**e-Project**

**Sprint #1 Report**

**Team Members:**

Lazaron Shyta - 100195078

Gabriel Kuka - 100189808

**1. Intro**

Our project consists in building a website which stores student projects online. For the development of this project, we decided to use React (javascript) as the frontend platform and Django REST (python) as the backend platform. For sprint 1, we had four user stories:

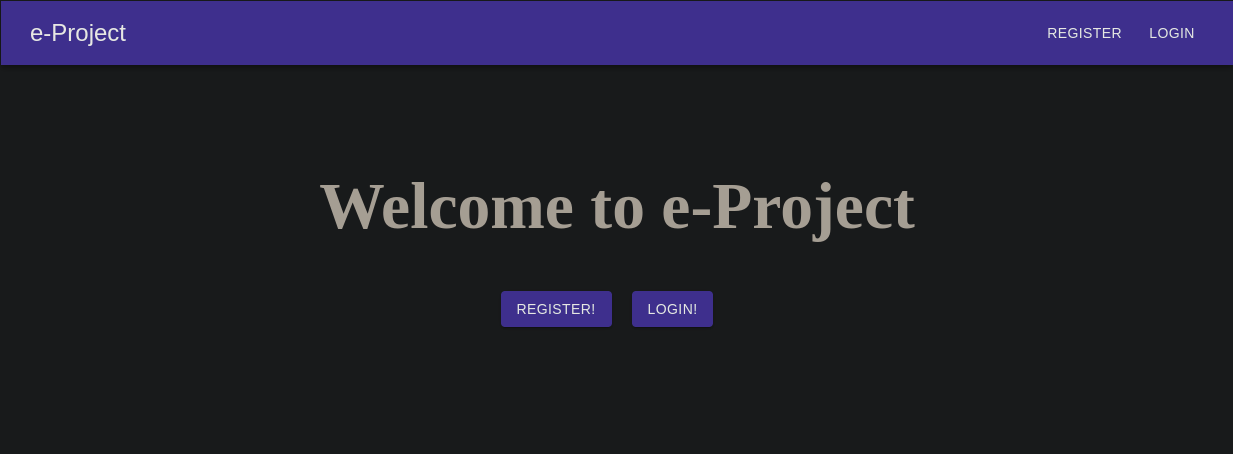
1. Display a “Welcome” page as a default landing page.
2. Implement a registration feature.
3. Implement a login feature using an email and password.
4. Implement a logout feature.

**2. Software Architecture**

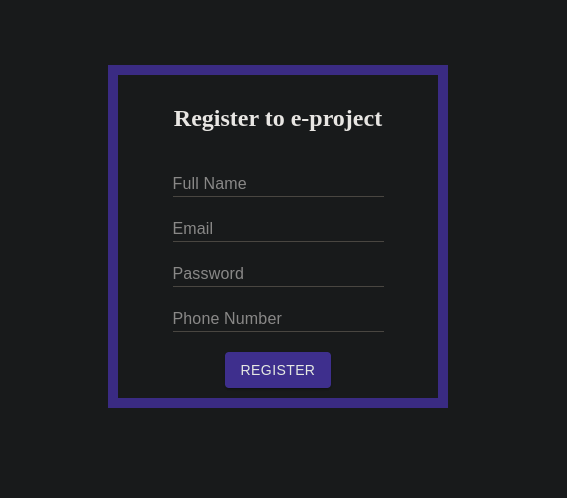
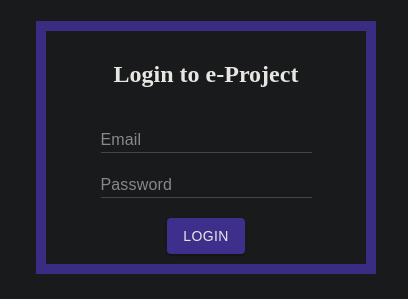
The project is divided into two folders: Backend and Frontend. Because we are using React for the frontend development, no specific software architecture pattern is used. Beneath the UI stand the React components, services, and reducers. Redux is used for state management and the connection Frontend-Backend is made in the services file. Each of them stored in a separate directory. On the backend side, the software architecture used is a fully RESTful API. Currently, the backend side consists of two main applications, accounts (which deals with user authentication) and api (will be used for other endpoints).

