host the script yourself:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.3.15/angular.min.js"></script>
```

To start using Angular, we'll need to create our app.js file and link it to our HTML file:

```
<script src="app.js"></script>
```

We'll also need to use the ng-app directive to designate the root element of our application:

```
<html ng-app="myapp">
```

Inside our app.js file, we'll need to create a controller, which I'm going to call "RegistrationController":

```
(function() {
    var app = angular.module('myapp', []);
    app.controller('RegistrationController', function($scope) {
    });
})();
```

Then, inside our HTML file, we'll have to link our controller to our form element using the ng-controller directive, in order to define its scope:

```
<form ng-controller="RegistrationController as registration">
```

Note: scopes are very important in Angular, since they enable data coherence and the separation of logic and concepts, within a given controller.

We'll use the ng-model directive on every input field, to save the data inside the scope of this controller.

We'll also use the "required" attribute to make sure everything is filled out, and we'll specify the length of our password using the "pattern" attribute.

```
<input type="text"
       id="username"
       name="username"
       ng-model="username"
       placeholder="john_smith"
required/>

<input type="password"
       id="password"
       name='password'
       ng-model="p"
       placeholder='***********'
       pattern=".{5,15}"
       title="The password must contain between 5-15 characters."
required/>

<input type="password"
       id="passwordconfirm"
       name='confirm_password'
       ng-model="pconfirm"
       placeholder='***********'
required/>
```

In order to disable the "Register" button whenever the passwords don't match, we'll use the `ng-disabled` directive.

This directive disables an element if a given condition is met. In this case, the condition is going to be the "passwordsMatch" function, which we'll need to create in our app.js file, inside our controller.

```
<button type="submit" ng-disabled="!passwordsMatch()">Register</button>
```

This function returns "true" only if the p value (`ng-model`="p") matches the pconfirm value (`ng-model`="pconfirm").

```
app.controller('RegistrationController', function($scope) {
        $scope.passwordsMatch = function() {
            if ($scope.p === $scope.pconfirm) {
                return true;
            } else {
                return false;
            }
        };
});
```

We'll use the `ng-show` directive with the same condition (expression), in order to show an error message whenever the passwords don't match. We'll put this p element below the Register button.

```
<p ng-show="!passwordsMatch()">
        The passwords don't match
</p>
```

Finally, we're going to add the `ng-submit` directive to our form element, which will trigger the "createUser" function whenever we submit the form.

