École Polytechnique Fédérale de Lausanne

D-Voting : Frontend

by Ambroise Borbely

Bachelor Project Report

Noémien Kocher
Project Supervisor

EPFL IC DEDIS
Bâtiment BC
Station 14
CH-1015 Lausanne

January 7, 2022

Contents

Chapter 1

Introduction

The DEDIS lab is currently developing solutions about e-voting, the project explained here is about the D-Voting project that aims to create an e-voting platform based on the DELA blockchain. In order to interact with the blockchain, a web application has been developed, and will communicate with the DELA blockchain trough an API that is on each node of the DELA blockchain. This project is about creating a "V2" of the currently existing web application by adding new features. The main feature added to this application is the login with the tequila service. As authentication makes significant changes in the architecture of the app, all the other features will come on top of this one to complete it.

This document explains how the application was at the beginning and it has been transformed during the semester with the new features. At the end, some ideas are given on possible next changes that could be made to the web application.

Chapter 2

Context

To begin, it is good to know how the site was working before its modification. The web application is composed of a single web application (process) based on React. This is a single web page app that uses react-router to choose which component should be displayed depending on the current url. To access the DELA nodes, the client makes requests to the react process and the server forward them to one of the DELA node : this is defined in the `package.json` in the `proxy` parameter. In the V1, also every user is able to do all the action on the website : every user can create election and close them, which not always suitable.

The requests that needs to go to the DELA nodes are sent via the React server because of a policy named CORS which is explained briefly later in this document.

Chapter 3

Use of a second process

In order to authenticate the users on the website, the first thing to do was to add a second process (web process) that will be the trusted process who decides and says to the DELA nodes and the frontend webpage which user is currently logged in and certifies that the data sent to the blockchain are actually from a genuine user. As React is a library only made for frontend purposes a second process needs to be created.

After some researches, the backend chosen for this project is ExpressJS which is very well explained in many tutorials found online. Furthermore, this backend framework is known to be quite popular, so it seemed to be a good choice to use it!

At this point, the client has to exchange data with 2 backends: something which is not possible with only the configuration stored in the `package.json`. So the first thing has been to use the middleware `http-proxy-middleware`. This allows to re-route the traffic to one of the two backends or the other depending on the url. In order to not mix calls that needs to go to the DELA nodes and the express server, for now all the DELA endoints start by `/evoting`, I choosed that all the endpoints on the express will start with `/api`. For information, this not absolutely needed to do so in the backends as url rewriting is possible with this middleware which means that is possible to only prefix the access url path on the frontend (i.e. on the react).

It also important to mention one point which seems important : why does the web client does not directly exchange its voting data with the DELA nodes instead of being sent to the react server ? The reason is because of a web policy which is in every web browser : the CORS policy. CORS stands for Cross-origin resource sharing. This policy has the goal to block some of the requests that the client sends to others websites that the one where the webpage is hosted. This is is the case of all the AJAX calls by default. So it means that if the express server is not behind the react server it would not be possible to do AJAX call to it. This policy is there for security purpose: as a web browser is "logged in" a lot of web application online it means that for instance with this kind of cross origin requests it could be possible to access the google drive data of the user via its web browser by sending requests to google drive from the client's web browser (this is just

a toy example, it does not means that it works all the time like this and talking about google drive is also just to explain the general idea of the possible attack). The decision has then been made to proxy the request via the React.

Express works the following way: first we create an `app` object, and then we add "routes" (which are endpoints) and we assign a function to each one of those routes that are triggered at each time that a web client is hitting the corresponding url. A response is then sent back to the clients. The code is located in `web/backend`.
Once this package have been installed (by doing a `npm install`, the app can be started with the `npm start`.

