# Workshop: Messages

Workshop description for the [“ASP.NET Core” course @ SoftUni](https://softuni.bg/courses/asp-net-mvc).

# Web API

## Messages

Let's create a **Messages** **Web API** in which Users can choose a username and chat with each other.

## Database

The **Database** of the **Messages** application needs to support only **1 entities**:

### Message

* Id - a GUID String, Primary Key
* Content - a **string**
* User - a **string**
* CreatedOn - a DateTime object.

Implement the entities with the **correct datatypes**.

## Functionality

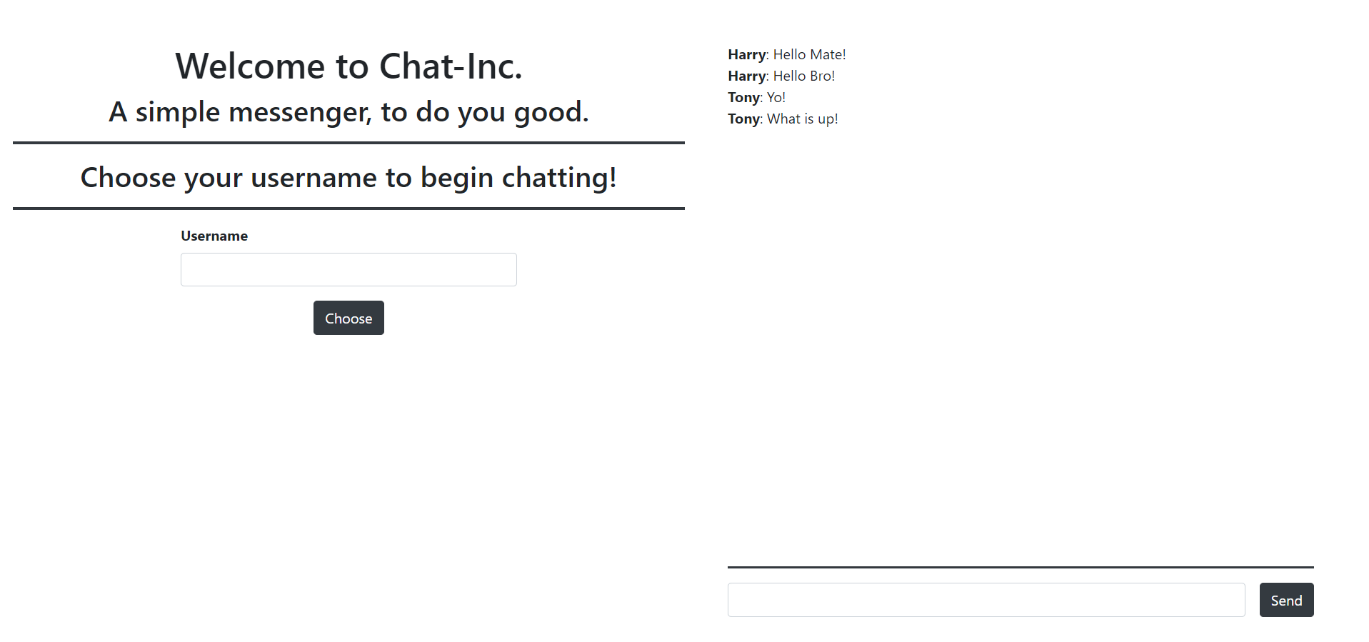
Users should be able to create messages and list all messages, ordered by createdOn.

There are 2 endpoints you must implement:

* /api/messages/all – **Returns** all **Messages**, ordered by CreatedOn
* /api/messages/create – **Creates** a **Message**, by a given **content** and **user**