**3. Frontend**

The website currently has four pages: registration page, login page, the homepage, and the user profile page. Below is shown the welcome page where an anonymous user first lands:

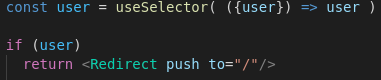


Below images show the login and the registration forms.



In the registration part, the user inputs his/her full name, email address, password, and phone number. The email must be unique, otherwise one is unable to create an account. Moreover, there are two restrictions when it comes to choosing a password. A password must be at least 6 characters long for security reasons and it must contain both letters and numbers. If one is missing, the user cannot create an account. Furthermore, the anonymous user provides a valid phone number to store in the account. The formats of the password and phone number are checked using regular expressions. The rest is done using React.

For a better web form performance, we decided to use Formik. Formik is a javascript library for React that aims to ease form handling. For data validation, we are using Yup which is another library commonly used in React for web forms. The login and register both use a function-based component to handle and render login forms. Before, we show the login page however, we make sure that the user is not logged in:



The above code first retrieves the status of the user, then checks its value. If the user is set, then there is no need for logging in. In this case, the user is redirected to his homepage. In case we are dealing with an anonymous user then the Formik component is rendered to the screen.



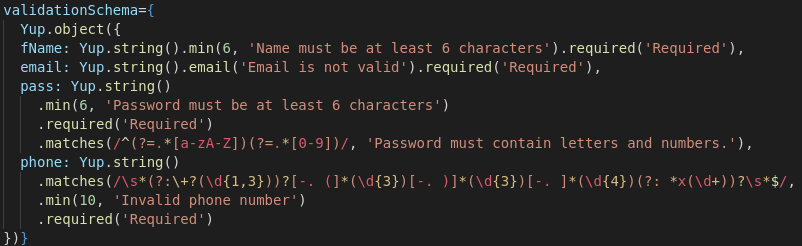
First, we set the initial values of the fields by using the “*initialValues*” attribute. By defaults, when the user lands on the page, the fields will be empty. The second attribute of the Formik component is “*onSubmit*”. Here, we create an anonymous function with a number of parameters needed to execute the code. This function is invoked when the user clicks the login button. The payload variable contains a dictionary with the entered values of the email and password. This variable is then passed to the dispatcher which handles the logging in process. The form is thereafter reset.

User reducer is the file that handles actions like logging in and out. The “*user\_login*” is a function which takes as a parameter the payload.

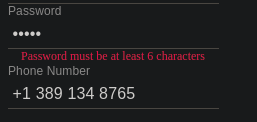
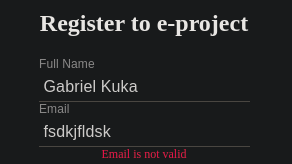
1. Send a request to retrieve the user token for authentication.
2. Retrieve and save user data.
3. Notify the user about the status of logging in.

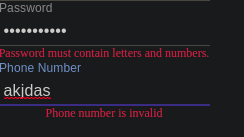


The registration process has a few extra steps compared to logging in. In this case we must perform data validation.