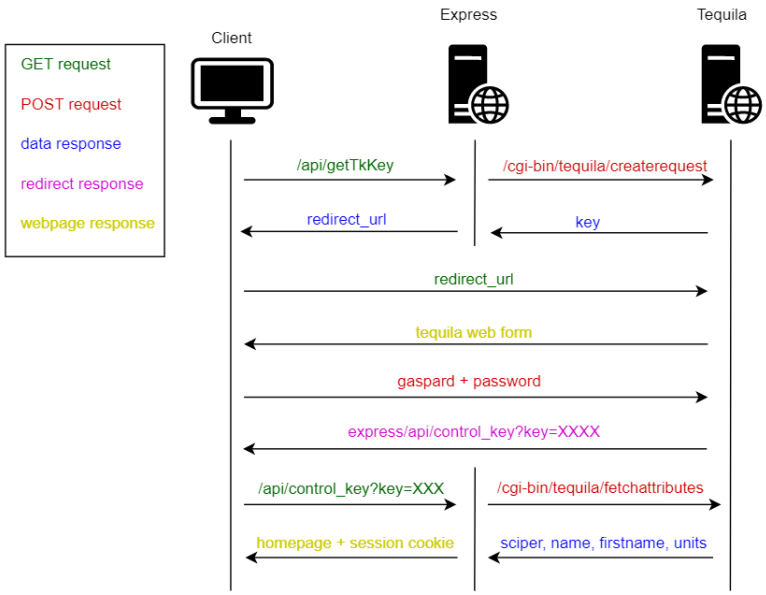
Chapter 4

Implenting the tequila

Tequila is the autentication service provided by EPFL to authenticate its users on websites. It is quite practical to have this service as it allows web developers to never have to deal with user's password directly. Furthermore it allows users to enter their credentials once and then to get logged in all the applications that uses tequila without entering again the credentials (as tequila "remembers" which user as already logged in for a particular session in a web browser).

Tequila works the following way: first the user goes on the website he wants to log in. The website reaches Tequila and ask it a token (which is called in this case a key), during this connection to the Tequila, the website will tell to the Tequila server which information on the user it needs and also the callback url. The callback url is the web address that the Tequila will use to redirect the client after entering his/her credentials on the Tequila's form. The website redirects the web client to the tequila frontend page (to a custom url which contains the key) which is just a form with a field for the username and a field for the password. The tequila check the credentials and if they are correct, it redirects the web client to the callback address it received in the previous step. In addition the Tequila add another token (or key) as a GET parameter in the callback url. When the website receive the GET parameter it sends the new key back to the Tequila. The Tequila will then confirm that the token is valid and if so, send all the asked data to the website (username, name, etc...). At this point the identity of the user is confirmed and the website needs to log in the user on its own pages.

For this "website login" we will use a common technique which is called a session, which is basically a trusted cookie on the web browser which contains at least enough information to identifies a user. Also the cookie is secure and no one should be able to create this kind of cookies to impersonate a user. In this project `express-session` is used which is a server-side session manager which creates sessions in express server-side (which means that the session data like the sciper, etc... are not stored in the cookie itself but on the server only). Once this module is installed and combined with the use of tequila we create the following endpoints:



- `/api/getTkKey` : requests a tequila key and forwards it to the client directly embeded in the complete url
- `/api/control_key` : this is the callback url we talked about just above, this is there that the sciper, name and firstname is set in the session of each user.
- `/api/logout` : this destroys the session
- `/api/getpersonnalinfo` : as the session is handled by the express backend, this route allows react to get the information of the currently logged in user. On the react side, those data are then stored in a state once they are fetched.

Chapter 5

User's identity on DELA Nodes

Since last chapter, the user is now authenticated in the react process and the express process (which is the trusted authority for authentication for the whole app). Now it is needed to create a trusted connection between the express (who knows the identity of each user's request) and the DELA node, which trust every random IDs for now.

To deal with that, 2 changes are needed : the redirection of the traffic that goes to the DELA nodes through the Express server too and then to certify that the user's data are actually coming from this user.