## Front-End

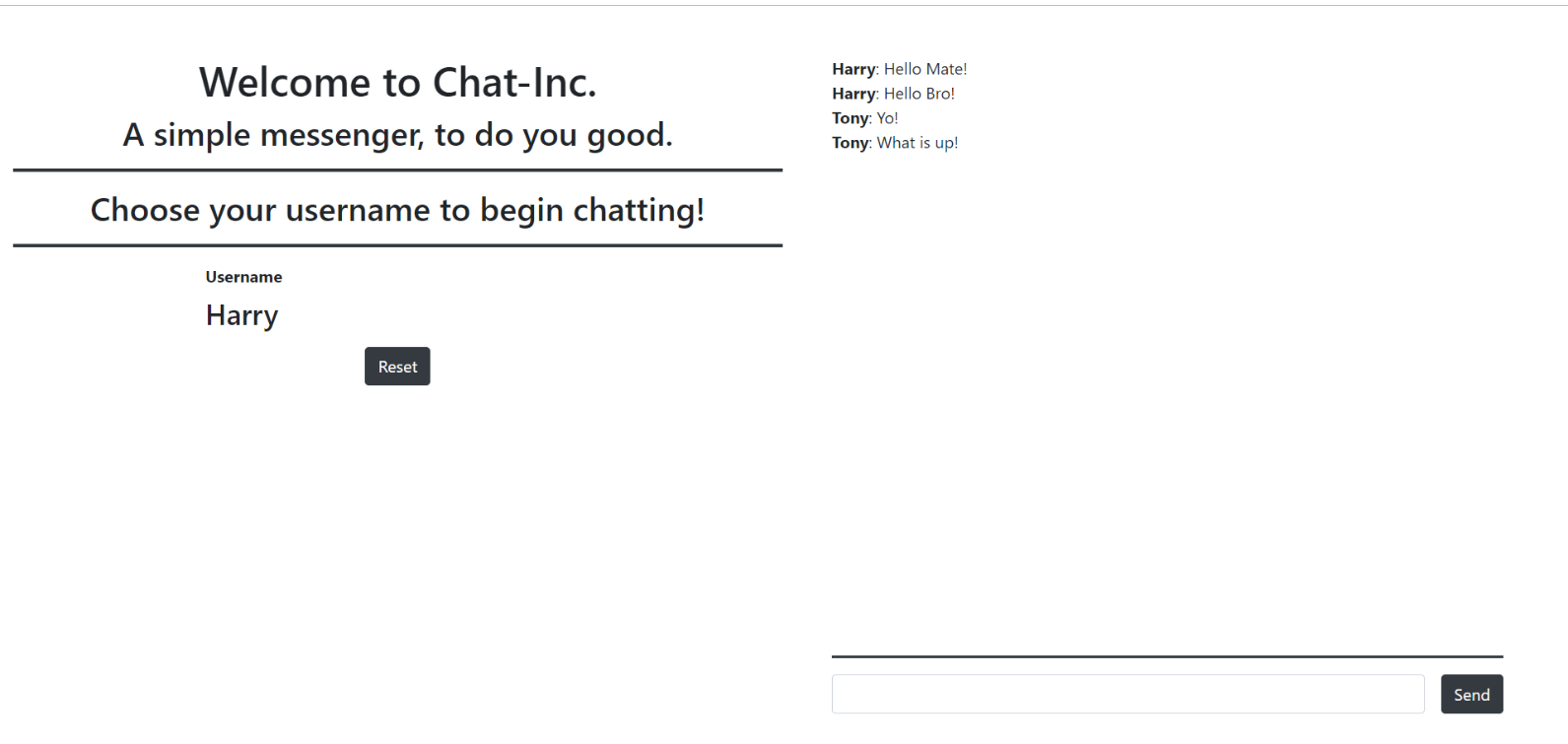
You will be given a simple HTML page, styled with Bootstrap. The page is constructed to be the Front-End of the Messages application. It has a simple form for choosing your current Username and a simple form for sending messages. It also has a list of messages sent, which is all the messages that are currently in the database of the application.



You do not need to touch any of the CSS. You will however be given a app.js file which you must implement. The Web API must be consumed, and you must do that with JavaScript.

### Username Functionality

Upon Choosing a Username (clicking on the [Choose] button) the following view should appear:



Upon clicking [Reset], the **Username** should be reset, and you should be able to choose another **Username**.

### Messaging Functionality

Upon clicking on the [Send] button a message should be sent to the **Web** **API** and it should be **created** in the database. All of the **messages** should be **refreshed** (**listed again**) so that the new message can be attached.