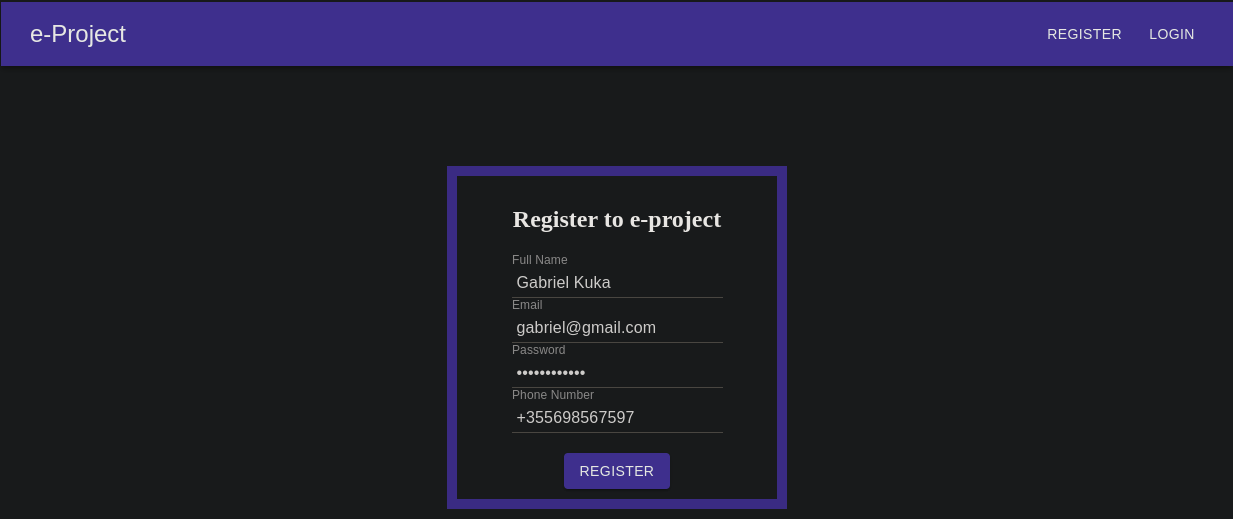
The image above shows how validation is handled. Besides the “*initialValues*” and the “*onSubmit*” attributes, we have added the “*validationSchema*”. The Yup library is used to determine the rules of the fields. For example, we decided that the full name should be at least 6 characters. The email field must have a proper email. All of these fields, together with the password and phone number field are required. The regex expression: *^(?=.\*[a-zA-Z])(?=.\*[0-9])* matches only if the user enters letters and numbers together, otherwise a message is displayed letting him/her know. The phone number regex expression matches only when the user enters a valid phone number. The number could begin with +, or prefixes or not. Moreover, the expression allows formatted numbers like (###) - (###) - (####)



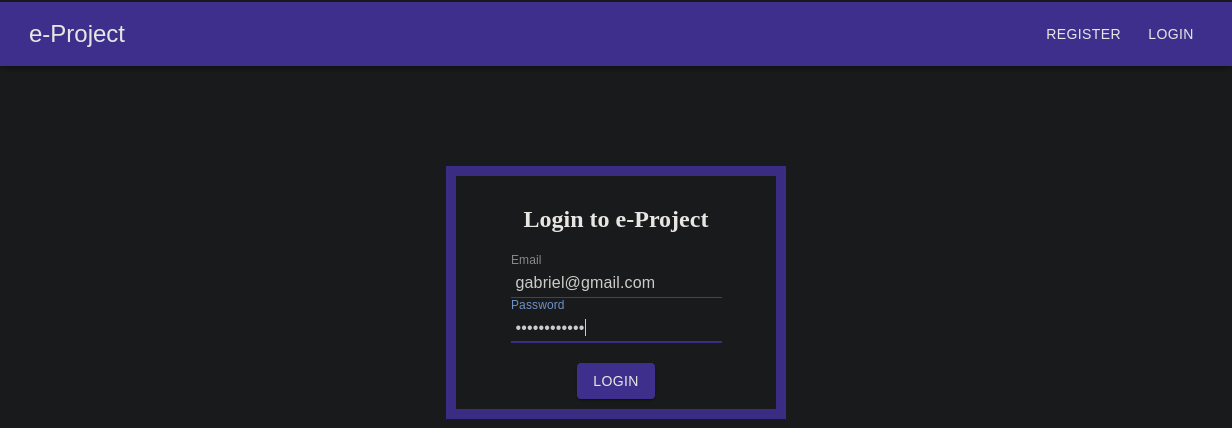


After submitting, a POST request is performed to the backend with the entered credentials. Below are shown a set of screenshots depicting the registration, login and logout process.