For the first point, it it just needed to modify the configuration of `http-proxy-middleware` to also redirect all the traffic that goes to `/evoting` through the express server. We create then a wildcard route on the express `/evoting/*` that handle all the requests that needs to be transmitted to the DELA nodes.

For the second point, we will create a way that the DELA nodes trust the Express server, the data that are sent to the DELA nodes must also resist man-in-the-middle attacks. As we know all the requests that needs to go to the DELA nodes pass through the Express server, we will then slightly modify all the payloads that comes from the client before to send them to the DELA nodes. The DELA node code will also need then to be modified in order that the application is still works. To make this, a wildcard route for `/evoting/*` is created and then the following things are made on each requests :

1. Recover the payload (the body) of the request
2. Add to the payload the UserID/AdminID (which is the sciper) to authenticate the user on the request
3. Encode the modified payload in base64
4. Hash in sha265 the encoded payload

5. Using the kyber library, we sign the hash
6. We send a request to a DELA node containing the base64 encoded modified payload and the signed hash.
7. Once a response is received back from the DELA node it is transfered without modification to the client (at least for its content/body)

The schnorr method is used for the signature and the `crypto` library is used for hashing the payload.

Chapter 6

Role management

The last feature added is role management: the idea is that in the V1, everybody can vote, create an election, and stop the election (for the last one this is only the creator of the election that can do the whole process that allows to show the results of the election at the end). The idea is now to have roles that are set on each users that allow or not allow them to do certain tasks.

Three roles have been defined:

- voter which can only vote
- operator which can do everything that a voter can do and can create / stop / show the results of an election
- admin which can do everything that an operator can do and can add/remove admin and operators

It is also important to note that there is also the possibility to not have any of those roles if the user is not logged in.

For the implementation, 3 things have been done : create an administration page for the admins and create all its dependencies, change the react to display the right pages depending on the user's role and finally secure the endpoints depending on the user's role.

The creation of the administration page has quite a lot of dependencies : as there is data to store (i.e. the role of the users) a database is needed. This one is a MySQL database which contains only one table with the following fields:

- `id` : the primary key of the table
- `sciper` : the sciper of the user

- `role` : the role of the user

Then, to access and manipulate this table, let's create three routes (endpoints):

- `GET /api/get_user_rights` : lists all the user and their role (which is basically a `SELECT *` on the table
- `POST /api/add_role` : add a new user's role, the two fields to send to this route in the body are `sciper` and `role`
- `POST /api/remove_role` : remove the role of a user, the parameter to send there is only the `sciper`

Important note : only the admins and operators are stored in the database, the "voters" are the default role set to each user if there is not their sciper in the DB.
Note : `mysql2` is used to access the database.

To create the administrative page I decided to use `data-grid` of the library `material-ui` which seemed to be quite adapted for this and also particularly nice. I also used `material-ui` to create two new modals : one to enter the information of a new user's role and one to confirm the deletion of a user's role. Those components are located respectively in the `admin` and `modal` folders.