### Micro-Validations

Introduce micro-validations such as:

* You should not be able to send a **message** **without having** chosen a **Username** first.
* You should not be able to choose an **empty Username**.
* You should not be able to send an **empty Message**.

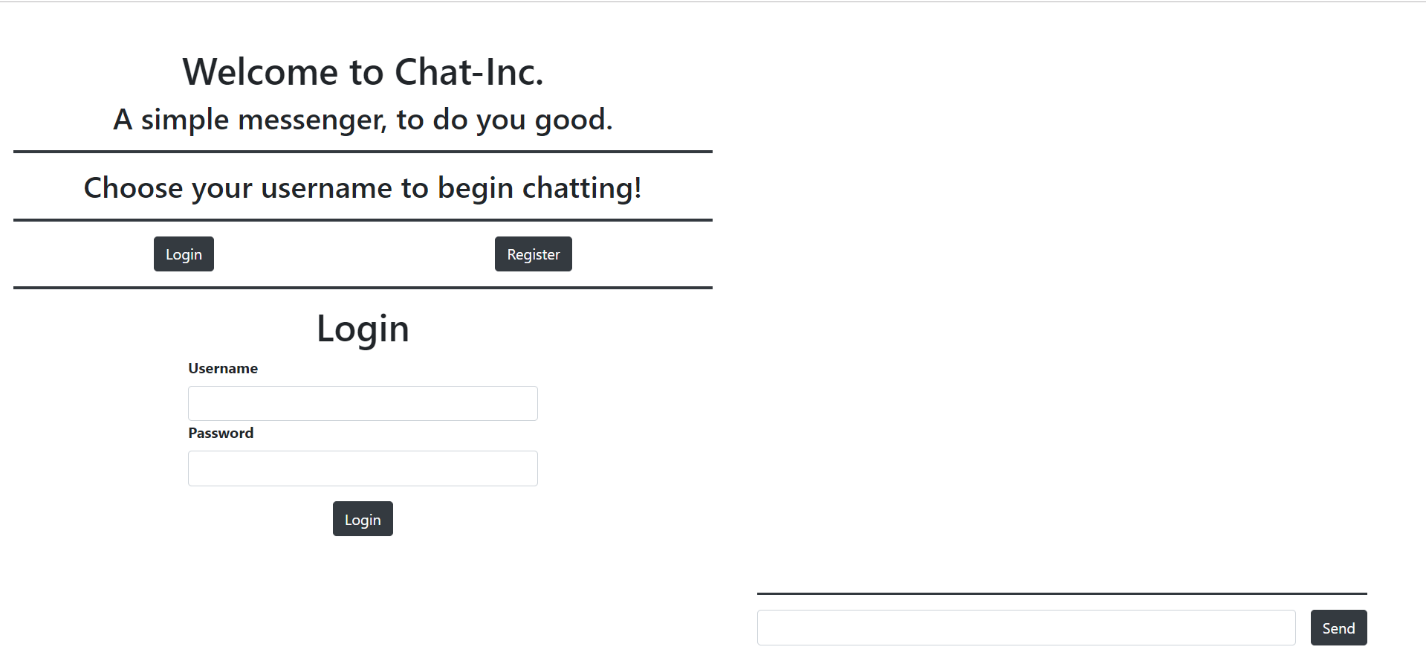
# Security and Identity

Let’s extend the **Messages** **Web API**, by adding User Registration and User Authentication to it. This will add Users to the database and remove the plain text users from the Messages.

## Front-End Changes

We will need to change the index.html a little.