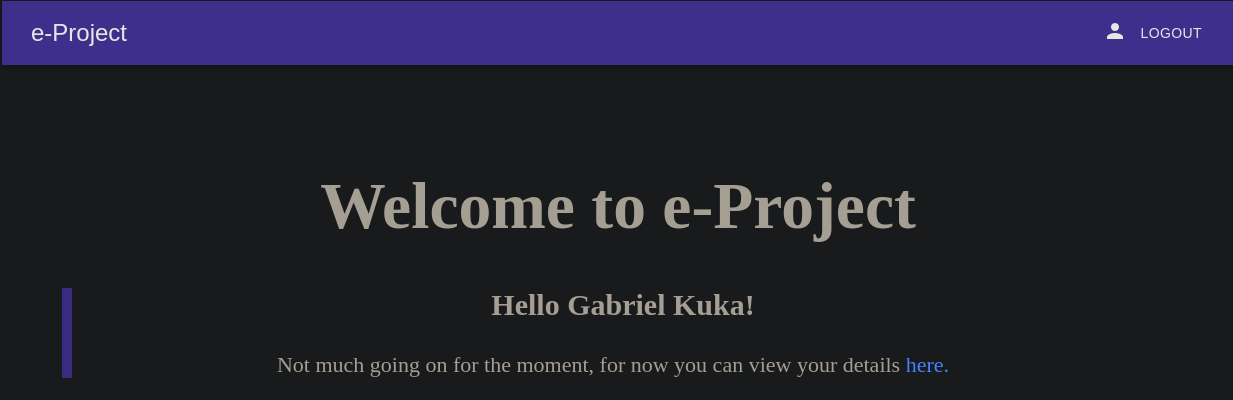
Registering a user



Logging in



The landing page after logging in.



**4. Backend**

The Backend we are using consists of 2 django applications:

**API**

This is the core application in which we will write all the application logic. There are 4 main files in this application: models.py, views.py, serializers.py and urls.py.

In the models.py we write all the models, or database tables that will take part in the project. Currently the only model we have is the User model. We customized the default User model provided by Django as it comes with certain limitations.

class User(AbstractBaseUser, PermissionsMixin):

email = models.EmailField(max\_length=255, unique=True,null=False)

name = models.CharField(max\_length=255, null=False)

phone = models.CharField(max\_length=50,null=False)

is\_active = models.BooleanField(default=True)

is\_staff = models.BooleanField(default=False)

objects = UserManager()

USERNAME\_FIELD = 'email'

As we can see, to create the User model we override the AbstractBaseUser and add some extra attributes (email, name, phone), is\_active and is\_staff are default attributes that we override to set the status of the user as active.

The objects = UserManager() line, just tells django that the model User, will be accessed from the UserManager() method that we will define in the same file:

class UserManager(BaseUserManager):

def create\_user(self, name,email,phone, password=None, \*\*extra\_fields):

""" Creates and saves a new user """

if not email:

raise ValueError("Users must have an email address")

user = self.model(email=email,name=name, phone=phone, \*\*extra\_fields)

user.set\_password(password)

user.save(using=self.\_db)

return user

def create\_superuser(self, email, password):

""" Creates and saves a new superuser """

user = self.model(email=email)

user.set\_password(password)

user.is\_staff = True

user.is\_superuser = True

user.save(using=self.\_db)

return user

The purpose of this method is to manage the way the User model can be used to create a particular type of user, like normal or superuser (set is\_superuser attribute as true).

After defining the User Model, we need a way to use that user in our application. All the logic for dealing with Users is implemented in the **accounts** application. We do so by using the views.py file. We create a method called CreateUserView(), which can handle the POST request sent to the **api/accounts/create** endpoint. The connection of views and particular urls is made in the urls.py file of the application:

app\_name = 'accounts'

urlpatterns = [

path('create/', views.CreateUserView.as\_view(), name='create'),

path('token/', views.CreateTokenView.as\_view(), name='token'),

path('me/', views.ManageUserView.as\_view(), name='me')

]

We can see that in the path with name ‘create’, the middle argument is views.CreateUserView.as\_view() , which means that the CreateUserView will get to handle the request sent.

class CreateUserView(generics.CreateAPIView):

""" Create a new user in the system """

serializer\_class = UserSerializer

That is how a generic.CreateAPIView class looks in DjangoREST. A lot of repetitive code is skipped here. The single line we have to write inside is to specify the serializer class of the view.