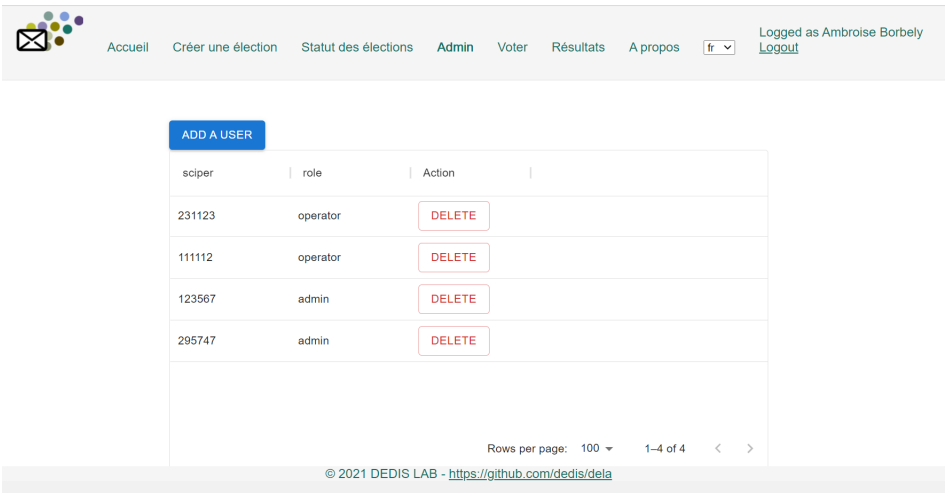


Figure 6.1: Administration panel view

The next thing to modify is the `react-router`, this step is quite simple as it is just needed to add the new administration component, then it is needed to modify the navigation bar to show/hide some of the fields of it.

To finish this feature we need to secure the routes on the express. For that I choosed to create a new very basic middleware that will forward the request (to the next middleware or to the

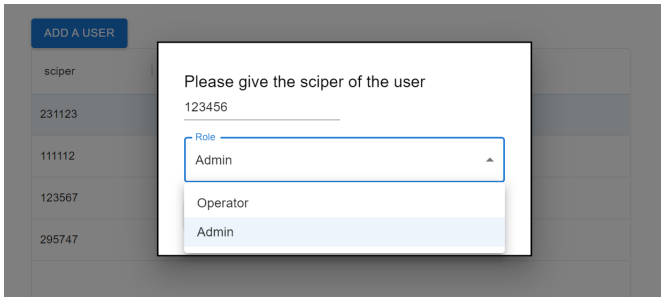


Figure 6.2: Modal to add a role to someone

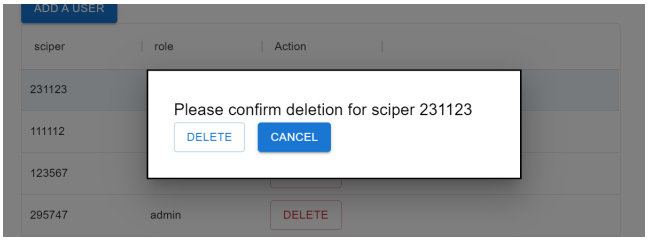


Figure 6.3: Confirmation modal to remove someone's role

correct endpoint's function) only if the user has the right to access the route asked. Otherwise it sends an error code. In order to classify which role is allowed to do what, a new file called `access_config.json` is on the root folder of the express source code.