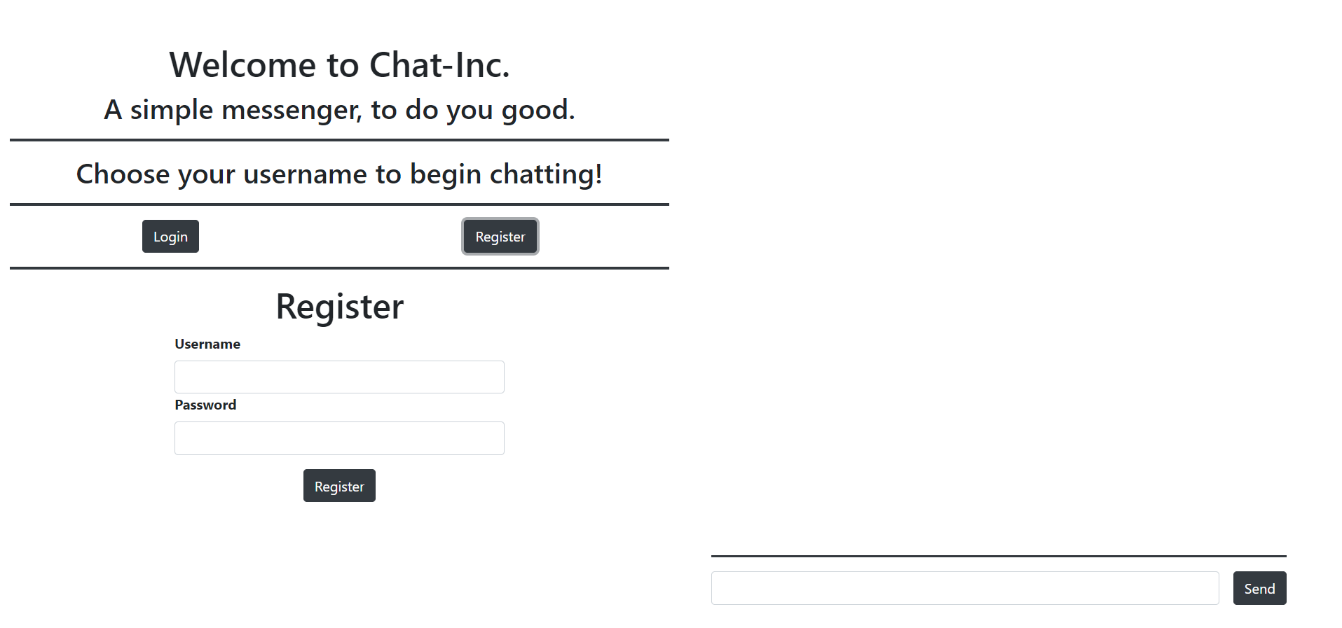
### Login Functionality



As you can see there is a new section with 2 buttons on it. When you click on the [Login] Button the Login Form should be visualized as shown above.