```
<form ng-controller="RegistrationController as registration" ng-submit="createUser()">
```

Again, we'll need to define this function in our app.js file, inside our controller, but we'll leave this part for later, once our webservice is set up.

At the end of this process, this is what our form looks like when we access the index page:
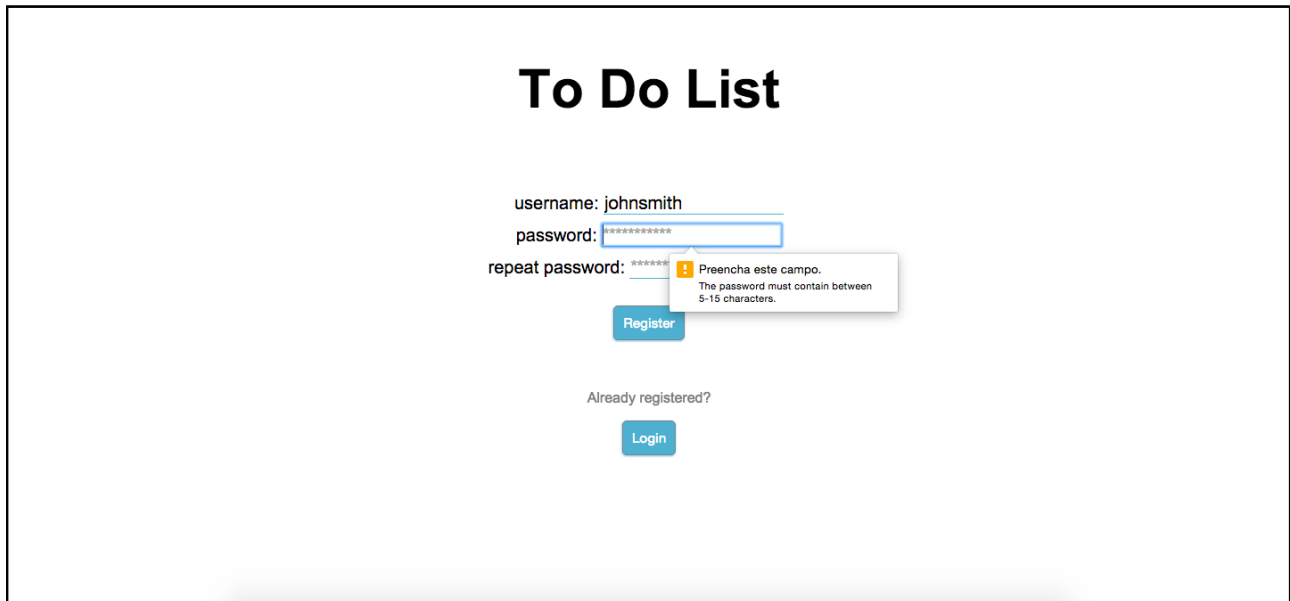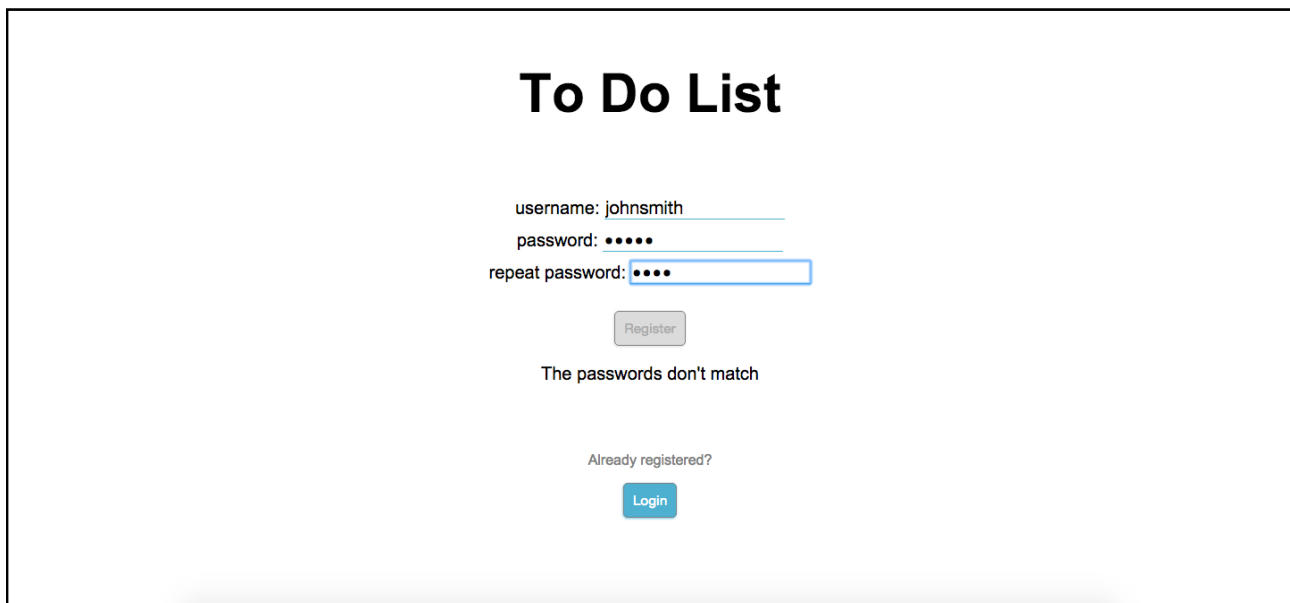
# To Do List

username: john_smith
password: ***********
repeat password: ***********

Register

Already registered?

Login

If we try to submit the form without filling out the password fields, we'll get this message:



If our passwords don't match, our Register button will be disabled and the error message we've created will appear below the button:



Now that we're sure the form will only be submitted if every field is filled out and if the passwords match, we can move on to setting up the database and creating our webservice.

Back-end Stuff

For this task, we're going to use MongoDB, Node.js, Express and Json Body Parse.

Node.js will be the "engine" that allows us to run the rest of the applications using Javascript, namely:

1. MongoDB library for Node —> allows us to run Mongo commands directly on Node, using Javascript;
2. Express —> provides a set of features for webservices. In simple terms, it creates a "listener" that waits in a specific port for our HTTP "post" and "get" methods, allowing us to create a webservice;
3. Json Body Parser —> allows us to interpret simple text as .json objects.

## Setting up our database

You can download MongoDB here and you can use this guide to help you install it.
Once Mongo is installed, we'll try to start the MongoDB daemon service with this command:

```
$ mongod
```

If you get a "permission denied" error, create a directory named "data", with a "db" directory inside, on your root directory (use the -p option). Then, you'll need to change the ownership of these directories to your user:

```
$ sudo mkdir -p /data/db
$ sudo chown [username] /data
$ sudo chown [username] /data/db
```

Once MongoDB is up and running, we're going to open up another console window and access the Mongo database, with this command:

```
$ mongo
```

In the same console window, we can now create a new database called "todos".

```
$ use todos
```

Inside our database, we'll create our collections – they're the equivalent of tables in Mongo. We'll create a "users" and a "todoItems" collection:

```
$ db.createCollection('users')
$ db.createCollection('todoItems')
```

Now, for the Node.js.
You can download it here.
Once Node is installed, we can install the rest of the applications mentioned above.

Installing the rest of the applications

We're going to start by creating a new directory ("server") inside our project's main directory.
For these next steps, we're going to use NPM:

First, we'll start a new NodeJS project inside our "server" directory using:

```
$ npm init
```

This will create a package.json file.
To install the rest of the applications and save these configs in this file, we'll use this command:

```
$ npm install mongodb express body-parser --save
```

Note: we can check if everything was correctly installed by reading the file:

```
$ cat package.json
```

The result should be something like this:

```
$ {
  "name": "server",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "[author's name]",
  "license": "ISC",
  "dependencies": {
      "body-parser": "^1.14.0",
      "express": "^4.13.3",
      "mongodb": "^2.0.43"
  },
  "devDependencies": {},
  "description": ""
}
```

## Creating our Node logic

Once everything is installed, we can start creating the Node.js logic:

We'll need to create a new js file inside our server directory ("server.js") and we'll begin by "importing" our dependencies using the following code:

```
var express = require('express');
var MongoClient = require('mongodb').MongoClient;
var bodyParser = require('body-parser');
var app = express();

app.use(bodyParser.json());
```

We'll have to add the following code to enable requests from multiple origins:

```
app.use(function(req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH,
DELETE');
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,content-type');
    res.setHeader('Access-Control-Allow-Credentials', true);
    next();
});
```

We'll also need to add a new variable ("server"), that'll establish the host and port where Express will be "listening":

```
var server = app.listen(3000, function() {
    var host = server.address().address;
    var port = server.address().port;
    console.log('Example app listening at http://%s:%s', host, port);
});
```

Next, we'll create 2 variables ("usersCollection" and "todosCollection") that match the 2 collections we've created inside our "todos" database.