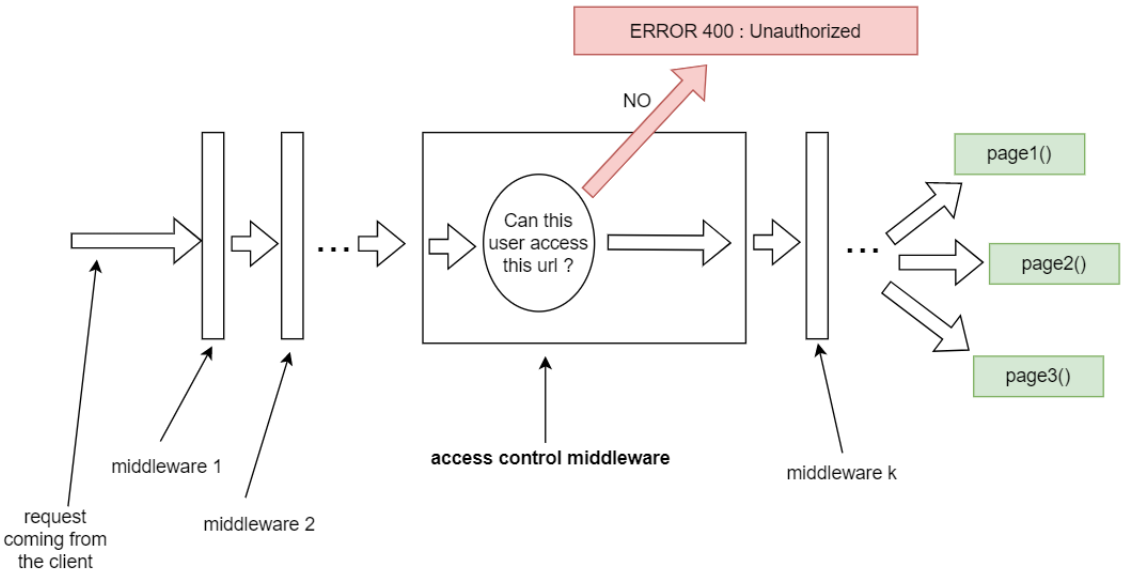


Figure 6.4: Middleware for access control

Chapter 7

Production settings

One of the main goal of this semester project was to bring the project to a production-ready state. In this mindset some work has also been done on a Linux server to prepare some configuration when the application will be used in production.

Two scenarios have been studied : one with both react and express processes on the same Linux server and one with the express and react not being on the same server.

All the information about that are directly located on the `/docs` folder of the project. There are there, the service file needed, the crond configuration and also the nginx configuration.

To explain a bit more the different files and what are their purposes, here is some more explanations:

- crond is a service available on Linux to create automated tasks that can run at various time schedules, as it is not very good to run a process for decades, crond is used here to restart express and react processes once a day (in the documentation it is at 2AM as there should not be too much traffic at this time).
- services files are the files that properly run processes on a server to create "service" those one can have multiple property and can for instance set to be restarted automatically when the server is rebooted (which is quite pratictal in case of a crash).
- NGINX is a web server it is used here to act as a reverse proxy : it will take all the requests that goes to the react and express processes and forward them. This is done because using NGINX can easily hold SSL and then make that the users use the app with HTTPS, the records are then decrypted by NGINX and sent via HTTP to the react server (as both processes are on the same server the data are not travelling outside the linux server and then it is not a problem to have unencrypted data).

Chapter 8

What's next

With the implementation of the Tequila and some other key features the application is almost production-ready: in theory the frontend completely works with all its new features but in practive, more tests would be great and also, as the backend team started working on some election with a richer content (and also some small changes should be done to make the signature part working on the backend side) the frontend is actually not compatible with the current version of the DELA nodes.

I wanted also to talk about a more personal thought about the maybe possible changes that could help the app to be better: the actual problem of this app is that the express server is a single point failure in terms of availability and security as all the votes goes through (for instance). One idea to at least maybe limit that would be that the client directly send the vote to the DELA nodes, which should be possible by configuring the react to allow the client to bypass in certain condition the CORS policy as it is explained in this website. The fact to bypass in certain condition led also to other problems as the express server is not able to filter the requests that goes to the blockchain and also it does not completely solve the problem of single point of failure but the problem is less significant.

Another idea that could be nice is the possibility to allow a vote only for a particular unit of EPFL. This option is not very complicated but needs to know exactly the current version of the API on the DELA nodes : to do this the units can be recovered during the last request to the tequila (in a login process of a particular user) with the same way as it is done for the firstname, name, etc... Then it could be stored in the session too.
Then if it is done the same way as the role, a table could be created with the unit info and its corresponding election ID (or the data could be also stored in the blockchain) for each election. Then the express would have to check on the vote request that the vote is valid knowing the user's units. The election list that a user asks should be also filtered by the express (or by DELA nodes) to display only the right election to each user. This feature would be a good idea in the context of EPFL which needs to do vote, take decision withing a section, a lab, a faculty, etc...

(which would need sometimes to specify more than one unit for each election).

Finally it could be a good idea to have the firstname and name in the administration panel. I personnally choosed to not do it because I would have to do it during the vacation and I would have wanted to confirm this idea first with my supervisor as it needs to scrap some webpage / EPFL service, which is not that trivial and depending also how it is done, maybe does not work 100 percent of the time.