A serializer is nothing more than a set of functions or a class that makes possible the conversion of objects to JSON or vice-versa.

class UserSerializer(serializers.ModelSerializer):

""" Serializer for the user object """

class Meta:

model = get\_user\_model()

fields = ('email', 'password', 'name', 'phone')

extra\_kwargs = { 'password': {'write\_only': True, 'min\_length': 6}}

def create(self, validated\_data):

""" Create a new user with encrypted password and return it """

return get\_user\_model().objects.create\_user(\*\*validated\_data)

def update(self, instance, validated\_data):

""" Update a user, setting the password correctly and return it """

password = validated\_data.pop('password', None)

user = super().update(instance, validated\_data)

if password:

user.set\_password(password)

user.save()

return user

Here, the validated data is received from the view and is used by two optional overridden functions like create() and update(). The create function gets the data from the view and calls the create\_user() method we defined earlier in the UserManager class and returns it to the view, to later be returned in the response with a HTTP\_201\_CREATED header. This is because the CreateUserView inherits from generics.CreateAPIView, which is an abstracted way of handling requests to prevent repetitive code. Basically, what the view does, it takes the payload from the request, send it to the serializer, the serializer serializes the JSON from payload and sends it to the UserManager to create the User. If all this is successful then the client will get a 201 Created Response with the user info (not the password).

**User Authentication**

For authentication we used Token-based authentication rather than using the default Django Session authentication. Basically each user can retrieve its own unique Token from the ‘api/accounts/token’ endpoint. The o punetor :P hahah

class CreateTokenView(ObtainAuthToken):

""" Create a new auth token for the user """

serializer\_class = AuthTokenSerializer

renderer\_classes = api\_settings.DEFAULT\_RENDERER\_CLASSES

The view above handles the request and uses the following serializer:

class AuthTokenSerializer(serializers.Serializer):

""" Serializer for the user authentication object """

email = serializers.CharField()

password = serializers.CharField(

style={'input\_type': 'password'},

trim\_whitespace=False

)

def validate(self, attrs):

""" Validate and authenticate the user """

email = attrs.get('email')

password = attrs.get('password')

user = authenticate(

request = self.context.get('request'),

username = email,

password = password

)

if not user:

msg = \_('Unable to authenticate with provided credentials')

raise serializers.ValidationError(msg, code='authentication')

attrs['user'] = user

return attrs

The serializer makes sure the email and password are valid and if so, return a response with the token.

**5. Backend and Frontend Connection**

The backend we developed is totally independent from the frontend. In the React project, we handle the requests with the ‘axios’ library, which is pretty much like fetch. We handle the requests in an asynchronous way using await async.

All the API calls are made in the services.js file of the React Project.

Example of a backend call in service.js:

const baseURL = 'http://localhost:8000/api/accounts/'

const register = async (credentials) => {

const response = await axios.post(`${baseURL}create/`, credentials)

return response.data

}

It receives the credentials from the Register.js component (from where the function is called) and sends the POST request to the server. Then it receives the response and returns it back to the Register.js where the call was made.

**6. Tests**

Authentication methods in the API are supported with tests in the tests.py file.

Example:

class AuthenticationTests(TestCase):

""" Test the users API """

def setUp(self):

self.client = APIClient()

def test\_create\_valid\_user\_success(self):

""" Test creating the user with valid payload is successful """

payload = {

'email' : 'lazo@test.com',

'password' : 'test123',

'name' : 'Test Name',

'phone': '+123 1234123 12'

}

res = self.client.post(CREATE\_USER\_URL, payload)

self.assertEqual(res.status\_code, status.HTTP\_201\_CREATED)

user = User.objects.get(\*\*res.data)

# Password shouldn't be in response

self.assertTrue(user.check\_password(payload['password']))

self.assertNotIn('password', res.data)

**Individual Contribution:**

* Lazaron Shyta: Backend, User Authentication Backend, Backend Tests, State management Redux, Design
* Gabriel Kuka: User Authentication Frontend, Forms validation Formik and Yup