```
var usersCollection, todosCollection;
MongoClient.connect('mongodb://localhost:27017/todos', function(err, db) {
    usersCollection = db.collection('users');
    todosCollection = db.collection('todoItems');
});
```

After that, we'll need to define the post method that will check if the user already exists in our 'usersCollection', and if not, will create a new user:

```javascript
app.post('/users/new', function(req, res) {
    var username = req.body.username,
        pwd = req.body.p;
    usersCollection.find({
        username: username
    }).toArray(function(err, items) {
        if (err) {
            res.status(500).send({
                error: err
            });
        } else {
            var userExists = items.length > 0;
            if (!userExists) {
                usersCollection.insert({
                    username: username,
                    pwd: pwd
                }, function(err, result) {
                    res.json(result);
                });
            } else {
                res.status(500).send({
                    error: 'User exists'
                });
            }
        }
    });
});
```

Note: don't forget to run this command whenever you want to try out your demo:

```
$ node server.js
```

## Front-end Stuff

So, now we'll go back to our app.js file and, just below our "passwordsMatch" function, we're going to define the "createUser" function.
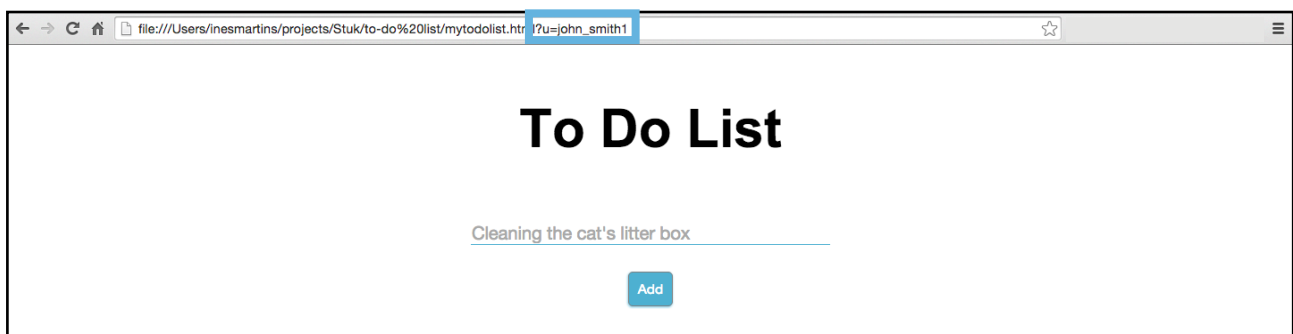
We'll begin by creating an HTTP post method for our server.
In the request body, we'll send a .json object with the user's data. This data will include the user's username and password, which we'll get from the values we'd stored with our ng-model directives.

Note that we've added the $window parameter to our Registration Controller and we've established that, in case of success, our user is going to be sent to a new page "mytodolist.html", which will have a "u" parameter in the URL, that will pass our user's username.

```
app.controller('RegistrationController', function($scope, $http, $window) {
...
        $scope.createUser = function() {
            $http.post("http://127.0.0.1:3000/users/new", {
                username: $scope.username,
                p: $scope.p
            }).success(function(data, status) {
                $window.location.href = 'mytodolist.html?u=' + $scope.username;
            }).error(function(data, status) {
                console.log("user already exists: ", data);
                $scope.errorMessage = data;
            });
        };
    });
```

We'll use this parameter later, in order to link the to-dos to a specific user.



Next, inside our HTML file, we'll create a new DIV that's going to present the error message we've defined in our server.js file, whenever the user tries to enter a username that already exists in our database.

```
<div class="error-message">
        <p ng-show="errorMessage !== undefined">{{ errorMessage.error }}</p>
</div>
```

The result should look something like this:



If the user doesn't exist, a new entry will be created in our collection, and the user will be sent to a new page ("mytodolist.html"), as we've mentioned before.

# LOGIN

The login form will be very similar to the registration form, except we'll only need to verify that the username and password match the data in our database.

So, starting with the same basic code, we'll need to change the AngularJS directives defined in our form element.

To do this, we'll need to create a new controller ("LoginController") and a new function ("loginUser") inside our app.js file:

```
app.controller('LoginController', function($scope, $http, $window) {
        $scope.loginUser = function() {};
});
```

Then, we'll be able to modify the ng-controller and ng-submit directives accordingly:

```
<form ng-controller="LoginController as login" ng-submit="loginUser()">
```

We'll keep the same structure for the "username" and "password" input fields and we'll remove the "repeat password" field and the ng-disabled directive from the button. We'll also need to change the button's text to "Login" instead of "Register".