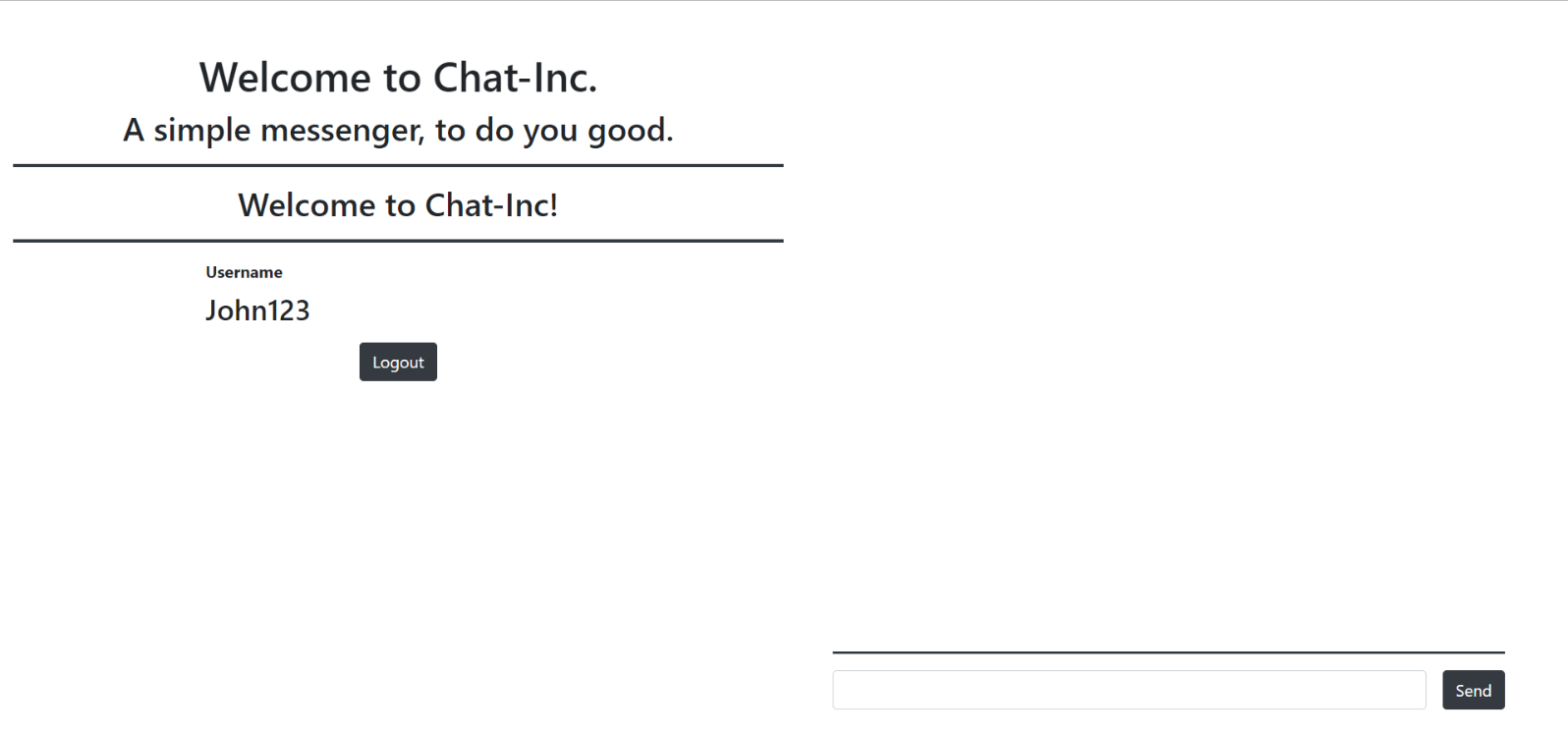
### Register Functionality

When you click on the [Register] button the Register Form should be visualized as shown below:



### Logged-In Functionality

Upon successful Back-End Authentication, you will get a JWT token. You need to store it in the Local Storage. Also, the following front-end should be visualized:



The caption above changed to "Welcome to Chat-Inc!".

The username of the currently-logged-in User is visualized.

The [Logout] button is visualized.

Upon clicking the [Logout] button the token should be deleted from the Local Storage and the Login Form should be visualized.

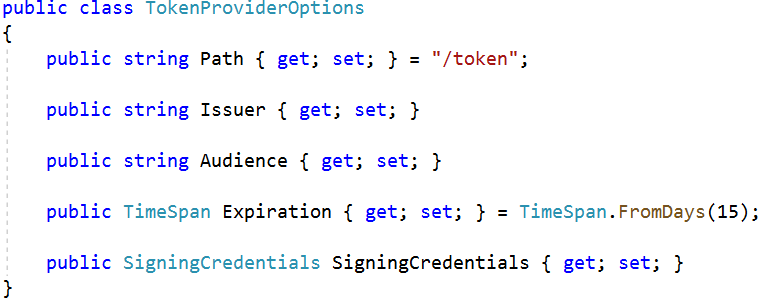
## Back-End Changes

You will need to add a simple User entity, which holds a Username and a Password. Later, this will be changed.

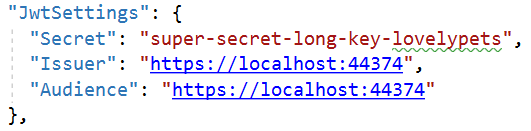
You will also need to implement appropriate Endpoints for User Registration and User Authentication.\