```
<!-- username —>
<label for="username">username: </label>
<input type="text" id="username" name="username" ng-model="username"
placeholder="john_smith" required/>

<!-- password -->
<label for="password">password: </label>
<input type="password" id="password" name='password' ng-model="p"
placeholder='***********' required/>

<!-- button -->
<button class="button-submit" type="submit">Login</button>
```

We can remove the message that was displayed whenever the passwords didn't match, but we can keep the second error message, the one that signaled that the user already existed.
Once we change the content of this message within our server.js file, the text will be updated automatically on our interface.

```
<div class="error-message">
        <p ng-show="errorMessage !== undefined">{{ errorMessage.error }}</p>
</div>
```

Next, in our app.js, we'll need to define the loginUser function inside the LoginController. Once again, we'll create an HTTP post method for our server that will send, in the request body, a .json object with the user's username and password.

We'll also establish the same success function – the user will be sent to the mytodolist.html page, and the URL will contain a "u" parameter with the user's username.

```
app.controller('LoginController', function($scope, $http, $window) {
        $scope.loginUser = function() {
                $http.post("http://127.0.0.1:3000/users/login", {
                        username: $scope.username,
                        p: $scope.p
                }).success(function(data, status) {
                        $window.location.href = 'mytodolist.html?u=' + $scope.username;
                }).error(function(data, status) {
                        $scope.errorMessage = data;
                });
        };
});
```

## Back-end Stuff

Inside our server.js file, we'll need to add the following code, which will check if the username exists inside our "users" collection and if so, validates if the password matches the user's password.

```
app.post('/users/login', function(req, res) {
    var username = req.body.username,
        pwd = req.body.p;
    usersCollection.findOne({
        username: username
    }, function(err, userObj) {
        if (err) {
            res.status(500).send({
                error: 'error occurred'
            });
        } else {
            if (userObj) {
                if (userObj.pwd === pwd)
                    res.send('login successful');
                else
                    res.status(403).send({
                        error: 'Wrong user/password'
                    });
            } else {
                res.status(404).send({
                    error: 'User not found'
                });
            }
        }
    });
});
```

Now, whenever we try to login with a nonexistent user or with the wrong password we'll see this message:

# To Do List

username: johnsmith

password: ●●●●●●

Login

**User not found**

As we've mentioned before, when the login is successful, the user is sent to the to-dos'
management module ("mytodolist.html"), which we're now going to create.

# THE TO-DOs PAGE

This page will follow the same basic structure as the previous ones.
Once again, inside our app.js file, we'll have to create a new controller ("TodoController") with a new function ("submitNew"), which we'll define later:

```
app.controller('TodoController', function($scope, $http, $window) {
        $scope.submitNew = function() {
        };
});
```

Except this time, before our submitNew function, we're going to add a new function which will be responsible for the parsing of our current URL, in order to find the "u" parameter, which contains the user's username.

```
app.controller('TodoController', function($scope, $http, $window) {
        /* Long version
        var url = $window.location.href;
        var urlSplits = usr.split('?');
        var userParam = urlSplits[1];
        var userParamSplits = userParam.split('=');
        $scope.user = userParamSplits[1];
        */
        /* Short version */
        $scope.user = $window.location.href.split('?')[1].split('=')[1];
});
```

Then, we'll have to modify the HTML accordingly. This time, instead of using our directives directly on the form element, we're going to create a div which will wrap all our elements.

```
<div ng-controller="TodoController as todo" ng-submit="submitNew()">
```

The form will be a bit more simple: we'll only need a text input and a submit type button.

Again, we're going to use the ng-model directive in our text input, to save the values inside the scope of the controller.

```
<input id="new-todo" type="text" ng-model="todoEntry" placeholder="Cleaning the cat's litter
box" required>
</input>

<button type="submit" name="add">Add</button>
```

We're also going to create a new div, just below our form, where we can later display the new to-dos.

This div will have a checkbox and a label which corresponds to the to-dos' "description" inside the "todoItems" collection.

We'll use the ng-repeat directive to iterate through every to-do in our collection and display the description for each one.

```
<div class="todos" ng-repeat="todo in todos">
        <input id="checkbox_{{$index}}" type="checkbox" ng-model="todo.checked"></
input><label for="checkbox_{{$index}}">{{ todo.description }}</label>
</div>
```

Note:
I've added the _{{$index}} at the end of the checkbox id to make sure that, when the ng-repeat directive runs through each of the to-dos in our collection, the checkbox elements are always generated with a unique ID.

Finally, we're going to define the "submitNew" function in our app.js file, inside the scope of our "TodoController" controller. This function will use an HTTP post method to send a .json object to our server, which will contain the to-do's description, status ("checked") and the username of the user that submitted the to-do:

```
$scope.submitNew = function() {
        $http.post("http://127.0.0.1:3000/todos/new", {
                description: $scope.todoEntry,
                checked: false,
                user: $scope.user
        }).success(function(data, status) {
                $scope.todos.push(data);
                $scope.todoEntry = '';
        }).error(function(data, status) {
                console.log(data);
        });
};
```

Then, we'll define a new function ("updateTodo"), that'll allow us to update the status of our to-dos every time we check and uncheck the correspondent checkbox. This function is basically telling our server to re-write the todo with the same id;

```
$scope.updateTodo = function(todo) {
    $http.post("http://127.0.0.1:3000/todos/update", todo)
    .success(function(data, status) {
        console.log('todo updated: ', data);
    }).error(function(data, status) {
        console.log(data);
    });
}
```

The final part of our app.js file, will be a new function ("initTodos") which is going to start by resetting (initializing) our "todos" variable, and then is going to get our user's to-dos from the todoItems collection.
Then, the "todos" variable will be re-defined as an array that includes our user's to-dos.

```
$scope.initTodos = function() {
    $http.get("http://127.0.0.1:3000/todos?user=" + $scope.user)
        .success(function(data, status) {
            $scope.todos = data;
            console.log($scope.todos);
        }).error(function(data, status) {
            console.log('error: ', data);
        });
}
$scope.todos = [];
$scope.initTodos();
```

## Back-end Stuff

Now, we'll go back to our server.js to create the Node logic that will allow us to send our to-dos to our database and to update their status.

We'll use the following code to define the post method for our "todos" database. This method will create a new entry in our collection.

```
app.post('/todos/new', function(req, res) {
    todosCollection.insert(req.body, function(err, result) {
        res.json(result.ops[0]);
    });
});
```

To update the to-do's status when we check and uncheck the to-do's checkbox, we'll use this post method:

```
app.post('/todos/update', function(req, res) {
        var todo = req.body;
        todosCollection.update({
                _id: mongo.ObjectID(todo._id)
        }, {
                $set: {
                        checked: todo.checked
                }
        }, function(err, result) {
                if (err) {
                        res.status(500).send({
                                error: 'error updated'
                        });
                } else
                        res.json(result);
        });
});
```

We'll use this code to define the get method for our "todos" database. This method returns our to-do list, which allows us to display this list below our form..

```
app.get('/todos', function(req, res) {
    todosCollection.find({
        user: req.query.user
    }).toArray(function(err, items) {
        res.json(items);
    });
});
```

Now, when we open up our "mytodolist.html" page, it looks something like this:

Then, we can write a new to-do in our text input field …



… and when we hit submit, our page looks like this:

We'll finish by altering our CSS, to create a line-through effect in our checked elements:

```
.strikethrough:checked + label {
    text-decoration: line-through;
    color: gray;
}
```

Now, when we check one of our to-dos, the label turns gray and has a line-through effect.