### Hint – Configuring the JWT

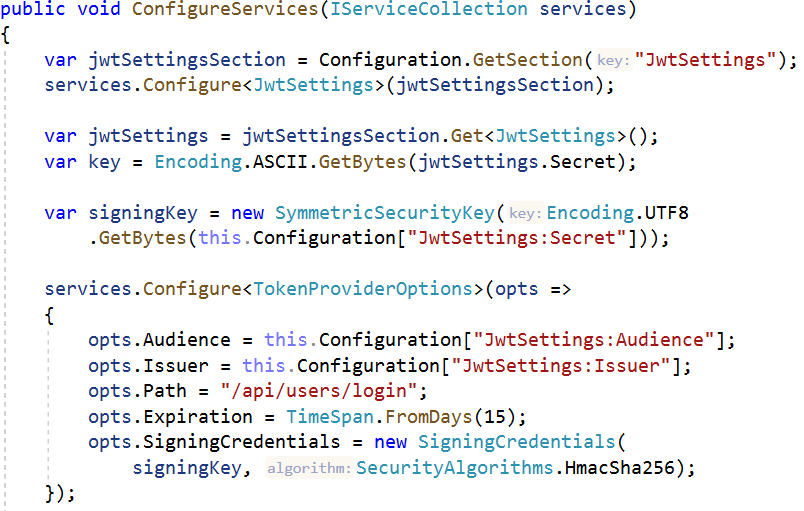
Let’s create a folder named **Jwt** where we will hold classes helping us to implement **JSON Web Tokens** to the application. Create two classes - **JwtSettings.cs** and **TokenProviderOptions.cs**. The first one will hold only one property – **Secret** (a **string**). The second one should look like this:



You can change the Expiration Days how you’d like. Now let’s add **JWT Settings** to **appsettings.json**.

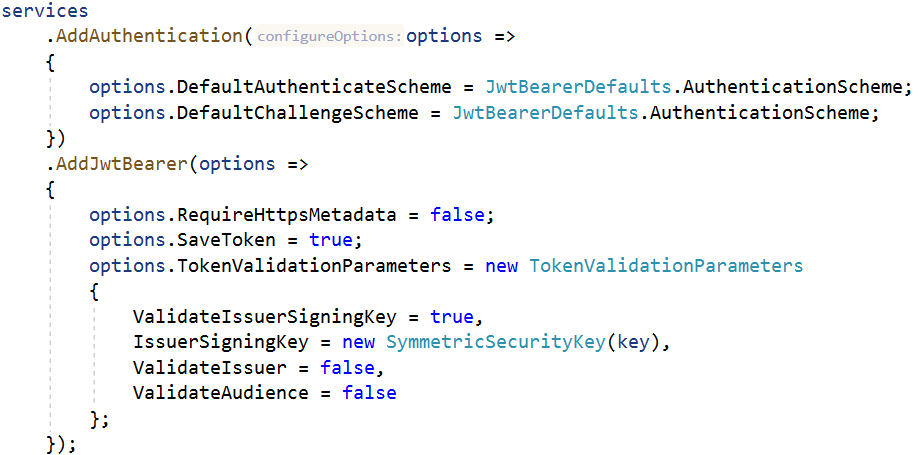


It’s time for the configuration in **StartUp.cs**. In **Configure()** we should have this code:



We get the **Secret** string from the class we wrote and create a **signing key**. Next we configure the **token options** with the options of the token and get the values from the **appsettings.json**.

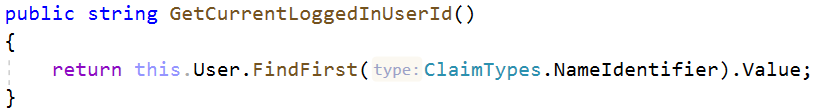
Next we say to our application that we will use **authentication** and **JWT Bearer** and we set options for them.



The magic of creating a token is in **Authenticate()** method in **UserService**. We will use it in our **Login()** method in **UsersController**. The implementation of the method is the same as in the presentation.



With the creation of **Claim** we can **access the logged in user** in the application like this:



## Common Functionalities

While implementing these functionalities, try to keep the old functionality of the application intact:

* The Front-End should still be a single page application.
* Messages should still be visualized to the User, even if he is logged out.
* The Validations should be the same as before.

## Bonus Task: Identity User

Try to change the **User** functionality on the back-end so that it works with the Identity User, from ASP.NET Core.